

Contents

Course Overview	2
The Primary Key	3
Configuring the Table Name	10
Generated Values for Primary Keys.....	13
Generation Types AUTO and IDENTITY	17
Generation Type IDENTITY for Multiple Tables	21
Generation Type SEQUENCE for Primary Keys.....	25
Generation Type TABLE for Primary Keys.....	29
Composite Keys Using Embeddable and Id.....	40
Composite Keys Using EmbeddedId	50
Composite Keys Using IdClass	53
The Column Annotation.....	56
Precision and Scale Specification	63
Not Null and Uniqueness Constraints	67
Non-persistable Fields With Transient	70
Temporal Annotation for Date Fields.....	74
Lob Annotation for Large Objects	78
Embeddable Entities for Persistent Fields	82
Sharing Embeddable Objects	86
Begin and Commit Transactions.....	90
Read Operations.....	97
Uses EntityManager Find().....	97
Uses JPQL select Query	101
Update and Delete Operations	103
Perform Update Operation in JPA uses EntityManager merge()	103
Perform Delete Operation in JPA uses EntityManager remove().....	105
Entity Specification Using XML.....	107
Course Summary	113

Course Overview

JPA or the Java persistence API is focused on persistence. Persistence can refer to any mechanism by which Java objects outlive the applications that create them. JPA is not a tool or a framework or an actual implementation. It defines a set of concepts that can be implemented by any tool or framework.

JPA supports object-relational mapping a mapping from objects and high level programming languages to tables which are the fundamental units of databases. JPA's model was originally based on Hibernate and initially only focused on relational databases. Today there are multiple persistent providers that support JPA, but Hibernate remains the most widely used. Because of their intertwined relationship and JPA's origins from Hibernate, they're often spoken off in the same breath. JPA today has many provider implementations, and Hibernate is just one amongst them. Having understood the basic persistence model that JPA uses, in this course, we can focus on primary keys that can be set up for the tables underlying your entities. And how you can manage the configuration of individual columns in your database using JPA annotations. We will also explore how you can Perform, Create, Read, Update and Delete operations using the JPA entity manager. Once you're done with this course, you will be able to structure your entities and their attributes exactly based on what your specifications are, and perform basic storage and retrieval operations using JPA and Hibernate.

The Primary Key

In this learning path, our objective is to understand how the Java Persistence API works. So in essence, we are building different prototypes to understand the different features and functionality offered by JPA. Which is why in most of the cases, the database action property that we'll choose will be **drop-and-create**. Each time we run our application, we'll drop any existing tables, and we'll recreate those tables within our application. If you're working with JPA in production, you'll typically set this value to none, you won't use drop-and-create. But *it's perfect for prototypes*.

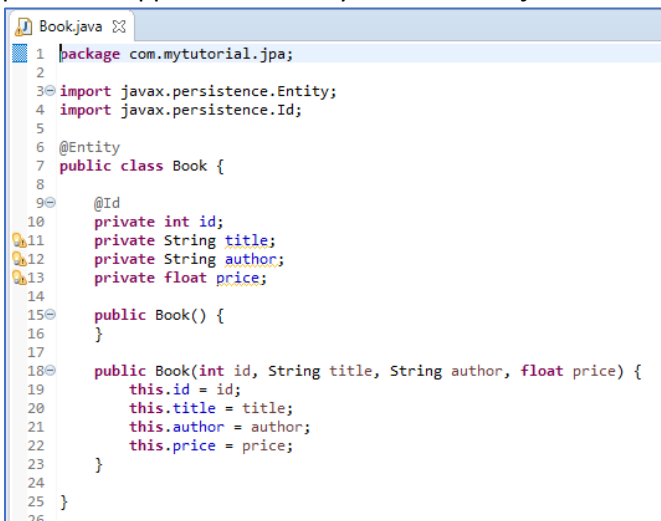
So let's work with this **persistence.xml**. We continue working with the BookstoreDB_Unit.



```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <persistence version="2.1"
3   xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
5
6   <persistence-unit name="BookstoreDB_Unit" >
7     <properties>
8       <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver" />
9       <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/bookstoredb" />
10      <property name="javax.persistence.jdbc.user" value="root" />
11      <property name="javax.persistence.jdbc.password" value="password" />
12
13      <property name="javax.persistence.schema-generation.database.action" value="drop-and-create"/>
14
15      <property name="hibernate.show_sql" value="true"/>
16      <property name="hibernate.format_sql" value="true"/>
17    </properties>
18  </persistence-unit>
19
20 </persistence>
  
```

Any tables that will be created will be based on the entities that we have in our application. Now, this particular application already has the **Book.java** class.

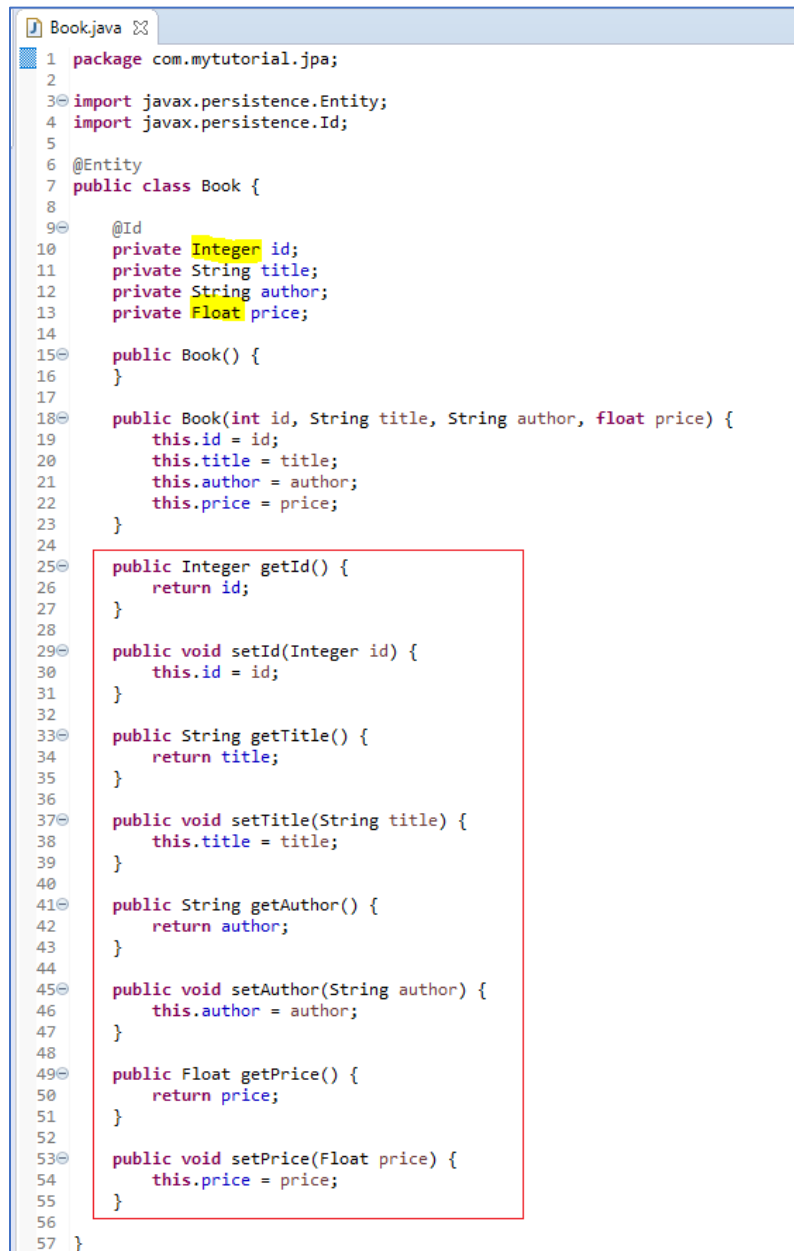


```

1 package com.mytutorial.jpaa;
2
3 import javax.persistence.Entity;
4 import javax.persistence.Id;
5
6 @Entity
7 public class Book {
8
9     @Id
10    private int id;
11    private String title;
12    private String author;
13    private float price;
14
15    public Book() {
16    }
17
18    public Book(int id, String title, String author, float price) {
19        this.id = id;
20        this.title = title;
21        this.author = author;
22        this.price = price;
23    }
24
25 }
26
  
```

- Book.java have annotated using the **@Entity** annotation indicating that Book is a table within our database and Book objects are records in that table.
- We also have an **@Id** annotation for the id member variable. We've discussed before that entity objects should have a default no argument constructor, which we do on lines 15 and 16. We also have additional constructors here as needed.
- We have just one additional constructor with four input arguments.

I'll now update the code contained within this **Book.java** entity class in a few subtle ways. I have the same member variables as before, *id*, *title*, *author*, and *price*.



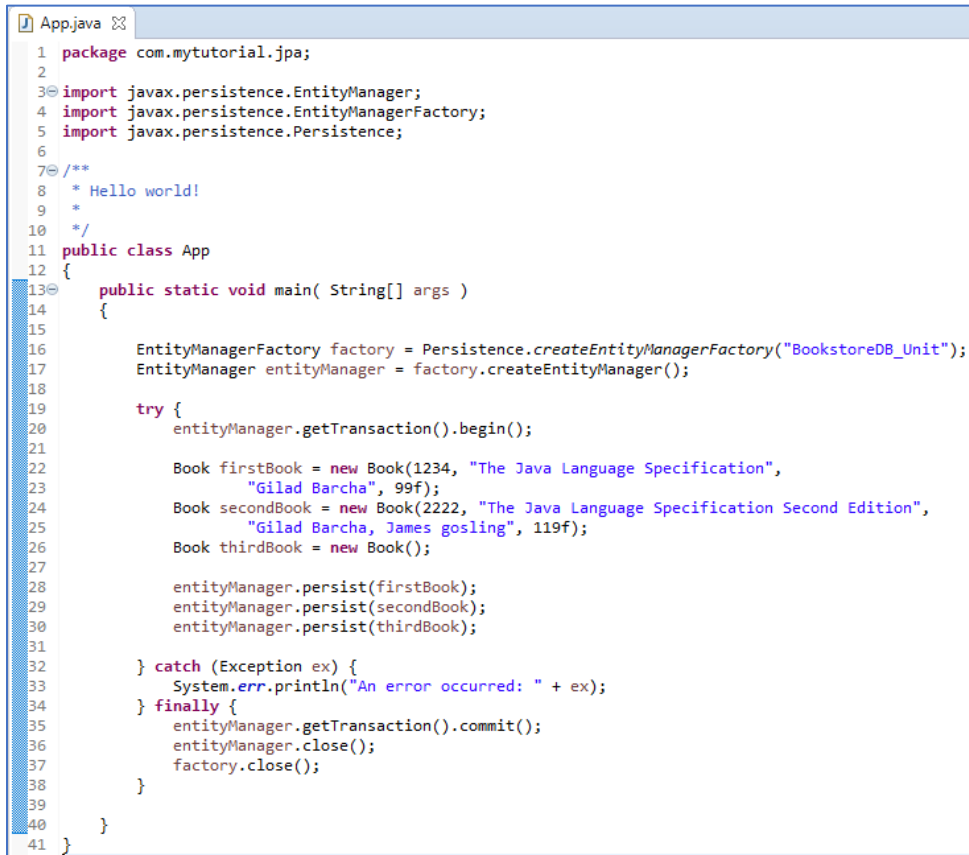
```
1 package com.mytutorial.jpa;
2
3 import javax.persistence.Entity;
4 import javax.persistence.Id;
5
6 @Entity
7 public class Book {
8
9     @Id
10    private Integer id;
11    private String title;
12    private String author;
13    private Float price;
14
15    public Book() {
16    }
17
18    public Book(int id, String title, String author, float price) {
19        this.id = id;
20        this.title = title;
21        this.author = author;
22        this.price = price;
23    }
24
25    public Integer getId() {
26        return id;
27    }
28
29    public void setId(Integer id) {
30        this.id = id;
31    }
32
33    public String getTitle() {
34        return title;
35    }
36
37    public void setTitle(String title) {
38        this.title = title;
39    }
40
41    public String getAuthor() {
42        return author;
43    }
44
45    public void setAuthor(String author) {
46        this.author = author;
47    }
48
49    public Float getPrice() {
50        return price;
51    }
52
53    public void setPrice(Float price) {
54        this.price = price;
55    }
56
57 }
```

But observe that the *id* member variable and the *price* member variable are both *classes*, and not *primitive types*. Which means both *id* and *price* can be set to null.

The *id* member variable is annotated using the **@Id** annotation, indicating that this is the primary key for this entity. And because *id* is the primary key, we can't really set that field to null.

For all of your JPA entity objects, it's good practice to specify getters and setters for all of your member variables. And that's exactly what I've added here. I've added getters and setters for the all fields in this *Book* class. I have getters and setters for *id*, *title*, *author*, and *price*.

Here we are in the **App.java** file. I've set up the import statements. The imports are the same as before.



```
1 package com.mytutorial.jpaa;
2
3 import javax.persistence.EntityManager;
4 import javax.persistence.EntityManagerFactory;
5 import javax.persistence.Persistence;
6
7 /**
8  * Hello world!
9  */
10
11 public class App
12 {
13     public static void main( String[] args )
14     {
15
16         EntityManagerFactory factory = Persistence.createEntityManagerFactory("BookstoreDB_Unit");
17         EntityManager entityManager = factory.createEntityManager();
18
19         try {
20             entityManager.getTransaction().begin();
21
22             Book firstBook = new Book(1234, "The Java Language Specification",
23                                     "Gilad Barcha", 99f);
24             Book secondBook = new Book(2222, "The Java Language Specification Second Edition",
25                                     "Gilad Barcha, James gosling", 119f);
26             Book thirdBook = new Book();
27
28             entityManager.persist(firstBook);
29             entityManager.persist(secondBook);
30             entityManager.persist(thirdBook);
31
32         } catch (Exception ex) {
33             System.err.println("An error occurred: " + ex);
34         } finally {
35             entityManager.getTransaction().commit();
36             entityManager.close();
37             factory.close();
38         }
39     }
40 }
41 }
```

And I've pasted in some code here, creating a few Book entities which will be records in my Book table within this database.

- Observe that I create the **EntityManagerFactory**, which is associated with the [BookstoreDB_Unit](#) persistence unit.
- And I create an [entityManager](#) using that factory.
- Observe that when I begin the transaction using my entityManager, which I do on line 19, I enclose all of my code within a **try**, **catch**, and **finally** block.
This is a best practice that you should follow when you create your own application managed entity managers. Make sure you *perform all of your transaction operations within a try block*. Make sure you have a catch block to catch any exceptions that are thrown. And commit your transaction and close your entityManager and entityManager factory within the *finally* block.
- Let's look at the Book records that we are about to insert into our Book table. We have firstBook, secondBook, and thirdBook.
I've instantiated new Book objects for firstBook and secondBook, with unique identifiers. The thirdBook, however, uses the empty default constructor.
- I then use the entityManager to persist all three entities.

Java Persistence API: Configuring Fields & Performing CRUD Operations

Remember that the id member variable has the **@Id** annotation within our Book entity. It cannot be null. Save all of your files within this IDE. Let's go ahead and run this application. I choose Run As and Java Application.

```
Hibernate:
drop table if exists Book
Dec 02, 2021 9:18:00 AM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionProv
Hibernate:
create table Book (
  id integer not null,
  author varchar(255),
  price float,
  title varchar(255),
  primary key (id)
) engine=MyISAM
Dec 02, 2021 9:18:00 AM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionProv
Dec 02, 2021 9:18:00 AM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HHH000476: Executing import script 'org.hibernate.tool.schema.internal.exec.ScriptSourceInputNonExistentImpl@2aledad4'
An error occurred: javax.persistence.PersistenceException: org.hibernate.id.IdentifierGenerationException: ids for this class must be manually a
Exception in thread "main" javax.persistence.RollbackException: Error while committing the transaction
    at org.hibernate.internal.ExceptionConverterImpl.convertCommitException(ExceptionConverterImpl.java:77)
    at org.hibernate.engine.transaction.internal.TransactionImpl.commit(TransactionImpl.java:71)
    at com.mytutorial.jpa.App.main(App.java:35)
Caused by: javax.persistence.PersistenceException: org.hibernate.TransactionException: Transaction was marked for rollback only; cannot commit
    at org.hibernate.internal.ExceptionConverterImpl.convert(ExceptionConverterImpl.java:149)
    at org.hibernate.internal.ExceptionConverterImpl.convert(ExceptionConverterImpl.java:157)
    at org.hibernate.internal.ExceptionConverterImpl.convertCommitException(ExceptionConverterImpl.java:56)
    ... 2 more
Caused by: org.hibernate.TransactionException: Transaction was marked for rollback only; cannot commit
    at org.hibernate.resource.transaction.backend.jdbc.internal.JdbcResourceLocalTransactionCoordinatorImpl$TransactionDriverControlImpl.com
    at org.hibernate.engine.transaction.internal.TransactionImpl.commit(TransactionImpl.java:68)
    ... 1 more
```

And within the console window, you'll immediately see that there is an exception. Why does this exception occur?

Well, we've successfully dropped the table Book, if it existed, and we've recreated the table. While recreating the table, the id member variable in our Book entity, which has been marked with the **@Id** annotation, has been designated the primary key in this table. It is an **Integer** and *it cannot be null*. But the thirdBook that we had instantiated has a null value for the id. And that's exactly why we encounter this **org.hibernate.id.IdentifierGenerationException**.

If you scroll to the very right, you can see that the message tells us that the ids for this class need to be assigned manually.

An error occurred: javax.persistence.PersistenceException:
org.hibernate.id.IdentifierGenerationException: ids for this class must be manually
assigned before calling save(): com.mytutorial.jpa.Book

That's because we haven't specified an automated way of generating IDs, you need to manually assign IDs. Everything should follow through intuitively here.

Java Persistence API: Configuring Fields & Performing CRUD Operations

Let's head over to **Book.java** and observe the `Id` annotation that we have specified on the `id` member variable. We can actually move this annotation over to the `getId` getter function.

```
6 @Entity
7 public class Book {
8
9     private Integer id;
10    private String title;
11    private String author;
12    private Float price;
13
14    public Book() {
15    }
16
17    public Book(int id, String title, String author, float price) {
18        this.id = id;
19        this.title = title;
20        this.author = author;
21        this.price = price;
22    }
23
24    @Id
25    public Integer getId() {
26        return id;
27    }
28
29    public void setId(Integer id) {
30        this.id = id;
31    }
32
33    public String getTitle() {
34        return title;
35    }
36}
```

The `Id` annotation that designates the primary key of an entity can be applied to the member variable, that is the field associated with the primary key, or to the getter.

To demonstrate this, I've moved the `@Id` annotation to the `getId` getter function. This still works. Let's head over to `App.java` and run our application once again.

```
INFO: HHH000400: Using dialect: org.hibernate.dialect.MySQL5Dialect
Hibernate:
    drop table if exists Book
Dec 02, 2021 9:25:15 AM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$Conne
Hibernate:
    create table Book (
        id integer not null,
        author varchar(255),
        price float,
        title varchar(255),
        primary key (id)
    ) engine=MyISAM
Dec 02, 2021 9:25:15 AM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$Conne
Dec 02, 2021 9:25:15 AM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HHH000476: Executing import script 'org.hibernate.tool.schema.internal.exec.ScriptSourceInputNonExistentImpl@2c1b9e4b'
```

And everything will work as it did before. If you look carefully at the output within our console window, you'll see the exception that says that ids for the `Book` class have to be assigned manually, before we persist these `Book` entities to our database.

```
An error occurred: javax.persistence.PersistenceException: org.hibernate.id.IdentifierGenerationException: ids for this class must be manually assigned before calling save(): com.mytutorial.jpa.Book
Exception in thread "main" javax.persistence.RollbackException: Error while committing the transaction
    at org.hibernate.internal.ExceptionConverterImpl.convertCommitException(ExceptionConverterImpl.java:77)
    at org.hibernate.engine.transaction.internal.TransactionImpl.commit(TransactionImpl.java:71)
    at com.mytutorial.jpa.App.main(App.java:35)
Caused by: javax.persistence.PersistenceException: org.hibernate.TransactionException: Transaction was marked for rollback only; cannot commit
    at org.hibernate.internal.ExceptionConverterImpl.convert(ExceptionConverterImpl.java:149)
    at org.hibernate.internal.ExceptionConverterImpl.convert(ExceptionConverterImpl.java:157)
    at org.hibernate.internal.ExceptionConverterImpl.convertCommitException(ExceptionConverterImpl.java:56)
    ... 2 more
Caused by: org.hibernate.TransactionException: Transaction was marked for rollback only; cannot commit
    at org.hibernate.resource.transaction.backend.jdbc.internal.JdbcResourceLocalTransactionCoordinatorImpl$TransactionDriverControlImpl.commit(JdbcResourceLocalTransactionCoordinatorImpl.java:228)
    at org.hibernate.engine.transaction.internal.TransactionImpl.commit(TransactionImpl.java:68)
```

We have one Book here with no id. That is thirdBook, that we've instantiated on line 26.

```
19     try {
20         entityManager.getTransaction().begin();
21
22         Book firstBook = new Book(1234, "The Java Language Specification",
23             "Gilad Barcha", 99f);
24         Book secondBook = new Book(2222, "The Java Language Specification Second Edition",
25             "Gilad Barcha, James gosling", 119f);
26         Book thirdBook = new Book();
27
28         entityManager.persist(firstBook);
29         entityManager.persist(secondBook);
30         entityManager.persist(thirdBook);
31
32     } catch (Exception ex) {
33         System.err.println("An error occurred: " + ex);
34     } finally {
35         entityManager.getTransaction().commit();
```

Unfortunately, this meant that our transaction could not be committed. The transaction was **rolled back**.

So it's pretty clear that every entity should have a unique id assigned to that entity. Now, it's kind of painful to assign ids manually. We need to fix that. But before we get there, let's quickly fix up our code so our thirdBook object also has a unique identifier, 3331.

```
16     EntityManagerFactory factory = Persistence.createEntityManagerFactory("BookstoreDB_Unit");
17     EntityManager entityManager = factory.createEntityManager();
18
19     try {
20         entityManager.getTransaction().begin();
21
22         Book firstBook = new Book(1234, "The Java Language Specification",
23             "Gilad Barcha", 99f);
24         Book secondBook = new Book(2222, "The Java Language Specification Second Edition",
25             "Gilad Barcha, James gosling", 119f);
26         Book thirdBook = new Book();
27         thirdBook.setId(3333);
28
29         entityManager.persist(firstBook);
30         entityManager.persist(secondBook);
31         entityManager.persist(thirdBook);
32
33     } catch (Exception ex) {
34         System.err.println("An error occurred: " + ex);
35     } finally {
36         entityManager.getTransaction().commit();
37         entityManager.close();
38         factory.close();
39     }
```

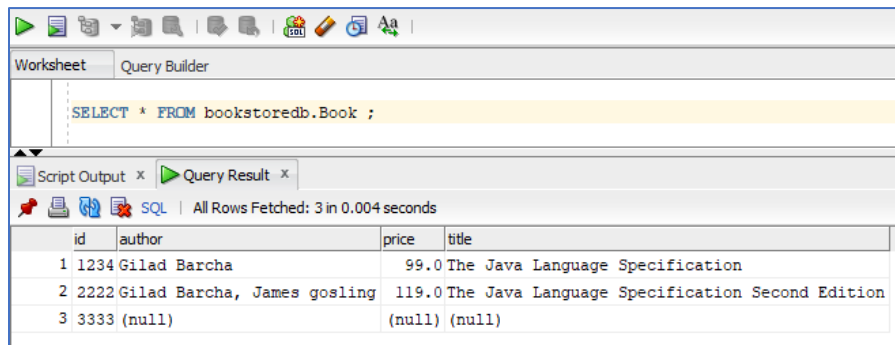
Because of the **@Id** annotation on our id getter, we know it is a required field, the primary key of our Book table. Let's go ahead and run this application once again.

Java Persistence API: Configuring Fields & Performing CRUD Operations

```
INFO: HHH000400: Using dialect: org.hibernate.dialect.MySQL5Dialect
Hibernate:
    drop table if exists Book
Dec 02, 2021 9:33:55 AM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionProv
Hibernate:
    create table Book (
      id integer not null,
      author varchar(255),
      price float,
      title varchar(255),
      primary key (id)
    ) engine=MyISAM
Dec 02, 2021 9:33:55 AM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionProv
Dec 02, 2021 9:33:55 AM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HHH000476: Executing import script 'org.hibernate.tool.schema.internal.exec.ScriptSourceInputNonExistentImpl@2c1b9e4b'
Hibernate:
    insert
    into
      Book
    (author, price, title, id)
    values
      (?, ?, ?, ?)
Hibernate:
    insert
    into
      Book
    (author, price, title, id)
    values
      (?, ?, ?, ?)
Hibernate:
    insert
    into
      Book
    (author, price, title, id)
    values
      (?, ?, ?, ?)
Dec 02, 2021 9:33:56 AM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl stop
INFO: HHH10001008: Cleaning up connection pool [jdbc:mysql://localhost:3306/bookstoredb]
```

And this time, this application should run through successfully. There should be no errors.

And that's exactly what a quick look at the console output tells us. The table was created successfully, and the insertion statements were also completed successfully. We can confirm this by heading over to our MySQL Workbench and running a select query from the bookstore table.



The screenshot shows the MySQL Workbench interface. The 'Query Builder' tab is active, displaying the SQL query: `SELECT * FROM bookstoredb.Book ;`. Below the query, the 'Query Result' tab shows the results of the query. The results are displayed in a table with 4 columns: id, author, price, and title. There are 3 rows of data.

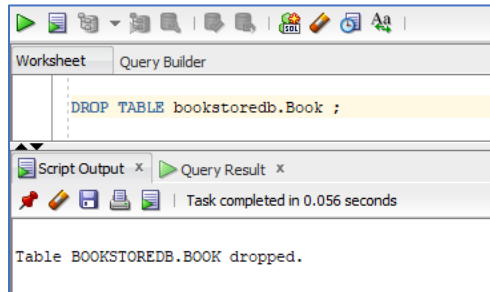
	id	author	price	title
1	1234	Gilad Barcha	99.0	The Java Language Specification
2	2222	Gilad Barcha, James gosling	119.0	The Java Language Specification Second Edition
3	3333	(null)	(null)	(null)

The results of the select query tell us that there are three book entries in our book table. We have the Java Language Specification, the Second Edition, and a third book entry with id 3331 where the remaining fields are all null.

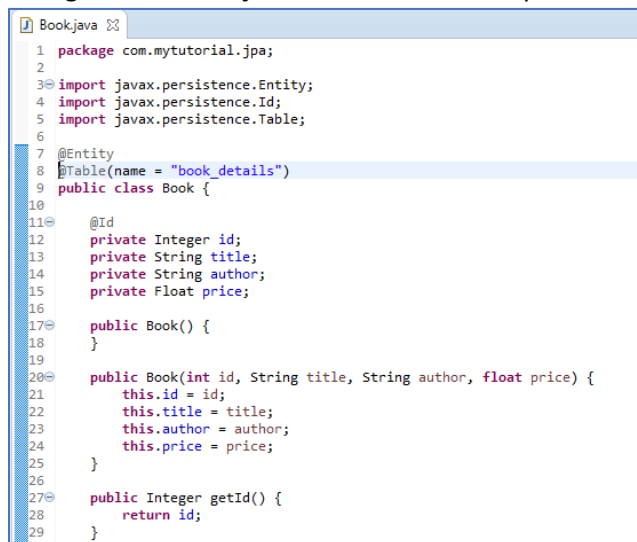
Configuring the Table Name

By default, the Java Persistence API persists entities using the names of the classes as tables, but it's possible for us to configure that.

Let's see how in this demo. Here I am in the SQL Workbench user interface. I'm going to drop the bookstoredb.book table. So I'll get rid of this table altogether.



Once this table has been dropped, let's switch over to our IDE, that is Eclipse. And let's see how we can configure our **Book.java** class so that it will persist its books in a table of a different name.



And the way we configure the name of the table where book records will be stored is by specifying an additional **@Table** annotation.

The **@Table** annotation, which is in **javax.persistence.Table**, takes in an additional input argument, **name**. This name is what you'd use to specify the name of the table where our books records will be stored. I've called the table `"book_details"`.

Observe that we continue to have the **@Entity** annotation for this book class, indicating that book objects will be records inserted into the `book_details` table. The only change here is the name of the table.

Go ahead and use Run As Java Application to run this code.

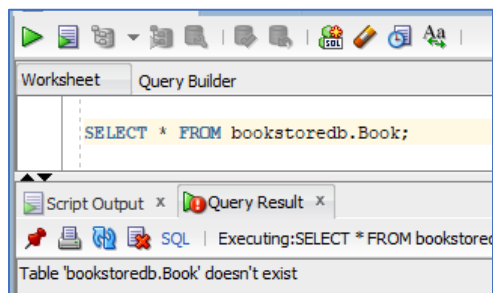
```
Hibernate:
    drop table if exists book_details
Dec 02, 2021 9:42:56 AM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImp
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnv
Hibernate:
    create table book_details (
        id integer not null,
        author varchar(255),
        price float,
        title varchar(255),
        primary key (id)
    ) engine=MyISAM
Dec 02, 2021 9:42:56 AM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImp
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnv
Dec 02, 2021 9:42:56 AM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HHH000476: Executing import script 'org.hibernate.tool.schema.internal.exec.ScriptSourceInputNonExistentIm
Hibernate:
    insert
    into
        book_details
    (author, price, title, id)
    values
    (?, ?, ?, ?)
Hibernate:
    insert
    into
        book_details
    (author, price, title, id)
    values
    (?, ?, ?, ?)
Hibernate:
    insert
    into
        book_details
    (author, price, title, id)
    values
    (?, ?, ?, ?)
Dec 02, 2021 9:42:57 AM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl stop
INFO: HHH10001008: Cleaning up connection pool [jdbc:mysql://localhost:3306/bookstoredb]
```

The code seems to run through fine. Let's scroll down within the console window and take a look at the SQL statements executed.

Observe that the drop table is now for the `book_details` table. And the new create table is also for the `book_details` table. So rather than use the name of the class directly as the name of the table, we have configured the name of the table to be what we want it to be.

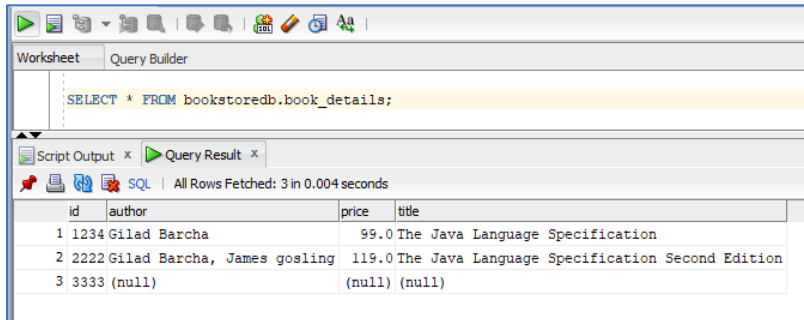
The remaining statements, the insert operations, remain the same, except that they insert into the `book_details` table.

Let's confirm what we already know using MySQL Workbench. I'm going to run a select * from the Book table within bookstoredb.



And when you run this command, you'll find that no book table exist. The book table hasn't been created.

However, if you change this command so that you select from `bookstoredb.book_details`, you'll find that this table now exist.



The screenshot shows a SQL query builder interface. The 'Query Builder' tab is active, displaying the query `SELECT * FROM bookstoredb.book_details;`. Below the query, the 'Query Result' tab is active, showing a table with 4 columns: `id`, `author`, `price`, and `title`. The table contains 3 rows of data. The status bar indicates 'All Rows Fetched: 3 in 0.004 seconds'.

	id	author	price	title
1	1234	Gilad Barcha	99.0	The Java Language Specification
2	2222	Gilad Barcha, James gosling	119.0	The Java Language Specification Second Edition
3	3333	(null)	(null)	(null)

The `book_details` table now contains the three records that we inserted using the `book` entity. The table contains three records corresponding to the three book objects that we had persisted using JPA and Hibernate.

Generated Values for Primary Keys

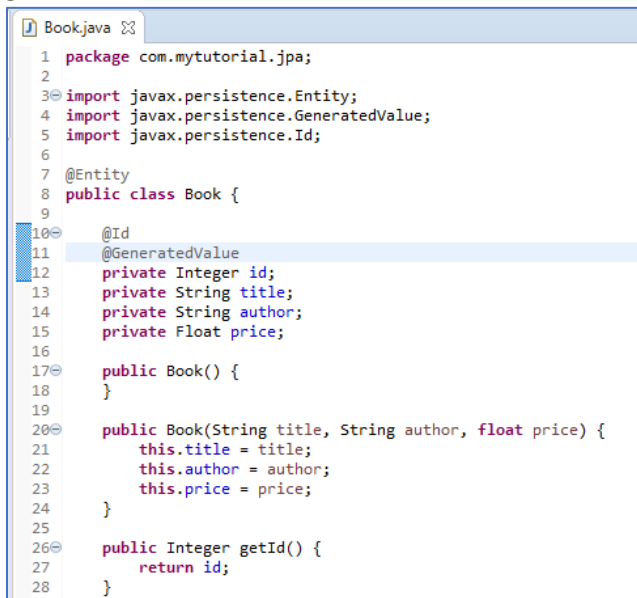
Let's head back to our **Book.java** class which represents the entity that we insert as records into our database table. Now I no longer want the table to be called `book_details`, I'm happy with the name just `Book`. So I'm going to get rid of the `@Table` annotation and get rid of the corresponding import as well.

So I have this `Book.java` as I did earlier with the `@Entity` annotation. But there is one thing about this current setup that is slightly annoying and that is the fact that we have to specify a unique identifier for each book.

Now if databases can generate unique values for primary keys, we should be able to do that within JPA Hibernate as well. And that's exactly what we'll do using the **GeneratedValue** annotation.

After setting up the import statement for generated value, which allows us to generate unique identifiers for primary keys. I'm going to change this book constructor so that it no longer accepts a unique identifier from the user. Notice my updated constructor here, it takes in three input arguments, the title of the book, the author and the price of the book, but no value for `id`.

That's going to be automatically generated when we add the **@GeneratedValue** annotation to our `getId` getter method.



```

1 package com.mytutorial.jpa;
2
3 import javax.persistence.Entity;
4 import javax.persistence.GeneratedValue;
5 import javax.persistence.Id;
6
7 @Entity
8 public class Book {
9
10     @Id
11     @GeneratedValue
12     private Integer id;
13     private String title;
14     private String author;
15     private Float price;
16
17     public Book() {
18     }
19
20     public Book(String title, String author, float price) {
21         this.title = title;
22         this.author = author;
23         this.price = price;
24     }
25
26     public Integer getId() {
27         return id;
28     }
29 }

```

This annotation can be applied to your member variable `id` as well. For the sake of consistency, this is always associated with whatever has the **@Id** annotation.

GeneratedValue means you will ask *the database to autogenerate unique values for this primary key*. When you don't specify any additional arguments for your `GeneratedValue` annotation, the **default generation strategy** used for the primary key is the **autogeneration** strategy. This means the JPA provider gets to choose the strategy it wants for the primary key generation.

In the case of Hibernate, under the hood Hibernate uses generation strategy **Sequence**.

The sequence strategy chosen by Hibernate uses a database sequence to generate primary key values. This requires the creation of a database sequence that is then accessed to get the next primary key and in sequence. There is an additional select statement required to access the next value in the sequence, but this is not a performance hit.

For almost all databases, this key generation strategy is a good one. Now that we know how to automatically generate keys, let's head over to `App.java` and fix the instantiations of the book object to not specify a unique key. Once again I've instantiated three book entities, *firstbook*, *secondbook*, and *thirdbook*. I haven't specified the IDs for any of these entities. In fact on line 26, observe that I've instantiated an empty book object with no ID specified at all. I have specified field values for the remaining two books, *firstbook* and *secondbook*, but I haven't specified the ID.

App.java

```
16 EntityManagerFactory factory = Persistence.createEntityManagerFactory("BookstoreDB_Unit");
17 EntityManager entityManager = factory.createEntityManager();
18
19 try {
20     entityManager.getTransaction().begin();
21
22     Book firstBook = new Book("The Java Language Specification",
23                             "Gilad Barcha", 99f);
24     Book secondBook = new Book("The Java Language Specification Second Edition",
25                              "Gilad Barcha, James Gosling", 119f);
26     Book thirdBook = new Book();
27
28     entityManager.persist(firstBook);
29     entityManager.persist(secondBook);
30     entityManager.persist(thirdBook);
31
32 } catch (Exception ex) {
33     System.err.println("An error occurred: " + ex);
34 } finally {
35     entityManager.getTransaction().commit();
36     entityManager.close();
37     factory.close();
38 }
```

I've then used the `entityManager` to persist all three books.

