JPA/Hibernate Fundamentals - Course course

JPA Fundamentals . . . 7
1. Relational Databases . . . 7
1.1. Overview . . . 7
1.2. SQL Usage . . . 7
1.3. SQL Syntax . . . 8
1.4. Choosing your Database . . . 8
1.5. Exercise: Project and DB Setup . . . 9
1.5.1. Using Another DB Product . . . 9
1.5.2. Easy Setup . . . 9
1.5.3. Manual Setup . . . 9
1.5.4. Using HSQL . . . 10
1.6. Referential Integrity . . . 10
1.7. First Chapter Exam . . . 10
2. JDBC . . . 11
2.1. Basic APIs . . . 11
2.1.1. What is JDBC? . . . 11
2.1.2. Getting a connection . . . 12
2.1.2.1. Opening . . . 13
2.1.2.2. Closing . . . 14
2.1.3. Statements and ResultSets . . . 15
2.1.3.1. Simple statements with java.sql.Statement . . . 15
2.1.3.2. Parameterized queries with java.sql.PreparedStatement . . . 16
2.2. Transactions (Optional) . . . 16
2.2.1. ACID requirements . . . 17
2.2.2. Concurrency . . . 17
2.2.3. JDBC Transaction Management . . . 18
2.3. Exercise . . . 19
2.3.1. Get database metadata . . . 19
2.3.2. Execute a Statement (Optional) . . . 19
2.4. Exam and Resources . . . 20
3. Hello World . . . 20
3.1. Entities, EntityManager, Persistence Unit . . . 20
3.1.1. Entities . . . 20
3.1.2. EntityManager . . . 21
3.1.3. Persistence Unit . . . 21
3.1.4. JavaDoc . . . 24
3.2. Hibernate setup . . . 24
3.2.1. Hibernate libraries and dependencies . . . 24
3.2.2. Create a new Eclipse project . . . 25
3.3. Hello World Exercise . . . 25
3.3.1. FAQs . . . 26

3.4. Javadoc Exercise . . . 26
3.5. Database Integration . . . 26
3.5.1. Show SQL . . . 26
3.5.2. Show SQL - Try it yourself . . . 27
3.5.3. DB Generation . . . 27
3.5.3.1. Usage during the exercises . . . 27
3.5.3.2. Usage during real-world development. . . . 28
3.5.4. Log4j . . . 28
3.5.5. DB Generation - Try it yourself . . . 29
3.6. Exam and Resources . . . 29
4. JPA Concepts . . . 30
4.1. DB History . . . 30
4.2. RDB Data in Your Program . . . 30
4.3. Paradigm Mismatch . . . 31
4.4. Manual Solution to the Mismatch . . . 31
4.5. JPA Solution to the Mismatch . . . 32
4.6. JPA's History . . . 32
4.7. Exam and Resources . . . 33
5. Entity . . . 33
5.1. Implementation . . . 33
5.1.1. Entity Example . . . 33
5.1.2. @Entity . . . 34
5.1.3. Constructor . . . 34
5.1.4. @Table . . . 34
5.1.5. @Column . . . 34
5.1.6. @Id . . . 35
5.1.6.1. Mandatory . . . 35
5.1.6.2. Surrogate key . . . 35
5.1.6.3. Null . . . 36
5.1.6.4. @GeneratedValue . . . 36
5.2. Exercise . . . 36
5.3. Exam . . . 37
6. Relationships . . . 37
6.1. Unidirectional Relationships . . . 37
6.1.1. ManyToOne . . . 37
6.1.2. OneToMany . . . 38
6.1.3. OneToOne . . . 38
6.1.4. JoinColumn . . . 38
6.2. Unidirectional Relationship Exercise . . . 39
6.3. Bidirectional Relationships . . . 39
6.3.1. mappedBy . . . 39