We are now ready to **run** this code and see the resulting records in our database. Everything seems to have run just fine. Let's take a look at the SQL commands that Hibernate has executed under the hood.

```
INFO: HHH000400: Using dialect: org.hibernate.dialect.MySQL5Dialect
Hibernate:
    drop table if exists Book
Dec 02, 2021 9:57:07 AM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNon
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.
Hibernate:
    drop table if exists hibernate_sequence
...
```

It drops the book table if one exist, and it also drops a table called **hibernate_sequence**, this is the database sequence generator.

We then create the **Book** table. And after the **Book** table, we go ahead and create the **hibernate_sequence** table as well.

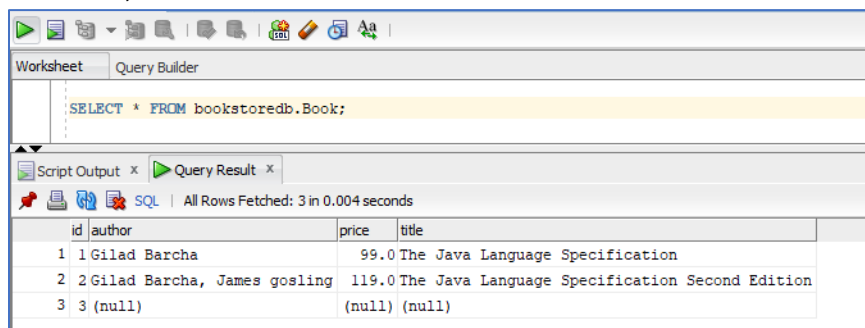
```
Hibernate:
    create table Book (
      id integer not null,
      author varchar(255),
      price float,
      title varchar(255),
      primary key (id)
    ) engine=MyISAM
Dec 02, 2021 9:57:07 AM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsC
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.in
Hibernate:
    create table hibernate_sequence (
      next_val bigint
    ) engine=MyISAM
Hibernate:
    insert into hibernate_sequence values ( 1 )
Dec 02, 2021 9:57:07 AM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HHH000476: Executing import script 'org.hibernate.tool.schema.internal.exec.ScriptSourceInpu
```

Which contains the next value of the primary key that we will use within the book table. Primary key generation by default starts at value 1. So we insert into the hibernate_sequence table, the value 1, this is what hibernate uses as the first value for primary key.

Each time we insert a book record into our book table, Hibernate chooses the next value from the sequence table as the primary key for this book entity.

```
Hibernate:
select
  next_val as id_val
from
  hibernate_sequence for update
Hibernate:
update
  hibernate_sequence
set
  next_val= ?
where
  next_val=?
Hibernate:
insert
into
  Book
(author, price, title, id)
values
  (?, ?, ?, ?)
```

Let's see what the records look like in our book table within MySQL Workbench. I'm going to run this command,



The screenshot shows the MySQL Workbench interface. The 'Query Builder' tab is active, displaying the SQL query: `SELECT * FROM bookstoredb.Book;`. Below the query, the 'Query Result' tab shows the results of the query. The results are displayed in a table with 4 columns: id, author, price, and title. There are 3 rows of data.

	id	author	price	title
1	1	Gilad Barcha	99.0	The Java Language Specification
2	2	Gilad Barcha, James gosling	119.0	The Java Language Specification Second Edition
3	3	(null)	(null)	(null)

and you can see that there are 3 records here having primary keys 1, 2, and 3. These are the primary keys autogenerated using the **hibernate_sequence** table. The **@GeneratedValue** annotation abstracts us away from the need to specify unique key IDs for our entities.

Java Persistence API: Configuring Fields & Performing CRUD Operations

The screenshot displays the SQL Developer interface for a database named 'SkillShareMySQL'. The 'hibernate_sequence' table is selected, and its structure is shown in the 'Columns' tab. The table has one column, 'next_val', which is a bigint with a precision of 19 and a scale of 0. The 'next_val' column is nullable and has a default value of 0. The 'Data' tab shows the current value of 'next_val' as 1. The 'bookstoredb' database structure is also visible in the left pane, showing tables, views, indexes, procedures, functions, and triggers.

COLUMN_NAME	ORDINAL_POSITION	COLUMN_DEFAULT	IS_NULLABLE	DATA_TYPE	NUMERIC_PRECISION	NUMERIC_SCALE	COLUMN_COMMENT
next_val	1	(null)	YES	bigint	19	0 (null)	

next_val

1 4

bookstoredb

- Tables
 - Author
 - Book
 - hibernate_sequence
- Views
- Indexes
- Procedures
- Functions
- Triggers

Generation Types AUTO and IDENTITY

Back to Book.java and this time I'm going to explicitly specify a generation strategy for the GeneratedValue annotation. This is what we'll use to generate primary keys.

Set up an import statement for **javax.persistence.GenerationType**, which contains the different enum values for the different generation strategies. Now this GeneratedValue annotation I'm going to update and specify a specific generation strategy, **GenerationType.AUTO**. Now this is the **default** strategy that Hibernate chooses when you don't specify an explicit strategy. When you use this **GenerationType.AUTO**, JPA lets the JPA provider, which in our case is Hibernate, choose the strategy for generating primary keys. And Hibernate chooses the sequence generation option.

```
Book.java
1 package com.mytutorial.jpa;
2
3 import javax.persistence.Entity;
4 import javax.persistence.GeneratedValue;
5 import javax.persistence.GenerationType;
6 import javax.persistence.Id;
7
8 @Entity
9 public class Book {
10
11     @Id
12     @GeneratedValue(strategy = GenerationType.AUTO)
13     private Integer id;
14     private String title;
15     private String author;
16     private Float price;
17
18     public Book() {
19     }
20
21     public Book(String title, String author, float price) {
22         this.title = title;
23         this.author = author;
24         this.price = price;
25     }
26
27     public Integer getId() {
28         return id;
29     }
30 }
```

Let's run this code and confirm that this auto generation strategy is the same strategy that we used earlier when we didn't explicitly specify a generation strategy for our primary key. Run this code and you'll find that the SQL statements under the hood are exactly the same.

```
Hibernate:
    drop table if exists Book
Dec 02, 2021 10:17:11 AM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIs
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironment
Hibernate:
    drop table if exists hibernate_sequence
Hibernate:
    create table Book (
        id integer not null,
        author varchar(255),
        price float,
        title varchar(255),
        primary key (id)
    ) engine=MyISAM
```

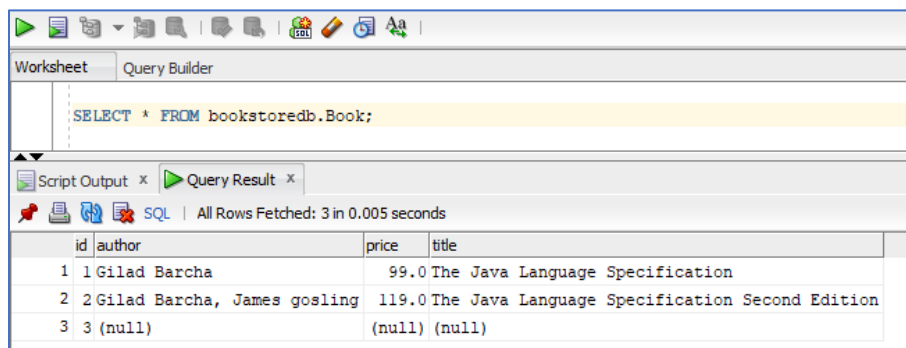
We drop the book table. We drop the **hibernate_sequence** table which Hibernate uses for sequence generation. We create the book table and create the **hibernate_sequence** table once again. The main

Java Persistence API: Configuring Fields & Performing CRUD Operations

thing you have to remember for this generation strategy, **AUTO**, when used with *hibernate* is that it *uses a sequence generator* within your database to generate primary key values.

```
Dec 02, 2021 10:17:11 AM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolate
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentIniti
Hibernate:
    create table hibernate_sequence (
      next_val bigint
    ) engine=MyISAM
Hibernate:
    insert into hibernate_sequence values ( 1 )
Dec 02, 2021 10:17:12 AM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HHH000476: Executing import script 'org.hibernate.tool.schema.internal.exec.ScriptSourceInputNonExistentImpl@30ea8c23'
Hibernate:
    select
      next_val as id_val
    from
      hibernate_sequence for update
Hibernate:
    update
      hibernate_sequence
    set
      next_val= ?
    where
      next_val=?
Hibernate:
    select
      next_val as id_val
    from
      hibernate_sequence for update
```

Let's run a select * from our **Book** table. And when you run this command, you'll find three book records in here with unique Ids, 1, 2, and 3.

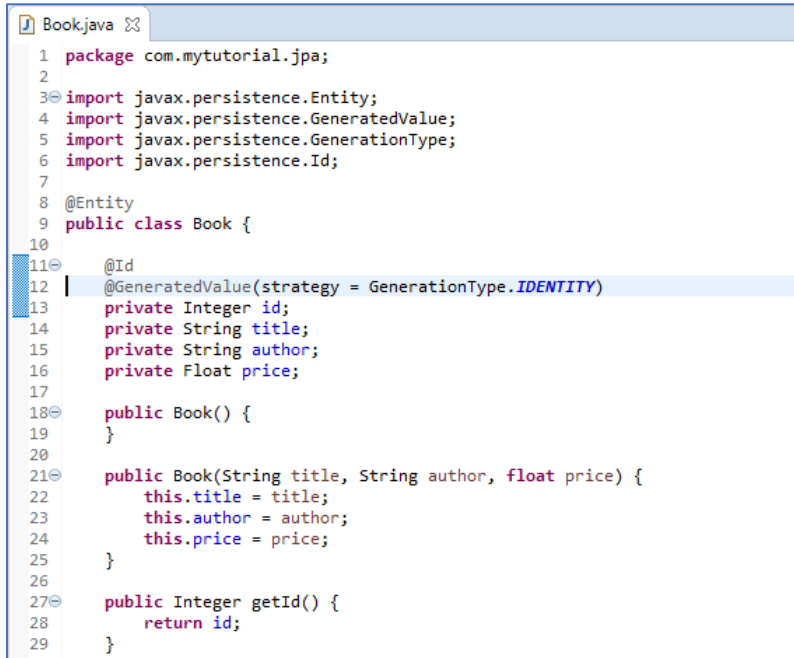


The screenshot shows a database query tool interface. At the top, there's a toolbar with various icons. Below it, a 'Worksheet' tab is active, showing a 'Query Builder' section with the SQL query: `SELECT * FROM bookstoredb.Book;`. Below the query, there's a 'Script Output' and 'Query Result' section. The 'Query Result' section shows the results of the query, indicating 'All Rows Fetched: 3 in 0.005 seconds'. The results are displayed in a table with four columns: 'id', 'author', 'price', and 'title'.

id	author	price	title
1	Gilad Barcha	99.0	The Java Language Specification
2	Gilad Barcha, James gosling	119.0	The Java Language Specification Second Edition
3	(null)	(null)	(null)

These are primary keys generated using the hibernate sequence table.

JPA allows primary key generation using different strategies. Let's head back to **Book.java** here, and I'm going to change the strategy used for the `@GeneratedValue`. I'm going to use **GenerationType.IDENTITY**.

A screenshot of a code editor showing the `Book.java` file. The code is as follows:

```
1 package com.mytutorial.jpa;
2
3 import javax.persistence.Entity;
4 import javax.persistence.GeneratedValue;
5 import javax.persistence.GenerationType;
6 import javax.persistence.Id;
7
8 @Entity
9 public class Book {
10
11     @Id
12     @GeneratedValue(strategy = GenerationType.IDENTITY)
13     private Integer id;
14     private String title;
15     private String author;
16     private Float price;
17
18     public Book() {
19     }
20
21     public Book(String title, String author, float price) {
22         this.title = title;
23         this.author = author;
24         this.price = price;
25     }
26
27     public Integer getId() {
28         return id;
29     }
30 }
```

The line `@GeneratedValue(strategy = GenerationType.IDENTITY)` is highlighted in blue.

This generation strategy is by far the easiest one to use, it does not require the setup of a separate database sequence. Instead, this strategy relies on an auto-incremented database column. This means for every insert statement into the book table, Hibernate lets the database generate a new value with each insert operation. That is MySQL is responsible for generating these primary keys. From the database point of view, this is efficient because these auto-increment columns are highly optimized.

And you don't have any additional select statements that you have to execute to get the next value of primary key. The database does all of this for you.

But in the case of Hibernate, this strategy is *not really super optimal*. This is because the need to generate a primary key with every insert operation prevents hibernate from performing other optimizations in your code. So keep that in mind if you're using hibernate as a JPA provider.

Now the results of this identity strategy remains the same. Let's run our code the same code in `App.java`. This is the code that inserts three records into our book table. Everything runs through successfully. The console log statements will give us an inkling of what's going on under the hood.

Java Persistence API: Configuring Fields & Performing CRUD Operations

```

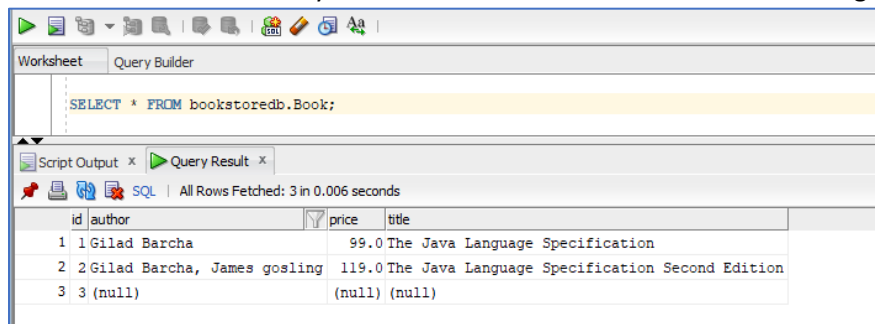
Hibernate:
    drop table if exists Book
Dec 02, 2021 10:25:44 AM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedC
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiat
Hibernate:
    create table Book (
      id integer not null auto_increment,
      author varchar(255),
      price float,
      title varchar(255),
      primary key (id)
    ) engine=MyISAM
Dec 02, 2021 10:25:44 AM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedC
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiat
Dec 02, 2021 10:25:44 AM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HHH000476: Executing import script 'org.hibernate.tool.schema.internal.exec.ScriptSourceInputNonExistentImpl@6326d182'
Hibernate:
insert
into
  Book
(author, price, title)
values
(?, ?, ?)

```

We drop table Book if it exists, observe that no additional hibernate sequence table has been set up. Instead when we create the Book table, the id column has been set to not null auto_increment.

Having flagged this column as auto increment, it's now MySQL's responsibility to generate primary keys, not Hibernate. You'll notice in our console log statements that there is no additional SQL command selecting from a separate hibernate sequence table.

Let's switch over to the MySQL Workbench and see the result of running this code.



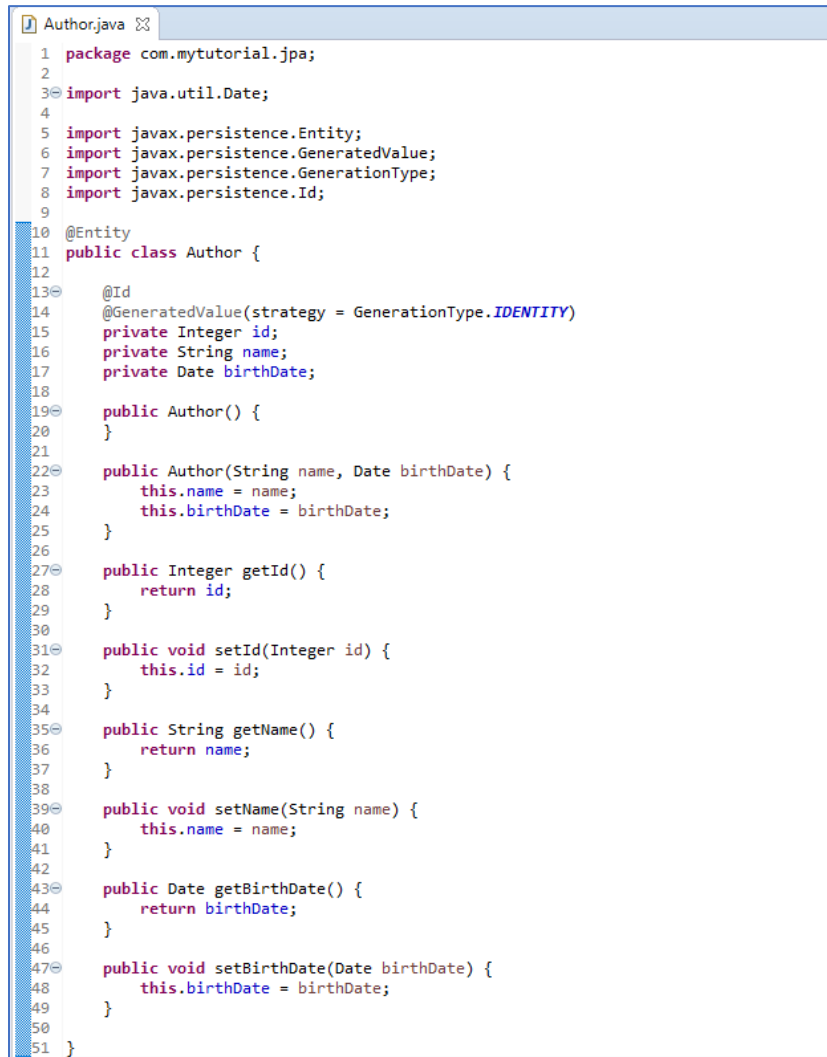
The screenshot shows the MySQL Workbench interface. The 'Query Builder' tab is active, displaying the query: `SELECT * FROM bookstoredb.Book;`. Below the query, the 'Script Output' and 'Query Result' tabs are visible. The 'Query Result' tab shows the results of the query, indicating 'All Rows Fetched: 3 in 0.006 seconds'. The results are displayed in a table with 4 columns: id, author, price, and title.

	id	author	price	title
1	1	Gilad Barcha	99.0	The Java Language Specification
2	2	Gilad Barcha, James gosling	119.0	The Java Language Specification Second Edition
3	3	(null)	(null)	(null)

We select from the [Book](#) table, and once we run this query, you'll see that we have three records in here with the primary key values 1, 2, and 3. These primary keys have been generated using the auto-incremented Id column.

Generation Type IDENTITY for Multiple Tables

Back to our Eclipse IDE and this time observe that I've set up a new class here, a Java class called **Author** in the **Author.java** file. This new class is going to represent a new entity that I want to persist within my database.



```

1 package com.mytutorial.jpaa;
2
3 import java.util.Date;
4
5 import javax.persistence.Entity;
6 import javax.persistence.GeneratedValue;
7 import javax.persistence.GenerationType;
8 import javax.persistence.Id;
9
10 @Entity
11 public class Author {
12
13     @Id
14     @GeneratedValue(strategy = GenerationType.IDENTITY)
15     private Integer id;
16     private String name;
17     private Date birthDate;
18
19     public Author() {
20     }
21
22     public Author(String name, Date birthDate) {
23         this.name = name;
24         this.birthDate = birthDate;
25     }
26
27     public Integer getId() {
28         return id;
29     }
30
31     public void setId(Integer id) {
32         this.id = id;
33     }
34
35     public String getName() {
36         return name;
37     }
38
39     public void setName(String name) {
40         this.name = name;
41     }
42
43     public Date getBirthDate() {
44         return birthDate;
45     }
46
47     public void setBirthDate(Date birthDate) {
48         this.birthDate = birthDate;
49     }
50
51 }

```

In a real world application, you won't be working with a single entity. In fact, you will have multiple entities and that's exactly what we'll see here in this demo. Each of these entities will have their own primary keys that can be auto generated. Setup the import statements for the objects and classes that we'll reference within this **Author** file.

Observe that we have **java.util.Date** indicating that this **Author.java** is going to have a date field. In addition, the other import statements are for our JPA annotations. I'm going to annotate this **Author** class with the **@Entity** annotation indicating that this references a table in my database. The table will be called **Author** by default. I'll now set up the fields within my **Author** class which correspond to the columns in my database table. I have an **id** field, a **name** field and a **birthDate** field which is of type **Date**.

I have a public no argument default **constructor**. And in addition, I have a **constructor** that takes in the **name** and **birthDate** for an **Author**.

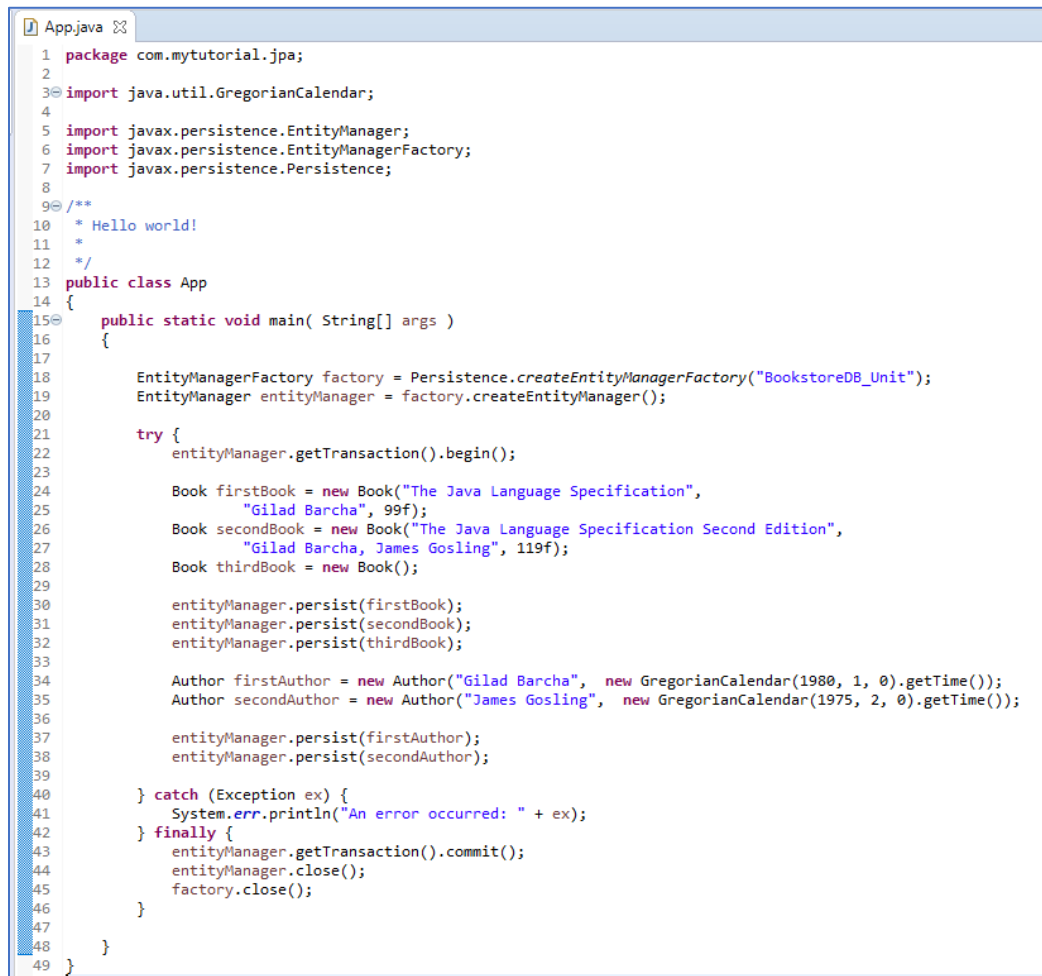
Java Persistence API: Configuring Fields & Performing CRUD Operations

I then set up the getters and setters for each of the member variables of this Author class. And I decorate the getId getter with **@Id** and **@GeneratedValue**.

This is the primary key of the **Author** table. We'll generate primary keys for this table, using the strategy **IDENTITY**, which means we will set up the id column with auto increment so that the database is responsible for generating the primary keys for this table.

The remaining methods here in this **Author** class are getters and setters for the rest of the fields in this table.

I'll now head over to **App.java** where I have code which instantiates an **EntityManagerFactory**, and get an **EntityManager** from that factory.

A screenshot of a code editor showing the App.java file. The code is in Java and uses the Java Persistence API (JPA). It starts with package and import statements for java.util.GregorianCalendar, javax.persistence.EntityManager, javax.persistence.EntityManagerFactory, and javax.persistence.Persistence. A comment says "Hello world!". The main method is public static void main. It creates an EntityManagerFactory with "BookstoreDB_Unit" and an EntityManager. A try block begins with entityManager.getTransaction().begin(). It creates three Book objects: firstBook (The Java Language Specification, Gilad Barcha, 99f), secondBook (The Java Language Specification Second Edition, Gilad Barcha, James Gosling, 119f), and thirdBook (empty). It persists firstBook, secondBook, and thirdBook. Then it creates two Author objects: firstAuthor (Gilad Barcha, 1980) and secondAuthor (James Gosling, 1975). It persists firstAuthor and secondAuthor. A catch block handles exceptions, printing "An error occurred: " + ex. A finally block calls entityManager.getTransaction().commit(), entityManager.close(), and factory.close(). The try block ends with entityManager.getTransaction().commit(). The main method ends with entityManager.close() and factory.close().

```
1 package com.mytutorial.jpaa;
2
3 import java.util.GregorianCalendar;
4
5 import javax.persistence.EntityManager;
6 import javax.persistence.EntityManagerFactory;
7 import javax.persistence.Persistence;
8
9 /**
10  * Hello world!
11  *
12  */
13 public class App
14 {
15     public static void main( String[] args )
16     {
17
18         EntityManagerFactory factory = Persistence.createEntityManagerFactory("BookstoreDB_Unit");
19         EntityManager entityManager = factory.createEntityManager();
20
21         try {
22             entityManager.getTransaction().begin();
23
24             Book firstBook = new Book("The Java Language Specification",
25                                     "Gilad Barcha", 99f);
26             Book secondBook = new Book("The Java Language Specification Second Edition",
27                                     "Gilad Barcha, James Gosling", 119f);
28             Book thirdBook = new Book();
29
30             entityManager.persist(firstBook);
31             entityManager.persist(secondBook);
32             entityManager.persist(thirdBook);
33
34             Author firstAuthor = new Author("Gilad Barcha", new GregorianCalendar(1980, 1, 0).getTime());
35             Author secondAuthor = new Author("James Gosling", new GregorianCalendar(1975, 2, 0).getTime());
36
37             entityManager.persist(firstAuthor);
38             entityManager.persist(secondAuthor);
39
40         } catch (Exception ex) {
41             System.err.println("An error occurred: " + ex);
42         } finally {
43             entityManager.getTransaction().commit();
44             entityManager.close();
45             factory.close();
46         }
47     }
48 }
49
```

Set up the import statements for all of the classes that we'll use here, and let's write some code to persist some books and authors into our database. We'll begin our transaction within a try block.

I have three book entities that I have instantiated here, three **Book** objects, *firstbook*, *secondbook*, and *thirdbook*. I've only specified the *name*, the *author*, and the *price* for each of these books. For the *thirdBook* I've specified nothing. Remember, the **primary key id**'s will be auto generated.

I call **entityManager.persist** to persist each of these objects as records in my Book table.

Java Persistence API: Configuring Fields & Performing CRUD Operations

Scroll down below, observe that I have now instantiated Author entities as well. I've instantiated two [Authors](#), along with their names and birth dates.

Now, let me just tell you that these birth dates are not actually real. I've just set them to random date values using the **GregorianCalendar** class available in **java.util**. In order to store these Author records in our database, I call **entityManager.persist** and pass in *firstAuthor* as well as *secondAuthor*. Our App.java code is now complete, save all of the files and let's run this Java application.

```
Hibernate:
    drop table if exists Author
Dec 02, 2021 10:38:18 AM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIs
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironment
Hibernate:
    drop table if exists Book
Hibernate:
    create table Author (
        id integer not null auto_increment,
        birthDate datetime,
        name varchar(255),
        primary key (id)
    ) engine=MyISAM
Dec 02, 2021 10:38:18 AM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIs
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironment
Hibernate:
    create table Book (
        id integer not null auto_increment,
        author varchar(255),
        price float,
        title varchar(255),
        primary key (id)
    ) engine=MyISAM
```

Let's take a look at the console output which tells us what SQL statements have been executed under the hood. We drop table if it exist. This is the Author table, that's because we have an [Author](#) entity class, a new class that we've set up in our project. You can see that Hibernate looks for all of the entity classes that exist within our project, and generates a drop table command for each of these tables. That's because in our **persistence.xml**, we specified the database action to be drop and create. Our settings will include dropping the book table if it exists as well, and the new Author table will be created.

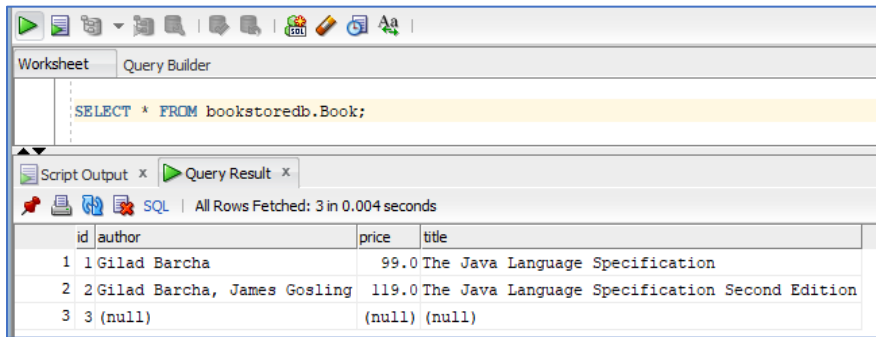
We create a table Author, where id is the **primary key** of this table. And this id is set to **auto_increment** the database will generate auto-incremented values for this primary key.

Corresponding to our second entity, we have another create table command, this time for the table [Book](#). And if you scroll down below, in the console output you'll see the insert statements for our [Book](#) objects as well as our [Author](#) objects.

```
Hibernate:
    insert
    into
        Book
        (author, price, title)
    values
        (?, ?, ?)
Hibernate:
    insert
    into
        Author
        (birthDate, name)
    values
        (?, ?)
```

Let's confirm that both tables have been set up with the right records using the MySQL Workbench.

I'll run a select on the [Book](#) table.

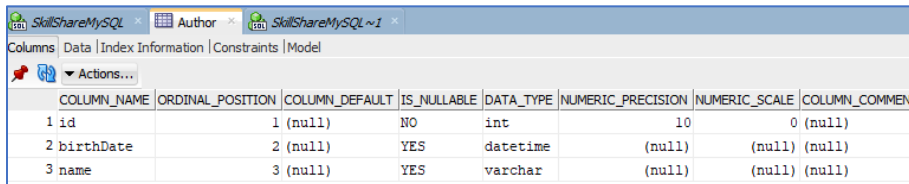


The screenshot shows the SQL Developer interface with a query window containing the SQL statement: `SELECT * FROM bookstoredb.Book;`. Below the query, the 'Query Result' tab is active, displaying the results of the query. The status bar indicates 'All Rows Fetched: 3 in 0.004 seconds'.

id	author	price	title
1	Gilad Barcha	99.0	The Java Language Specification
2	Gilad Barcha, James Gosling	119.0	The Java Language Specification Second Edition
3	(null)	(null)	(null)

And you can see that the [Book](#) table has three entries corresponding to the three books that we persisted. The primary key for these book entries start with the value 1, and they're auto-incremented.

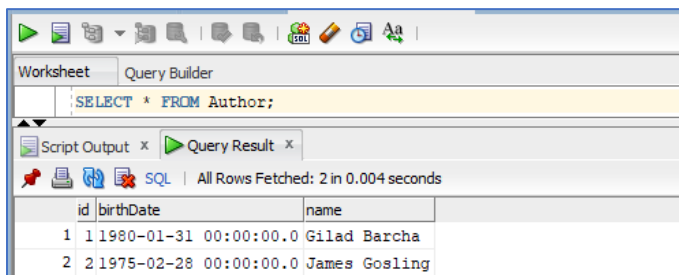
Let's run another command here. I'm going to run a describe command on the [author](#) table to make sure that the [Author](#) table exists. The results show me that the [Author](#) table does exist with three columns, id, birthDate, and name. And the id column is a primary key with auto_increment.



The screenshot shows the SQL Developer interface with the 'Columns' tab selected for the 'Author' table. The 'Actions...' menu is open, and the 'DESCRIBE' command has been executed. The results show the table structure with three columns: id, birthDate, and name.

COLUMN_NAME	ORDINAL_POSITION	COLUMN_DEFAULT	IS_NULLABLE	DATA_TYPE	NUMERIC_PRECISION	NUMERIC_SCALE	COLUMN_COMMENT
id	1	(null)	NO	int	10	0	(null)
birthDate	2	(null)	YES	datetime	(null)	(null)	(null)
name	3	(null)	YES	varchar	(null)	(null)	(null)

I'll quickly run a select * from the [Author](#) table to make sure that our two author entries are present, and yes indeed, they are.



The screenshot shows the SQL Developer interface with a query window containing the SQL statement: `SELECT * FROM Author;`. Below the query, the 'Query Result' tab is active, displaying the results of the query. The status bar indicates 'All Rows Fetched: 2 in 0.004 seconds'.

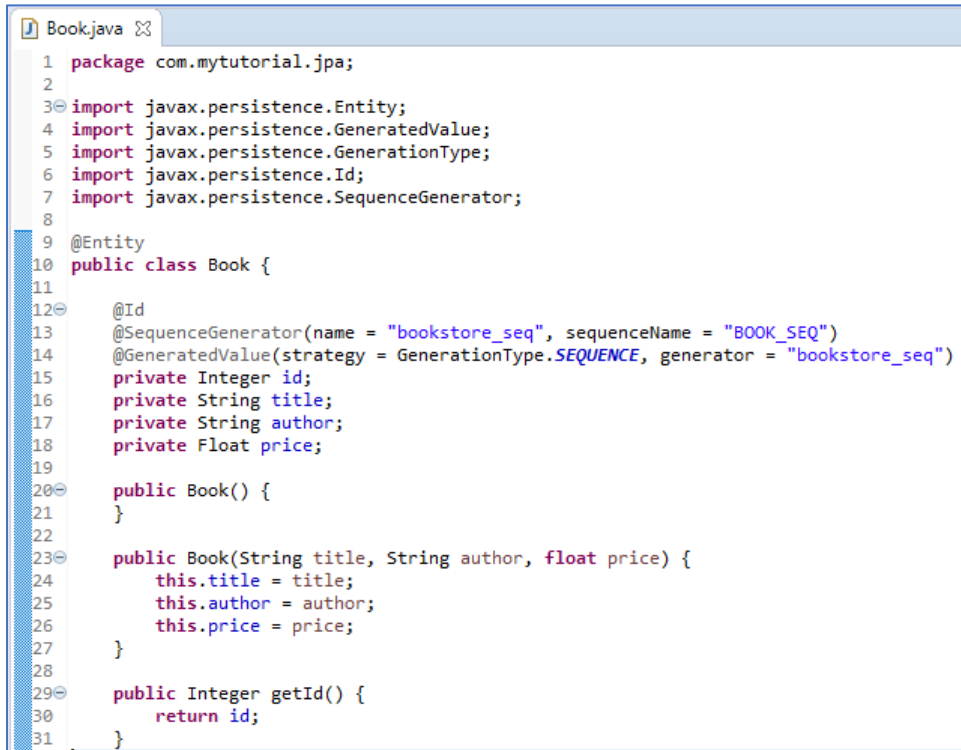
id	birthDate	name
1	1980-01-31 00:00:00.0	Gilad Barcha
2	1975-02-28 00:00:00.0	James Gosling

The id column you can see contains unique values starting from the value 1. The auto-increment on the individual id column, generates separate unique id's for each of the database tables that we have set up. Each id generator starts from 1.

Generation Type SEQUENCE for Primary Keys

In this demo, we'll understand and explore another primary key generation strategy that you can use with JPA and that is the **sequence** generator. We've seen that Hibernate uses the sequence generator by default when you configure the auto strategy, but here we'll see how we can explicitly use the sequence generator strategy to generate sequences for our primary keys. Go ahead and set up the import statement for **javax.persistence.SequenceGenerator**.