6.3.2. Where is the foreign key? . . . 40
6.3.3. Keeping both directions in sync. . . . 40
6.4. Bidirectional Relationships Exercise . . . 41
6.5. Recap video & Exam . . . 41
7. Lifecycle . . . 41
7.1. Entity States . . . 42
7.2. EntityManager methods . . . 42
7.3. Handling Entity Equality . . . 43
7.3.1. Object Equality . . . 43
7.3.2. Entity Equality . . . 44
7.3.2.1. Generated Key . . . 44
7.3.2.2. Business Key . . . 45
7.3.2.3. Choice . . . 45
7.4. Summary Exercise . . . 45
7.5. Exam and References . . . 45
8. Fetching Strategies . . . 46
8.1. Understanding the Lazy Loading principle . . . 46
8.1.1. The Lazy/Eager Principle . . . 46
8.1.2. Default . . . 47
8.1.3. When to use eager? . . . 47
8.1.4. More references . . . 48
8.2. Hibernate Uses Proxies . . . 48
8.2.1. Where's the Magic? . . . 48
8.2.2. Lazy-loading gotcha . . . 49
8.3. Summary Diagram . . . 49
8.4. Exercise . . . 49
9. Query Language . . . 50
9.1. Syntax Basics . . . 50
9.1.1. Java Names . . . 50
9.1.2. API . . . 50
9.1.3. Parameters . . . 51
9.1.4. Short Form . . . 51
9.1.5. Implicit Joins . . . 52
9.2. Exercise . . . 52
9.3. Exam . . . 52
10. DAO . . . 53
10.1. DAO Basics . . . 53
10.1.1. What are DAOs? . . . 53
10.1.2. Where is my EntityManager? . . . 53
10.1.3. More about DAOs . . . 53
10.2. Spring shows up 54

10.2.1. @PersistenceContext 54
10.2.2. ApplicationContext 54
10.3. Exercise 55
11. More Query Language 55
11.1. Explicit Joins 55
11.1.1. Syntax 56
11.1.2. When Are You Forced to Use Explicit Joins? 56
11.2. Aggregation Functions 58
11.3. Further Reading 58
11.4. Practice 58
11.4.1. Exercise 58
11.4.2. Exam 59
12. Project Setup Summary 59
12.1. Overview 59
12.2. Hibernate Setup 59
12.3. Spring setup 60
13. Exceptions 60
13.1. Class Diagram 60
13.2. BatchUpdateException 60
13.2.1. When does it happen ? 60
13.2.2. Stack trace 61
13.2.3. Using getNextException method 61
13.3. Common exceptions 62
13.3.1. NullPointerException 62
13.3.2. IllegalArgumentException 63
13.3.3. ClassNotFoundException 63
13.3.4. ClassCastException 64
13.3.5. NoSuchMethodException 64
13.3.6. SqlException 65
13.3.7. QuerySyntaxException 65
13.3.8. PersistenceException 66
13.3.9. NoSuchBeanDefinitionException 66
13.3.10. TransientObjectException 67
13.3.11. BeanCreationException 67
13.3.12. AnnotationException 68
14. Conclusion & Exam 69
15. Exercise Solutions 69
15.1. reminder 69
15.2. solutions 70
15.2.1. fetching strategy 70
15.2.1.1. Invoice 70

15.2.1.2. InvoiceLine 71
15.2.1.3. Main 72
15.2.1.4. Fetching Strategy 73
15.2.1.4.1. Lazy loading 74
15.2.1.4.2. Eager loading 75
15.3. DAO 76

JPA Fundamentals

Welcome to this **Java Persistence API course**. It will teach you, in a hands-on way, how JPA builds up from SQL and Java's low-level standard libraries to bridge the gap between relational databases and object orientation.

Each chapter has a theory section, points to some online resources for further information and suggests a few practical exercises. We've also created one short exam per chapter to make sure you've assimilated the material.

No prior knowledge of JPA or any other persistence framework is necessary. In fact, we'll walk you through setting up a database and a JPA project with a simple domain model.

If you have not already done so, please create an account on JavaBlackBelt before starting the course. This will allow you to pass the exams and record your progress.

By the end of this course, you should have a solid grasp of what JPA is and why it is useful, how to map your objects to a database using annotations and how to manipulate your persistent objects.

Before starting, feel free to have a look at this video where John Rizzo explains how a BlackBeltFactory course happen.

Watch this video on
http://knowledgeblackbelt.com/#CoursePage/12647612/EN

The following chapters are typically performed in 16 hours (2 business days).

1. Relational Databases

1.1. Overview

Prior to executing any JPA exercise, we need a relational database (RDB). In the prerequisites of this course we assumed you already have knowledge of relational databases, so this chapter is here as a quick refresher that covers the basics you have to know before starting with JPA.

1.2. SQL Usage

Why should you bother about SQL ? Won't JPA/Hibernate generate SQL statements for you ? Well ... yes but developers using JPA typically use SQL directly during development phase:

- to verify data values (set by JPA)

- to apply data fixes (change data directly in the DB)
- to change the DB structure (migrate to the next version of the application)
- to verify SQL statements generated/logged by JPA (for debugging)

1.3. SQL Syntax

This part contains a short reminder about SQL syntax with examples of the main SQL commands.

Source

```
// Select all fields from the product that has the id 234.
select * from PRODUCT where ID=234

// Select the id and the label of all products which label is "computer".
select ID, LABEL from PRODUCT where LABEL='computer'

// Change the label of the product 234 to "book".
update table PRODUCT set LABEL='book' where ID=234

// Add a book with the id 567.
insert into PRODUCT (ID, LABEL) values (567, 'book')

// Delete all the products having an id < 100.
delete from table PRODUCT where ID < 100

// Create the product table.
create table PRODUCT (
 ID integer not null,
 LABEL varchar(255)
)
```

There are plenty of resources on the internet that explain SQL syntax. If you feel like a deeper explanation on SQL is needed, we suggest you the following sites:

- W3Schools
- Sql Zoo
- SqlCourse.com

1.4. Choosing your Database

There are tons of database products under various licenses out there and JPA can deal with any of them as long as they offer a JDBC driver to communicate with. During the exercises, we use HSQL DB, but you can choose any other DB:

- free: MySQL, PostgreSQL,…
- commercial: Oracle, DB/2,…

We choose HSQL because it requires no setup on the your PC, only a JAR to run (in some classrooms and companies, students have no administrator rights on their machines and cannot install software).

The exercises' solutions and the initial ready-to-import project that we provide include HSQL DB (embedded inside

the Java project). If you feel confident with another DB product, feel free to use it: the exercises adapt easily to another system.

1.5. Exercise: Project and DB Setup

In this exercise, we install HSQL DB (or another DB) and execute a few simple SQL statements. We also create a Java project with the necessary JPA/Hibernate jar files.

The setup part exists in two versions:

- the **easy** version where you import an existing project that includes all the needed jar files for Hibernate and HSQL DB.
- the **longer** manual version where you create a blank new project and download/select all the necessary jar files yourself.

Choose the version you prefer. You will end to the same result.

After the setup, we'll execute a few SQL commands.

1.5.1. Using Another DB Product

If you already have a RDB environment and feel comfortable with it, you may want to use it instead of HSQL DB. Follow any version (easy or long) of this exercise to setup the JPA/Hibernate stuff in the Java project. With the easy version, you'll get also HSQL, even if you don't need to it.

1.5.2. Easy Setup

The easy setup consists in importing a pre-packaged project.

The link below is a chapter of a generic common course that explains how to proceed. Install the Eclipse project, then come back here (the JPA/Hibernate Fundamentals course). You will have the choice between 2 zip files. Select the first: *Jpa**Simple**.zip*.

- Pre-packaged project installation

If you have finished the setup, you can skip the "Manual Setup" section, down to "Using HSQL".

1.5.3. Manual Setup

The manual setup lets you download, install the required libraries and create a new project yourself. If you have done the previous "easy setup" part, skip this "manual setup" part.

In this exemple we use HSQLDB but you can choose another database: PostgreSQL, Oracle, MySQL, ...

Create a new Eclipse project

- Launch Eclipse and create a new project (not a JPA project, but a regular Java project).
- Open File > New > Java Project, name it JpaSimple then click the Finish button.
- Add a "Lib" folder in your project

Setup HSQL Database access

- Outside Eclipse create a Temporary directory where you will receive the downloaded jar files all along the course
- Open the hsqldb.org website in your web browser
- Find the download page and download the latest released version in your temporary directory.
- Locate the hsqldb.jar file in the Lib sub-folder of the unzipped file and import it to the "Lib" folder of your project as created above.
- Modify the build path to include the imported jar file. This will enable the DB accesses.