And let's change the strategy for our generated value here to be **GenerationType SEQUENCE**.



```

1 package com.mytutorial.jpa;
2
3 import javax.persistence.Entity;
4 import javax.persistence.GeneratedValue;
5 import javax.persistence.GenerationType;
6 import javax.persistence.Id;
7 import javax.persistence.SequenceGenerator;
8
9 @Entity
10 public class Book {
11
12     @Id
13     @SequenceGenerator(name = "bookstore_seq", sequenceName = "BOOK_SEQ")
14     @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "bookstore_seq")
15     private Integer id;
16     private String title;
17     private String author;
18     private Float price;
19
20     public Book() {
21     }
22
23     public Book(String title, String author, float price) {
24         this.title = title;
25         this.author = author;
26         this.price = price;
27     }
28
29     public Integer getId() {
30         return id;
31     }

```

Notice that in addition to the generated value and the Id annotations, there is a new annotation that I've specified here called **@SequenceGenerator**. The name of the sequence that we'll generate is called **"bookstore_seq"**, and the sequence name property has been set to **"BOOK_SEQ"**, all in caps.

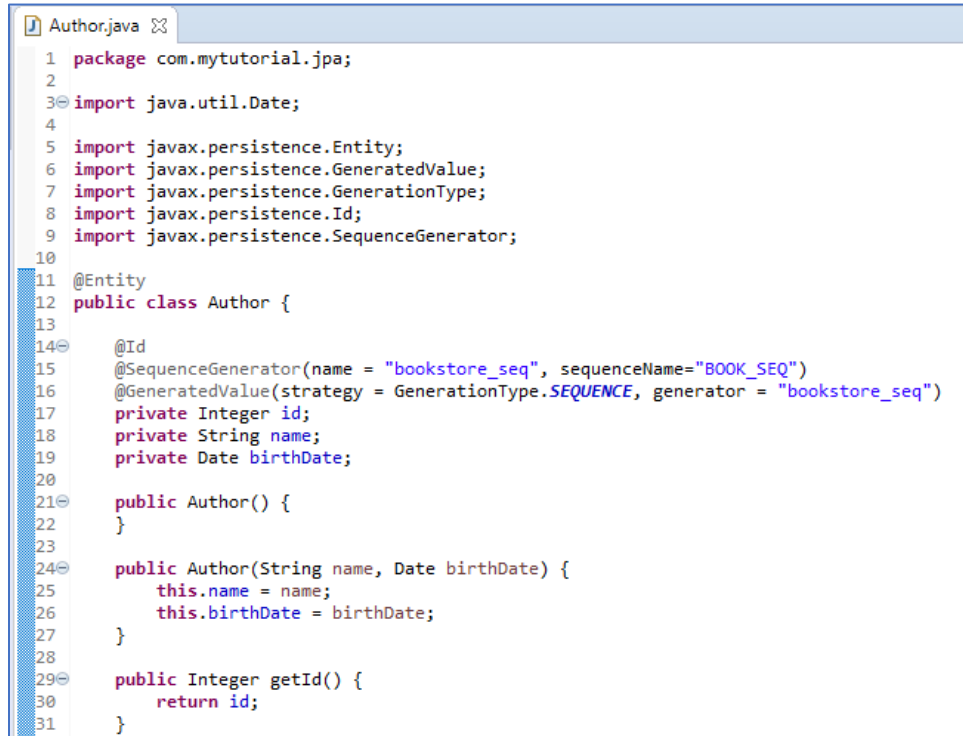
When you use the **SEQUENCE** Generation Strategy for your primary keys, your JPA provider will set up a database sequence which is a separate table that will contain the numbers or sequence for your primary keys.

Now with the generation strategy sequence you have fine grain control over this database table that is your sequence generator. In fact, you can have the same sequence generate primary keys for multiple tables.

With this in mind, let's go back and understand this **@SequenceGenerator** Annotation. Its name is **"bookstore_seq"**, and we refer to this sequence from within our Java code using this name. The sequence name property refers to the database table that contains the underlying sequence.

This is **"BOOK_SEQ"** in our example here. In order to use this generated sequence, the **@GeneratedValue** annotation specifies the strategy as **GenerationType.SEQUENCE**. And the generator refers to the sequence by name, generator equal to book sequence, which is the same as the name of our sequence generator. The rest of the code in this Book table remains exactly the same.

So let's head over to **Author.java**, and we'll have this entity use a generated sequence as well.



```
1 package com.mytutorial.jpa;
2
3 import java.util.Date;
4
5 import javax.persistence.Entity;
6 import javax.persistence.GeneratedValue;
7 import javax.persistence.GenerationType;
8 import javax.persistence.Id;
9 import javax.persistence.SequenceGenerator;
10
11 @Entity
12 public class Author {
13
14     @Id
15     @SequenceGenerator(name = "bookstore_seq", sequenceName="BOOK_SEQ")
16     @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "bookstore_seq")
17     private Integer id;
18     private String name;
19     private Date birthDate;
20
21     public Author() {
22     }
23
24     public Author(String name, Date birthDate) {
25         this.name = name;
26         this.birthDate = birthDate;
27     }
28
29     public Integer getId() {
30         return id;
31     }
32 }
```

Go ahead and set up the import statement for the sequence generator class. And I'm going to update this **@GeneratedValue** annotation to use **GenerationType.SEQUENCE** generator. So I have a **@SequenceGenerator** annotation which we have seen earlier, the name of the sequence is the same **"bookstore_seq"**. The **sequenceName** property refers to the database table that will contain our sequence that is **"BOOK_SEQ"**. The **@GeneratedValue** annotation uses the generation type **SEQUENCE**, and the generator is the bookstore sequence. We've used the same sequence generator for both of our tables, author as well as book. The use of the sequence as a generation strategy allows us to configure our tables in this manner.

The App.java code here hasn't changed at all. We've instantiated three book objects and persisted those in our database. We instantiate two author objects and persist those as well.

We are now ready to run this JPA Hibernate application and take a look at the results. Let's run this code through, there are no errors.

Java Persistence API: Configuring Fields & Performing CRUD Operations

```
Hibernate:
    drop table if exists Author
Dec 02, 2021 10:58:05 AM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolator
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal
Hibernate:
    drop table if exists Book
Hibernate:
    drop table if exists BOOK_SEQ
```

Let's observe what we see here within the console window. Thanks to our **drop-and-create** database action. The table Author if it exists, is dropped as is the table Book. Observe that Hibernate goes ahead and drops the table **BOOK_SEQ** if it exists. This is the database table that is our sequence generator.

If you scroll down further,

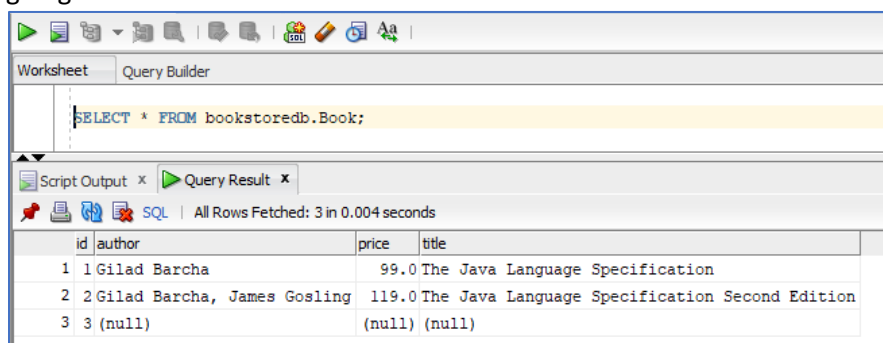
```
Hibernate:
    create table Author (
        id integer not null,
        birthDate datetime,
        name varchar(255),
        primary key (id)
    ) engine=MyISAM
Dec 02, 2021 10:58:05 AM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getI
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmen
Hibernate:
    create table Book (
        id integer not null,
        author varchar(255),
        price float,
        title varchar(255),
        primary key (id)
    ) engine=MyISAM
Hibernate:
    create table BOOK_SEQ (
        next_val bigint
    ) engine=MyISAM
```

you'll see that Hibernate creates a new table for our sequence generator. This is **BOOK_SEQ**, with one column called **next_val**.

```
Hibernate:
    insert into BOOK_SEQ values ( 1 )
Hibernate:
    insert into BOOK_SEQ values ( 1 )
Dec 02, 2021 10:58:05 AM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HHH000476: Executing import script 'org.hibernate.tool.schema.internal.exec.ScriptSourceInputNonExistentImpl@26fb628'
```

The initial value for the sequence generator is 1 which is inserted into the **BOOK_SEQ** table.

Now, how exactly did this sequence generator work? Let's head over to the MySQL Workbench and I'm going to run a select * from the **Book** table. Let's take a look at the results here.

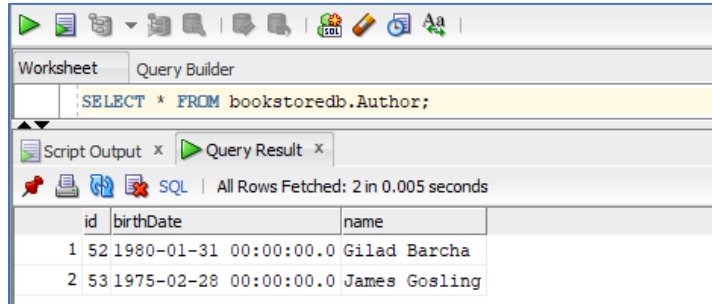


The screenshot shows the MySQL Workbench interface. The 'Query Builder' tab is active, displaying the query: `SELECT * FROM bookstoredb.Book;`. Below the query, the 'Query Result' tab shows the results of the query. The results are displayed in a table with 4 columns: id, author, price, and title. There are 3 rows of data.

id	author	price	title
1	Gilad Barcha	99.0	The Java Language Specification
2	Gilad Barcha, James Gosling	119.0	The Java Language Specification Second Edition
3	(null)	(null)	(null)

Observe that we have three entries in the [Book](#) table, the sequence for the primary key starts at 1 and then has values 2 and 3. So the sequence for this table that is the Book table started at 1.

Let's take a look at the Author table, which if you remember uses the same sequence generator. So I'm going to run the select * from the Author table, and let's take a look at the entries here.



The screenshot shows a database query tool interface. The top bar includes icons for running queries, saving, and other functions. Below the toolbar, there are tabs for 'Worksheet' and 'Query Builder'. The 'Query Builder' tab is active, showing the SQL query: `SELECT * FROM bookstoredb.Author;`. Below the query, there are tabs for 'Script Output' and 'Query Result'. The 'Query Result' tab is active, displaying the results of the query. The results are shown in a table with columns 'id', 'birthDate', and 'name'. The table contains two rows of data.

	id	birthDate	name
1	52	1980-01-31 00:00:00.0	Gilad Barcha
2	53	1975-02-28 00:00:00.0	James Gosling

Observe that the first entry has id 52, and the second entry has id 53. Each time the sequence generator allocates a new sequence of primary keys, two records inserted in a table. Within a transaction, it uses a default allocation size to increment the sequence. The default allocation size when you haven't specified anything explicitly, is 50. Which is why the first table has records starting from 1 and the second table has record starting from 52.

Generation Type TABLE for Primary Keys

In this demo, we'll understand and explore the fourth primary key generation strategy that JPA supports, and that is the table generator. In order to see how the table generator works, I'm going to modify my **persistence.xml**.



```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <persistence version="2.1"
3   xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
5
6   <persistence-unit name="BookstoreDB_Unit" >
7     <properties>
8       <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver" />
9       <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/bookstoredb" />
10      <property name="javax.persistence.jdbc.user" value="root" />
11      <property name="javax.persistence.jdbc.password" value="password" />
12
13      <property name="javax.persistence.schema-generation.database.action" value="drop-and-create"/>
14
15      <property name="javax.persistence.schema-generation.create-source" value="script"/>
16      <property name="javax.persistence.schema-generation.create-script-source" value="META-INF/create-table.sql"/>
17      <property name="javax.persistence.schema-generation.drop-source" value="script"/>
18      <property name="javax.persistence.schema-generation.drop-script-source" value="META-INF/drop-table.sql"/>
19
20      <property name="hibernate.show_sql" value="true"/>
21      <property name="hibernate.format_sql" value="true"/>
22    </properties>
23  </persistence-unit>
24
25 </persistence>
  
```

We'll continue to use the database action **drop-and-create**, and we'll pre-create some database tables that we'll use to generate our sequence.

In order to be able to use scripts with **drop-and-create**, we need to specify additional properties as well.

- The `javax.persistence.schema-generation.create-source` property is set to the value `"script"`, and
- The `javax.persistence.schema-generation.create-script-source` is the `"META-INF/create-table.sql"` script.
- Similarly, The `javax.persistence.schema-generation.drop-source` is set to the value `"script"`, and
- The `javax.persistence.schema-generation.drop-script-source` is the `"META-INF/drop-table.sql"` script.

Within the **META-INF/create-table.sql**, let's paste in a number of commands.

```

1 CREATE TABLE bookstore_table (gen_name VARCHAR(16) NOT NULL, gen_val INT NOT NULL);
2 INSERT INTO bookstore_table values ('book_id', 1);
3 INSERT INTO bookstore_table values ('author_id', 1000);
4 CREATE TABLE book (id INT, title VARCHAR(128), author VARCHAR(64), price FLOAT);
5 CREATE TABLE author (id INT, name VARCHAR(128), birthDate DATETIME);
6

```

One thing that you need to keep in mind when you use SQL script files from within JPA Hibernate is that every SQL command should be in its own line, it cannot span multiple lines. Otherwise you'll encounter an error.

Let's take a look at the SQL commands in here.

- The first is a **CREATE TABLE** command which creates a new table called *bookstore_table*.

```
CREATE TABLE bookstore_table (gen_name VARCHAR(16) NOT NULL, gen_val INT NOT NULL);
```

This *bookstore_table* will be the table that contains our sequences for both the author as well as book tables. *bookstore_table* has two columns, *gen_name* which is a **String** column, and *gen_val* that is an **Integer** column. Both of these are **NOT NULL**.

- We then insert two rows into the bookstore table.
One row corresponding to each of the tables that we are about to create, the *book* table and the *author* table, both of which will use sequences generated using this *bookstore_table*.

```
INSERT INTO bookstore_table values ('book_id', 1);
```

The first row here contains the starting sequence for our *book* table. So the values that I've inserted is *book_id* with a starting sequence of 1.

```
INSERT INTO bookstore_table values ('author_id', 1000);
```

The second row contains the starting sequence for our *author* table. So the values that I've inserted is *author_id* with a value of 1,000.

The *book_id* row corresponds to the sequence for the *book* table and the *author_id* row corresponds to the sequence for the *author* table.

- I then create the *book* and *author* tables.

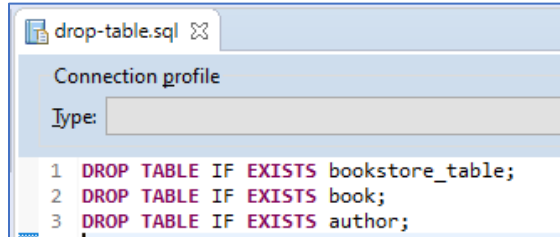
```
CREATE TABLE book (id INT, title VARCHAR(128), author VARCHAR(64), price FLOAT);
```

The *book* table has columns *id*, *title*, *author*, and *price*.

```
CREATE TABLE author (id INT, name VARCHAR(128), birthDate DATETIME);
```

The *author* table has columns *id*, *name*, and *birthDate*.

This completes our creation script in exactly the same way I need to set up the **META-INF/drop-table.sql** script as well.



And again how to use SQL script files from within JPA Hibernate is that every SQL command should be in its own line, it cannot span multiple lines. Otherwise you'll encounter an error.

This is the *drop-table.sql* script which we contained commands, allowing us to clean up after ourselves once we've exited the application. This contains three drop table commands for the [bookstore_table](#), [book](#) and [author](#) tables as well. The cleanup script will ensure that these tables are dropped and recreated each time we run our application, giving us a fresh start.

Let's head over to our `book.java` class which contains our entity. Set up the import statement for **`javax.persistence.TableGenerator`**. This is what we'll use to generate our primary keys.

Now let's go ahead and add the annotations that will allow us to use the **`@TableGenerator`** to generate our primary keys. In addition to the **`@Id`** and the **`@GeneratedValue`** annotation, we have a third annotation on our get id variable, that is the **`@TableGenerator`** annotation.

Book.java

```
Book.java
1 package com.mytutorial.jpaa;
2
3 import javax.persistence.Entity;
4 import javax.persistence.GeneratedValue;
5 import javax.persistence.GenerationType;
6 import javax.persistence.Id;
7 import javax.persistence.Table;
8 import javax.persistence.TableGenerator;
9
10 @Entity
11 @Table(name = "book")
12 public class Book {
13
14     @Id
15     @TableGenerator(name = "bookstore_generator",
16                     table = "bookstore_table",
17                     pkColumnName = "gen_name",
18                     pkColumnValue = "book_id",
19                     valueColumnName = "gen_val",
20                     allocationSize = 10)
21     @GeneratedValue(strategy = GenerationType.TABLE,
22                     generator = "bookstore_generator")
23     private Integer id;
24     private String title;
25     private String author;
26     private Float price;
27
28     public Book() {
29     }
30
31     public Book(String title, String author, float price) {
32         this.title = title;
33         this.author = author;
34         this.price = price;
35     }
36
37     public Integer getId() {
38         return id;
39     }
40 }
```

We've specified a number of properties for this **`@TableGenerator`** annotation. Let's understand what each of these properties represent.

- The first is the **`name`** property, and that is the name of our **`@TableGenerator`**, **`"bookstore_generator"`**. This is the name that we'll use to reference this table generator within our Java code.
- The **`table`** property in the **`@TableGenerator`** annotation, is what we use to identify which table in the underlying database we'll use to generate our primary key sequences. This is a table that we have set up using our create script. The create script is here the table is called **`"bookstore_table"`**.


```
create-table.sql
1 CREATE TABLE bookstore_table (gen_name VARCHAR(16) NOT NULL, gen_val INT NOT NULL);
2 INSERT INTO bookstore_table values ("book_id", 1);
3 INSERT INTO bookstore_table values ("author_id", 1000);
4 CREATE TABLE book (id INT, title VARCHAR(128), author VARCHAR(64), price FLOAT);
5 CREATE TABLE author (id INT, name VARCHAR(128), birthDate DATETIME);
```

- Now within this bookstore table, you need to tell Hibernate the name of the column which contains the name of the sequence that you have to use. The **pkColumnName** is "gen_name", which is the same as the *name* of our column in the *bookstore_table*.

```
9 @Entity
10 public class Book {
11
12     @Id
13     @TableGenerator(name = "bookstore_generator",
14                     table = "bookstore_table",
15                     pkColumnName = "gen_name",
16                     pkColumnValue = "book_id",
17                     valueColumnName = "gen_val",
18                     allocationSize = 10)
19     @GeneratedValue(strategy = GenerationType.TABLE,
20                     generator = "bookstore_generator")
21     private Integer id;
22     private String title;
```

```
create-table.sql
1 CREATE TABLE bookstore_table (gen_name VARCHAR(16) NOT NULL, gen_val INT NOT NULL);
2 INSERT INTO bookstore_table values ("book_id", 1);
3 INSERT INTO bookstore_table values ("author_id", 1000);
4 CREATE TABLE book (id INT, title VARCHAR(128), author VARCHAR(64), price FLOAT);
5 CREATE TABLE author (id INT, name VARCHAR(128), birthDate DATETIME);
```

- Property **pkColumnValue** gives us the *value* within this *gen_name* Column that we should use to generate our sequence for the *book* table. **pkColumnValue** is "book_id".

```
9 @Entity
10 public class Book {
11
12     @Id
13     @TableGenerator(name = "bookstore_generator",
14                     table = "bookstore_table",
15                     pkColumnName = "gen_name",
16                     pkColumnValue = "book_id",
17                     valueColumnName = "gen_val",
18                     allocationSize = 10)
19     @GeneratedValue(strategy = GenerationType.TABLE,
20                     generator = "bookstore_generator")
21     private Integer id;
22     private String title;
```

This is the record that we have explicitly inserted into the bookstore table with a corresponding value of 1.

```
create-table.sql
1 CREATE TABLE bookstore_table (gen_name VARCHAR(16) NOT NULL, gen_val INT NOT NULL);
2 INSERT INTO bookstore_table values ("book_id", 1);
3 INSERT INTO bookstore_table values ("author_id", 1000);
4 CREATE TABLE book (id INT, title VARCHAR(128), author VARCHAR(64), price FLOAT);
5 CREATE TABLE author (id INT, name VARCHAR(128), birthDate DATETIME);
```

- The Property **valueColumnName** is "gen_val", that is the name of the column in our bookstore table.

```

9 @Entity
10 public class Book {
11
12     @Id
13     @TableGenerator(name = "bookstore_generator",
14                     table = "bookstore_table",
15                     pkColumnName = "gen_name",
16                     pkColumnValue = "book_id",
17                     valueColumnName = "gen_val",
18                     allocationSize = 10)
19     @GeneratedValue(strategy = GenerationType.TABLE,
20                     generator = "bookstore_generator")
21     private Integer id;
22     private String title;

```

Now these three configuration parameters are a little confusing **pkColumnName**, **pkColumnValue**, and **valueColumnName**.

```

9 @Entity
10 public class Book {
11
12     @Id
13     @TableGenerator(name = "bookstore_generator",
14                     table = "bookstore_table",
15                     pkColumnName = "gen_name",
16                     pkColumnValue = "book_id",
17                     valueColumnName = "gen_val",
18                     allocationSize = 10)
19     @GeneratedValue(strategy = GenerationType.TABLE,
20                     generator = "bookstore_generator")
21     private Integer id;
22     private String title;

```

If you look at the corresponding entries in our `bookstore_table`, you'll understand exactly what each of these are referred to.

```

create-table.sql
1 CREATE TABLE bookstore_table (gen_name VARCHAR(16) NOT NULL, gen_val INT NOT NULL);
2 INSERT INTO bookstore_table values ("book_id", 1);
3 INSERT INTO bookstore_table values ("author_id", 1000);
4 CREATE TABLE book (id INT, title VARCHAR(128), author VARCHAR(64), price FLOAT);
5 CREATE TABLE author (id INT, name VARCHAR(128), birthDate DATETIME);

```

So make sure you double check and follow.

- The **allocationSize** that I have chosen for this generator is **10**. So after assigning our entries' primary keys, the sequence will be incremented by a count of 10. This is for the book sequence.
- One last detail to note here, the **@GeneratedValue** annotation uses **GenerationType.TABLE** as its strategy and the generator points to the "bookstore_generator", that is the *name* of our table generator.

Now I'm going to configure the `author` entity to use the same table generator, but a different sequence within that table generator.

Author.java

```

1 package com.mytutorial.jpaa;
2
3 import java.util.Date;
4
5 import javax.persistence.Entity;
6 import javax.persistence.GeneratedValue;
7 import javax.persistence.GenerationType;
8 import javax.persistence.Id;
9 import javax.persistence.Table;
10 import javax.persistence.TableGenerator;
11
12 @Entity
13 @Table(name = "author")
14 public class Author {
15
16     @Id
17     @TableGenerator(name = "bookstore_generator",
18         table = "bookstore_table",
19         pkColumnName = "gen_name",
20         pkColumnValue = "author_id",
21         valueColumnName = "gen_val",
22         allocationSize = 100)
23     @GeneratedValue(strategy = GenerationType.TABLE,
24         generator = "bookstore_generator")
25     private Integer id;
26     private String name;
27     private Date birthDate;
28
29     public Author() {
30     }
31
32     public Author(String name, Date birthDate) {
33         this.name = name;
34         this.birthDate = birthDate;
35     }
36
37     public Integer getId() {
38         return id;
39     }

```

Let's set up the right annotations for our fields `id`. We have the `@Id` annotation and the `@GeneratedValue` annotation, both of which we are familiar with, no explanation is required there. What is really interesting here is the `@TableGenerator` annotation, and that's what we'll focus our attention on.

- Our `@TableGenerator` annotation takes in a `name` property. The `name` of this table generator is simply `"bookstore_generator"`. This is how we refer to this table generator from within our Java code.
- The `table` property points to the underlying table in our database which contains the values that we use to generate our primary key sequences. Our create script for this application has already set up this `bookstore_table`.

```

1 CREATE TABLE bookstore_table (gen_name VARCHAR(16) NOT NULL, gen_val INT NOT NULL);
2 INSERT INTO bookstore_table values ("book_id", 1);
3 INSERT INTO bookstore_table values ("author_id", 1000);
4 CREATE TABLE book (id INT, title VARCHAR(128), author VARCHAR(64), price FLOAT);
5 CREATE TABLE author (id INT, name VARCHAR(128), birthDate DATETIME);

```

```

11 @Entity
12 public class Author {
13
14     @Id
15     @TableGenerator(name = "bookstore_generator",
16                     table = "bookstore_table",
17                     pkColumnName = "gen_name",
18                     pkColumnValue = "author_id",
19                     valueColumnName = "gen_val",
20                     allocationSize = 100)
21     @GeneratedValue(strategy = GenerationType.TABLE,
22                     generator = "bookstore_generator")
23     private Integer id;
24     private String name;

```

- The **pkColumnName** is the *name* of the column that contains the names of the sequence of values that we've inserted into this table. "gen_name" is the *name* of this column, which is of type varchar 16, it cannot be null.

```

create-table.sql
1 CREATE TABLE bookstore_table (gen_name VARCHAR(16) NOT NULL, gen_val INT NOT NULL);
2 INSERT INTO bookstore_table values ("book_id", 1);
3 INSERT INTO bookstore_table values ("author_id", 1000);
4 CREATE TABLE book (id INT, title VARCHAR(128), author VARCHAR(64), price FLOAT);
5 CREATE TABLE author (id INT, name VARCHAR(128), birthDate DATETIME);

```

- Property **pkColumnValue** tells us the *value* within the gen_name column that we should use to generate the sequence for this author entity. The value here is `author_id`, which is a record that we've inserted into this bookstore table.

```

create-table.sql
1 CREATE TABLE bookstore_table (gen_name VARCHAR(16) NOT NULL, gen_val INT NOT NULL);
2 INSERT INTO bookstore_table values ("book_id", 1);
3 INSERT INTO bookstore_table values ("author_id", 1000);
4 CREATE TABLE book (id INT, title VARCHAR(128), author VARCHAR(64), price FLOAT);
5 CREATE TABLE author (id INT, name VARCHAR(128), birthDate DATETIME);

```

`author_id` corresponds to a value of **1,000**, which means we'll start that primary key sequence close to 1,000, it's based on the **allocation** size.

- The **valueColumnName** is `gen_val`. This is the *name* of the column that holds the sequence values in our `bookstore_table`.

```

create-table.sql
1 CREATE TABLE bookstore_table (gen_name VARCHAR(16) NOT NULL, gen_val INT NOT NULL);
2 INSERT INTO bookstore_table values ("book_id", 1);
3 INSERT INTO bookstore_table values ("author_id", 1000);
4 CREATE TABLE book (id INT, title VARCHAR(128), author VARCHAR(64), price FLOAT);
5 CREATE TABLE author (id INT, name VARCHAR(128), birthDate DATETIME);

```

- And finally, the **allocationSize** that I've chosen for this generator is 100. This is the number by which our sequence will be auto incremented.
- And finally, ensure that your **@GeneratedValue** annotation has strategy set to **GenerationType.TABLE** and the generator that it refers to is the `bookstore_generator` that we have set up.

Java Persistence API: Configuring Fields & Performing CRUD Operations

Now let's head over to `app.java` and run the same code as before. This is the code that persists three book entities and two author entities in our database.

```
JUnit Console Markers Properties Servers Data Source Explorer Snippets Problems Progress Search Debug Coverage
<terminated> App my-jpa-app [Java Application] C:\Program Files\Java\jdk1.8.0_202\bin\javaw.exe (Dec 2, 2021, 3:06:32 PM - 3:06:35 PM)
Dec 02, 2021 3:06:32 PM org.hibernate.jpa.internal.util.LogHelper logPersistenceUnitInformation
INFO: HHH000204: Processing PersistenceUnitInfo [
    name: BookstoreDB_Unit
    ...]
Dec 02, 2021 3:06:32 PM org.hibernate.Version logVersion
INFO: HHH000412: Hibernate Core (5.2.12.Final)
Dec 02, 2021 3:06:32 PM org.hibernate.cfg.Environment <clinit>
INFO: HHH000206: hibernate.properties not found
Dec 02, 2021 3:06:32 PM org.hibernate.annotations.common.reflection.java.JavaReflectionManager <clinit>
INFO: HCAN000001: Hibernate Commons Annotations (5.0.1.Final)
Dec 02, 2021 3:06:32 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl configure
WARN: HHH10001002: Using Hibernate built-in connection pool (not for production use!)
Dec 02, 2021 3:06:32 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl buildCreator
INFO: HHH10001005: using driver [com.mysql.cj.jdbc.Driver] at URL [jdbc:mysql://localhost:3306/bookstoredb]
Dec 02, 2021 3:06:32 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl buildCreator
INFO: HHH10001001: Connection properties: {user=root, password=****}
Dec 02, 2021 3:06:32 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl buildCreator
INFO: HHH10001003: Autocommit mode: false
Dec 02, 2021 3:06:32 PM org.hibernate.engine.jdbc.connections.internal.PooledConnections <init>
INFO: HHH000115: Hibernate connection pool size: 20 (min=1)
Dec 02, 2021 3:06:35 PM org.hibernate.dialect.Dialect <init>
INFO: HHH000400: Using dialect: org.hibernate.dialect.MySQL5Dialect
Hibernate:
    DROP TABLE IF EXISTS bookstore_table
Dec 02, 2021 3:06:35 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionProviderJdbcConnecti
Hibernate:
    DROP TABLE IF EXISTS book
Hibernate:
    DROP TABLE IF EXISTS author
```

The code seems to have run through fine, let's see what's happening under the hood. We drop the table `bookstore_table` and Hibernate recreates this.

```
Hibernate:
    CREATE TABLE bookstore_table (
        gen_name VARCHAR(16) NOT NULL,
        gen_val INT NOT NULL
    )
Dec 02, 2021 3:06:35 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$Connect
Hibernate:
    INSERT INTO bookstore_table values ("book_id", 1)
Hibernate:
    INSERT INTO bookstore_table values ("author_id", 1000)
```

Notice that this sequence generator table (`bookstore_table`) has two columns, `gen_name` which is of type **VARCHAR** and `gen_val` that is of type **INT**. Both of these columns cannot be null. We now insert two entries into this `bookstore_table`. The first record contains the values “`book_id`” and 1, the second record “`author_id`” and 1000.

The “`book_id`” record will be used to generate primary key sequences. For the `book` table, the “`author_id`” record will be used to generate primary key sequences for the `author` table.

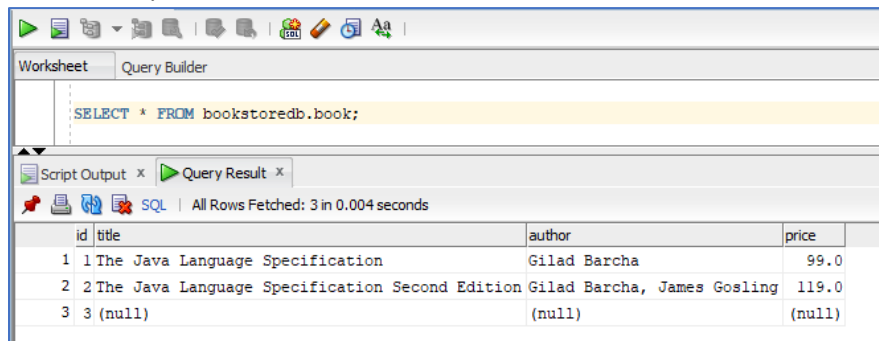
```
Hibernate:
    CREATE TABLE book (
        id INT,
        title VARCHAR(128),
        author VARCHAR(64),
        price FLOAT
    )
Hibernate:
    CREATE TABLE author (
        id INT,
        name VARCHAR(128),
        birthDate DATETIME
    )
Dec 02, 2021 3:06:35 PM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HHH000476: Executing import script 'org.hibernate.tool.schema.internal.exec.ScriptSourceInputNonExistentImpl@26fb628'
```

Java Persistence API: Configuring Fields & Performing CRUD Operations

The remaining insertion statements seem to have been executed successfully.

```
Hibernate:
insert
into
  book
  (author, price, title, id)
values
  (?, ?, ?, ?)
Hibernate:
insert
into
  author
  (birthDate, name, id)
values
  (?, ?, ?)
Dec 02, 2021 3:06:35 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl stop
INFO: HHH100010008: Cleaning up connection pool [jdbc:mysql://localhost:3306/bookstoredb]
```

So what's interesting now is the final result which we'll view using MySQL workbench. I'm going to run a select star operation from the `book` table and take a look at the entries that exist.



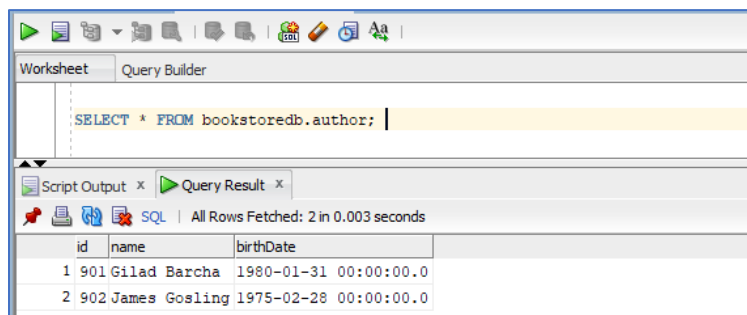
The screenshot shows the MySQL Workbench interface. The 'Query Builder' tab is active, displaying the query: `SELECT * FROM bookstoredb.book;`. Below the query, the 'Query Result' tab shows the results of the query. The results are displayed in a table with 4 columns: `id`, `title`, `author`, and `price`. There are 3 rows of data.

id	title	author	price
1	The Java Language Specification	Gilad Barcha	99.0
2	The Java Language Specification Second Edition	Gilad Barcha, James Gosling	119.0
3	(null)	(null)	(null)

You can see that there are three entries here. The primary keys associated with these entries start at one, then we have two and three. And this squares with what we had specified.

We had inserted into the bookstore table `book_id` and a value of one, which is the first primary key in the sequence generated for the book table.

Now let's take a look at our `author` table. I'm going to run a select star command to view the entries that exist in here.

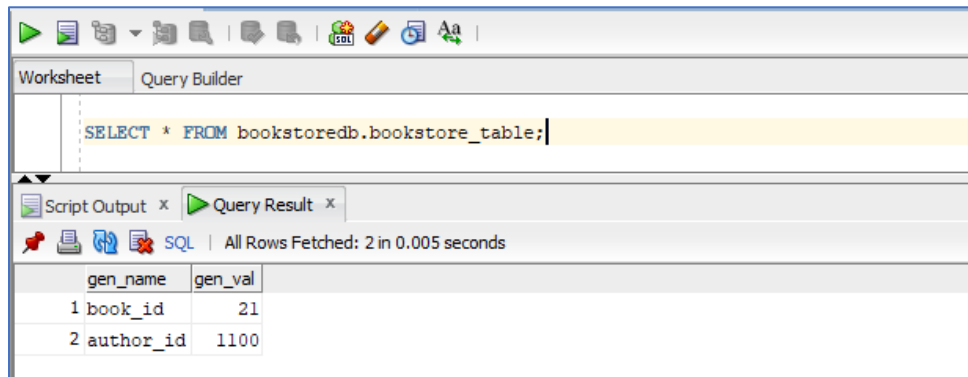


The screenshot shows the MySQL Workbench interface. The 'Query Builder' tab is active, displaying the query: `SELECT * FROM bookstoredb.author;`. Below the query, the 'Query Result' tab shows the results of the query. The results are displayed in a table with 4 columns: `id`, `name`, and `birthDate`. There are 2 rows of data.

id	name	birthDate
1 901	Gilad Barcha	1980-01-31 00:00:00.0
2 902	James Gosling	1975-02-28 00:00:00.0

The `author` table contains two entries with primary key starting at 901, then the second one is 902. That's because we've inserted the initial value as 1,000 and specified an allocation size of 100. So the table generator use the allocation size and started allocating primary keys for entries in this `author` table starting at 901.

There is one last interesting detail left for us to see. Let's run a select star operation on the `bookstore_table`. This is the table that we use to generate primary key values.



You can see that the allocation size that we had specified for the `book_id` and the `author_id` records have been used to increment the corresponding values for these records.

Composite Keys Using Embeddable and Id

So far in our JPA Hibernate implementation, we've seen a number of different strategies that we can use to generate primary keys. But in each case, the primary key is just one column in your database table. It's possible that your data is such that you have no column that uniquely identifies a record, which means you might need to use composite keys.

Composite keys are combinations of two or more columns which uniquely identify a row. A single column by itself may not hold a unique value for each record, but multiple columns together allow you to uniquely identify rows.

The Java Persistence API allows you to model composite keys in a variety of different ways, some of which are intuitive, some not so much.

Let's explore the different techniques that you can use to model composite keys. For this particular demo, our **persistence.xml** file is back to having the database action as drop and create and we don't have any additional scripts. We still continue to use the [BookstoreDB_Unit](#) that connects to the [bookstoredb](#) database, in our MySQL database server.

```
persistence.xml
1 <?xml version="1.0" encoding="UTF-8"?>
2 <persistence version="2.1"
3   xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
5
6   <persistence-unit name="BookstoreDB_Unit" >
7     <properties>
8       <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver" />
9       <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/bookstoredb" />
10      <property name="javax.persistence.jdbc.user" value="root" />
11      <property name="javax.persistence.jdbc.password" value="password" />
12
13      <property name="javax.persistence.schema-generation.database.action" value="drop-and-create"/>
14
15      <property name="hibernate.show_sql" value="true"/>
16      <property name="hibernate.format_sql" value="true"/>
17    </properties>
18  </persistence-unit>
19
20 </persistence>
```


Now in this example, I'm going to create a new class to represent a composite key. This composite key will be a class called **BookKey.java**, which contains multiple fields that make up our composite key.

```
BookKey.java
1 package com.mytutorial.jpaa;
2
3 import java.io.Serializable;
4 import java.util.Objects;
5
6 import javax.persistence.Embeddable;
7
8 @Embeddable
9 public class BookKey implements Serializable {
10
11     private static final long serialVersionUID = 1L;
12
13     private Integer titleHash;
14     private Float price;
15
16     public BookKey() {
17     }
18
19     public BookKey(String title, Float price) {
20         this.titleHash = Objects.hash(title);
21         this.price = price;
22     }
23
24     public Float getPrice() {
25         return price;
26     }
27
28     public void setPrice(Float price) {
29         this.price = price;
30     }
31
32     @Override
33     public int hashCode() {
34         return Objects.hash(titleHash, price);
35     }
36
37     @Override
38     public boolean equals(Object obj) {
39         if (obj == null)
40             return false;
41         if (!(obj instanceof BookKey))
42             return false;
43         BookKey other = (BookKey) obj;
44         return this.titleHash.equals(other.titleHash) && this.price.equals(other.price);
45     }
46 }
47 }
```

This new Java class BookKey is not going to be an entity, but instead is going to be an object that is embedded within an entity and this object represents a key in that other entity within which it's embedded. Which means there are certain requirements that the BookKey class must fulfill in order to be an embedded entity as well as a complex key.

- The BookKey should be an **Embeddable**, which is why we import the annotation **javax.persistence.Embeddable**. This BookKey class will represent objects that are not entities in their own right but are instead referenced by other entities that we have within our application. Which means this BookKey class is embedded into those other entities. We need to tag this with an **@Embeddable** annotation. Such embeddable classes do not generate tables of their own. All fields that we'll define within this embeddable class will be mapped into the owner entity table.
- It's good practice to have your embeddable implement a **serializable** interface. In fact, as a best practice, all of the entities that you write within your application should implement serializable, indicating that they can be serialized to file all across the wire.

When you work with JPA and Hibernate, it's quite possible that you're working within a spring application or you use Java Beans. It's good practice to have your JPA persistence objects, be Java Beans, serializable and implementing a no argument constructor.

- When your class implements the serializable interface, you need to have a **static final long serial version UID**, which are set to the default value of "1".

```
private static final long serialVersionUID = 1L;
```

- An embeddable object may or may not be a composite key in the parent entity in this example, our embeddable is indeed a composite key. So the fields that you see within this embeddable, the `titleHash`, and the floating point `price` fields, together make up the composite key in our book object.

```
private Integer titleHash;  
private Float price;
```

These two columns together guarantee the uniqueness of every record in our book table.

- Here is our public default no argument constructor.

```
public BookKey() {}
```

- In addition, you can have other constructors that take in input arguments. We have one here which takes in the String `title` and the floating point `price`.

```
public BookKey(String title, Float price) {  
    this.titleHash = Objects.hash(title);  
    this.price = price;  
}
```

Within this constructor, we use the **Objects.hash()** method to generate a hash of the `title` String. Rather than using the title directly as a composite key, we generate a hash of the title and `this.titleHash` is what will be our composite key. We'll store the title separately as well, but we want only the hash to form one column in our composite key. The second column is the `price` of the book.

- I have a getter and a setter for the `price` variable in order to access the `price` using this composite key,

```
public Float getPrice() {  
    return price;  
}  
  
public void setPrice(Float price) {  
    this.price = price;  
}
```

- And Then I have two methods that are absolutely required. This is the overridden version of the **hashCode()** method in the base class and the **equals()** method in the base class.

```
@Override
public int hashCode() {
    return Objects.hash(titleHash, price);
}

@Override
public boolean equals(Object obj) {
    if (obj == null)
        return false;
    if (! (obj instanceof BookKey))
        return false;
    BookKey other = (BookKey) obj;
    return this.titleHash.equals(other.titleHash) &&
        this.price.equals(other.price);
}
```

Whenever you have an embeddable object that is a composite key for an entity, that key needs to be unique. And the way you compare keys for uniqueness is to check to see whether two keys are *equal* to one another. This means that we have to implement the overridden equals method from the object base class in order to compare two keys for equality correctly.

Best practices in Java programming states that if you *override* the **equals()** method from the object base class, you should also *override* the **hashCode()** method to uniquely generate a hash for every object of this class. And both of these methods are *overridden*.

Take a look at the overridden **hashCode()** method, we simply use the **Objects.hash()** utility to generate a hash of the **titleHash**, which we've already generated, and the **price** of the book. And we also override the equals method.

The **equals()** method takes as an input argument an object. If that other object is equal to null, obviously the current object is not equal to the other, we return false. If the other object is not an instance of BookKey, we return false. Otherwise, we go ahead and compare the title hash and the price of the current BookKey with the other BookKey to see if they match if they match. If they match, the two keys are equal, otherwise, they are not.

Now let's head over to **Book.java** which we are going to rewrite to use the book key embeddable as a composite key.

```
Book.java
1 package com.mytutorial.jpaa;
2
3 import javax.persistence.Entity;
4 import javax.persistence.Id;
5
6 @Entity
7 public class Book {
8
9     @Id
10    private BookKey bookKey;
11
12    private String title;
13    private String author;
14
15    public Book() {
16    }
17
18    public Book(String title, String author, float price) {
19        this.bookKey = new BookKey(title, price);
20        this.title = title;
21        this.author = author;
22    }
23
24    public BookKey getBookKey() {
25        return bookKey;
26    }
27
28    public void setBookKey(BookKey bookKey) {
29        this.bookKey = bookKey;
30    }
31
32    public String getTitle() {
33        return title;
34    }
35
36    public void setTitle(String title) {
37        this.title = title;
38    }
39
40    public String getAuthor() {
41        return author;
42    }
43
44    public void setAuthor(String author) {
45        this.author = author;
46    }
47
48 }
```

- This book class marked with the **@Entity** annotation that we have defined is an entity which means we have a table corresponding to the book object in our database.
- Observe that we have a member variable **bookKey** of type **BookKey**. And we have annotated this member variable with the **@Id** annotation. Now this **@Id** annotation combined with the fact that the **bookKey** is an embeddable, makes the **bookKey** a *Composite Key* of the **Book** object.
- The fields within the **BookKey**, that is the **titleHash** and the **price** of the **Book**, will be the columns that make up our composite key.

```
8 @Embeddable
9 public class BookKey implements Serializable {
10
11    private Integer titleHash;
12    private Float price;
13
14    public BookKey(String title, Float price) {
15        this.titleHash = Objects.hash(title);
16        this.price = price;
17    }
18 }
```

- In addition, we have two other fields in the book object that is the `author` and the `title`. These will be separate columns in our `Book` table.

```
private String title;  
private String author;
```

- We have the default no argument constructor, as we did earlier, we also have a constructor that takes an input arguments. The `title` of the `Book`, the `author`, and the `price`.

```
public Book() {  
}  
  
public Book(String title, String author, float price) {  
    this.bookKey = new BookKey(title, price);  
    this.title = title;  
    this.author = author;  
}
```

- Setup the getters and setters for every member variable that we have within this book class for the book key, the title and the author.

We also have the author class representing the *Author* entity.

```
Author.java
1 package com.mytutorial.jpaa;
2
3 import java.util.Date;
4
5
6
7
8
9
10 @Entity
11 public class Author {
12
13     @Id
14     @GeneratedValue(strategy = GenerationType.IDENTITY)
15     private Integer id;
16     private String name;
17     private Date birthDate;
18
19     public Author() {
20     }
21
22     public Author(String name, Date birthDate) {
23         this.name = name;
24         this.birthDate = birthDate;
25     }
26
27     public Integer getId() {
28         return id;
29     }
30
31     public void setId(Integer id) {
32         this.id = id;
33     }
34
35     public String getName() {
36         return name;
37     }
38
39     public void setName(String name) {
40         this.name = name;
41     }
42
43     public Date getBirthDate() {
44         return birthDate;
45     }
46
47     public void setBirthDate(Date birthDate) {
48         this.birthDate = birthDate;
49     }
50
51 }
```

- The author entity we've kept simple, it has an *id*, *name* and *birthDate* columns.
- The primary key for the author will be a single column and that is the ID column. We use the **@Id** annotation on the getter for the ID, and we use the **GeneratedValue** with the Strategy of *IDENTITY*. We asked the database to be responsible for generating primary keys for this table, using auto incremented ID values.

Now let's head over to **App.java**, where we use the entity manager to persist new entities of the *Book* type as well as the *Author* type.

```
App.java
1 package com.mytutorial.jpa;
2
3 import java.util.GregorianCalendar;
4
5 import javax.persistence.EntityManager;
6 import javax.persistence.EntityManagerFactory;
7 import javax.persistence.Persistence;
8
9 /**
10  * Hello world!
11  *
12  */
13 public class App
14 {
15     public static void main( String[] args )
16     {
17
18         EntityManagerFactory factory = Persistence.createEntityManagerFactory("BookstoreDB_Unit");
19         EntityManager entityManager = factory.createEntityManager();
20
21         try {
22             entityManager.getTransaction().begin();
23
24             Book firstBook = new Book("The Java Language Specification",
25                                     "Gilad Barcha", 99f);
26             Book secondBook = new Book("The Java Language Specification Second Edition",
27                                     "Gilad Barcha, James Gosling", 119f);
28             Book thirdBook = new Book("Core Java Volume I", "Cay S. Horstmann", 59f);
29
30             entityManager.persist(firstBook);
31             entityManager.persist(secondBook);
32             entityManager.persist(thirdBook);
33
34             Author firstAuthor = new Author("Gilad Barcha", new GregorianCalendar(1980, 1, 0).getTime());
35             Author secondAuthor = new Author("James Gosling", new GregorianCalendar(1975, 2, 0).getTime());
36
37             entityManager.persist(firstAuthor);
38             entityManager.persist(secondAuthor);
39
40         } catch (Exception ex) {
41             System.err.println("An error occurred: " + ex);
42         } finally {
43             entityManager.getTransaction().commit();
44             entityManager.close();
45             factory.close();
46         }
47     }
48 }
49 }
```

The code here is not very different from what we had before. The third book with no input arguments, I'm going to change so that it's a regular book with a *title*, the *name* of the author and the *price*. Remember, the hash of the title and the price make up our composite key, so both of those are required.

Now we have three books, each with titles, *authors* and a *price*. In addition to the three books that we persist in our database, we also persist to author entities, for "Gilad Barcha" and "James Gosling".

Let's go ahead and run this code, and see what the results look like when we use a composite key within our table.

```

Hibernate:
    drop table if exists Author
Dec 03, 2021 9:36:54 AM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJt
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.Jdb
Hibernate:

    drop table if exists Book
Hibernate:

    create table Author (
        id integer not null auto_increment,
        birthDate datetime,
        name varchar(255),
        primary key (id)
    ) engine=MyISAM
Dec 03, 2021 9:36:54 AM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJt
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.Jdb
Hibernate:

    create table Book (
        price float not null,
        titleHash integer not null,
        author varchar(255),
        title varchar(255),
        primary key (price, titleHash)
    ) engine=MyISAM
    
```

Now the code runs through without error. Let's scroll down and see what happens. We drop the *Author* and the *Book* table.

And notice when we recreate the *Book* table, how the **primary key** has been set up.

- First, let's observe the columns in this *Book* table that we have created. We have the *price* and the *titleHash*. These are the member variables from our *BookKey Composite Key*, *embeddable*.

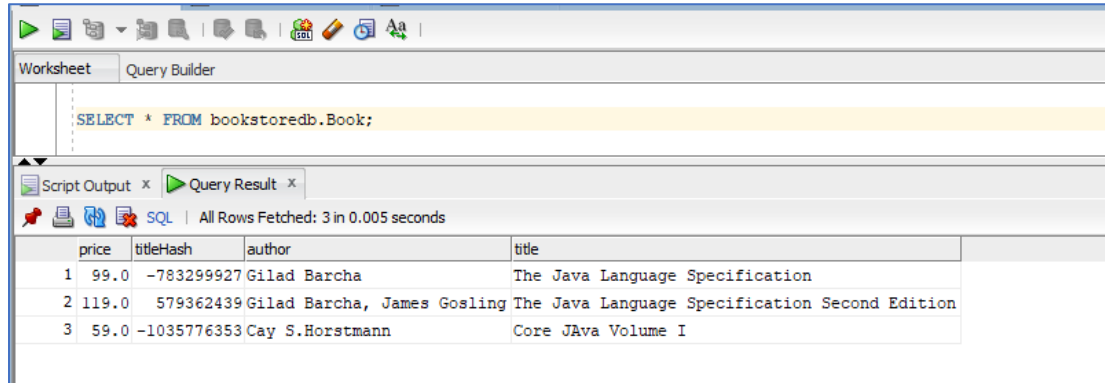
```

6 @Entity
7 public class Book {
8
9     @Id
10    private BookKey bookKey;
11
12    private String title;
13    private String author;
14
15    public Book() {
16    }
17
18    public Book(String title, String author, float price) {
19        this.bookKey = new BookKey(title, price);
20        this.title = title;
21        this.author = author;
22    }
23
24    8 @Embeddable
25    9 public class BookKey implements Serializable {
26
27        10
28        11 private static final long serialVersionUID = 1L;
29        12
30        13 private Integer titleHash;
31        14 private Float price;
32        15
33        16 public BookKey() {
34        17
35        18
36        19 public BookKey(String title, Float price) {
37            20 this.titleHash = Objects.hash(title);
38            21 this.price = price;
39            22
40        }
41    }
    
```

- Then we have the *author* and the *title* from our *Book* entity
- and the primary key is a combination of the *price* and the *titleHash*.

Java Persistence API: Configuring Fields & Performing CRUD Operations

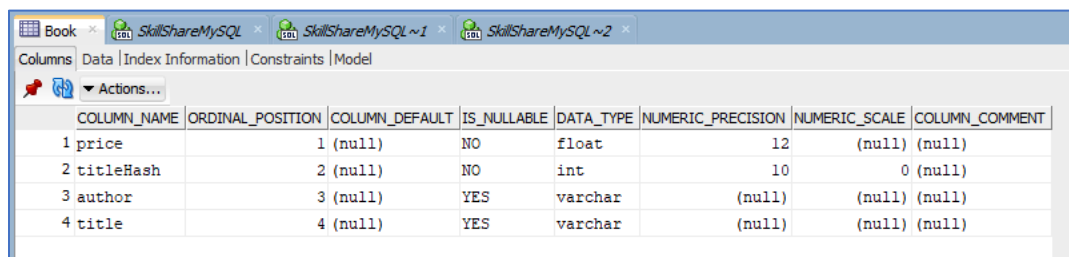
Everything else looks okay, so we can head over to the MySQL Workbench and see how the records have been inserted into our various tables. Let's do a select star from the *Book* table. And here are the three books that have been inserted.



The screenshot shows the MySQL Workbench interface. The Query Builder tab is active, displaying the query `SELECT * FROM bookstoredb.Book;`. Below the query, the 'Query Result' tab shows the results of the query. The results are displayed in a table with 4 columns: price, titleHash, author, and title. There are 3 rows of data.

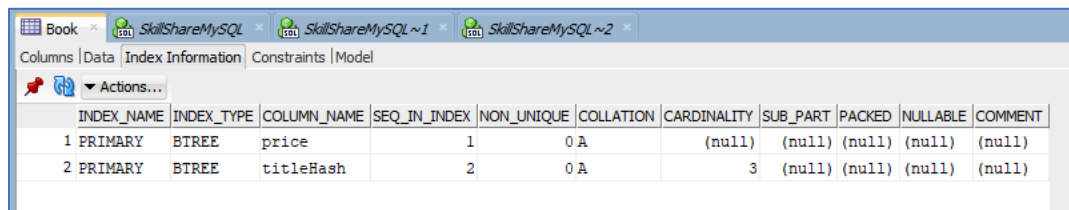
	price	titleHash	author	title
1	99.0	-783299927	Gilad Barcha	The Java Language Specification
2	119.0	579362439	Gilad Barcha, James Gosling	The Java Language Specification Second Edition
3	59.0	-1035776353	Cay S. Horstmann	Core Java Volume I

The *price* and the *titleHash* together make up our composite key, and every book has these fields. Let's execute a describe command to see the structure of this book table with the composite key. And this will show you that the fields *price* and *titleHash* together make up our **primary key**. This is a composite key with multiple columns.



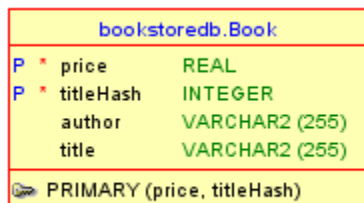
The screenshot shows the MySQL Workbench interface with the 'Columns' tab selected. It displays the output of a DESCRIBE command for the 'Book' table. The table has 4 columns: price, titleHash, author, and title. The 'price' and 'titleHash' columns are part of a primary key.

	COLUMN_NAME	ORDINAL_POSITION	COLUMN_DEFAULT	IS_NULLABLE	DATA_TYPE	NUMERIC_PRECISION	NUMERIC_SCALE	COLUMN_COMMENT
1	price	1	(null)	NO	float	12	(null)	(null)
2	titleHash	2	(null)	NO	int	10	0	(null)
3	author	3	(null)	YES	varchar	(null)	(null)	(null)
4	title	4	(null)	YES	varchar	(null)	(null)	(null)



The screenshot shows the MySQL Workbench interface with the 'Index Information' tab selected. It displays the output of an INDEX INFORMATION command for the 'Book' table. The table has 2 indexes: PRIMARY and BTREE. The PRIMARY index is a composite key on the 'price' and 'titleHash' columns.

	INDEX_NAME	INDEX_TYPE	COLUMN_NAME	SEQ_IN_INDEX	NON_UNIQUE	COLLATION	CARDINALITY	SUB_PART	PACKED	NULLABLE	COMMENT
1	PRIMARY	BTREE	price	1	0	A	(null)	(null)	(null)	(null)	(null)
2	PRIMARY	BTREE	titleHash	2	0	A	3	(null)	(null)	(null)	(null)



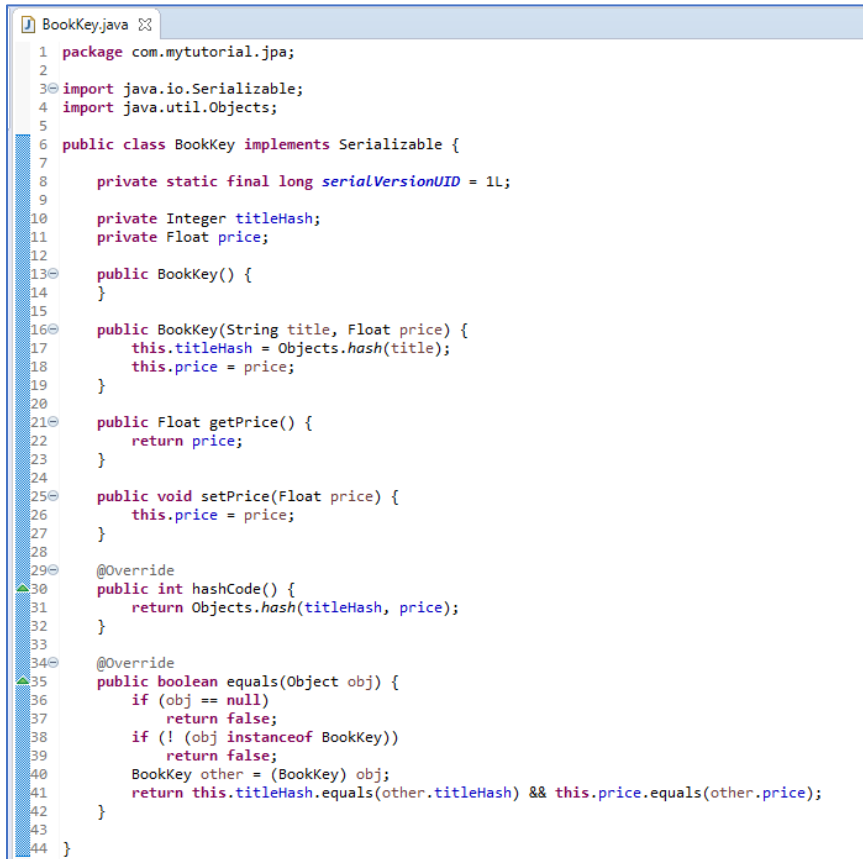
The diagram shows the structure of the *bookstoredb.Book* table. It lists the columns and their data types, and indicates that the *price* and *titleHash* columns form a primary key.

bookstoredb.Book	
P *	price REAL
P *	titleHash INTEGER
	author VARCHAR2 (255)
	title VARCHAR2 (255)
PRIMARY (price, titleHash)	

Composite Keys Using EmbeddedId

It's possible for you to specify composite keys for your JPA entities in a slightly different way as well. Instead of using the separate **@Embeddable** and **@Id** annotation, you can use a single annotation, the **@EmbeddableId**. Let's see how.

Here is our updated **BookKey.java** file. This is the object that represents our composite key, and it contains the field's **titleHash** and **price**. Notice though that I've *gotten rid* of the **@Embeddable** annotation for this **BookKey**.



```

1 package com.mytutorial.jpa;
2
3 import java.io.Serializable;
4 import java.util.Objects;
5
6 public class BookKey implements Serializable {
7
8     private static final long serialVersionUID = 1L;
9
10    private Integer titleHash;
11    private Float price;
12
13    public BookKey() {
14    }
15
16    public BookKey(String title, Float price) {
17        this.titleHash = Objects.hash(title);
18        this.price = price;
19    }
20
21    public Float getPrice() {
22        return price;
23    }
24
25    public void setPrice(Float price) {
26        this.price = price;
27    }
28
29    @Override
30    public int hashCode() {
31        return Objects.hash(titleHash, price);
32    }
33
34    @Override
35    public boolean equals(Object obj) {
36        if (obj == null)
37            return false;
38        if (! (obj instanceof BookKey))
39            return false;
40        BookKey other = (BookKey) obj;
41        return this.titleHash.equals(other.titleHash) && this.price.equals(other.price);
42    }
43
44 }

```

This **BookKey** is simply a plain old Java object with no annotations, it implements **Serializable**. It has a default no argument constructor on lines 13, 14, and 15. But it has no additional JPA annotation identifying this class as an object that represents a composite key.

We'll now head over to the **Book.java** class to make changes that'll use the `@EmbeddedId` annotation to identify a composite key.

```
Book.java
1 package com.mytutorial.jpaa;
2
3 import javax.persistence.EmbeddedId;
4 import javax.persistence.Entity;
5
6 @Entity
7 public class Book {
8
9     @EmbeddedId
10    private BookKey bookKey;
11
12    private String title;
13    private String author;
14
15    public Book() {
16    }
17
18    public Book(String title, String author, float price) {
19        this.bookKey = new BookKey(title, price);
20        this.title = title;
21        this.author = author;
22    }
23
24    public BookKey getBookKey() {
25        return bookKey;
26    }
27
28    public void setBookKey(BookKey bookKey) {
29        this.bookKey = bookKey;
30    }
31
32    public String getTitle() {
33        return title;
34    }
35
36    public void setTitle(String title) {
37        this.title = title;
38    }
39
40    public String getAuthor() {
41        return author;
42    }
43
44    public void setAuthor(String author) {
45        this.author = author;
46    }
47
48 }
```

- Now this *Book* object, as you know, is an entity.
- Instead of using the `@Id` annotation for the member variable, which is the *Composite Key*, the *bookKey*, I'm going to change this annotation. I'm going to use `@EmbeddedId`. You can imagine that this `@EmbeddedId` annotation is the combination of the `@Embeddable` and the `@Id` annotation. It indicates that the *BookKey* class represents a composite key. And that composite key is embedded within this *Book* entity. And this is all the change that you need to make.

```
5 import java.io.Serializable;
6
7 public class BookKey implements Serializable {
8
9     private static final long serialVersionUID = 1L;
10
11     private Integer titleHash;
12     private Float price;
13
14     public BookKey() {
15     }
16
17     public BookKey(String title, Float price) {
18         this.titleHash = Objects.hash(title);
19         this.price = price;
20     }
21 }
```

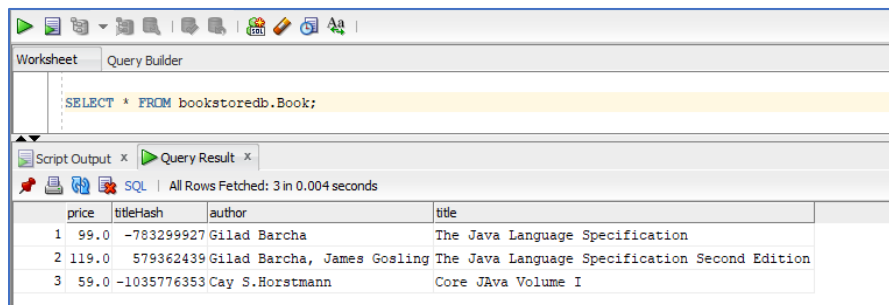
Java Persistence API: Configuring Fields & Performing CRUD Operations

Let's head over to **App.java** and run the same code that we had earlier, persisting three Book entities and two author entities into our relational database.

```
Hibernate:
    drop table if exists Author
Dec 03, 2021 10:18:00 AM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironment]
Hibernate:
    drop table if exists Book
Hibernate:
    create table Author (
      id integer not null auto_increment,
      birthDate datetime,
      name varchar(255),
      primary key (id)
    ) engine=MyISAM
Dec 03, 2021 10:18:00 AM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironment]
Hibernate:
    create table Book (
      price float not null,
      titleHash integer not null,
      author varchar(255),
      title varchar(255),
      primary key (price, titleHash)
    ) engine=MyISAM
```

We look at the console logs to see the structure of the Book table that is created with our new set of annotations. Observe that the Book table continues to have the columns **price**, **titleHash**, **author** and **title**. Thanks to our **@EmbeddedId** annotation the primary key includes the columns **price** as well as **titleHash**.

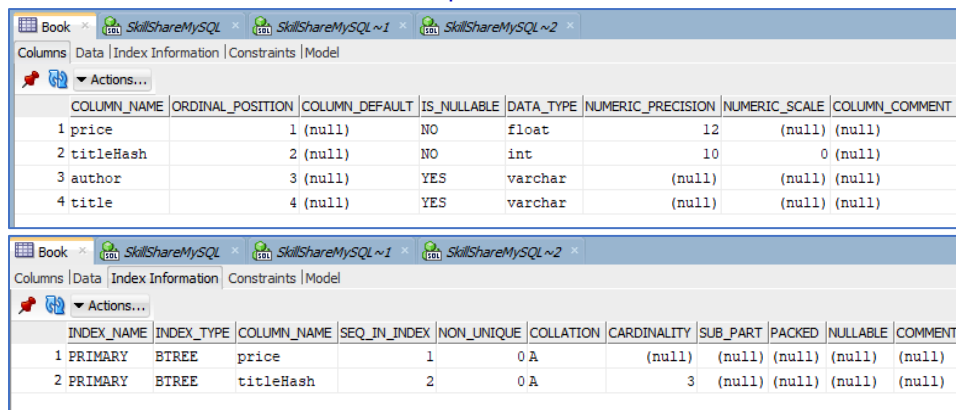
We can quickly verify all of this by heading over to the MySQL Workbench, and viewing the entries in the table. Let's run a `select * from 'bookstoredb'.Book`.



The screenshot shows the MySQL Workbench interface. The 'Query Builder' tab is active, displaying the query `SELECT * FROM bookstoredb.Book;`. Below the query, the 'Script Output' and 'Query Result' tabs are visible. The 'Query Result' tab shows the results of the query, with columns **price**, **titleHash**, **author**, and **title**. The results are as follows:

	price	titleHash	author	title
1	99.0	-783299927	Gilad Barcha	The Java Language Specification
2	119.0	579362439	Gilad Barcha, James Gosling	The Java Language Specification Second Edition
3	59.0	-1035776353	Cay S. Horstmann	Core Java Volume I

You can see that there are three entries here, **price** and **titleHash** are present, and these together form the *Composite Key* for this table. Let's run a describe on the **Book** table to confirm this. And you can see that this is indeed true. Both **price** and **titleHash** have been marked as primary keys.



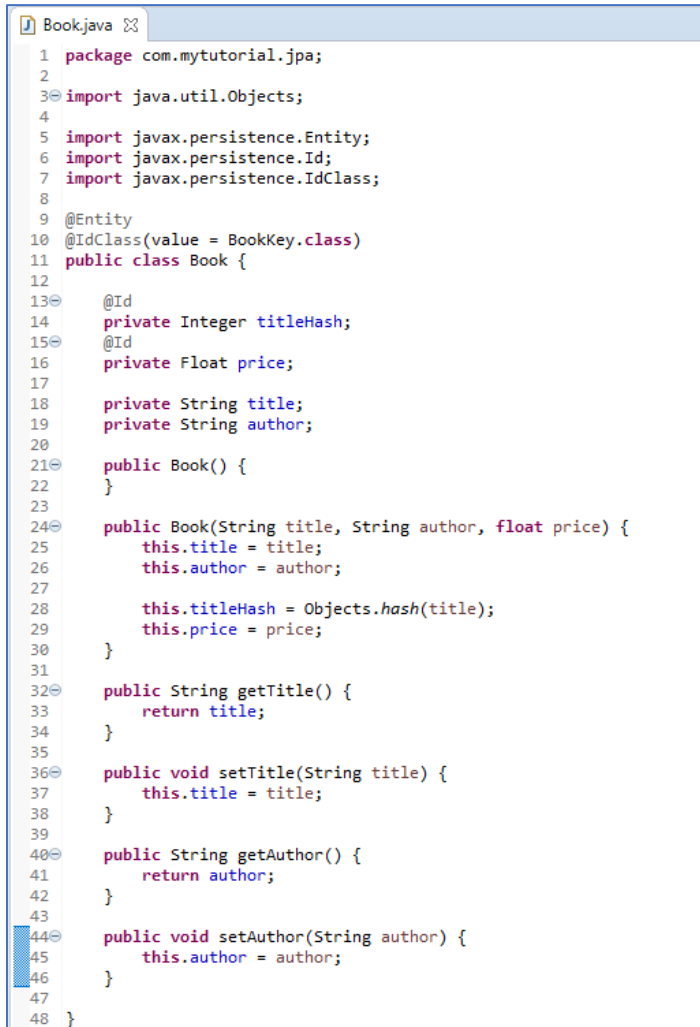
The screenshot shows the MySQL Workbench interface with the 'Book' table selected. The 'Data' tab is active, displaying the 'Describe' output for the table. The output is as follows:

COLUMN_NAME	ORDINAL_POSITION	COLUMN_DEFAULT	IS_NULLABLE	DATA_TYPE	NUMERIC_PRECISION	NUMERIC_SCALE	COLUMN_COMMENT
1 price	1	(null)	NO	float	12	(null)	(null)
2 titleHash	2	(null)	NO	int	10	0	(null)
3 author	3	(null)	YES	varchar	(null)	(null)	(null)
4 title	4	(null)	YES	varchar	(null)	(null)	(null)

INDEX_NAME	INDEX_TYPE	COLUMN_NAME	SEQ_IN_INDEX	NON_UNIQUE	COLLATION	CARDINALITY	SUB_PART	PACKED	NULLABLE	COMMENT
1 PRIMARY	BTREE	price	1	0	A	(null)	(null)	(null)	(null)	(null)
2 PRIMARY	BTREE	titleHash	2	0	A	3	(null)	(null)	(null)	(null)

Composite Keys Using IdClass

There is yet another technique that you can use in JPA to identify composite keys for your entity and that's exactly what we'll look at now. We won't use any of the previous annotations. Instead, we will identify composite keys using the **@IdClass** annotation. **javax.persistence.IdClass**. We have the Book entity, tagged to the **@Entity** annotation.



```

1 package com.mytutorial.jpa;
2
3 import java.util.Objects;
4
5 import javax.persistence.Entity;
6 import javax.persistence.Id;
7 import javax.persistence.IdClass;
8
9 @Entity
10 @IdClass(value = BookKey.class)
11 public class Book {
12
13     @Id
14     private Integer titleHash;
15     @Id
16     private Float price;
17
18     private String title;
19     private String author;
20
21     public Book() {
22     }
23
24     public Book(String title, String author, float price) {
25         this.title = title;
26         this.author = author;
27
28         this.titleHash = Objects.hash(title);
29         this.price = price;
30     }
31
32     public String getTitle() {
33         return title;
34     }
35
36     public void setTitle(String title) {
37         this.title = title;
38     }
39
40     public String getAuthor() {
41         return author;
42     }
43
44     public void setAuthor(String author) {
45         this.author = author;
46     }
47
48 }

```

- Observe that it has an additional, **@IdClass** annotation. And the value passed into IdClass is the *BookKey*. This annotation tells our JPA provider which happens to be hibernate, that the fields of the *BookKey* class will be the fields that make up the composite key for this Book entity.

In addition to specifying the **@IdClass** annotation, you need to individually tag these fields within the Book Entity with the **@Id** annotation. So in addition to the *titleHash* and *price* being part of the *BookKey* class, you need to have member variables for them within this Book class as well, which is our Entity.

- Observe that we have specified the `@Id` annotation for both `titleHash` as well as `price`. These fields are also present in the `BookKey` class. The other member variables and the constructors remain the same.

Book.java

```
9 @Entity
10 @IdClass(value = BookKey.class)
11 public class Book {
12
13     @Id
14     private Integer titleHash;
15     @Id
16     private Float price;
17
18     private String title;
19     private String author;
20
21     public Book() {
22     }
23
24     public Book(String title, String author, float price) {
25         this.title = title;
26         this.author = author;
27
28         this.titleHash = Objects.hash(title);
29         this.price = price;
30     }
31 }
```

BookKey.java

```
4 import java.util.Objects;
5
6 public class BookKey implements Serializable {
7
8     private static final long serialVersionUID = 1L;
9
10     private Integer titleHash;
11     private Float price;
12
13     public BookKey() {
14     }
15
16     public BookKey(String title, Float price) {
17         this.titleHash = Objects.hash(title);
18         this.price = price;
19     }
20 }
```

- We have getters and setters for all of the member variables of the `Book` entity. And this is all we need to do to set up the properties of the `BookKey` class as the composite key columns in our entity.

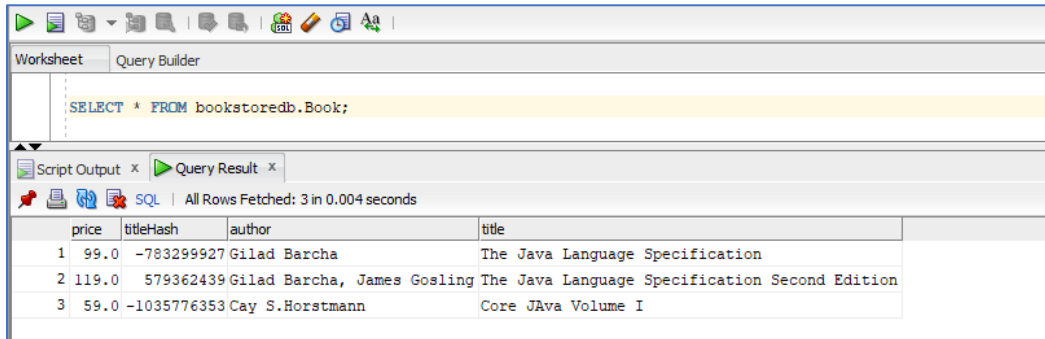
Let's head over to `App.java` and run this code. The code runs through successfully.

```
Dec 03, 2021 10:35:41 AM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTrans
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdb
Hibernate:
    create table Book (
      price float not null,
      titleHash integer not null,
      author varchar(255),
      title varchar(255),
      primary key (price, titleHash)
    ) engine=MyISAM
```

Let's take a look at the structure of the `Book` table that has been created. Notice that we have the columns `price`, `titleHash`, `author`, and `title` within the `Book` table. And the **primary key** is the *Composite Key*. Which includes the columns, `price` as well as `titleHash`.