In chapter 3.2 we shall see how to integrate Hibernate in your environment file.

1.5.4. Using HSQL

Follow these instructions to start the HSQL DB and execute a SQL statement on it.

Then come back to this course.

1.6. Referential Integrity

Foreign Keys and Primary Keys enable relationships between records in a relational database. Any RDB table must have a primary key. It may be a single field, as the ID in our dog and person tables below. It may also be composed of multiple fields. The primary key enables to uniquely retrieve any record.

A table may also refer another table. For example, a dog may refer its owner (person). The dog table would have a foreign key referring the person table. In every dog record, we store the id of the person.

If you are not comfortable with this concept, the following resources should help you:

- Introduction to Relational Databases (Database Journal)
- What are relational databases? (HowStuffWorks)
- Part of SQL for Web Nerds. Includes a very good introduction on why hell we need relational databases.

1.7. First Chapter Exam

Close this chapter by taking the following simple 5 questions RDB/SQL test.

Review the exam objectives (defining what you are supposed to be able to do) before taking it. Should you feel uncomfortable with any topic, it's a good time to review.

2. JDBC

Hibernate is built on top of JDBC, so when configuring and debugging applications using Hibernate/JPA it is somewhat useful to have at least a basic understanding of JDBC. If you feel comfortable with using the JDBC APIs and if JDBC Drivers, Connections, Statements hold no secrets for you, feel free to skip these instructions and jump to the next chapter or the JDBC exercise.

In the following chapter, you will learn:

- What JDBC is.
- How to get a connection to the database.
- How to send statements and retrieve resultsets to get/modify data with JDBC.
- Optionally, use explicit programmatic transactions.

In case you'd like to review the general principles of relational databases first, read the Relational Database overview.

2.1. Basic APIs

2.1.1. What is JDBC?

JDBC stands for Java Database Connectivity

Aimed to connect to relational databases, JDBC is a Java SE API. It can be used in standard Java applications and Enterprise applications (Java EE) as well.

The whole JDBC API consists in two packages. A class using JDBC typically will import both of them:

1. import java.sql.*;
2. import javax.sql.*;

Note: The hsqldb.jar contains the DB server software, **and** the client JDBC driver. Usually, in your Java project, you only add the JDBC driver jar file (i.e. *ojdbc14.jar* for Oracle or *postgresql-9.0-801.jdbc4.jar* for Postgresql 9.0).

JDBC and SQL

Using JDBC is all about sending SQL statements to the database and processing query results. Basically, the JDBC developer:

1. Gets a connection to the database
2. Manages transactions (optional)
3. Sends SQL statements to the database
4. Processes query results (optional)

On one hand, JDBC is easy to learn and use. But on the other hand, it needs a good grasp on SQL from the developer, and SQL experience as well.

That's one of the reasons for which it's a good practice to clearly separate DB access through JDBC calls from the business layer. We'll come back on that later.

JDBC is not the only way to access relational databases, but most alternatives (O/R mapping tools like JPA, Hibernate and TopLink) may use JDBC behind the scene.

Drivers & Vendor-Independence

Thanks to its architecture, JDBC is a vendor-independent technology: the same API will be used to access an Oracle, MS-SQL or IBM DB2 database, or even multiple different DB systems at the same time. JDBC mainly is a set of interfaces whose real implementations (the classes) are provided by a JDBC driver.

While the developer codes to interfaces, the driver executes code which is specific to the database in use.

Drivers offer the concrete implementation of the JDBC interfaces for a given database.

A JDBC driver carries the SQL statements embedded in your JDBC calls to the database you are connected to, using a proprietary communication API.

Take a look at the below code to get your first grasp of JDBC, the details will be explained later.