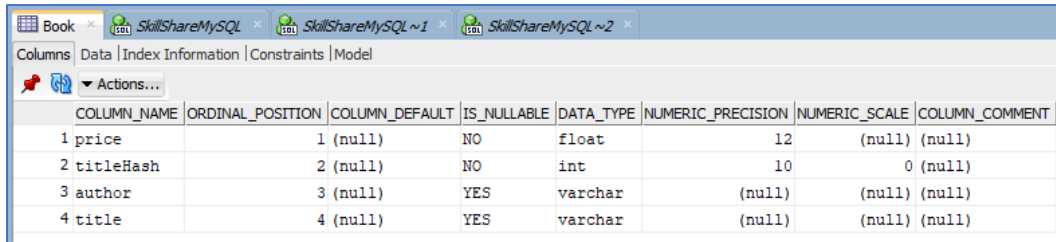
Java Persistence API: Configuring Fields & Performing CRUD Operations

Let's quickly verify all of this within our database using the MySQL Workbench. I'm going to run a select * on the Book table.

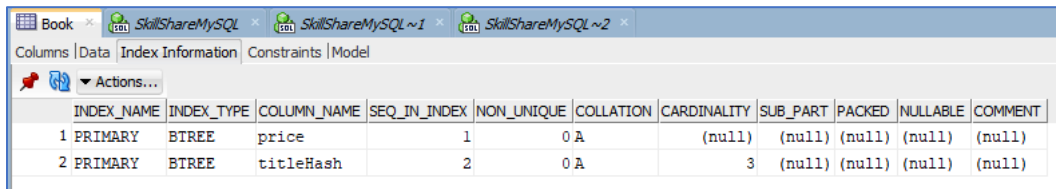


	price	titleHash	author	title
1	99.0	-783299927	Gilad Barcha	The Java Language Specification
2	119.0	579362439	Gilad Barcha, James Gosling	The Java Language Specification Second Edition
3	59.0	-1035776353	Cay S. Horstmann	Core JAVa Volume I

We have three entries here and every entry includes the `titleHash`. If you run a describe on the Book table, you'll see the structure of the individual columns. And you can see the `price` and `titleHash` together form a composite key.



	COLUMN_NAME	ORDINAL_POSITION	COLUMN_DEFAULT	IS_NULLABLE	DATA_TYPE	NUMERIC_PRECISION	NUMERIC_SCALE	COLUMN_COMMENT
1	price	1	(null)	NO	float	12	(null)	(null)
2	titleHash	2	(null)	NO	int	10	0	(null)
3	author	3	(null)	YES	varchar	(null)	(null)	(null)
4	title	4	(null)	YES	varchar	(null)	(null)	(null)



	INDEX_NAME	INDEX_TYPE	COLUMN_NAME	SEQ_IN_INDEX	NON_UNIQUE	COLLATION	CARDINALITY	SUB_PART	PACKED	NULLABLE	COMMENT
1	PRIMARY	BTREE	price	1	0	A	(null)	(null)	(null)	(null)	(null)
2	PRIMARY	BTREE	titleHash	2	0	A	3	(null)	(null)	(null)	(null)

The Column Annotation

So far we've seen how we can use JPA to model primary keys and composite keys for the entities that will persist in our database. In this demo, we'll explore some of the annotations that you can apply to other persistent fields that are not keys in our database. Here is our **persistence.xml**, our database action has been set to **drop-and-create** and this is what we'll work with unless we specify otherwise.

```
persistence.xml
1 <?xml version="1.0" encoding="UTF-8"?>
2 <persistence version="2.1"
3   xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
5
6   <persistence-unit name="BookstoreDB_Unit" >
7     <properties>
8       <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver" />
9       <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/bookstoredb" />
10      <property name="javax.persistence.jdbc.user" value="root" />
11      <property name="javax.persistence.jdbc.password" value="password" />
12
13      <property name="javax.persistence.schema-generation.database.action" value="drop-and-create"/>
14
15      <property name="hibernate.show_sql" value="true"/>
16      <property name="hibernate.format_sql" value="true"/>
17    </properties>
18  </persistence-unit>
19
20 </persistence>
```


Let's head over to our **Book.java** file which contains the Book entity.

```
Book.java
1 package com.mytutorial.jpaa;
2
3 import javax.persistence.Column;
4 import javax.persistence.Entity;
5 import javax.persistence.GeneratedValue;
6 import javax.persistence.GenerationType;
7 import javax.persistence.Id;
8
9 @Entity
10 public class Book {
11
12     @Id
13     @GeneratedValue(strategy = GenerationType.IDENTITY)
14     private Integer id;
15
16     @Column(name = "author_name", columnDefinition = "VARCHAR(55)")
17     private String author;
18
19     @Column(name = "book_title")
20     private String title;
21
22     private Float price;
23
24
25     public Book() {
26     }
27
28     public Book(String title, String author, float price) {
29         this.title = title;
30         this.author = author;
31         this.price = price;
32     }
33
34     public Integer getId() {
35         return id;
36     }
37
38     public void setId(Integer id) {
39         this.id = id;
40     }
41
42     public String getTitle() {
43         return title;
44     }
45
46     public void setTitle(String title) {
47         this.title = title;
48     }
49
50     public String getAuthor() {
51         return author;
52     }
53
54     public void setAuthor(String author) {
55         this.author = author;
56     }
57
58     public Float getPrice() {
59         return price;
60     }
61
62     public void setPrice(Float price) {
63         this.price = price;
64     }
65
66 }
```

And set up the import statements for the annotations that we'll use. Now all of the annotations here are familiar to us except for the column annotation, **javax.persistence.Column**. This is a specific annotation that you can apply to any field that corresponds to a column in your database table. The column annotation allows you to define in a very granular manner, how exactly you want to structure that particular column.

- This *Book* class is tag as an **@Entity** indicating that *Book* objects are entries in a database table named Book.

Next, I set up the member variables of this *Book* class, which correspond to columns in our database table. Every member variable can be thought of as a persistent field.

- The member variable `id` of type `Integer` is the primary key for this `Book` entity and it has the `@Id` annotation and the `@GeneratedValue` annotation with generation type `IDENTITY`.

I then have three other fields here corresponding to three columns `author`, `title` and `price`.

- The `author` as well as the `title` fields have the `@Column` annotation. And this `@Column` annotation allows us to define in a very granular manner how exactly we want those columns to look in our database table.

```
@Column(name = "author_name", columnDefinition = "VARCHAR(55)")
private String author;
```

Let's take a look at the `@Column` annotation on this line for the `author` field. By default, the database column has the `name`, `"author"`, which is the name of the member variable within this `Book` entity. The `name` property of the `@Column` annotation allows us to change the `name` of the database column.

Here I want the author's name column to be called `"author_name"`, and I specify this in the `name` property. I also want additional configuration of this column. By default, the string columns are `VARCHAR (255)`, but that's too long. I want the column to be of type `VARCHAR(55)` and I specify this in the `columnDefinition` property.

- Let's take a look at the `title` field, I have the `@Column` annotation here specifying that the name of the column should be `book_title`.

```
@Column(name = "book_title")
private String title;
```

- The `price` field does not have an `@Column` annotation and it will be set up with the defaults.

```
private Float price;
```

The name of the column will be `price` and it's of type floating point.

- I have the default no argument constructor and an additional constructor which takes in `title`, `author`, and `price`. There's not much change in the remaining code of the `Book` entity class.
- We'll set up getters and setters for each of our member variables.

Let's now move on to the next entity and that is **Author.java**.

```
Author.java
1 package com.mytutorial.jpaa;
2
3 import java.util.Date;
4
5 import javax.persistence.Column;
6 import javax.persistence.Entity;
7 import javax.persistence.GeneratedValue;
8 import javax.persistence.GenerationType;
9 import javax.persistence.Id;
10
11 @Entity
12 public class Author {
13
14     @Id
15     @GeneratedValue(strategy = GenerationType.IDENTITY)
16     private Integer id;
17
18     @Column(name = "author_name", columnDefinition = "VARCHAR(55)")
19     private String name;
20
21     @Column(name = "birth_date")
22     private Date birthDate;
23
24     public Author() {
25     }
26
27     public Author(String name, Date birthDate) {
28         this.name = name;
29         this.birthDate = birthDate;
30     }
31
32     public Integer getId() {
33         return id;
34     }
35
36     public void setId(Integer id) {
37         this.id = id;
38     }
39
40     public String getName() {
41         return name;
42     }
43
44     public void setName(String name) {
45         this.name = name;
46     }
47
48     public Date getBirthDate() {
49         return birthDate;
50     }
51
52     public void setBirthDate(Date birthDate) {
53         this.birthDate = birthDate;
54     }
55 }
56
```

- The `Author` class is tagged with the **@Entity** annotation indicating it corresponds to a table in our database.

Let's set up the member variables of this entity class.

- The `id` variable is the primary key for the `Author` and it has a generated value with strategy **IDENTITY**.
- We have the `name` field for the author and we've used an **@Column** definition to indicate that this field maps to the `author_name` column. Rather than the default **VARCHAR (255)**, we specify that the column definition of this column should be **VARCHAR (55)**.

Java Persistence API: Configuring Fields & Performing CRUD Operations

- Then we have the `birthDay` column, and we use the `@Column` annotation to indicate that this information should be stored in a column named `birth_date`.
- The rest of the author code remains the same with getters and setters for each of the member variables `id`, `name`, and `birthDate`.

Our `App.java` code remains the same. There's not much change here.

```
App.java
1 package com.mytutorial.jpa;
2
3 import java.util.GregorianCalendar;
4
5 import javax.persistence.EntityManager;
6 import javax.persistence.EntityManagerFactory;
7 import javax.persistence.Persistence;
8
9 /**
10  * Hello world!
11  *
12  */
13 public class App
14 {
15     public static void main( String[] args )
16     {
17
18         EntityManagerFactory factory = Persistence.createEntityManagerFactory("BookstoreDB_Unit");
19         EntityManager entityManager = factory.createEntityManager();
20
21         try {
22             entityManager.getTransaction().begin();
23
24             Book firstBook = new Book("The Java Language Specification",
25                                     "Gilad Barcha", 99f);
26             Book secondBook = new Book("The Java Language Specification Second Edition",
27                                     "Gilad Barcha, James Gosling", 119f);
28             Book thirdBook = new Book("Core Java Volume I", "Cay S.Horstmann", 59f);
29
30             entityManager.persist(firstBook);
31             entityManager.persist(secondBook);
32             entityManager.persist(thirdBook);
33
34             Author firstAuthor = new Author("Gilad Barcha", new GregorianCalendar(1980, 1, 0).getTime());
35             Author secondAuthor = new Author("James Gosling", new GregorianCalendar(1975, 2, 0).getTime());
36
37             entityManager.persist(firstAuthor);
38             entityManager.persist(secondAuthor);
39
40         } catch (Exception ex) {
41             System.err.println("An error occurred: " + ex);
42         } finally {
43             entityManager.getTransaction().commit();
44             entityManager.close();
45             factory.close();
46         }
47     }
48 }
49 }
```

We create three `Book` entities, `firstBook`, `secondBook` and `thirdBook`. And we use `entityManager.persist()` to persist each of these entities to the database. These entities correspond to records in the `Book` table. If you scroll down below, you'll see that we've instantiated two `Author` entities as before and use the `entityManager` to persist these entities. Make sure you enclose all of this in a **try catch finally block**. It's time for us to run this code and take a look at the results.

Java Persistence API: Configuring Fields & Performing CRUD Operations

Now the code runs through successfully. There are no errors in the console output.

```
Hibernate:
    create table Author (
      id integer not null auto_increment,
      birth_date datetime,
      author_name VARCHAR(55),
      primary key (id)
    ) engine=MyISAM
```

Let's take a look at the tables that were created. Here is the "Author" table, note that the columns have been defined based on the column definition that we had specified in the [Author](#) entity.

```
11 @Entity
12 public class Author {
13
14     @Id
15     @GeneratedValue(strategy = GenerationType.IDENTITY)
16     private Integer id;
17
18     @Column(name = "author_name", columnDefinition = "VARCHAR(55)")
19     private String name;
20
21     @Column(name = "birth_date")
22     private Date birthDate;
23
24     public Author() {
25     }
26
27     public Author(String name, Date birthDate) {
28         this.name = name;
29         this.birthDate = birthDate;
30     }
}
```

We have the [birthDay](#) column, which is called [birth_date](#) and the name of the author is in the column [author_name](#). The author name is of type **VARCHAR(55)**, which was part of our column definition.

Scroll down a little further and let's take a look at the [Book](#) table that was created.

```
Hibernate:
    create table Book (
      id integer not null auto_increment,
      author_name VARCHAR(55),
      price float,
      book_title varchar(255),
      primary key (id)
    ) engine=MyISAM
```

The column name for the Author field is [author_name](#), it is of type **VARCHAR (55)**. And the column name for the Book title is [book_title](#). We hadn't specified a column definition, so by default it's of **VARCHAR(255)**, that's the type.

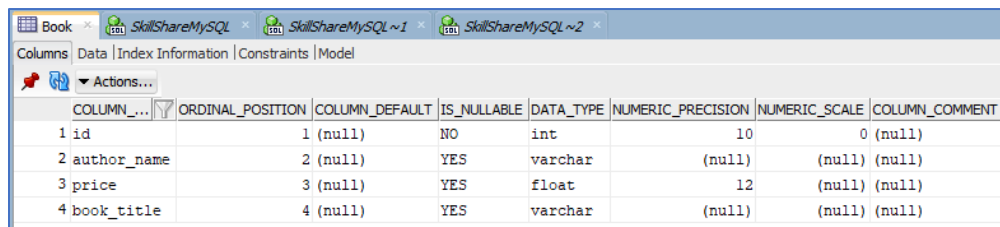
```
9 @Entity
10 public class Book {
11
12     @Id
13     @GeneratedValue(strategy = GenerationType.IDENTITY)
14     private Integer id;
15
16     @Column(name = "author_name", columnDefinition = "VARCHAR(55)")
17     private String author;
18
19     @Column(name = "book_title")
20     private String title;
21
22     private Float price;
23
24
25     public Book() {
26     }
27
28     public Book(String title, String author, float price) {
29         this.title = title;
30         this.author = author;
31         this.price = price;
32     }
}
```

Java Persistence API: Configuring Fields & Performing CRUD Operations

Note that the insert statements that hibernate executes to add our entities to the respective tables use the correct column names for *Book* as well as for the *Author* tables.

```
Hibernate:
insert
into
  Book
  (author_name, price, book_title)
values
  (?, ?, ?)
Hibernate:
insert
into
  Author
  (birth_date, author_name)
values
  (?, ?)
```

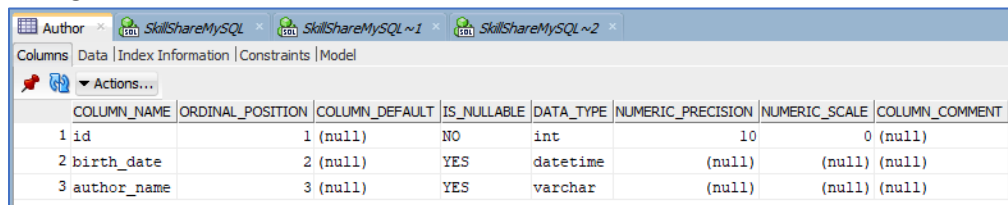
Time for us to verify that everything looks okay using the MySQL Workbench. I'm going to describe the *Book* table, run this command and you can see the column names that we had specified using the **@Column** definition, *author_name* and *book_title*.



The screenshot shows the MySQL Workbench interface with the 'Book' table selected. The 'Columns' tab is active, displaying a table with 8 columns: COLUMN_NAME, ORDINAL_POSITION, COLUMN_DEFAULT, IS_NULLABLE, DATA_TYPE, NUMERIC_PRECISION, NUMERIC_SCALE, and COLUMN_COMMENT. The data rows are as follows:

COLUMN_NAME	ORDINAL_POSITION	COLUMN_DEFAULT	IS_NULLABLE	DATA_TYPE	NUMERIC_PRECISION	NUMERIC_SCALE	COLUMN_COMMENT
1 id	1	(null)	NO	int	10	0	(null)
2 author_name	2	(null)	YES	varchar	(null)	(null)	(null)
3 price	3	(null)	YES	float	12	(null)	(null)
4 book_title	4	(null)	YES	varchar	(null)	(null)	(null)

Let's run a describe on the *Author* table. And once again, you'll find that the table has been created with the right column names based on our column definition.



The screenshot shows the MySQL Workbench interface with the 'Author' table selected. The 'Columns' tab is active, displaying a table with 8 columns: COLUMN_NAME, ORDINAL_POSITION, COLUMN_DEFAULT, IS_NULLABLE, DATA_TYPE, NUMERIC_PRECISION, NUMERIC_SCALE, and COLUMN_COMMENT. The data rows are as follows:

COLUMN_NAME	ORDINAL_POSITION	COLUMN_DEFAULT	IS_NULLABLE	DATA_TYPE	NUMERIC_PRECISION	NUMERIC_SCALE	COLUMN_COMMENT
1 id	1	(null)	NO	int	10	0	(null)
2 birth_date	2	(null)	YES	datetime	(null)	(null)	(null)
3 author_name	3	(null)	YES	varchar	(null)	(null)	(null)

Precision and Scale Specification

In this demo, we'll see how we can use JPA annotations to configure how we want to represent floating point numbers. We'll specify how these numbers should be represented using scale and precision.

I'm now going to add an **@Column** annotation to the price field of my **Book.java** entity.

```

9  @Entity
10 public class Book {
11
12     @Id
13     @GeneratedValue(strategy = GenerationType.IDENTITY)
14     private Integer id;
15
16     @Column(name = "author_name", columnDefinition = "VARCHAR(55)")
17     private String author;
18
19     @Column(name = "book_title")
20     private String title;
21
22     @Column(precision = 7, scale = 4)
23     private Float price;
24
25
26     public Book() {
27     }
28
29     public Book(String title, String author, float price) {
30         this.title = title;
31         this.author = author;
32         this.price = price;
33     }

```

This **@Column** annotation has two properties that I have configured, `precision = 7`, `scale = 4`.

- The precision property defines how many digits you'll use to represent your floating point numbers. I want to use a **maximum of 7 digits**, no more.
- In addition, I've also specified the **scale** property set to **4**. The scale property configures how many digits we want to represent in our floating point to the right of the decimal point. So precision is the total number of digits, scale is the number of **digits to the right of the decimal**.

So the price of a book can have a maximum of 7 digits with 4 digits to the right of the decimal point, that's what this column definition specifies.

Let's switch over to **Author.java** and configure our name column a little differently.

```

11 @Entity
12 public class Author {
13
14     @Id
15     @GeneratedValue(strategy = GenerationType.IDENTITY)
16     private Integer id;
17
18     @Column(name = "author_name", length = 44)
19     private String name;
20
21     @Column(name = "birth_date")
22     private Date birthDate;
23
24     public Author() {
25     }
26
27     public Author(String name, Date birthDate) {
28         this.name = name;
29         this.birthDate = birthDate;
30     }
31

```

Instead of using `columnDefinition = "VARCHAR(55)"`, I'm simply going to specify the `length = 44` of this column. That is the number of characters that we can store in this column, I've set it to be 44.

Java Persistence API: Configuring Fields & Performing CRUD Operations

The length property on the column annotation is a more intuitive way of specifying the largest possible length of a particular column in our database.

Let's switch over to **App.java**, and we'll run the same code as before where we persist three Book entities and two Author entities.

```
15 public static void main( String[] args )
16 {
17     EntityManagerFactory factory = Persistence.createEntityManagerFactory("BookstoreDB_Unit");
18     EntityManager entityManager = factory.createEntityManager();
19
20     try {
21         entityManager.getTransaction().begin();
22
23         Book firstBook = new Book("The Java Language Specification",
24             "Glad Barcha", 99.99999f);
25         Book secondBook = new Book("The Java Language Specification Second Edition",
26             "Glad Barcha, James Gosling", 119f);
27         Book thirdBook = new Book("Core Java Volume I", "Cay S. Horstmann", 59.9999f);
28
29         entityManager.persist(firstBook);
30         entityManager.persist(secondBook);
31         entityManager.persist(thirdBook);
32
33         Author firstAuthor = new Author("Glad Barcha", new GregorianCalendar(1980, 1, 0).getTime());
34         Author secondAuthor = new Author("James Gosling", new GregorianCalendar(1975, 2, 0).getTime());
35
36         entityManager.persist(firstAuthor);
37         entityManager.persist(secondAuthor);
38
39     } catch (Exception ex) {
40         System.err.println("An error occurred: " + ex);
41     } finally {
42         entityManager.getTransaction().commit();
43         entityManager.close();
44         factory.close();
45     }
46 }
```

But take a look at the prices that I've specified for my books.

The **firstBook**, *"The Java Language Specification"*, has a **price** of **\$99.99999**.

Note that this floating point number has a total of seven digits with five digits after the decimal point. Our precision was seven digits with a scale of four. We only allow four digits after the decimal point.

Turn your attention to the **thridBook** entity here, this is the core Java Volume I, the **price** is **\$59.9999** Cents. There are four digits after the decimal point, our scale was four, these four digits should be allowed. The precision was seven, which means the total number of digits in our floating point number could be no more than seven, here it is six.

The rest of the code here remains the same, we persist a total of five entities, three books and two Authors. Run this code and the code runs through with no errors.

```
Hibernate:
    create table Author (
        id integer not null auto_increment,
        birth_date datetime,
        author_name varchar(44),
        primary key (id)
    ) engine=MyISAM
Dec 03, 2021 2:13:15 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTrans
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jd
Hibernate:
    create table Book (
        id integer not null auto_increment,
        author_name VARCHAR(55),
        price float,
        book_title varchar(255),
        primary key (id)
    ) engine=MyISAM
```

Let's scroll down and take a look at the individual tables that were created. Here is the *Author* table, if you remember in this *Author* table we had specified the length of the string stored in the **author_name** to be 44. So you can see that the **author_name** has been defined as a **varchar(44)**.

Java Persistence API: Configuring Fields & Performing CRUD Operations

Let's look at the creation of the *Book* table. We have a precision and scale on the Price column. But that's not really visible here within the creation. But it will be visible when we see the records in our database. And let's do exactly that by heading over to the MySQL Workbench. We first run a describe command on the *Book* table in the 'bookstoredb'.

The image shows two screenshots from MySQL Workbench. The top screenshot displays the 'Columns' tab for the 'Book' table, showing a table with 8 columns: COLUMN_NAME, ORDINAL_POSITION, COLUMN_DEFAULT, IS_NULLABLE, DATA_TYPE, NUMERIC_PRECISION, NUMERIC_SCALE, and COLUMN_COMMENT. The bottom screenshot shows the 'Model' tab, displaying a visual representation of the 'bookstoredb.Book' table with columns: price (REAL), titleHash (INTEGER), author (VARCHAR2(255)), and title (VARCHAR2(255)). A primary key constraint is shown on the price and titleHash columns.

COLUMN_NAME	ORDINAL_POSITION	COLUMN_DEFAULT	IS_NULLABLE	DATA_TYPE	NUMERIC_PRECISION	NUMERIC_SCALE	COLUMN_COMMENT
1 id	1	(null)	NO	int	10	0 (null)	
2 author_name	2	(null)	YES	varchar	(null)	(null)	(null)
3 price	3	(null)	YES	float	12	(null)	(null)
4 book_title	4	(null)	YES	varchar	(null)	(null)	(null)

bookstoredb.Book

- P * price REAL
- P * titleHash INTEGER
- author VARCHAR2(255)
- title VARCHAR2(255)
- PRIMARY (price, titleHash)

You can see that the columns are *id*, *author_name*, *price*, and *book_title*. We have a scale and precision on the price but that's not really visible here when we describe the columns.

Let's now run a describe on the *Author* table within the bookstoredb.

The image shows two screenshots from MySQL Workbench. The top screenshot displays the 'Columns' tab for the 'Author' table, showing a table with 8 columns: COLUMN_NAME, ORDINAL_POSITION, COLUMN_DEFAULT, IS_NULLABLE, DATA_TYPE, NUMERIC_PRECISION, NUMERIC_SCALE, and COLUMN_COMMENT. The bottom screenshot shows the 'Model' tab, displaying a visual representation of the 'bookstoredb.Author' table with columns: id (INTEGER), birth_date (DATE), and author_name (VARCHAR2(44)). A primary key constraint is shown on the id column.

COLUMN_NAME	ORDINAL_POSITION	COLUMN_DEFAULT	IS_NULLABLE	DATA_TYPE	NUMERIC_PRECISION	NUMERIC_SCALE	COLUMN_COMMENT
1 id	1	(null)	NO	int	10	0 (null)	
2 birth_date	2	(null)	YES	datetime	(null)	(null)	(null)
3 author_name	3	(null)	YES	varchar	(null)	(null)	(null)

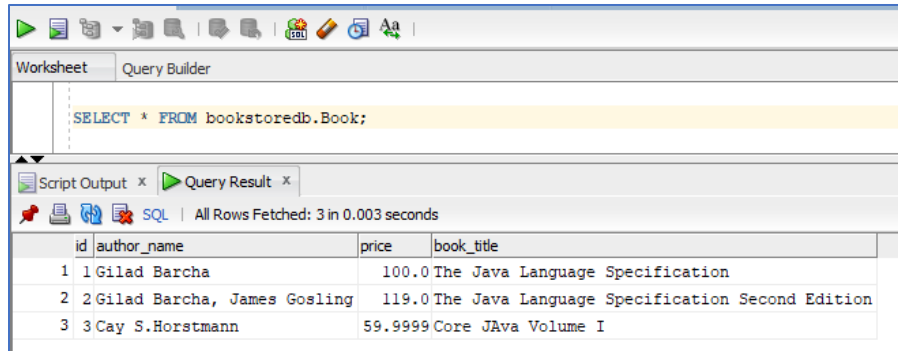
bookstoredb.Author

- P * id INTEGER
- birth_date DATE
- author_name VARCHAR2(44)
- PRIMARY (id)

There are three columns here in the result, *id*, *birth_date*, and *author_name*, and the *author_name* is of type **varchar(44)**.

Let's run a select * on the *Book* table to see how the prices represented in our database. Run this, and you'll immediately see how scale and precision come into play.

Java Persistence API: Configuring Fields & Performing CRUD Operations



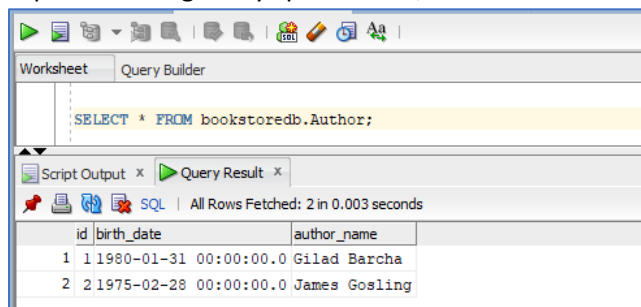
The screenshot shows a SQL query tool interface. The top bar contains icons for various functions. Below it, the 'Worksheet' tab is active, displaying the SQL query: `SELECT * FROM bookstoreb.Book;`. The 'Query Result' tab is also visible, showing the results of the query. The results are displayed in a table with columns: `id`, `author_name`, `price`, and `book_title`. The table contains three rows of data.

	id	author_name	price	book_title
1	1	Gilad Barcha	100.0	The Java Language Specification
2	2	Gilad Barcha, James Gosling	119.0	The Java Language Specification Second Edition
3	3	Cay S. Horstmann	59.9999	Core JAVa Volume I

The `firstBook`, “The Java Language Specification”, which had a price of \$99, and then five 9's after the decimal point (**\$99.9999**) has been rounded up to **\$100**. Because the price specification did not fit within scale 4 and precision 7, a rounding operation was performed on the price of this book.

Let's take a look at the `thirdBook` with a price of **\$59.9999**. And this is within the precision and scale defined for this column, and that is why this floating point price is represented as is.

Just for completeness, you can run a `select *` on the `Author` table, and the entries here should be as you expect. Nothing really special here, we've seen these entries before.



The screenshot shows a SQL query tool interface. The top bar contains icons for various functions. Below it, the 'Worksheet' tab is active, displaying the SQL query: `SELECT * FROM bookstoreb.Author;`. The 'Query Result' tab is also visible, showing the results of the query. The results are displayed in a table with columns: `id`, `birth_date`, and `author_name`. The table contains two rows of data.

	id	birth_date	author_name
1	1	1980-01-31 00:00:00.0	Gilad Barcha
2	2	1975-02-28 00:00:00.0	James Gosling

Not Null and Uniqueness Constraints

The **@Column** annotation can be used to specify additional constraints on our column values. For example, if you want to make a column **non-nullable**, that is it cannot accept null values, or you want a column to have a **unique** value.

Let's see how we can do this. **Book.java**

```

9  @Entity
10 public class Book {
11
12     @Id
13     @GeneratedValue(strategy = GenerationType.IDENTITY)
14     private Integer id;
15
16     @Column(nullable = false)
17     private String author;
18
19     @Column(nullable = false, unique = true, length = 55)
20     private String title;
21
22     @Column(precision = 7, scale = 4)
23     private Float price;
24
25
26     public Book() {
27     }
28
29     public Book(String title, String author, float price) {
30         this.title = title;
31         this.author = author;
32         this.price = price;
33     }
34 }

```

- I'm changing the **@Column** annotation on the **author** field of the book entity to set **nullable** equal to **false**. The name of the column will be **author** but we *cannot* have null values for author, so nullable equal to false.
- Let's specify another annotation on the **title** column. I want this to also have **nullable** equal to **false**, that is it cannot accept null values but in addition I want this column to be **unique**, so you can't have two books with the same title. The unique property in the **@Column** annotation I have set to true. In addition, I've specified **length** is equal to 55 indicating that this *varchar* column should have length 55.

Now let's head over to **Author.java**.

```

11 @Entity
12 public class Author {
13
14     @Id
15     @GeneratedValue(strategy = GenerationType.IDENTITY)
16     private Integer id;
17
18     @Column(nullable = false, unique = true, length = 55)
19     private String name;
20
21     @Column(name = "birth_date")
22     private Date birthDate;
23
24     public Author() {
25     }
26
27     public Author(String name, Date birthDate) {
28         this.name = name;
29         this.birthDate = birthDate;
30     }
31 }

```

Let's redefine our columns once again,

- I'm going to change the definition of the **author** name column to set **nullable** equal to **false**. I want **author** names to be **unique**, so unique is set to true, and the **length** of this column is 55. So the author name will be varchar 55.

The App.java code will remain the same, we'll persist three book entities and two author entities. Run this code and it runs through successfully.

```
Hibernate:
    create table Author (
      id integer not null auto_increment,
      birth_date datetime,
      name varchar(55) not null,
      primary key (id)
    ) engine=MyISAM
Dec 03, 2021 2:33:07 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIs
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.i
Hibernate:
    create table Book (
      id integer not null auto_increment,
      author varchar(255) not null,
      price float,
      title varchar(55) not null,
      primary key (id)
    ) engine=MyISAM
Hibernate:
    alter table Author
      add constraint UK_soak1hrrvg12k041wmdawe6t5 unique (name)
Hibernate:
    alter table Book
      add constraint UK_odppys651q7q1xbx8o6p6fgxj unique (title)
```

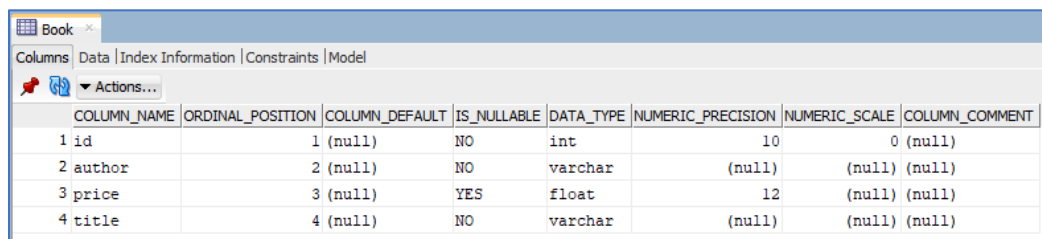
Let's take a look at some of the tables that we've created and look at the column definitions.

- Notice that for the *Author* table, the **name** column is **varchar(55)**, length we had specified as 55, and it has the **not null** constraint. This is the constraint that is applied when we set nullable equal to false.
- If you scroll down a little bit, you'll see the creation of the *Book* table. Notice that the **title** column here has **varchar(55)**, we had specified length is equal to 55 and it has the **not null** constraint. We had set nullable equal to false.

Now we had two uniqueness constraint and these are specified separately using an alter table command.

- The first alter table command alters the *Author* table and adds an additional **uniqueness** constraint on the **name** column.
- And the second alter table command alters the *Book* table and adds an additional **uniqueness** constraint on the title column.

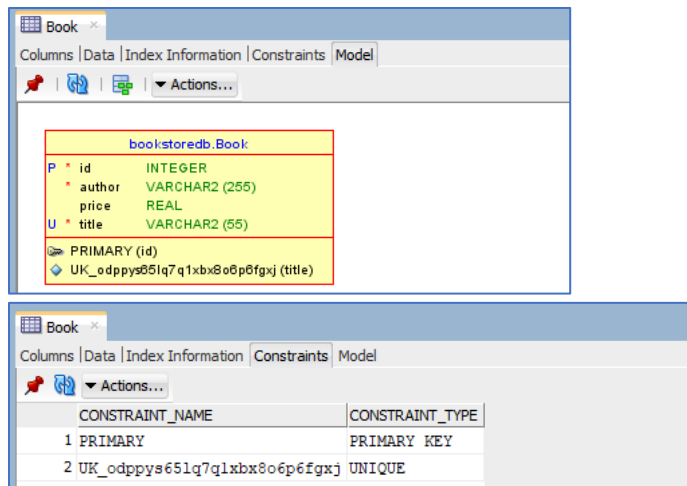
Let's head over to MYSQL Workbench and see the results of running these queries. We'll describe the *Book* table within the bookstoredb,



COLUMN_NAME	ORDINAL_POSITION	COLUMN_DEFAULT	IS_NULLABLE	DATA_TYPE	NUMERIC_PRECISION	NUMERIC_SCALE	COLUMN_COMMENT
id	1	(null)	NO	int	10	0	(null)
author	2	(null)	NO	varchar	(null)	(null)	(null)
price	3	(null)	YES	float	12	(null)	(null)
title	4	(null)	NO	varchar	(null)	(null)	(null)

and you can see that the **author** name cannot be null, null values are not allowed, and the **title** field also cannot be null.

Java Persistence API: Configuring Fields & Performing CRUD Operations



The **title** field also has an additional **uniqueness** constraint, which we had specified using the **@Column** annotation.

Let's describe the *Author* table and take a look.

The screenshot shows the 'Author' table model. The columns are:

- id**: INTEGER, PRIMARY KEY
- birth_date**: DATE
- name**: VARCHAR2 (55), UNIQUE

The constraints are listed below the columns:

- PRIMARY (id)
- UK_soak1hrrvg12k041wmdawe6t5 (name)

The screenshot shows the 'Author' table model. The columns are:

- id**: INTEGER, PRIMARY KEY
- birth_date**: DATE
- name**: VARCHAR2 (55), UNIQUE

The constraints are listed below the columns:

- PRIMARY (id)
- UK_soak1hrrvg12k041wmdawe6t5 (name)

The screenshot shows the 'Author' table model. The columns are:

- id**: INTEGER, PRIMARY KEY
- birth_date**: DATE
- name**: VARCHAR2 (55), UNIQUE

The constraints are listed below the columns:

- PRIMARY (id)
- UK_soak1hrrvg12k041wmdawe6t5 (name)

Once you run this code, you'll see that the constraints that we specified apply to this table as well. Take a look at the **name** column, it's **varchar(55)**, it cannot have null values, null column says NO. And there is an additional constraint that the author **name** should be **unique**.