Source

```
import java.sql.*;
 // ...
 // 1° Loading the vendor-specific driver class
 Class.forName("oracle.jdbc.driver.OracleDriver");

 // 2° Getting a Connection
 // The parameters depend on your DB vendor
 Connection con = DriverManager.getConnection("jdbc:oracle:thin:@10.0.0.20:1521:xe

 // 3° Creating and executing a statement
 Statement stmt = con.createStatement();
 ResultSet rs = stmt.executeQuery("SELECT * FROM employee");

 // 4° Processing results
 while (rs.next()) {
   System.out.println(
   rs.getString("ID") + " | " + rs.getString("NAME"));
 }

 // 5° Closing the Connection
 con.close();
```

2.1.2. Getting a connection

Working with a database using JDBC starts with getting a connection to the database, which can be done in two ways: through the DriverManager class or through the DataSource interface. As both techniques have pros and cons and different constraints, they are not interchangeable though.

2.1.2.1. Opening

A database connection corresponds to an instance of a class which implements the java.sql.Connection interface.

There are two ways for getting a JDBC Connection. Through:

- java.sql.DriverManager
- javax.sql.DataSource

In both cases, the connection is obtained by a call to one of the overloaded getConnection() methods.

Using the java.sql.DriverManager class

Two-phases process :

1. Driver registration, either via the Java command line, by setting the jdbc.drivers system property. DriverManager then loads the drivers by itself.

   Source
   ```
   java -Djdbc.drivers=DriverClassName MyJavaApp
   ```

   Or by loading the driver(s) class(es) explicitly in your program:

   Source
   ```
   Class.forName("com.caucho.jdbc.mysql.Driver");
   ```

   Once loaded, the driver class creates an instance of itself and registers it with the DriverManager.
2. Connection reference

   Source
   ```
   Connection conn = DriverManager.getConnection("JDBC URL", "userid", "passwor
   ```

   The JDBC URL has the form jdbc:protocol:subprotocol
   Protocol: which database engine to connect to (ex: mysql)
   Subprotocol: provides driver-specific connection information (host, port number,…)

Note (optional detail): JDBC 4.0 drivers running with Java SE 6 don't need to explicitly register with the DriverManager by calling Class.forName(). The DriverManager has been enhanced to support the Java SE Service Provider Mechanism. SPM auto-registers services that provide configuration files stored in the META-INF/services directory. So JDBC 4.0 driver libraries must include the META-INF/services/java.sql.Driver file.

Using the javax.sql.DataSource interface

This is the preferred way, but it's generally available only in application servers. The DataSource way of getting a Connection is preferred over the DriverManager one, because typical implementations of the DataSource interface

include the connection pooling feature, which boosts performance when many clients connect to a database through an application server.

This code snippet shows you how to lookup a DataSource from JNDI and use it to get a Connection.

Source

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup( "jdbc/AcmeDB" );
Connection con = ds.getConnection( "genius", "abracadabra" );
```

2.1.2.2. Closing

Connection has a close() method used to release JDBC and database resources.

Notice though that when Connection.close() is called, all associated Statement instances are automatically closed as well, as are closed the ResultSet instances when their Statement is closed.

Database connections are limited resources. As with any such a resource, it is good practice to make sure that it is released, whatever happens (Exception, Error) after its allocation. This can be done with a try-finally construct.

Source

```
Connection con = null;
try {
   con = DriverManager.getConnection(...);
   // ...
   // use the connection
   // ...
} finally {
   // Whatever happens between try and finally, the
   // finally block is guaranteed to be executed.
   if (con != null) {
       try {
           con.close()
       } catch (SQLException e) {
           throw new RuntimeException(e);
       }
   }
}
```

If you want to get an extended explanation on how to close a JDBC connection feel free to read the following resources:

- Blog entry - How to Close JDBC Resources Properly - Every Time

2.1.3. Statements and ResultSets

JDBC provides three interfaces to send SQL statements:

- Statement
- PreparedStatement (statements with parameters)
- CallableStatement (to call stored procedures)

and one interface to retrieve query results : ResultSet.

This section briefly presents the main JDBC interfaces and classes, describes the use of the Statement interface and shows how to work with ResultSets.

2.1.3.1. Simple statements with java.sql.Statement

A statement instance is obtained from the Connection:

Source

```
Statement stmt = conn.createStatement();
```

If the statement is a query (SELECT) the executeQuery() method must be used. It returns a ResultSet object.

Source

```
ResultSet rs = stmt.executeQuery("SELECT * FROM EMPLOYEE");
```

A loop is then typically used to process the query results

Source

```
    while (rs.next()) {
        System.out.println("ID: " + rs.getInt(1) + ", " + rs.getString(2));
    }
```

Field values are retrieved through calls to getXXX(int) or getXXX(String) overloaded methods. There is one getXXX() methods pair per JDBC type supported, to give access to field values by position (starting with 1) or by name. The while loop above could have been written:

Source

```
    while (rs.next()) {
        System.out.println("ID: " + rs.getInt("ID") + ", " +
        rs.getString("NAME"));
    }
```

2.1.3.2. Parameterized queries with java.sql.PreparedStatement

PreparedStatement is a sub-interface of Statement. It offers three main advantages over Statement:

- It is sent to the database engine at creation time in order to be pre-compiled, which improves performance in case of reuse of the same statement.
- It accepts parameters, which makes the code easier to write, read, maintain and enforces security by reducing the risk of SQL Injection attack
- Most data sources supporting connection pooling also support prepared statement pooling. When such a prepared statement is closed, it returns in the pool for later reuse.

Example:

Source

```
PreparedStatement stmt = conn.prepareStatement("DELETE FROM EMPLOYEE WHERE ID =
stmt.setInt(1, 100); // sets the first parameter
stmt.executeUpdate();
```

PreparedStatement may be used for UPDATE as well as INSERT, SELECT, DELETE. The question marks ('?') in the statement act as parameters placeholders. When used in a loop, a call to clearParameters() clears all parameters values.

We've reviewed the PreparedStatment interface. Hibernate, a popular JPA engine, uses them behind the scene. You will not use them directly, but see them in the log files. The following (very optional) resources will give you more information:

- Use PreparedStatements in Java
- Why Prepared Statements are important and how to use them "properly"

2.2. Transactions (Optional)

Most of database engines support transactions. After a recall of what transactions are and the ACID requirements, you'll find there

- How to solve possible issues in relation with concurrency
- How to deal practically with transactions from JDBC

Transactions are related database operations grouped together in a logical unit of work. The database must ensure that the whole sequence of operations carried by the transaction either completes or fails.

To check if your database supports transactions, you must, follow these steps :

1. Get a Connection Object

2. Get an instance of java.sql.DatabaseMetaData from the connection

3. Ask that object wether the db supports transactions as in :

Source

```
Connection con = DriverManager.getConnection(bla bla bla);
java.sql.DatabaseMetaData dbmd = con.getMetaData();
System.out.println( dbmd.supportsTransactions() );
```

2.2.1. ACID requirements

ACID stands for Atomicity, Consistency, Isolation, Durability, the four properties that a transaction ideally must guarantee.

- Atomicity: indivisible unit of work
- Consistency: the data state remains consistent after the transaction completes (relations included)
- Isolation: the transaction ignores other transactions
- Durability: after a commit, changes are permanent, even in the event of a system failure

2.2.2. Concurrency

Concurrency of a system measures the ability to grant concurrent access to shared resources to multiple users.

As database transactions use locks (especially to guarantee data consistency and isolation), strictly fulfilling the ACID requirements (and especially the isolation one) goes against concurrency, since DBMS needs to block operations on locked data to prevent other transactions to affect them. This can increase the possibility of a deadlock situation, where two or more competing actions are each waiting for the other to finish. In other words, isolation and concurrency are conflicting elements of ACID requirements.

There are several levels of isolation defined by the ANSI/ISO SQL standard:

- Serializable: This isolation level specifies that all transactions occur in a completely isolated fashion;
- Repeatable Read: All data records read by a SELECT statement cannot be changed;
- Read Committed: Data records retrieved by a query are not prevented from modification by some other transactions;
- Read Uncommitted: One transaction may see uncommitted changes made by some other transaction (this isolation level allows *dirty reads*).

More about isolation and deadlock concept can be found on Wikipedia articles.

Two best practices may help to increase database concurrency:

- keeping the transactions as short as possible;
- breaking a little the isolation ACID requirement, by setting the isolation level at its least acceptable value in the transaction context.

2.2.3. JDBC Transaction Management

By default, each SQL statement forms a single auto-committed transaction.

Managing transactions by code means:

- Setting off the auto-commit feature:

Source