Non-persistable Fields With Transient

The main objective in our using the Java Persistence API, and a JPA provider such as Hibernate is to abstract us away from dealing with the database directly. We can live in our object-oriented programming world. And this world will map to an underlying database. All of that is abstracted away from us. But it's quite possible that your objects that you deal with may have member variables and fields that you don't want persisted in your database.

You should be able to control which field maps to a column in your underlying table and which field doesn't within your entity. And that's exactly where the transient annotation comes in.

Book.java

```

1 package com.mytutorial.jpa;
2
3 import javax.persistence.Column;
4 import javax.persistence.Entity;
5 import javax.persistence.GeneratedValue;
6 import javax.persistence.GenerationType;
7 import javax.persistence.Id;
8 import javax.persistence.Transient;
9
10 @Entity
11 public class Book {
12
13     @Id
14     @GeneratedValue(strategy = GenerationType.IDENTITY)
15     private Integer id;
16
17     @Column(nullable = false)
18     private String author;
19
20     @Column(nullable = false, unique = true, length = 55)
21     private String title;
22
23     @Column(precision = 7, scale = 4)
24     private Float price;
25
26     @Transient
27     private boolean inStock;
28
29     public Book() {
30     }
31
32     public Book(String title, String author, float price) {
33         this.title = title;
34         this.author = author;
35         this.price = price;
36         this.inStock = true;
37     }
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64 public Float getPrice() {}
65
66
67
68 public void setPrice(Float price) {}
69
70
71
72 public boolean isInStock() {
73     return inStock;
74 }
75
76 public void setInStock(boolean inStock) {
77     this.inStock = inStock;
78 }
79
80 }

```

Observe we have an import for **javax.persistence.Transient**. We'll mark variables in our entity object using this annotation to indicate they're not to be persisted in the database. We've seen that by default, all the member variables that we defined within our entity object map to columns in an underlying database table.

Take a look at this example here,

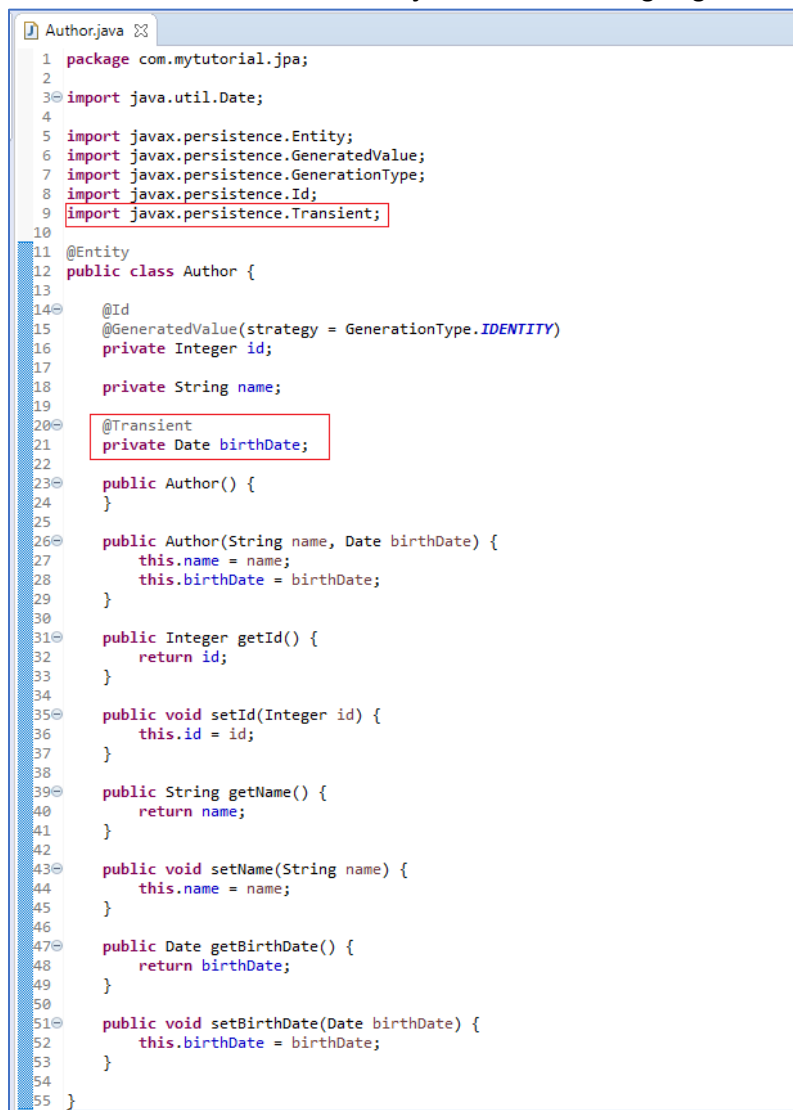
```
@Transient
private boolean inStock;
```

We have an additional member variable here called `inStock`. In the object-oriented world where we write our code, we want this variable to be populated with a value, maybe from some other source, not our underlying database.

Now, if we just declare this member variable within our book entity class, this will translate to a column in the underlying table. To prevent this, we annotate `inStock` using the **@Transient** annotation.

@Transient basically, tells the underlying Hibernate provider that *this is not a column in the Book table*. Instead, this will be populated using other means so don't do anything about it. The rest of the code in this book entity class is familiar to us. We have getters and setters for all of our member variables including our Transient variable.

I'll now switch over to the **Author.java** file where I'm going to annotate a field as transient.



```
1 package com.mytutorial.jpa;
2
3 import java.util.Date;
4
5 import javax.persistence.Entity;
6 import javax.persistence.GeneratedValue;
7 import javax.persistence.GenerationType;
8 import javax.persistence.Id;
9 import javax.persistence.Transient;
10
11 @Entity
12 public class Author {
13
14     @Id
15     @GeneratedValue(strategy = GenerationType.IDENTITY)
16     private Integer id;
17
18     private String name;
19
20     @Transient
21     private Date birthDate;
22
23     public Author() {
24     }
25
26     public Author(String name, Date birthDate) {
27         this.name = name;
28         this.birthDate = birthDate;
29     }
30
31     public Integer getId() {
32         return id;
33     }
34
35     public void setId(Integer id) {
36         this.id = id;
37     }
38
39     public String getName() {
40         return name;
41     }
42
43     public void setName(String name) {
44         this.name = name;
45     }
46
47     public Date getBirthDate() {
48         return birthDate;
49     }
50
51     public void setBirthDate(Date birthDate) {
52         this.birthDate = birthDate;
53     }
54
55 }
```

Java Persistence API: Configuring Fields & Performing CRUD Operations

I'm removing these additional constraints that I have on the `name` column. I'll just get rid of the `@Column` annotation. I'm now going to change the `@Column` annotation for the `birthDate` field and mark it as `@Transient`. This means that the `birthDate` will not be stored in the underlying `Author` table. It may be populated by other means, but it will not map to a column.

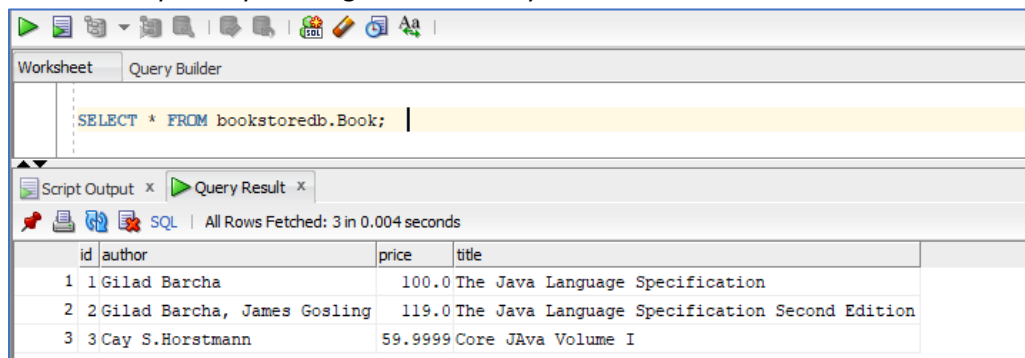
Let's run this code and make sure that it works as we would expect it to. Here is our `App.java`. No change here we persist three book entities and two author entities to the underlying database. Hit run, either using a shortcut or using the UI. And you can see that everything has run through without error.

```
Hibernate:
    create table Author (
      id integer not null auto_increment,
      name varchar(255),
      primary key (id)
    ) engine=MyISAM
Dec 03, 2021 3:07:17 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIso
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.in
Hibernate:
    create table Book (
      id integer not null auto_increment,
      author varchar(255) not null,
      price float,
      title varchar(55) not null,
      primary key (id)
    ) engine=MyISAM
Hibernate:
    alter table Book
      add constraint UK_odppys65lq7q1xbx8o6p6fgxj unique (title)
```

Take a look at the `Author` table here. We had marked the `birthDate` column as `Transient` in the `Author` table. And notice that in this create table SQL command, the `birthDate` column is not present. We have an `id` column, a `name` column but no `birthDate`.

Let's scroll down below and take a look at the `Book` table that we have created. The `inStock` member variable was tagged as `Transient` and there is no column corresponding to that member variable from our book entity class. So we have `id`, `author`, `price`, and `title`, but no `inStock`.

We can verify this by heading over to the MySQL Workbench. Let's run a select star on the `Book` table.

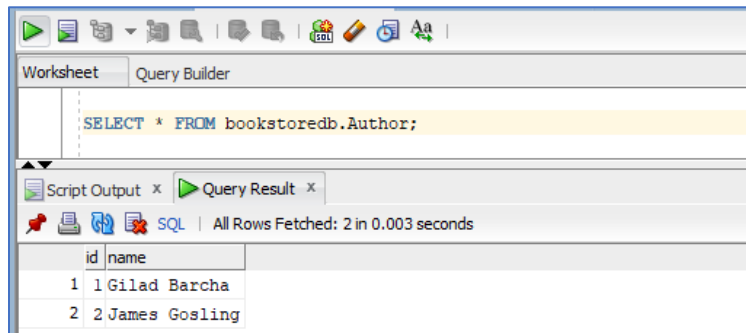


The screenshot shows the MySQL Workbench interface. The 'Query Builder' tab is active, displaying the SQL query: `SELECT * FROM bookstoredb.Book;`. Below the query, the 'Query Result' tab shows the results of the query. The results are displayed in a table with 4 columns: `id`, `author`, `price`, and `title`. There are 3 rows of data.

	id	author	price	title
1	1	Gilad Barcha	100.0	The Java Language Specification
2	2	Gilad Barcha, James Gosling	119.0	The Java Language Specification Second Edition
3	3	Cay S. Horstmann	59.9999	Core JAVa Volume I

You see that there is no column corresponding to `inStock`.

Let's run a select star on the *Author* table and when we execute this command, you will find that the *Author* table has just two columns *id* and *name*.



There is no column for the *birthDate* variable which we had marked as Transient.

Temporal Annotation for Date Fields

When you're working with **DateTime** fields in your entities, it's quite likely that you want more control over how exactly that **DateTime** field is stored in the database. This is exactly what you get using the **@Temporal** annotation.

```

3 import java.util.Date;
4
5 import javax.persistence.Entity;
6 import javax.persistence.GeneratedValue;
7 import javax.persistence.GenerationType;
8 import javax.persistence.Id;
9 import javax.persistence.Temporal;
10 import javax.persistence.TemporalType;
11
12 @Entity
13 public class Author {
14
15     @Id
16     @GeneratedValue(strategy = GenerationType.IDENTITY)
17     private Integer id;
18
19     private String name;
20
21     @Temporal(TemporalType.DATE)
22     private Date birthDate;
23
24     public Author() {
25     }
26
27     public Author(String name, Date birthDate) {
28         this.name = name;
29         this.birthDate = birthDate;
30     }
31

```

And I'm going to add in two new import statements that will allow us to deal with **DateTime** member variables and how they're represented in our database, **javax.persistence.Temporal** and **javax.persistence.TemporalType**. These are the two new imports.

You must have observed that by default when you have a member variable of Type **DATE** within your entity, the column is of data type **DateTime**. So the **Date** as well as the **Time** is represented within that column.

But for the **birthDate** field, it's kind of strange to have the time associated with the date of birth as well. Let's change how we represent this date field within our database by using the **@Temporal** annotation, and the **TemporalType** that I have specified here is **DATE**.

The **@Temporal** annotation is to be used with timestamp related fields, and *Date* is an example of such a field. The **TemporalType.DATE** is an indication to our hibernate implementation that we want the underlying column to be of Type **DATE**, the *Time* is not really relevant. So this allows us to control this date column.

Now let's switch over to **App.java**, and we'll run the same code as before where we persist three *Book* entities, and two *Author* entities. Our application code has run through fine, let's see how the **Temporal** annotation affects the *Author* table.

```

Hibernate:
    create table Author (
      id integer not null auto_increment,
      birthDate date,
      name varchar(255),
      primary key (id)
    ) engine=MyISAM

```

Notice that the type of the `birthDate` column has changed. It's no longer the **datetime** default type. But instead it is simply the **date** type. This will store only date information, and not time information.

And we can confirm this by running a select * on the Author table.

id	birthDate	name
1	1980-01-31	Gilad Barcha
2	1975-02-28	James Gosling

If you look at the `birthDate` column for the two records that we've inserted, you can see that we only store the date of birth. The associated timestamp has not been stored. **TemporalType.DATE** means store just the date, and not the time. That's exactly what we want for the `birthDate` column.

Let's now try another enum value for the `@Temporal` annotation. The `birthDate` field continues to have the `@Temporal` annotation. But the **TemporalType** that I've specified here is the **TIME**.

```

3 import java.util.Date;
4
5 import javax.persistence.Entity;
6 import javax.persistence.GeneratedValue;
7 import javax.persistence.GenerationType;
8 import javax.persistence.Id;
9 import javax.persistence.Temporal;
10 import javax.persistence.TemporalType;
11
12 @Entity
13 public class Author {
14
15     @Id
16     @GeneratedValue(strategy = GenerationType.IDENTITY)
17     private Integer id;
18
19     private String name;
20
21     @Temporal(TemporalType.TIME)
22     private Date birthDate;
23
24     public Author() {
25     }
26
27     public Author(String name, Date birthDate) {
28         this.name = name;
29         this.birthDate = birthDate;
30     }
31

```

This will cause hibernate to map this column as a **time** column, the **date** will not be stored, instead the **time** will be stored. Now you have to realize that this annotation doesn't really make much sense for the `birthDate` column. But it serves for the purposes of our demo, we'll see how **TemporalType.TIME** works. Let's switch over to **App.java**, we run the same code as before. And when you run this application, everything seems to work fine.

```

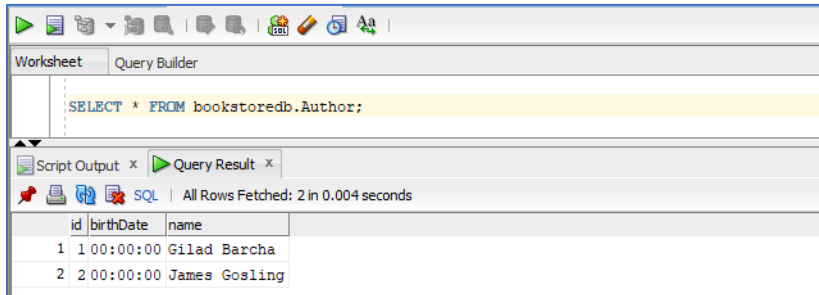
Hibernate:
    create table Author (
        id integer not null auto_increment,
        birthDate time,
        name varchar(255),
        primary key (id)
    ) engine=MyISAM

```

Java Persistence API: Configuring Fields & Performing CRUD Operations

Let's take a look at the creation of the *Author* table. Notice that the `birthDate` column now has a different data type, it's of type **time**, so only **timestamp** information will be stored.

We can confirm this by running a `select *` from the *Author* table. And in the resulting records, you can see that the **timestamp** information is present in the database. The **date** information has not been stored.



id	birthDate	name
1	1 00:00:00	Gilad Barcha
2	2 00:00:00	James Gosling

A **date** field without the Temporal annotation saves the date in the **datetime** format.

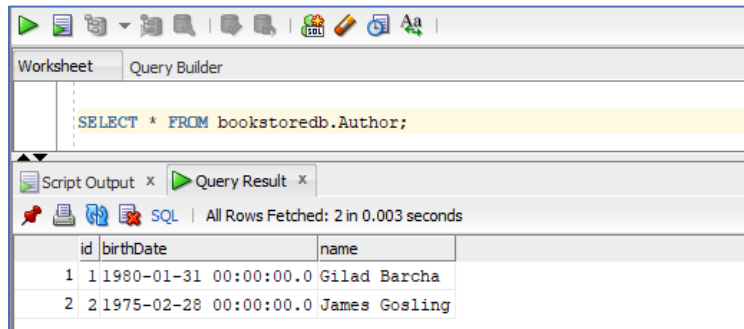
And if you want to explicitly specify this, you can use the **@Temporal** annotation with **TemporalType.TIMESTAMP**.

```
3 import java.util.Date;
4
5 import javax.persistence.Entity;
6 import javax.persistence.GeneratedValue;
7 import javax.persistence.GenerationType;
8 import javax.persistence.Id;
9 import javax.persistence.Temporal;
10 import javax.persistence.TemporalType;
11
12 @Entity
13 public class Author {
14
15     @Id
16     @GeneratedValue(strategy = GenerationType.IDENTITY)
17     private Integer id;
18
19     private String name;
20
21     @Temporal(TemporalType.TIMESTAMP)
22     private Date birthDate;
23
24     public Author() {
25     }
26
27     public Author(String name, Date birthDate) {
28         this.name = name;
29         this.birthDate = birthDate;
30     }
31 }
```

This is how we've been saving the `birthDate` information so far, we include the **date** as well as the **time** of birth. Let's switch over to **App.java**, run our code as before, and take a look at the creation of the *Author* table. You can see that the `birthDate` column has type **datetime**.

```
Hibernate:
create table Author (
  id integer not null auto_increment,
  birthDate datetime,
  name varchar(255),
  primary key (id)
) engine=MyISAM
```

datetime means store the **date** of birth as well as the **time** of birth, which by default is zero. When we haven't specified a time, let's run a select * from the *Author* table.



The screenshot shows a database query tool interface. At the top, there's a toolbar with various icons. Below it, a tab labeled 'Query Builder' is active, displaying the SQL query: `SELECT * FROM bookstoredb.Author;`. Below the query editor, there's a section for 'Script Output' and 'Query Result'. The 'Query Result' tab is selected, showing the results of the query. The results are displayed in a table with three columns: 'id', 'birthDate', and 'name'. There are two rows of data.

	id	birthDate	name
1	1	1980-01-31 00:00:00.0	Gilad Barcha
2	2	1975-02-28 00:00:00.0	James Gosling

And in the result, you can see that the date information as well as the time information is present in the `birthDate` column.

Lob Annotation for Large Objects

Within your relational database table, it's possible to have columns that store very large values. These large values can be large objects of the character type, or large objects of the binary type. And both of these can be modeled using JPA annotations.

Here we are in our **Author.java** file.

```

1 package com.mytutorial.jpa;
2
3 import java.util.Date;
4
5 import javax.persistence.Basic;
6 import javax.persistence.Entity;
7 import javax.persistence.FetchType;
8 import javax.persistence.GeneratedValue;
9 import javax.persistence.GenerationType;
10 import javax.persistence.Id;
11 import javax.persistence.Lob;
12 import javax.persistence.Temporal;
13 import javax.persistence.TemporalType;
14
15 @Entity
16 public class Author {
17
18     @Id
19     @GeneratedValue(strategy = GenerationType.IDENTITY)
20     private Integer id;
21
22     private String name;
23
24     @Basic(fetch = FetchType.LAZY)
25     @Lob
26     private String bio;
27
28     @Temporal(TemporalType.DATE)
29     private Date birthDate;
30
31     @Basic(fetch = FetchType.LAZY)
32     @Lob
33     private byte[] image;
34
35     public Author() {
36     }
37
38     public Author(String name, Date birthDate) {
39         this.name = name;
40         this.birthDate = birthDate;
41     }
42
43     public Integer getId() {}
44     public void setId(Integer id) {}
45     public String getName() {}
46     public void setName(String name) {}
47     public String getBio() {
48         return bio;
49     }
50     public void setBio(String bio) {
51         this.bio = bio;
52     }
53
54     public Date getBirthDate() {}
55     public void setBirthDate(Date birthDate) {}
56     public byte[] getImage() {
57         return image;
58     }
59     public void setImage(byte[] image) {
60         this.image = image;
61     }
62 }

```

- Two import statements here in order to model large objects stored within the columns of this *Author* table.

The first import **javax.persistence.Basic** statement is for the **@Basic** annotation.

And the second one **javax.persistence.Lob** is for the large object **@Lob** annotation.

- The **@Basic** annotation is a very simple annotation typically used with primitive data types such as Strings and Integers. When you don't specify an annotation for a particular member variable in your entity class, the **@Basic** annotation is implicitly assumed by JPA and Hibernate.

The **@Basic** annotation is just a way to tell our JPA provider to map that field to a basic type, such as **floats**, **doubles**, **integers** and so on. This is why we don't really focus on the basic annotation.

However, the **@Basic** annotation has some other properties, which we use when we use the **@Basic** annotation along with large objects and that's what we'll study here.

```
@Basic(fetch = FetchType.LAZY)
@Lob
private String bio;
```

Within the *Author* class I've defined a member variable called **bio** which is of type String. This holds the biography of the *author*. Now if the *author* has been very prolific, it's quite possible that the biography is really *long*. So I have tagged this with the **@Lob** annotation. Because this field is of the string data type this **@Lob** annotation indicates to Hibernate that this is a **Character Large Object** or a **CLOB**.

In addition to the Lob annotation, I have an additional **@Basic** annotation here and I have specified an input argument **fetch**. And here I've specified that FetchType is **LAZY**. What does this mean? This basically tells Hibernate that when you're fetching all of the records from the *Author* table, *do not fetch this bio column unless we explicitly access this field*.

FetchType. **LAZY** means that the contents of the **bio** column will be *lazily loaded* into our program only when we explicitly use the getter or some other means to access this bio field.

When you define columns using the **@Lob** annotation this is a best practice that you should follow because you don't want to be loading in large amounts of data unless the data is absolutely required.

Now, we won't really see how exactly this lazy loading works. In this demo we'll see it in a later demo in this learning path. But you should understand that it's good practice to tag your large objects using FetchType. **LAZY** so they're not loaded in unless they're absolutely required.

- Before we move on and look at another example of the Lob annotation. Let me just clean up this **birthDate** field so that it has the **@Temporal** annotation with TemporalType. **DATE** we'll only

store the **Date** information of the author and not the time information associated with the `birthDate`.

- I'm now going to add in another field which holds an `image` for this particular author. Now, you may want to store this `image` information in the binary format in your database. I use the `@Lob` annotation here. This is a **Binary Large Object** or a **BLOB**.

Once again, I use the `@Basic` annotation and specify that the `FetchType` is `LAZY` the `image` won't be loaded into our program unless he explicitly access the `image` field.

```
@Basic(fetch = FetchType.LAZY)
@Lob
private byte[] image;
```

Let's see how we can work with these fields.

- Make sure you set up the *Getters* and *Setters* for the `bio` field as well as the `image` fields so we can assign values to these fields.

I'll head over to `App.java`,

```
18 EntityManagerFactory factory = Persistence.createEntityManagerFactory("BookstoreDB_Unit");
19 EntityManager entityManager = factory.createEntityManager();
20
21 try {
22     entityManager.getTransaction().begin();
23
24     Book firstBook = new Book("The Java Language Specification",
25         "Gilad Barcha", 99.99999f);
26     Book secondBook = new Book("The Java Language Specification Second Edition",
27         "Gilad Barcha, James Gosling", 119f);
28     Book thirdBook = new Book("Core Java Volume I", "Cay S.Horstmann", 59.9999f);
29
30     entityManager.persist(firstBook);
31     entityManager.persist(secondBook);
32     entityManager.persist(thirdBook);
33
34     Author firstAuthor = new Author("Gilad Barcha", new GregorianCalendar(1980, 1, 0).getTime());
35     firstAuthor.setBio("Some very long personal bio here");
36     firstAuthor.setImage("Pretend this is an image".getBytes());
37     Author secondAuthor = new Author("James Gosling", new GregorianCalendar(1975, 2, 0).getTime());
38     secondAuthor.setImage("Pretend this is also an image".getBytes());
39
40     entityManager.persist(firstAuthor);
41     entityManager.persist(secondAuthor);
42
43 } catch (Exception ex) {
44     System.err.println("An error occurred: " + ex);
45 } finally {
46     entityManager.getTransaction().commit();
47     entityManager.close();
48     factory.close();
49 }
```

where we'll persist three *Book* entities and two *Author* entities. No change in the *Book* entities. However, we've updated the *Author* entities to have the `firstAuthor` have both a `bio` as well as an `image`, a fake image that is, I call `firstAuthor.setBio()`. And the `bio` here is really brief, but imagine that it's really a long description.

I call `firstAuthor.setImage()` and assign some `bytes` to represent `image` data. We pretend that this is an `image`, it isn't really. In exactly the same way I have assigned a fake image to the `secondAuthor` as well.

Java Persistence API: Configuring Fields & Performing CRUD Operations

This will serve for the purposes of our demo. Make sure you persist both the `firstAuthor` and the `secondAuthor`.

Let's run this code and see how the resulting database table is set up. Everything runs through fine. The interesting bit will be in the creation of the Author table.

```
Hibernate:
    create table Author (
      id integer not null auto increment,
      bio longtext,
      birthDate date,
      image longblob,
      name varchar(255),
      primary key (id)
    ) engine=MyISAM
```

Notice that the `bio` column is of type `longtext` that's because of the `@Lob` annotation. And the `image` column is of type `longblob`. That is **Long binary** data.

Let's quickly head over to our MySQL Workbench and run a select `*` on the `Author` field to see how this data is represented in our relational database.

The screenshot shows the MySQL Workbench interface. The top panel displays a query: `SELECT * FROM bookstordb.Author;`. The bottom panel shows the query result with 2 rows fetched. The first row contains a long bio string, a date, a long hexadecimal string for the image, and the name 'Gilad Barcha'. The second row contains a null bio, a date, a null image, and the name 'James Gosling'.

id	bio	birthDate	image	name
1	Some very long personal bio here	1980-01-31	50726574656E64207468697320697320616E20696D616765	Gilad Barcha
2	(null)	1975-02-28	50726574656E64207468697320697320616C736F20616E20696D616765	James Gosling

The bottom panel shows the table structure for the `Author` table. The columns are `id`, `bio`, `birthDate`, `image`, and `name`. The data types are `int`, `longtext`, `date`, `longblob`, and `varchar` respectively.

COLUMN_NAME	ORDINAL_POSITION	COLUMN_DEFAULT	IS_NULLABLE	DATA_TYPE	NUMERIC_PRECISION	NUMERIC_SCALE	COLUMN_COMMENT
1 id	1 (null)	NO	int	10	0 (null)		
2 bio	2 (null)	YES	longtext	(null)	(null)	(null)	
3 birthDate	3 (null)	YES	date	(null)	(null)	(null)	
4 image	4 (null)	YES	longblob	(null)	(null)	(null)	
5 name	5 (null)	YES	varchar	(null)	(null)	(null)	

You can see that the `bio` column has some **longtext** that's represented here. The `image` column is of type **blob**. The binary data can't really be displayed. The Workbench user interface simply tags it as a **BLOB**.

Embeddable Entities for Persistent Fields

When you're working within an object oriented programming language such as Java, you may want to represent information using nested objects. And you want these to map correctly to the underlying database table.

You can do this using the **@Embeddable** annotation. We've already seen how the **javax.persistence.Embeddable** annotation works for composite keys. Here we'll see how we can use the embeddable annotation for ordinary persistent fields.

Address.java

```
Address.java
1 package com.mytutorial.jpa;
2
3 import javax.persistence.Embeddable;
4
5 @Embeddable
6 public class Address {
7
8     private String city;
9     private String country;
10
11     public Address() {
12     }
13
14     public Address(String city, String country) {
15         this.city = city;
16         this.country = country;
17     }
18
19     public String getCity() {
20         return city;
21     }
22
23     public void setCity(String city) {
24         this.city = city;
25     }
26
27     public String getCountry() {
28         return country;
29     }
30
31     public void setCountry(String country) {
32         this.country = country;
33     }
34
35 }
```

- Now this *Address* class that I have set up here, I have tagged using the **@Embeddable** annotation. This tells the underlying Hibernate provider that *this is a nested object that is embedded into another entity*. This object is *not an entity* by itself and it should *not be mapped to a table* directly. Instead, the member variables within this embeddable map to fields within the **outer entity** that embeds this embeddable.
- This address embeddable has two member variables, **city** and **country**.
- It has a public default constructor with no arguments, and you can set up additional constructors with whatever arguments you choose.
- It's good practice to have your embeddable classes implement the **serializable** interface, but it's not really necessary for them to do so. So go ahead and implement serializable when you're working in a production environment.

- Here, I'm going to just use this *Address* class as-is. We now need to set up the getters and setters for all of the member variables of this *Address* class, get and set *city*, as well as *country*.

I'll now embed this address embeddable object within the *Author* entity.

```
Author.java
1 package com.mytutorial.jpa;
2
3 import java.util.Date;
4
5 import javax.persistence.Embedded;
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
14 public class Author {
15
16     @Id
17     @GeneratedValue(strategy = GenerationType.IDENTITY)
18     private Integer id;
19
20     private String name;
21
22     @Temporal(TemporalType.DATE)
23     private Date birthDate;
24
25     @Embedded
26     private Address address;
27
28     public Author() {
29     }
30
31     public Author(String name, Date birthDate) {
32         this.name = name;
33         this.birthDate = birthDate;
34     }
35
36     public Integer getId() {}
37
38     public void setId(Integer id) {}
39
40     public String getName() {}
41
42     public void setName(String name) {}
43
44     public Date getBirthDate() {}
45
46     public void setBirthDate(Date birthDate) {}
47
48     public Address getAddress() {
49         return address;
50     }
51
52     public void setAddress(Address address) {
53         this.address = address;
54     }
55
56 }
```

Here is *Author.java*,

- and I have tagged the *Author* class with the **@Entity** annotation.

Java Persistence API: Configuring Fields & Performing CRUD Operations

- I'll now specify the other member variables of this author class which will map to columns in my database. `id` is the primary key. We have the `name` of the author, the `birthDate` of the author and the `address` member variable.
- Observe that the address member variable is of type `address` that is our `@Embeddable`, and we have tagged this with the `@Embedded` annotation. This annotation tells Hibernate that this is a complex object that is embedded within this `Author` entity. And we should map the fields of this complex nested object to the columns of the `Author` table.
- I now set up the getters and setters and the constructors for the `Author` class exactly as before.
- Remember to have a default no argument constructor.

Let's switch over and update our `App.java` code to work with this new `Author` entity. I'm going to persist three `Book` entities, they remain exactly the same as before, and I have two `Author` entities.

```
18 EntityManagerFactory factory = Persistence.createEntityManagerFactory("BookstoreDB_Unit");
19 EntityManager entityManager = factory.createEntityManager();
20
21 try {
22     entityManager.getTransaction().begin();
23
24     Book firstBook = new Book("The Java Language Specification",
25                               "Gilad Barcha", 99.99999f);
26     Book secondBook = new Book("The Java Language Specification Second Edition",
27                                "Gilad Barcha, James Gosling", 119f);
28     Book thirdBook = new Book("Core Java Volume I", "Cay S.Horstmann", 59.9999f);
29
30     entityManager.persist(firstBook);
31     entityManager.persist(secondBook);
32     entityManager.persist(thirdBook);
33
34     Author firstAuthor = new Author("Gilad Barcha", new GregorianCalendar(1980, 1, 0).getTime());
35     Address firstAddress = new Address("New York", "USA");
36     firstAuthor.setAddress(firstAddress);
37
38     Author secondAuthor = new Author("James Gosling", new GregorianCalendar(1975, 2, 0).getTime());
39     Address secondAddress = new Address("San Francisco", "USA");
40     secondAuthor.setAddress(secondAddress);
41
42     entityManager.persist(firstAuthor);
43     entityManager.persist(secondAuthor);
44
45 } catch (Exception ex) {
46     System.err.println("An error occurred: " + ex);
47 } finally {
48     entityManager.getTransaction().commit();
49     entityManager.close();
50     factory.close();
51 }
```

Each `Author` has an address.

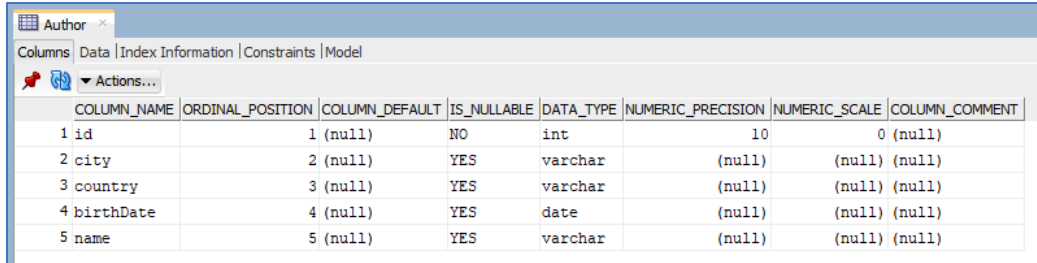
- The `firstAuthor` has the `firstAddress` with City, New York and country USA. I call `firstAuthor.setAddress()` to set the address for "Gilad Barcha".
- Then I have the `secondAddress`, which is the city is "San Francisco", country "USA". And I set this on the `secondAuthor`.
- Go ahead and persist both the `firstAuthor` and the `secondAuthor` entities that will automatically persist the embeddable objects as well.

We'll run this code and everything runs through successfully.

```
Hibernate:
create table Author (
  id integer not null auto_increment,
  city varchar(255),
  country varchar(255),
  birthDate date,
  name varchar(255),
  primary key (id)
) engine=MyISAM
```

What's interesting here is how the *Author* table is set up. Remember the *Author* table is the one which references our address embeddable object. *city* and *country* which are member variables in our nested embeddable have been mapped as columns in the *Author* table.

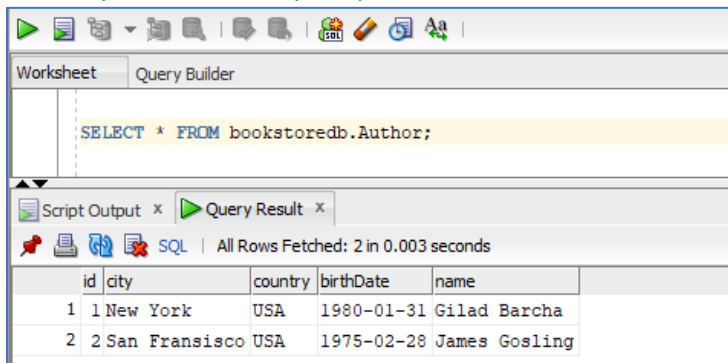
Let's quickly verify this by heading over to our MySQL Workbench and running a describe on the *Author* table. You can see that *city* and *country* are columns within our nested **embeddable** *address* object.



	COLUMN_NAME	ORDINAL_POSITION	COLUMN_DEFAULT	IS_NULLABLE	DATA_TYPE	NUMERIC_PRECISION	NUMERIC_SCALE	COLUMN_COMMENT
1	id	1 (null)	NO	int	10	0 (null)		
2	city	2 (null)	YES	varchar	(null)	(null)	(null)	(null)
3	country	3 (null)	YES	varchar	(null)	(null)	(null)	(null)
4	birthDate	4 (null)	YES	date	(null)	(null)	(null)	(null)
5	name	5 (null)	YES	varchar	(null)	(null)	(null)	(null)

They have been mapped as columns within this *Author* table.

For completeness, let's quickly run a select * on the *Author* table.



	id	city	country	birthDate	name
1	1	New York	USA	1980-01-31	Gilad Barcha
2	2	San Francisco	USA	1975-02-28	James Gosling

And you can see that our two authors have addresses. The author with Id 1 lives in New York, USA. The one with Id 2 lives in San Francisco, USA.

Sharing Embeddable Objects

We saw that the address entity was tagged with the **@Embeddable** annotation. Allowing us to use and embed the address object fields within another table. The cool thing about having embeddable classes within your Java application is that **embeddable objects can be shared across persistent entities**.

Here I've set up a new class called **Publisher** representing the publisher of books. I'm going to embed the address object within the Publisher entity as well.

```

1 package com.mytutorial.jpaa;
2
3 import javax.persistence.Embedded;
4 import javax.persistence.Entity;
5 import javax.persistence.GeneratedValue;
6 import javax.persistence.GenerationType;
7 import javax.persistence.Id;
8
9 @Entity
10 public class Publisher {
11
12     @Id
13     @GeneratedValue(strategy = GenerationType.IDENTITY)
14     private Integer id;
15
16     private String name;
17
18     @Embedded
19     private Address address;
20
21     public Publisher() {
22     }
23
24     public Publisher(String name) {
25         this.name = name;
26     }
27
28     public Integer getId() {
29         return id;
30     }
31
32     public void setId(Integer id) {
33         this.id = id;
34     }
35
36     public String getName() {
37         return name;
38     }
39
40     public void setName(String name) {
41         this.name = name;
42     }
43
44     public Address getAddress() {
45         return address;
46     }
47
48     public void setAddress(Address address) {
49         this.address = address;
50     }
51
52 }

```

- Set up the import statement for the publisher. And tag this Publisher class using **@Entity**. Publisher will map to a table, Publisher, within our relational database.
- We have the **id** column, that is the **primary key** using the generated value of type **IDENTITY**.
- We have the **name** of the publisher, and

- We have the `address` of the publisher. The address member variable here is of type `Address`, which is an **embeddable**. So we tag this within this entity class as **@Embedded**, indicating that the fields within the `Address` object should map to columns within the `Publisher` table.
- We'll set up the remaining code for the publisher in a manner that is familiar to us. Just getters and setters for each member variable.

Time for us to switch over to **App.java**. And instantiate a few `Publisher` entities that we'll persist in our database.

```
18 EntityManagerFactory factory = Persistence.createEntityManagerFactory("BookstoreDB_Unit");
19 EntityManager entityManager = factory.createEntityManager();
20
21 try {
22     entityManager.getTransaction().begin();
23
24     Book firstBook = new Book("The Java Language Specification",
25                             "Gilad Barcha", 99.99999f);
26     Book secondBook = new Book("The Java Language Specification Second Edition",
27                               "Gilad Barcha, James Gosling", 119f);
28     Book thirdBook = new Book("Core Java Volume I", "Cay S.Horstmann", 59.99999f);
29
30     entityManager.persist(firstBook);
31     entityManager.persist(secondBook);
32     entityManager.persist(thirdBook);
33
34     Author firstAuthor = new Author("Gilad Barcha", new GregorianCalendar(1980, 1, 0).getTime());
35     Address firstAddress = new Address("New York", "USA");
36     firstAuthor.setAddress(firstAddress);
37
38     Author secondAuthor = new Author("James Gosling", new GregorianCalendar(1975, 2, 0).getTime());
39     Address secondAddress = new Address("San Fransisco", "USA");
40     secondAuthor.setAddress(secondAddress);
41
42     entityManager.persist(firstAuthor);
43     entityManager.persist(secondAuthor);
44
45     Publisher firstPublisher = new Publisher("Apress");
46     firstAddress = new Address("Paris", "France");
47     firstPublisher.setAddress(firstAddress);
48
49     Publisher secondPublisher = new Publisher("Manning");
50
51     entityManager.persist(firstPublisher);
52     entityManager.persist(secondPublisher);
53
54 } catch (Exception ex) {
55     System.err.println("An error occurred: " + ex);
56 } finally {
57     entityManager.getTransaction().commit();
58     entityManager.close();
59     factory.close();
60 }
```

- In addition to the three `Book` entities and the two `Author` entities, I'm going to add in two `Publisher` entities, the `firstPublisher` and the `secondPublisher`.
- The `firstPublisher` is `"Apress"`. And I've made up an address here. I've set the `firstPublisher` to be in `"Paris"`, `"France"`, by using `firstPublisher.setAddress()`.
- The `secondPublisher` is `"Manning"` and I've set no `Address` for this `secondPublisher`.

Java Persistence API: Configuring Fields & Performing CRUD Operations

Time for us to run this code and see how our embeddable objects can be shared across persistent entities.

```
Hibernate:
    create table Author (
        id integer not null auto_increment,
        city varchar(255),
        country varchar(255),
        birthDate date,
        name varchar(255),
        primary key (id)
    ) engine=MyISAM
Dec 03, 2021 5:13:28 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJta
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.Jdbc
Hibernate:
    create table Book (
        id integer not null auto_increment,
        author varchar(255) not null,
        price float,
        title varchar(55) not null,
        primary key (id)
    ) engine=MyISAM
Hibernate:
    create table Publisher (
        id integer not null auto_increment,
        city varchar(255),
        country varchar(255),
        name varchar(255),
        primary key (id)
    ) engine=MyISAM
Hibernate:
    alter table Book
        add constraint UK_odppys65lq7q1xbx8o6p6fgxj unique (title)
```

- Let's take a look at the create table for the *Publisher* entity here that embeds the *Address* object.
- Notice that *city* and *country* are columns from our embeddable *Address* that have been mapped into this *Publisher* table.

Let's quickly head over to the MySQL Workbench, and describe the Publisher table here.

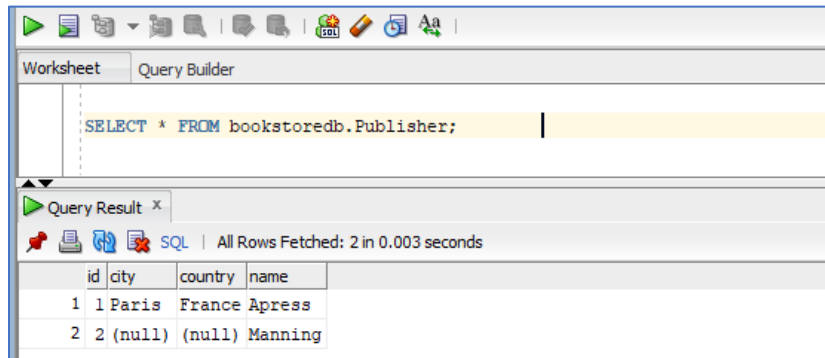
Publisher							
Columns							
COLUMN_NAME	ORDINAL_POSITION	COLUMN_DEFAULT	IS_NULLABLE	DATA_TYPE	NUMERIC_PRECISION	NUMERIC_SCALE	COLUMN_COMMENT
1 id	1 (null)	NO	int	10	0 (null)		
2 city	2 (null)	YES	varchar	(null)	(null)	(null)	
3 country	3 (null)	YES	varchar	(null)	(null)	(null)	
4 name	4 (null)	YES	varchar	(null)	(null)	(null)	

bookstoredb.Publisher	
P * id	INTEGER
city	VARCHAR2 (255)
country	VARCHAR2 (255)
name	VARCHAR2 (255)
PRIMARY (id)	

And you can see that we have city and country as columns.

Java Persistence API: Configuring Fields & Performing CRUD Operations

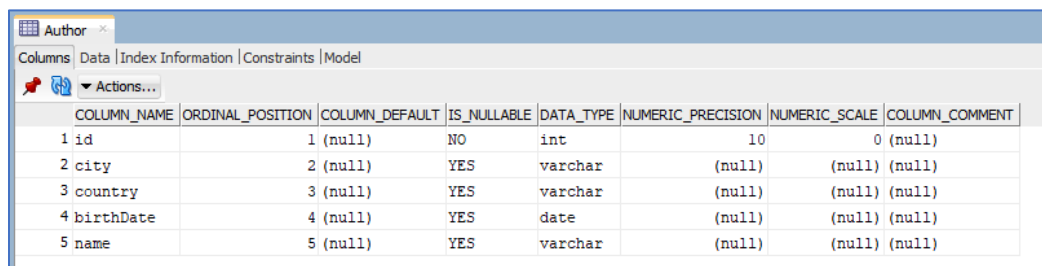
One last sanity check, where we do a select * on the Publisher table.



The screenshot shows the SQL Developer interface with a query window containing the SQL statement: `SELECT * FROM bookstoredb.Publisher;`. Below the query window, the 'Query Result' tab is active, displaying the results of the query. The status bar indicates 'All Rows Fetched: 2 in 0.003 seconds'.

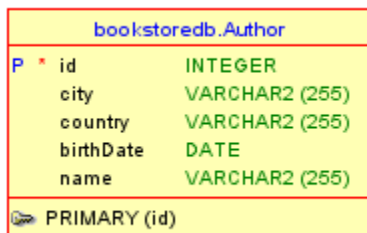
	id	city	country	name
1	1	Paris	France	Apress
2	2	(null)	(null)	Manning

There are two entries here, **Apress** located in **Paris, France** and **Manning** with no address. Both of those fields are set to null.



The screenshot shows the 'Author' table structure in SQL Developer. The 'Columns' tab is selected, displaying a table with columns: COLUMN_NAME, ORDINAL_POSITION, COLUMN_DEFAULT, IS_NULLABLE, DATA_TYPE, NUMERIC_PRECISION, NUMERIC_SCALE, and COLUMN_COMMENT.

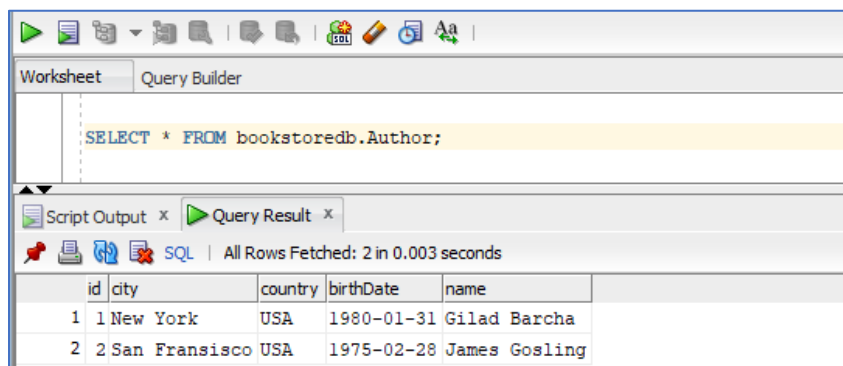
	COLUMN_NAME	ORDINAL_POSITION	COLUMN_DEFAULT	IS_NULLABLE	DATA_TYPE	NUMERIC_PRECISION	NUMERIC_SCALE	COLUMN_COMMENT
1	id	1	(null)	NO	int	10	0	(null)
2	city	2	(null)	YES	varchar	(null)	(null)	(null)
3	country	3	(null)	YES	varchar	(null)	(null)	(null)
4	birthDate	4	(null)	YES	date	(null)	(null)	(null)
5	name	5	(null)	YES	varchar	(null)	(null)	(null)



The diagram shows the structure of the `bookstoredb.Author` table. It lists the columns and their data types, with the `id` column marked as the primary key.

	id	city	country	birthDate	name
P *	INTEGER	VARCHAR2 (255)	VARCHAR2 (255)	DATE	VARCHAR2 (255)

PRIMARY (id)




The screenshot shows the SQL Developer interface with a query window containing the SQL statement: `SELECT * FROM bookstoredb.Author;`. Below the query window, the 'Query Result' tab is active, displaying the results of the query. The status bar indicates 'All Rows Fetched: 2 in 0.003 seconds'.

	id	city	country	birthDate	name
1	1	New York	USA	1980-01-31	Gilad Barcha
2	2	San Francisco	USA	1975-02-28	James Gosling

Begin and Commit Transactions

In this demo, we'll continue working with the bookstore database that we've used so far, and we'll use the same persistence unit, the *BookstoreDB_Unit*.

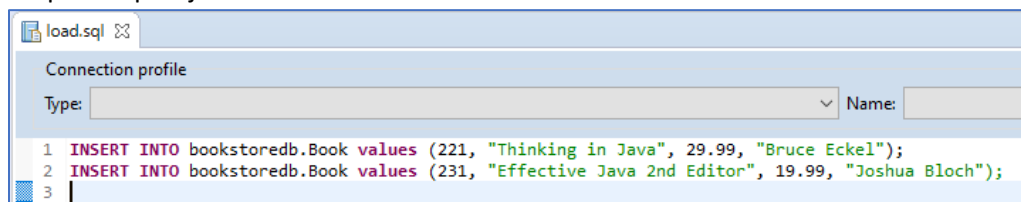


```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <persistence version="2.1"
3   xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
5
6   <persistence-unit name="BookstoreDB_Unit" >
7     <properties>
8       <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver" />
9       <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/bookstoredb" />
10      <property name="javax.persistence.jdbc.user" value="root" />
11      <property name="javax.persistence.jdbc.password" value="password" />
12
13      <property name="javax.persistence.schema-generation.database.action" value="drop-and-create"/>
14      <property name="javax.persistence.sql-load-script-source" value="META-INF/load.sql"/>
15
16      <property name="hibernate.show_sql" value="true"/>
17      <property name="hibernate.format_sql" value="true"/>
18    </properties>
19  </persistence-unit>
20
21 </persistence>
  
```

- Now observe here that my database action has been set to **drop-and-create**. This is great for prototyping because all of the existing tables will be dropped and then recreated each time you run your application.
- Observe, that I have specified a value for the property **javax.persistence.sql-load-script-source**. If you want to preload your data into your relational database, this is the property that you'll specify within **persistence.xml**. The value of this property is set to a script **load.sql** that I'm going to create within the **META-INF** folder under my resources. The script that you execute can load data into multiple tables if you choose. And this is a way to prepopulate your tables before your test or your prototypes.

And that's exactly how we'll use the *load.sql* script. This is where I'm going to set up a few insert statements to insert records into the Book table. Make sure that each SQL command within the SQL script occupies just one line.



```

1 INSERT INTO bookstoredb.Book values (221, "Thinking in Java", 29.99, "Bruce Eckel");
2 INSERT INTO bookstoredb.Book values (231, "Effective Java 2nd Editor", 19.99, "Joshua Bloch");
3
  
```

So don't format your commands to be on multiple lines. Make sure each command is on its own line and doesn't contain any additional new line characters.

If commands are formatted across multiple lines, you'll find that you'll run into parsing errors with hibernate.

Let's head over to **Book.java**.

```

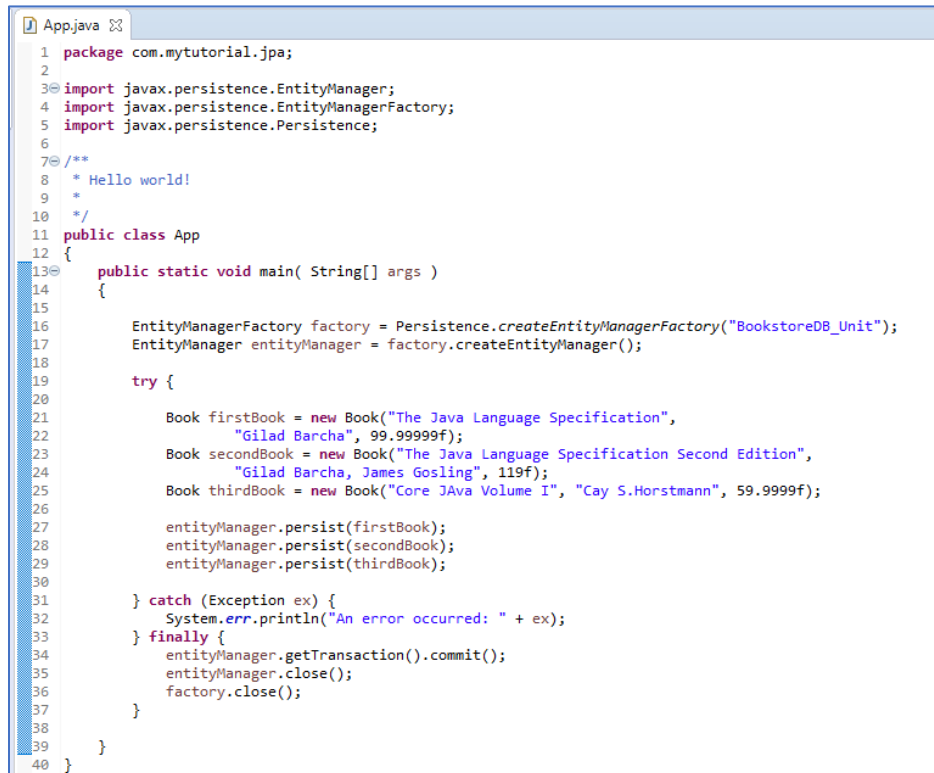
1 package com.mytutorial.jpa;
2
3 import javax.persistence.Column;
4 import javax.persistence.Entity;
5 import javax.persistence.GeneratedValue;
6 import javax.persistence.GenerationType;
7 import javax.persistence.Id;
8
9 @Entity
10 public class Book {
11
12     @Id
13     @GeneratedValue(strategy = GenerationType.IDENTITY)
14     private Integer id;
15
16     @Column(nullable = false)
17     private String author;
18
19     @Column(nullable = false, unique = true, length = 55)
20     private String title;
21
22     @Column(precision = 7, scale = 4)
23     private Float price;
24
25     public Book() {
26     }
27
28     public Book(String title, String author, float price) {
29         this.title = title;
30         this.author = author;
31         this.price = price;
32     }
33
34     public Integer getId() {}
35
36     public void setId(Integer id) {}
37
38     public String getTitle() {}
39
40     public void setTitle(String title) {}
41
42     public String getAuthor() {}
43
44     public void setAuthor(String author) {}
45
46     public Float getPrice() {}
47
48     public void setPrice(Float price) {}
49
50     @Override
51     public String toString() {
52         return String.format("{ %d, %s, %s, %f}", id, title, author, price);
53     }
54 }

```

- Set up the import statements and specify the **@Entity** annotation for the *Book* class.
- Let's specify the fields that we mapped to columns in the *Book* table. We are already familiar with all of this. The *Id* field has the **@Id** and **@GeneratedValue** annotation that is the primary key for *Book*.
- The *author* key is a **VARCHAR** that cannot hold a null value, nullable is equal to false.
- The *title* of the *Book* needs to be unique, that's also not nullable and has a length of 55.
- The *price* of the *Book* is a float with precision=7 and scale=4.
- The rest of the class remains the same. We have the constructor, the public default, no argument constructor, the specific constructor with input arguments, and the getters and setters for all of the fields.

- I have a two string overridden method here at the very bottom of book. This displays a string representation of each book. And it's useful when we want to print out a book instance using **System.out.println()**.

I'll now switch over to **App.java** and write some code.



```
1 package com.mytutorial.jpa;
2
3 import javax.persistence.EntityManager;
4 import javax.persistence.EntityManagerFactory;
5 import javax.persistence.Persistence;
6
7 /**
8  * Hello world!
9  */
10
11 public class App
12 {
13     public static void main( String[] args )
14     {
15
16         EntityManagerFactory factory = Persistence.createEntityManagerFactory("BookstoreDB_Unit");
17         EntityManager entityManager = factory.createEntityManager();
18
19         try {
20
21             Book firstBook = new Book("The Java Language Specification",
22                                     "Gilad Barcha", 99.99999f);
23             Book secondBook = new Book("The Java Language Specification Second Edition",
24                                     "Gilad Barcha, James Gosling", 119f);
25             Book thirdBook = new Book("Core Java Volume I", "Cay S.Horstmann", 59.99999f);
26
27             entityManager.persist(firstBook);
28             entityManager.persist(secondBook);
29             entityManager.persist(thirdBook);
30
31         } catch (Exception ex) {
32             System.err.println("An error occurred: " + ex);
33         } finally {
34             entityManager.getTransaction().commit();
35             entityManager.close();
36             factory.close();
37         }
38     }
39 }
40 }
```

Now we've already seen how we can create entities to persist them as rows in our database table. Observe here that I've created three Book entities, **firstBook**, **secondBook**, and **thirdBook**. And I've used **entityManager.persist()** to persist these three entities.

But there is an important detail that is wrong with this code. Within the try block I haven't called **entityManager.getTransaction.begin()**. I haven't begun the transaction but on line 34 I try to commit the transaction.

Basically, each time you update your database, that is you alter it in any way, you have to have a **begin()** and **commit()** transaction, only the commit is not sufficient.

Java Persistence API: Configuring Fields & Performing CRUD Operations

Let's try and run this code and see what happens.

```
Hibernate:
    drop table if exists Author
Dec 04, 2021 12:54:39 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl g
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnviro
Hibernate:
    drop table if exists Book
Hibernate:
    drop table if exists Publisher
Hibernate:
    create table Author (
      id integer not null auto_increment,
      city varchar(255),
      country varchar(255),
      birthDate date,
      name varchar(255),
      primary key (id)
    ) engine=MyISAM
Dec 04, 2021 12:54:39 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl g
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnviro
Hibernate:
    create table Book (
      id integer not null auto_increment,
      author varchar(255) not null,
      price float,
      title varchar(55) not null,
      primary key (id)
    ) engine=MyISAM
Hibernate:
    create table Publisher (
      id integer not null auto_increment,
      city varchar(255),
      country varchar(255),
      name varchar(255),
      primary key (id)
    ) engine=MyISAM
Hibernate:
    alter table Book
      add constraint UK_odppys651q7qlxbx8o6p6fgxj unique (title)
```

Hibernate behind the scenes will execute statements to **drop** all of the existing tables and recreate tables. Now it seems like everything is working just fine. In fact, all of the stuff that happens under the hood works perfectly. That is, all of the statements created because of the **drop-and-create** database action in **persistence.xml** will work fine.

```
Dec 04, 2021 12:59:17 PM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HHH000476: Executing import script 'ScriptSourceInputFromUrl(file:/C:/SkillUpKnowledge/JPA/my-jpa-app/target/classes/META-INF/load.sql)'
Hibernate:
    INSERT INTO bookstoredb.Book values (221, "Thinking in Java", 29.99, "Bruce Eckel")
Hibernate:
    INSERT INTO bookstoredb.Book values (231, "Effective Java 2nd Editor", 19.99, "Joshua Bloch")
```

Observe that hibernate performs insert operations. These are the two insert statements that are present in our **load.sql** script. This will run fine as well.

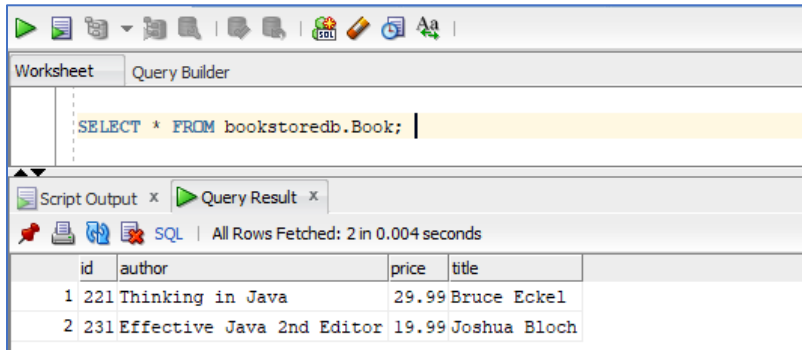
```
Dec 04, 2021 12:59:17 PM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HHH000476: Executing import script 'org.hibernate.tool.schema.internal.exec.ScriptSourceInputNonExistentImpl@5c48c0c0'
Exception in thread "main" java.lang.IllegalStateException: Transaction not successfully started
    at org.hibernate.engine.transaction.internal.TransactionImpl.commit(TransactionImpl.java:63)
    at com.mytutorial.jpa.App.main(App.java:34)
```

But after that, when we try to use our code, that is our Java code, to persist *Book* entities, that's where we'll run into an error. You can see that the error is very clear here. It says **java.lang.IllegalStateException: Transaction not successfully started**.

So all of the entities persisted within our code will not be present in our *Book* table.

Java Persistence API: Configuring Fields & Performing CRUD Operations

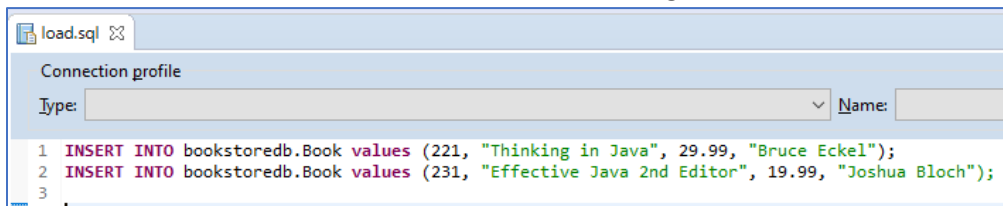
Let's run a select * from the *Book* table and we'll find out. Execute this command and you'll find that the *Book* table has just two entries.



The screenshot shows a SQL query builder window with a 'Query Builder' tab. The query entered is `SELECT * FROM bookstoredb.Book;`. Below the query, the 'Query Result' tab is active, displaying a table with 2 rows and 5 columns: `id`, `author`, `price`, `title`, and an unlabeled column. The data is as follows:

id	author	price	title	
1	221 Thinking in Java	29.99	Bruce Eckel	
2	231 Effective Java 2nd Editor	19.99	Joshua Bloch	

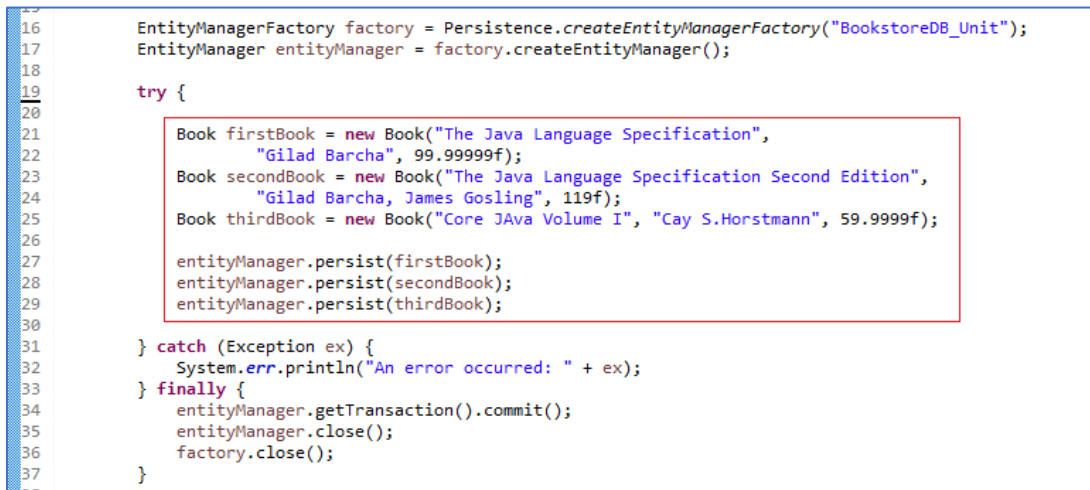
These are the two book records that were inserted using the SQL commands in the **load.sql** file.



The screenshot shows a SQL script editor with a file named `load.sql`. The script contains two `INSERT INTO` statements:

```
1 INSERT INTO bookstoredb.Book values (221, "Thinking in Java", 29.99, "Bruce Eckel");
2 INSERT INTO bookstoredb.Book values (231, "Effective Java 2nd Editor", 19.99, "Joshua Bloch");
3
```

The book entities that are persisted as a part of our code that is in **Book.java**, those don't exist in this table.



The screenshot shows a Java code editor with the `Book.java` file. The code is as follows:

```
16 EntityManagerFactory factory = Persistence.createEntityManagerFactory("BookstoreDB_Unit");
17 EntityManager entityManager = factory.createEntityManager();
18
19 try {
20     Book firstBook = new Book("The Java Language Specification",
21                             "Gilad Barcha", 99.999999f);
22     Book secondBook = new Book("The Java Language Specification Second Edition",
23                              "Gilad Barcha, James Gosling", 119f);
24     Book thirdBook = new Book("Core Java Volume I", "Cay S.Horstmann", 59.99999f);
25
26     entityManager.persist(firstBook);
27     entityManager.persist(secondBook);
28     entityManager.persist(thirdBook);
29
30 } catch (Exception ex) {
31     System.err.println("An error occurred: " + ex);
32 } finally {
33     entityManager.getTransaction().commit();
34     entityManager.close();
35     factory.close();
36 }
37
```

That's because we hadn't begun the transaction before committing the transaction.

Java Persistence API: Configuring Fields & Performing CRUD Operations

Let's go back to our Java code and fix this. Here we are on **App.java**, and I've added in an `entityManager.getTransaction().begin();` on line 20.

```
16 EntityManagerFactory factory = Persistence.createEntityManagerFactory("BookstoreDB_Unit");
17 EntityManager entityManager = factory.createEntityManager();
18
19 try {
20     entityManager.getTransaction().begin();
21
22     Book firstBook = new Book("The Java Language Specification",
23                             "Gilad Barcha", 99.99999f);
24     Book secondBook = new Book("The Java Language Specification Second Edition",
25                               "Gilad Barcha, James Gosling", 119f);
26     Book thirdBook = new Book("Core Java Volume I", "Cay S. Horstmann", 59.99999f);
27
28     entityManager.persist(firstBook);
29     entityManager.persist(secondBook);
30     entityManager.persist(thirdBook);
31
32 } catch (Exception ex) {
33     System.err.println("An error occurred: " + ex);
34 } finally {
35     entityManager.getTransaction().commit();
36     entityManager.close();
37     factory.close();
38 }
```

Now that we've wrapped our entity persistence within a begin and end transaction, everything should work just fine. We run our code and you can see within the console output that there are no errors.

```
Dec 04, 2021 1:10:04 PM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HHH000476: Executing import script 'ScriptSourceInputFromUrl(file:/C:/SkillUpKnowledge/JPA/my-jpa-app/target/classes/META-INF/load.sql)'
Hibernate:
INSERT INTO bookstoredb.Book values (221, "Thinking in Java", 29.99, "Bruce Eckel")
Hibernate:
INSERT INTO bookstoredb.Book values (231, "Effective Java 2nd Editor", 19.99, "Joshua Bloch")
Dec 04, 2021 1:10:04 PM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HHH000476: Executing import script 'org.hibernate.tool.schema.internal.exec.ScriptSourceInputNonExistentImpl@5c48c0c0'
Hibernate:
insert
into
Book
(author, price, title)
values
(?, ?, ?)
Hibernate:
insert
into
Book
(author, price, title)
values
(?, ?, ?)
Hibernate:
insert
into
Book
(author, price, title)
values
(?, ?, ?)
Dec 04, 2021 1:10:05 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl stop
INFO: HHH10001008: Cleaning up connection pool [jdbc:mysql://localhost:3306/bookstoredb]
```

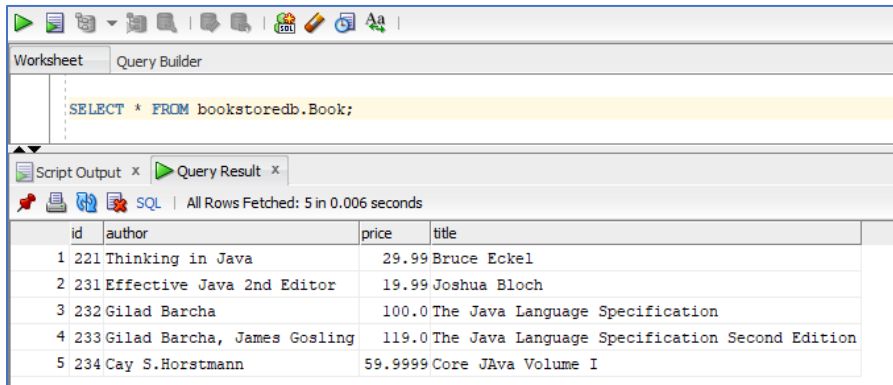
Here are the insert statements from our **load.sql** script, which is run before our application.

And if you scroll down below you'll see other insert commands persisting each of our book entities, the `firstBook`, `secondBook`, and `thirdBook` to our relational database.

Now everything looks good.

Java Persistence API: Configuring Fields & Performing CRUD Operations

Let's switch over to our MySQL Workbench and run a select * for the *Book* table. This will allow us to see whether all entries are present and yes, indeed, they are. There are a total of five records in this *Book* table.



The screenshot shows the MySQL Workbench interface. The 'Query Builder' tab is active, displaying the SQL query: `SELECT * FROM bookstoredb.Book;`. Below the query editor, the 'Query Result' tab shows the results of the query. The results are displayed in a table with four columns: 'id', 'author', 'price', and 'title'. There are five rows of data.

	id	author	price	title
1	221	Thinking in Java	29.99	Bruce Eckel
2	231	Effective Java 2nd Editor	19.99	Joshua Bloch
3	232	Gilad Barcha	100.0	The Java Language Specification
4	233	Gilad Barcha, James Gosling	119.0	The Java Language Specification Second Edition
5	234	Cay S. Horstmann	59.9999	Core Java Volume I

The first two records with ids **221** and **231** are from the **load.sql** script.

The remaining three records for “**The Java language specification**”, “**The second edition**”, and the “**Core Java Volume 1**”, these are entities that we persisted from within our Java code.

Read Operations

In this learning path, so far we've had lots of experience with create operations. We know how to configure different kinds of entities and insert those entities as records in our database tables. Now we'll get some experience reading data from our SQL tables.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <persistence version="2.1"
3   xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
5
6   <persistence-unit name="BookstoreDB_Unit" >
7     <properties>
8       <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver" />
9       <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/bookstoredb" />
10      <property name="javax.persistence.jdbc.user" value="root" />
11      <property name="javax.persistence.jdbc.password" value="password" />
12
13      <property name="javax.persistence.schema-generation.database.action" value="drop-and-create"/>
14      <property name="javax.persistence.sql-load-script-source" value="META-INF/load.sql"/>
15
16      <property name="hibernate.show_sql" value="true"/>
17      <property name="hibernate.format_sql" value="true"/>
18    </properties>
19  </persistence-unit>
20
21 </persistence>

```

I'm going to modify this *load.sql* script here to insert four book entities into the *Book* table. Here are four insert commands. Remember, each SQL command should be on its own line. So don't format your SQL commands on multiple lines.

```

1 INSERT INTO bookstoredb.Book values (221, "Thinking in Java", 29.99, "Bruce Eckel");
2 INSERT INTO bookstoredb.Book values (231, "Effective Java 2nd Editor", 19.99, "Joshua Bloch");
3 INSERT INTO bookstoredb.Book values (241, "The Java Language Specification", 99.99, "Gilad Bracha");
4 INSERT INTO bookstoredb.Book values (251, "Core Java Volume I", 59.99, "Cay S. Horstmann");
5

```

We've inserted four books with ids 221, 231, 241, and 251. Each Book has a [name](#), an [author](#), and a [price](#). Now, we'll have some data in our database, in order to read from our *Book* table.

Uses EntityManager Find()

I'm going to change the **App.java** code to use the find operation, which is what we use to read from our table.

```

16 EntityManagerFactory factory = Persistence.createEntityManagerFactory("BookstoreDB_Unit");
17 EntityManager entityManager = factory.createEntityManager();
18
19 try {
20   Book bookOne = entityManager.find(Book.class, 221);
21   System.out.println(String.format("Book 1 : %s", bookOne));
22
23   Book bookTwo = entityManager.find(Book.class, 251);
24   System.out.println(String.format("Book 2 : %s", bookTwo));
25 } catch (Exception ex) {
26   System.err.println("An error occurred: " + ex);
27 } finally {
28   entityManager.close();
29   factory.close();
30 }

```

The find method is a generic method and you need to specify the class of the entity that you're looking for within your database persistence unit. Within our bookstoredb database that is our persistence unit, I'm looking for an entity of the *Book.class*, and the primary key of that entity is 221.

The entityManager will return the data in the form of a Book object, which I'll then print out to screen on line 21.

I then retrieve another *Book*, this time with the id 251, that is the primary key of the Book. So we use the find method once again, to get a *Book* entity, *Book.class* is the class type that we pass in. And once we access the Book, we'll print out the contents of the book to screen, **System.out.println(bookTwo)**.

Observe that because we are just reading from our database table and not altering the table in any way, we don't have to enclose our methods within a **begin** transaction and a **commit** transaction. Read operations can be performed outside of a transaction as well.

Go ahead and run this code, and let's take a look at the output in the console window.

```
Dec 04, 2021 1:38:49 PM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HHH000476: Executing import script 'ScriptSourceInputFromUrl(file:/C:/SkillUpKnowledge/JPA/my-jpa-app/target/classes/META-INF/load.sql)'
Hibernate:
INSERT INTO bookstoredb.Book values (221, "Thinking in Java", 29.99, "Bruce Eckel")
Hibernate:
INSERT INTO bookstoredb.Book values (231, "Effective Java 2nd Editor", 19.99, "Joshua Bloch")
Hibernate:
INSERT INTO bookstoredb.Book values (241, "The Java Language Specification", 99.99, "Gilad Bracha")
Hibernate:
INSERT INTO bookstoredb.Book values (251, "Core Java Volume I", 59.99, "Cay S. Horstmann")
Dec 04, 2021 1:38:49 PM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HHH000476: Executing import script 'org.hibernate.tool.schema.internal.exec.ScriptSourceInputNonExistentImpl@5c48c0c0'
```

There are no errors in the console output. And notice, the insert statements loading in four books into our Book table. This is what we had configured in **persistence.xml** and *load.sql*.

persistence.xml

```
6  <persistence-unit name="BookstoreDB_Unit" >
7    <properties>
8      <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver" />
9      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/bookstoredb" />
10     <property name="javax.persistence.jdbc.user" value="root" />
11     <property name="javax.persistence.jdbc.password" value="password" />
12
13     <property name="javax.persistence.schema-generation.database.action" value="drop-and-create"/>
14     <property name="javax.persistence.sql-load-script-source" value="META-INF/load.sql"/>
15
16     <property name="hibernate.show_sql" value="true"/>
17     <property name="hibernate.format_sql" value="true"/>
18   </properties>
19 </persistence-unit>
```

META-INF/load.sql

```
load.sql
Connection profile
Type: Name:
1 INSERT INTO bookstoredb.Book values (221, "Thinking in Java", 29.99, "Bruce Eckel");
2 INSERT INTO bookstoredb.Book values (231, "Effective Java 2nd Editor", 19.99, "Joshua Bloch");
3 INSERT INTO bookstoredb.Book values (241, "The Java Language Specification", 99.99, "Gilad Bracha");
4 INSERT INTO bookstoredb.Book values (251, "Core Java Volume I", 59.99, "Cay S. Horstmann");
5
```

Java Persistence API: Configuring Fields & Performing CRUD Operations

When we invoke the find method to search for a particular record by primary key, under the hood, Hibernate runs a select statement on our *Book* table to look for a book with that specified **primary key**.

App.java

```
20      Book bookOne = entityManager.find(Book.class, 221);
21      System.out.println(String.format("Book 1 : %s", bookOne));
22
23      Book bookTwo = entityManager.find(Book.class, 251);
24      System.out.println(String.format("Book 2 : %s", bookTwo));
```

Console Output:

```
Hibernate:
select
  book0_.id as id1_1_0_,
  book0_.author as author2_1_0_,
  book0_.price as price3_1_0_,
  book0_.title as title4_1_0_
from
  Book book0_
where
  book0_.id=?
Book 1 : { 221, Bruce Eckel, Thinking in Java, 29.990000}
Hibernate:
select
  book0_.id as id1_1_0_,
  book0_.author as author2_1_0_,
  book0_.price as price3_1_0_,
  book0_.title as title4_1_0_
from
  Book book0_
where
  book0_.id=?
Book 2 : { 251, Cay S. Horstmann, Core Java Volume I, 59.990002}
Dec 04, 2021 1:38:49 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl stop
INFO: HHH10001008: Cleaning up connection pool [jdbc:mysql://localhost:3306/bookstoredb]
```

The select statement is what you can see here on the console output. And just below that you can see our **System.out.println** result. The primary key 221 is associated with the book “Thinking in Java” by “Bruce Eckel”. This shows us that this Book entity has been successfully retrieved from the underlying table.

If you scroll down further, you'll see the second select statement for the book with primary key 251. Once Hibernate executes this query, the *Book* “Core Java Volume I” is retrieved, and the contents have been printed out to screen.

I'm now going to add in some more code here, we'll try retrieving a book that does not exist in our database. I'm going to use `entityManager.find()` to find a book with primary key `281`. "There is no such entry in our underlying Book table".

```
25
26         Book bookThree = entityManager.find(Book.class, 281);
27         System.out.println(String.format("Book 3 : %s", bookThree));
28
```

Let's run this code and see what the result looks like. There is no error, so everything seems to be working fine.

```
Hibernate:
  select
    book0_.id as id1_1_0_,
    book0_.author as author2_1_0_,
    book0_.price as price3_1_0_,
    book0_.title as title4_1_0_
  from
    Book book0_
  where
    book0_.id=?
Book 3 : null
```

When we try and retrieve the `bookThree` with id `281`, we just get a *null* entity, and null has been printed out to screen.

Uses JPQL select Query

JPA with Hibernate allows you to retrieve multiple entities from the underlying database table using queries. Later on in this learning path, we'll cover in some detail how you can run native queries as well as JPQL queries from within your JPA application. But for now we'll see how a simple select query works.

import **java.util.List** we'll retrieve a list of entities, and add the **SuppressWarnings** annotation to your main method. The SuppressWarnings annotation is needed because we are about to perform an unchecked cast.

I now write a simple JPQL select query, in order to retrieve all of the book entities that exists in the underlying Book table, **"SELECT b FROM Book b"**.

```
App.java
1 package com.mytutorial.jpa;
2
3 import java.util.List;
4
5 import javax.persistence.EntityManager;
6 import javax.persistence.EntityManagerFactory;
7 import javax.persistence.Persistence;
8
9 public class App
10 {
11     public static void main( String[] args )
12     {
13
14         EntityManagerFactory factory = Persistence.createEntityManagerFactory("BookstoreDB_Unit");
15         EntityManager entityManager = factory.createEntityManager();
16
17         try {
18
19             List<Book> books = entityManager.createQuery("SELECT b FROM Book b", Book.class).getResultList();
20             System.out.println(books);
21
22         } catch (Exception ex) {
23             System.err.println("An error occurred: " + ex);
24         } finally {
25             entityManager.close();
26             factory.close();
27         }
28     }
29 }
30 }
```

- JPQL queries look very much like SQL queries, but they're actually independent of the database that you're using under the hood.
- The structure of the queries also tend to be a little bit different. We'll understand many of those nuances later on in this learning path. For now, we'll create a query using the **createQuery()** method on the **entityManager**. And **getResultList()** will execute the query on our MySQL database return the results in the form of entities. We'll get a list of books in return, and we'll print out those books.

Let's go ahead and run this code and see whether all of the books in our underlying table have been retrieved.

```
Dec 04, 2021 2:00:44 PM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HHH000476: Executing import script 'org.hibernate.tool.schema.internal.exec.ScriptSourceInputNonExistentImpl@5c48c0c0'
Dec 04, 2021 2:00:44 PM org.hibernate.hql.internal.QueryTranslatorFactoryInitiator initiateService
INFO: HHH000397: Using ASTQueryTranslatorFactory
Hibernate:
select
  book0_.id as id1_1_,
  book0_.author as author2_1_,
  book0_.price as price3_1_,
  book0_.title as title4_1_
from
  Book book0_
[{" 221, Bruce Eckel, Thinking in Java, 29.990000}, {" 231, Joshua Bloch, Effective Java 2nd Editor, 19.990000}, {" 241, Gilad Bracha, The Java Language Specification, 99.989998}, {" 251, Cay S. Horstmann, Core Java Volume I, 59.990002}]
Dec 04, 2021 2:00:45 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl stop
INFO: HHH10001008: Cleaning up connection pool [jdbc:mysql://localhost:3306/bookstoredb]
```

I'm going to scroll down to the very bottom and there you see it the list of books, all of the books that exist in our *Book* table. The select query has retrieved all four book records present in the *Book* table.

Update and Delete Operations

So far we've seen how to create entities, we've also seen how to read entities from our database table.

Perform Update Operation in JPA uses EntityManager merge()

Let's take a look at the U in CRUD, that is **update** operations for entities that already exist in our database.

In order to update an entity, we first need to retrieve that entity from the underlying table, and we'll do this using the find method.

I've performed two find invocations to retrieve the entities with primary keys 221 and 251. bookOne and bookTwo are those entities I'll first print them out to screen before I perform an update.

```

12 EntityManagerFactory factory = Persistence.createEntityManagerFactory("BookstoreDB_Unit");
13 EntityManager entityManager = factory.createEntityManager();
14
15 try {
16     Book bookOne = entityManager.find(Book.class, 221);
17     Book bookTwo = entityManager.find(Book.class, 251);
18
19     System.out.println(String.format("Book One : %s", bookOne));
20     System.out.println(String.format("Book Two : %s", bookTwo));
21
22     entityManager.getTransaction().begin();
23     bookOne.setPrice(25.22f);
24     bookTwo.setTitle("Core Java Volume I - Fundamentals");
25
26     entityManager.merge(bookOne);
27     entityManager.merge(bookTwo);
28
29 } catch (Exception ex) {
30     System.err.println("An error occurred: " + ex);
31 } finally {
32     entityManager.getTransaction().commit();
33     entityManager.close();
34     factory.close();
35 }

```

So on line 19 and 20, I invoke **System.out println()** commands to print the books out, I then **begin** my **Transaction**. Remember update operations alter the state of our database, and we have to enclose this within a Transaction, `entityManager.getTransaction().begin()`.

I'm going to update the price of `bookOne`, so that it's 25.22f, and I'm going to change the title of `bookTwo`, so that it now reads "Core Java Volume I - Fundamentals". Once you've updated the entity objects, we can invoke the `entityManager.merge()` method.

The **merge** method merges the changes that we've made to the *Book* entities with the original values that already exist in the database table. Merge is what you use to **perform update operations** in JPA. Now, when you update the table, make sure that you commit your Transaction as well, we wrapped our update operation in a **begin()** Transaction and **commit()**.

Java Persistence API: Configuring Fields & Performing CRUD Operations

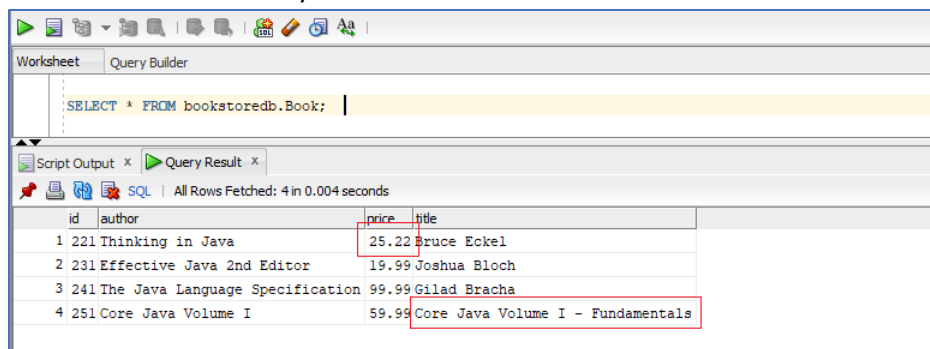
Let's run this code, and let's see whether the updates have been made successfully.

```
Hibernate:
select
  book0_.id as id1_1_0_,
  book0_.author as author2_1_0_,
  book0_.price as price3_1_0_,
  book0_.title as title4_1_0_
from
  Book book0_
where
  book0_.id=?
Hibernate:
select
  book0_.id as id1_1_0_,
  book0_.author as author2_1_0_,
  book0_.price as price3_1_0_,
  book0_.title as title4_1_0_
from
  Book book0_
where
  book0_.id=?
Book One : { 221, Bruce Eckel, Thinking in Java, 29.99}
Book Two : { 251, Cay S. Horstmann, Core Java Volume I, 59.99}
Hibernate:
update
  Book
set
  author=?,
  price=?,
  title=?
where
  id=?
Hibernate:
update
  Book
set
  author=?,
  price=?,
  title=?
where
  id=?
Dec 04, 2021 2:21:22 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl stop
INFO: HHH10001008: Cleaning up connection pool [jdbc:mysql://localhost:3306/bookstoredb]
```

I'm going to scroll down in the console window, you'll see when we retrieve the two books.

These are the original records before update, “*Thinking in Java*” with the price of 29.99 and “*Core Java Volume 1*”. We then update these books, one where we change the **price** and another where we change the **name** of the *Book*. In the output console log statements, we can see update commands were executed, but whether the update was successful, only SQL will tell us.

So let's switch over to MySQL Workbench and run a select * on the *Book* table.



The screenshot shows the MySQL Workbench interface. The 'Query Builder' tab is active, displaying the query: `SELECT * FROM bookstoredb.Book;`. Below the query, the 'Script Output' and 'Query Result' tabs are visible. The 'Query Result' tab shows the results of the query, which are 4 rows of book data. The first two rows are highlighted with red boxes, corresponding to the books mentioned in the text: 'Thinking in Java' and 'Core Java Volume I - Fundamentals'.

	id	author	price	title
1	221	Thinking in Java	25.22	Bruce Eckel
2	231	Effective Java 2nd Editor	19.99	Joshua Bloch
3	241	The Java Language Specification	99.99	Gilad Bracha
4	251	Core Java Volume I	59.99	Core Java Volume I - Fundamentals

Once we run this query, we'll see the actual values that are present in the database, and you can see that the two records that we updated have indeed been updated in the underlying table. “*Core Java Volume 1*” is now called “*Core Java Volume I – Fundamentals*”.

Perform Delete Operation in JPA uses EntityManager remove()

We now know how to perform update operations on our database. Now let's see the last of the CRUD operations, we'll see how we can delete entities from the underlying table, this we do using `entityManager.remove()`.

App.java

```
12     EntityManagerFactory factory = Persistence.createEntityManagerFactory("BookstoreDB_Unit");
13     EntityManager entityManager = factory.createEntityManager();
14
15     try {
16         Book bookOne = entityManager.find(Book.class, 221);
17         Book bookTwo = entityManager.find(Book.class, 251);
18
19         System.out.println(String.format("Book One : %s", bookOne));
20         System.out.println(String.format("Book Two : %s", bookTwo));
21
22         entityManager.getTransaction().begin();
23
24         if (bookOne != null) entityManager.remove(bookOne);
25         if (bookTwo != null) entityManager.remove(bookTwo);
26
27     } catch (Exception ex) {
28         System.err.println("An error occurred: " + ex);
29     } finally {
30         entityManager.getTransaction().commit();
31         entityManager.close();
32         factory.close();
33     }
```

We use the find method as we did before to access the entities that we want to delete. And if the entity is present, we call `entityManager.remove()` and pass in the entity that we want to delete.

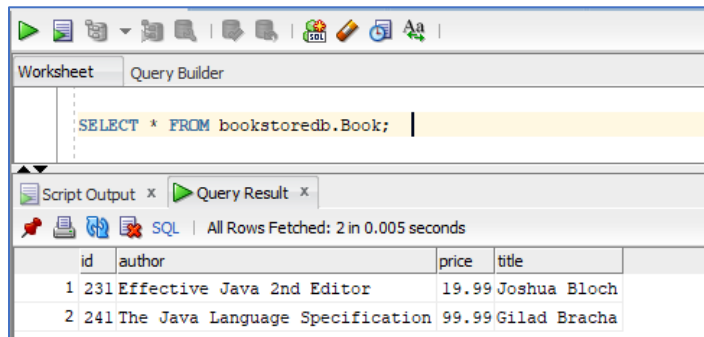
Let's see how this works we'll run this code, and if you scroll down to the very bottom, you'll see that a few delete operations were executed.

```
Hibernate:
select
  book0_.id as id1_1_0_,
  book0_.author as author2_1_0_,
  book0_.price as price3_1_0_,
  book0_.title as title4_1_0_
from
  Book book0_
where
  book0_.id=?
Hibernate:
select
  book0_.id as id1_1_0_,
  book0_.author as author2_1_0_,
  book0_.price as price3_1_0_,
  book0_.title as title4_1_0_
from
  Book book0_
where
  book0_.id=?
Book One : { 221, Bruce Eckel, Thinking in Java, 29.99}
Book Two : { 251, Cay S. Horstmann, Core Java Volume I, 59.99}
Hibernate:
delete
from
  Book
where
  id=?
Hibernate:
delete
from
  Book
where
  id=?
Dec 04, 2021 2:28:07 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl stop
INFO: HHH10001008: Cleaning up connection pool [jdbc:mysql://localhost:3306/bookstoredb]
```

The delete complete successfully.

Java Persistence API: Configuring Fields & Performing CRUD Operations

Let's switch over to MySQL Workbench and run a select * on the *Book* table. Once you execute this code, you'll see that we have just two records in our database.

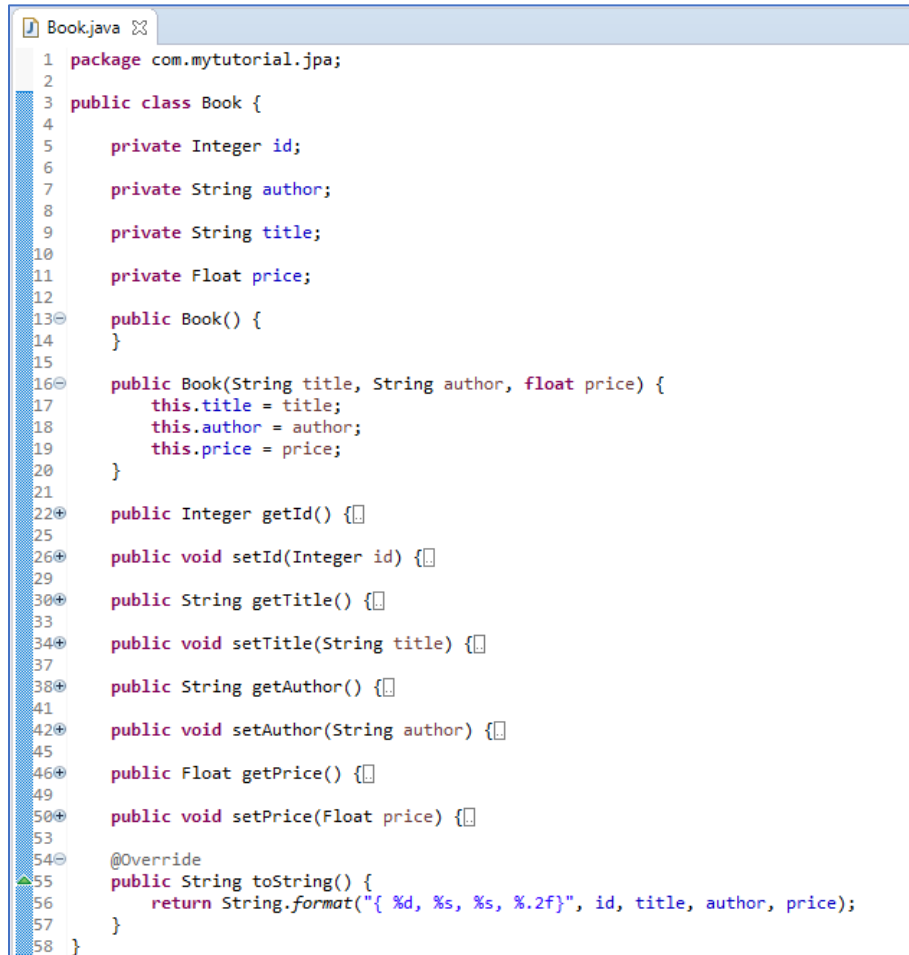


Two entries have been removed, thanks to our `entityManager.remove()` operation which performed a delete on those records.

Entity Specification Using XML

When you work with the JAVA Persistence API and you want to model the underlying database tables that correspond to your objects in code, you'll typically use annotations within your code. Because that technique is more readable. But you can model your underlying database tables using **XML mapping** as well. And that's exactly what we'll do here in this demo, just to show you that it's possible.

Using annotations is considered a best practice because it makes your code more maintainable, because all of the relationships and structure are present together within your Java objects. Here is the **Book** class, this is the entity that we've been working with so far.



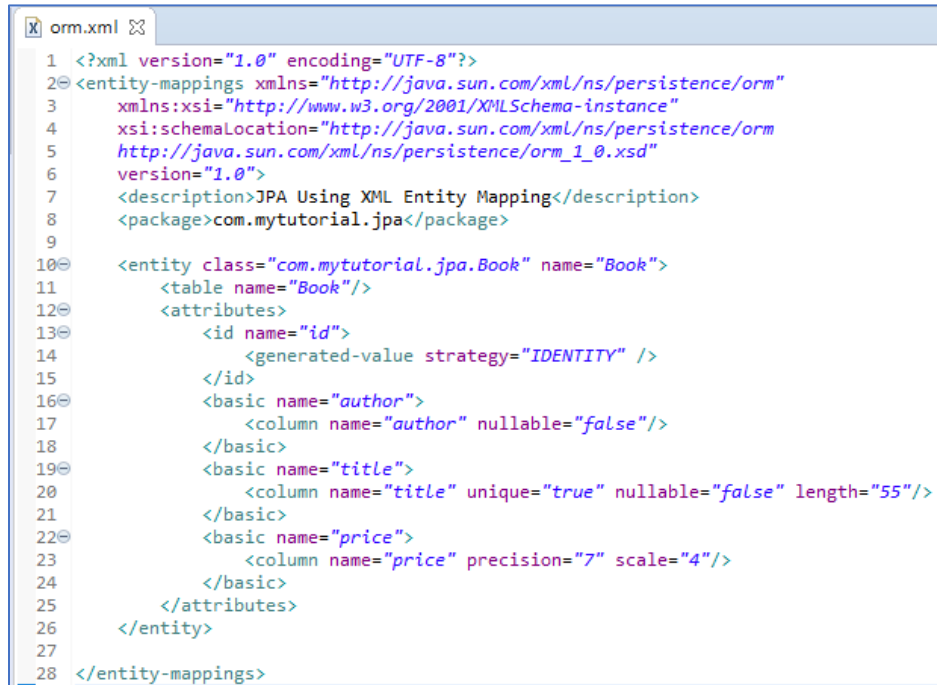
```
1 package com.mytutorial.jpaa;
2
3 public class Book {
4
5     private Integer id;
6
7     private String author;
8
9     private String title;
10
11     private Float price;
12
13     public Book() {
14     }
15
16     public Book(String title, String author, float price) {
17         this.title = title;
18         this.author = author;
19         this.price = price;
20     }
21
22     public Integer getId() {}
23
24     public void setId(Integer id) {}
25
26     public String getTitle() {}
27
28     public void setTitle(String title) {}
29
30     public String getAuthor() {}
31
32     public void setAuthor(String author) {}
33
34     public Float getPrice() {}
35
36     public void setPrice(Float price) {}
37
38     @Override
39     public String toString() {
40         return String.format("{ %d, %s, %s, %.2f}", id, title, author, price);
41     }
42 }
```

- Every *Book* has an *id*, *author*, *title*, and *price*.
- This *Book* class here represents an entity.
- We have set up the constructors, getters, and setters

Exactly like we would normally, except that I haven't annotated this code at all.

You can see there are no annotations on any of my methods. And even the *Book* class itself is not annotated with the **@Entity** annotation.

We'll specify all of the annotations for the Book class, as well as member variables in the Book class using a separate file. I'm going to create a new XML file called **orm.xml**, created under template/META-INF folder.



```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
5     http://java.sun.com/xml/ns/persistence/orm_1_0.xsd"
6   version="1.0">
7   <description>JPA Using XML Entity Mapping</description>
8   <package>com.mytutorial.jpa</package>
9
10  <entity class="com.mytutorial.jpa.Book" name="Book">
11    <table name="Book"/>
12    <attributes>
13      <id name="id">
14        <generated-value strategy="IDENTITY" />
15      </id>
16      <basic name="author">
17        <column name="author" nullable="false"/>
18      </basic>
19      <basic name="title">
20        <column name="title" unique="true" nullable="false" length="55"/>
21      </basic>
22      <basic name="price">
23        <column name="price" precision="7" scale="4"/>
24      </basic>
25    </attributes>
26  </entity>
27
28 </entity-mappings>
  
```

Let's set up the XML mappings for our entity.

- The outermost **XML** tag is `<entity-mappings/>`. Within this, you'll use **entity-mapping** tags to specify the mappings for every entity that you have in your application.

```

<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
    http://java.sun.com/xml/ns/persistence/orm_1_0.xsd"
  version="1.0">
  
```

- Within the `<entity-mappings/>` tag you'll specify a `<description>`

```

<description>JPA Using XML Entity Mapping</description>
  
```

- for your `<entity-mappings/>` you'll also need specify the Java `<package>` where your entity lives.

```

<package>com.mytutorial.jpa</package>
  
```

- And then you'll start the `<entity>` specification, give the complete path to the **entity class** `com.mytutorial.jpa.Book`, the **name** of the entity is `Book` the table name is also `Book` you can choose a different table name if you want.

```

<entity class="com.mytutorial.jpa.Book" name="Book">
  
```

- The table tag corresponds to the **@Table** annotation.

```
<table name="Book"/>
```

- You then specify the attributes of the entity, the annotations that you typically associated with the member variables.

```
<attributes>
```

- The `<id/>` tag is for **primary key** *ids* the generated-value strategy="*IDENTITY*".

```
<id name="id">  
    <generated-value strategy="IDENTITY"/>  
</id>
```

- The `<basic/>` tag is for basic variables such as **Strings**, **floats**, etc.
The basic type *author* corresponds to the column name *author*. It's not nullable, nullable is equal to false.

```
<basic name="author">  
    <column name="author" nullable="false"/>  
</basic>
```

- The basic type *title* correspond to the column name *title*. It's unique and not nullable and has a length of 55.

```
<basic name="title">  
    <column name="title" unique="true" nullable="false" length="55"/>  
</basic>
```

- And finally, the basic field *price* has precision seven scale four

```
<basic name="price">  
    <column name="price" precision="7" scale="4"/>  
</basic>
```

Observe how everything that we could do using annotations can be performed in XML. We'll see that this works in exactly the same way.

orm mapping xml file vs @Entity Class

<pre> 1 <?xml version="1.0" encoding="UTF-8"?> 2 <entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm" 3 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" 4 xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm 5 http://java.sun.com/xml/ns/persistence/orm_1_0.xsd" 6 version="1.0"> 7 <description>JPA Using XML Entity Mapping</description> 8 <package>com.mytutorial.jpa</package> 9 10 <entity class="com.mytutorial.jpa.Book" name="Book"> 11 <table name="Book"/> 12 <attributes> 13 <id name="id"> 14 <generated-value strategy="IDENTITY" /> 15 </id> 16 <basic name="author"> 17 <column name="author" nullable="false"/> 18 </basic> 19 <basic name="title"> 20 <column name="title" unique="true" nullable="false" length="55"/> 21 </basic> 22 <basic name="price"> 23 <column name="price" precision="7" scale="4"/> 24 </basic> 25 </attributes> 26 </entity> 27 </entity-mappings> </pre>	<pre> 1 package com.mytutorial.jpa; 2 3 4 5 6 7 8 9 10 @Entity 11 @Table(name = "Book") 12 public class Book { 13 14 @Id 15 @GeneratedValue(strategy = GenerationType.IDENTITY) 16 private Integer id; 17 18 @Column(name = "author", nullable = false) 19 private String author; 20 21 @Column(name = "title", nullable = false, unique = true, length = 55) 22 private String title; 23 24 @Column(name = "price", precision = 7, scale = 4) 25 private Float price; 26 27 public Book() { 28 } 29 30 public Book(String title, String author, float price) { 31 this.title = title; 32 this.author = author; 33 this.price = price; 34 } </pre>
--	--

We'll now head over to the **persistence.xml** file.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <persistence version="2.1"
3   xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
5
6   <persistence-unit name="BookstoreDB Unit" >
7     <mapping-file>META-INF/orm.xml</mapping-file>
8     <properties>
9       <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver" />
10      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/bookstoredb" />
11      <property name="javax.persistence.jdbc.user" value="root" />
12      <property name="javax.persistence.jdbc.password" value="password" />
13
14      <property name="javax.persistence.schema-generation.database.action" value="drop-and-create"/>
15      <property name="javax.persistence.sql-load-script-source" value="META-INF/load.sql"/>
16
17      <property name="hibernate.show_sql" value="true"/>
18      <property name="hibernate.format_sql" value="true"/>
19    </properties>
20  </persistence-unit>
21 </persistence>

```

We need to indicate within this file the **name** of the **mapping-file**. The **name** of the **mapping-file** here is **META-INF/orm.xml**.

Everything else here remains the same. This is what we use to specify the connection to our underlying database. I've used the database action **drop-and-create**. And I have a load script source that is to **load.sql** under **META-INF**.

I'll quickly configure the **load.sql** file here under the **META-INF** folder.

```

1 INSERT INTO bookstoredb.Book values (221, "Bruce Eckel", 29.99, "Thinking in Java");
2 INSERT INTO bookstoredb.Book values (231, "Joshua Bloch", 19.99, "Effective Java 2nd Edition");
3

```

I'm inserting two books into my **Book** table in the **bookstoredb**.

Now, let's write some code in our main method to persist some entities into our **Book** table. We'll do this in exactly the same way that we have done before.

Java Persistence API: Configuring Fields & Performing CRUD Operations

```
12 EntityManagerFactory factory = Persistence.createEntityManagerFactory("BookstoreDB_Unit");
13 EntityManager entityManager = factory.createEntityManager();
14
15 try {
16     entityManager.getTransaction().begin();
17
18     Book firstBook = new Book("The Java Language Specification", "Gilad Bracha", 99.999999f);
19     Book secondBook = new Book("The Java Language Specification Second Edition",
20                               "Gilad Bracha", 119f);
21     Book thirdBook = new Book("Core Java Volume I", "Cay S. Horstmann", 59.99999f);
22
23     entityManager.persist(firstBook);
24     entityManager.persist(secondBook);
25     entityManager.persist(thirdBook);
26
27 } catch (Exception ex) {
28     System.err.println("An error occurred: " + ex);
29 } finally {
30     entityManager.getTransaction().commit();
31     entityManager.close();
32     factory.close();
33 }
```

I've instantiated three *Book* entities here. "The Java Language Specification", "The second edition", and "Core Java Volume 1". I then invoke `entityManager.persist()` to persist all three entities.

Go ahead and run this code and you'll find that our *Book* entities are persisted in exactly the same way as though we'd use Java annotations.

```
Hibernate:
    drop table if exists Book
Dec 04, 2021 5:53:18 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolat
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentIni
Hibernate:
    create table Book (
        id integer not null auto_increment,
        author varchar(255) not null,
        price float,
        title varchar(55) not null,
        primary key (id)
    ) engine=MyISAM
Dec 04, 2021 5:53:18 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolat
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentIni
Hibernate:
    alter table Book
        add constraint UK_odppys65lq7q1xbx8o6p6fgxj unique (title)
Dec 04, 2021 5:53:18 PM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HHH000476: Executing import script 'ScriptSourceInputFromUrl(file:/C:/SkillUpKnowledge/JPA/my-jpa-app/target/classes/
Hibernate:
    INSERT INTO bookstoredb.Book values (221, "Bruce Eckel", 29.99, "Thinking in Java")
Hibernate:
    INSERT INTO bookstoredb.Book values (231, "Joshua Bloch", 19.99, "Effective Java 2nd Editor")
Dec 04, 2021 5:53:18 PM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HHH000476: Executing import script 'org.hibernate.tool.schema.internal.exec.ScriptSourceInputNonExistentImpl@47289387
Hibernate:
    insert
    into
        Book
        (author, price, title)
    values
        (?, ?, ?)
Hibernate:
    insert
    into
        Book
        (author, price, title)
    values
        (?, ?, ?)
Hibernate:
    insert
    into
        Book
        (author, price, title)
    values
        (?, ?, ?)
Dec 04, 2021 5:53:18 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl stop
INFO: HHH10001008: Cleaning up connection pool [jdbc:mysql://localhost:3306/bookstoredb]
```

Java Persistence API: Configuring Fields & Performing CRUD Operations

Here is the table Book created in the underlying database.

Book							
Columns Data Index Information Constraints Model							
Actions...							
COLUMN_NAME	ORDINAL_POSITION	COLUMN_DEFAULT	IS_NULLABLE	DATA_TYPE	NUMERIC_PRECISION	NUMERIC_SCALE	COLUMN_COMMENT
1 id	1 (null)		NO	int	10	0 (null)	
2 author	2 (null)		NO	varchar	(null)	(null) (null)	
3 price	3 (null)		YES	float	12	(null) (null)	
4 title	4 (null)		NO	varchar	(null)	(null) (null)	

Book										
Columns Data Index Information Constraints Model										
Actions...										
INDEX_NAME	INDEX_TYPE	COLUMN_NAME	SEQ_IN_INDEX	NON_UNIQUE	COLLATION	CARDINALITY	SUB_PART	PACKED	NULLABLE	COMMENT
1 PRIMARY	BTREE	id	1	0 A		5 (null)	(null)	(null)	(null)	(null)
2 UK_odppys65lq7q1xbx8o6p6fgxj	BTREE	title	1	0 A		5 (null)	(null)	(null)	(null)	(null)

bookstoredb.Book	
P * id	INTEGER
* author	VARCHAR2 (255)
price	REAL
U * title	VARCHAR2 (55)
PRIMARY (id)	
UK_odppys65lq7q1xbx8o6p6fgxj (title)	

Switch over to the MySQL Workbench and run a select * on the Book table and you will find that there are five entries here.

Worksheet

Query Builder

```
SELECT * FROM bookstoredb.Book;
```

Script Output x

Query Result x

SQL | All Rows Fetched: 5 in 0.003 seconds

id	author	price	title
1 221	Bruce Eckel	29.99	Thinking in Java
2 231	Joshua Bloch	19.99	Effective Java 2nd Editor
3 232	Gilad Bracha	100.0	The Java Language Specification
4 233	Gilad Barcha	119.0	The Java Language Specification Second Edition
5 234	Cay S. Horstmann	59.9999	Core Java Volume I

Two that we set up using **load.sql**,

```
INSERT INTO bookstoredb.Book values (221, "Bruce Eckel", 29.99, "Thinking in Java");
INSERT INTO bookstoredb.Book values (231, "Joshua Bloch", 19.99, "Effective Java 2nd Edition");
```

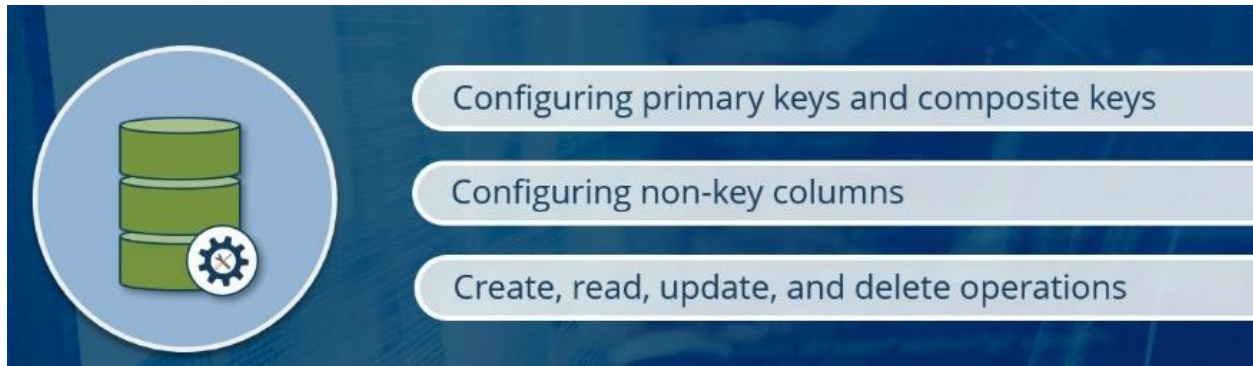
and three that we persisted using the entity manager.

```
Book firstBook = new Book("The Java Language Specification", "Gilad Bracha", 99.99999f);
Book secondBook = new Book("The Java Language Specification Second Edition",
    "Gilad Bracha", 119f);
Book thirdBook = new Book("Core Java Volume I", "Cay S. Horstmann", 59.9999f);

entityManager.persist(firstBook);
entityManager.persist(secondBook);
entityManager.persist(thirdBook);
```

You can see that it's possible to use XML specification for your entities but it's generally not used because it's much harder to maintain.

Course Summary



In this course we saw how the fields of individual entities can be configured in a very granular manner using JPA annotations.

- We started off by configuring the primary keys of our entity objects. JPA supports four techniques to generate primary key values, namely, auto, identity, sequence and table. We understood each of these generation strategies and implemented them on our relational tables. We then moved on to configuring our entities using composite keys, where multiple columns in the underlying database tables make up our key. We saw that there were a number of different strategies that could be followed here as well, and we considered the pros and cons of each approach.
- We then explored the use of JPA annotations for non-key columns. We apply these annotations to manage column definitions, precision and scale values for columns holding floating point data. We imposed non-null and uniqueness constraints on our columns, tagged non-persistable fields in our entities and dealt with date time data and large objects in our tables.
- Finally, we saw how basic storage and retrieval operations could be performed using the JPA EntityManager. We specifically explored, create, read, update and delete operations.