```
// this line automatically starts a new transaction
conn.setAutoCommit(false);
```

- Executing multiple statements ...

- And finally either committing or rolling back the transaction:

Source

```
if (everythingWasOK) {
 conn.commit() ;
} else {
 conn.rollback() ;
}
```

The recommended way of wrapping this code safely is to use a try-catch-finally construct.

This is a better way of writing the above code:

Source

```
boolean everythingWasOK = true;

try {
   conn.setAutoCommit(false);

   ... do interesting stuff
   ... if there are any functional problems just set the everythingWasOK to false

} catch ( SQLException e ) {
   everythingWasOK = false;

   // log the exception
   log.error( e );

   // re-throw the exception
   throw se;

} finally {
   try {
       if ( everythingWasOK ) {
           conn.commit() ;
       } else {
           conn.rollback() ;
       }
```

```
            conn.close();
        } catch ( Exception e ) {
        }
    }
```

Sure this is quite a bit more verbose but it is a proven way of making sure that everything gets cleaned up properly.

More about JDBC transaction can be found in the JDK Javadoc

2.3. Exercise

The goal of this exercise is to have you practice JDBC and to make sure the necessary driver library is present in your project classpath.

This exercise requires you to have finished the Database Setup Exercise done in the previous chapter.

2.3.1. Get database metadata

Steps:

- Open your project and create a new java class called JDBCMain in the blackbelt.main package
- Make it an executable class by adding a main(String[] args) method
- Write code that opens a new JDBC connection to your database
  You'll need the HSQL driver name and connection string. Hint: The DatabaseManagerSwing application (see previous exercise) shows correct values when starting up (don't forget to select "server" in the combo, to get the correct value).
- Using the connection, retrieve the database metadata (Hint) and display the database name and version number. This will verify that we have a valid connection to the server

Source

```
...
DatabaseMetaData dbdata = connection.getMetaData();
System.out.println(dbdata.getDatabaseProductName() + " " + dbdata.getDatabaseProdu
```

- Execute your code

2.3.2. Execute a Statement (Optional)

Modify the code written in the previous step

- Create a new PreparedStatement object to insert a new record in the PRODUCT table created in the DB Setup Exercise.
- Reuse the statement object to insert 3 new records
- Execute your code and verify using the Swing DB Client that the records were correctly inserted

2.4. Exam and Resources

To validate your understanding of the basic JDBC APIs take this little quiz. When passed continue with the next chapter.

If you want to deepen your knowledge of JDBC, you may read the following:

- Sun Microsystems Official JDBC Tutorial
- One page introduction on JDBC, basic APIs and Drivers

3. Hello World

This chapter describes the minimal requirements to make a first JPA application work. You will make a first JPA example work during this chapter. You will create a Product class and map it to the PRODUCT table in your database. You will create a product object in memory and store it in the DB.

A TechMap gathers these elements into one diagram. Refer to this diagram and try to identify each element while reading this chapter.

Click to download

3.1. Entities, EntityManager, Persistence Unit

Before going any further make sure you understand the TechMap diagram and identify the following elements:

- the PRODUCT table, in the DB
- the class Product
- the class Main, using the EntityManager
- the persistence.xml file

When done, read the following sections that describe some JPA classes. JPA is a specification. To use it, you need a product implementing this specification. Hibernate is such a product. The necessary jar files represented on the diagram can be downloaded from the Hibernate.org website.

3.1.1. Entities

You turn your domain model's classes into “Entities” managed by JPA. Entities are mapped to RDB tables.

For example, the Java Product class will be mapped to the PRODUCT table. Each attribute (id, label, price,…) of the Product class will have a corresponding field in the PRODUCT table. JPA will automatically transfer data between the instances of the Product class and records of the PRODUCT table.

The @Entity annotation marks a class as a JPA managed entity.

Source

```
@Entity
class Product {
        @Id
   private Long id;
   private String label;
   …
}
```

3.1.2. EntityManager

The EntityManager interface provides the main methods to interact with JPA:

- find (retrieves an entity from its id),
- persist (inserts a new entity in the DB table),
- merge (updates an existing record or insert a new one),
- getTransaction (obtains an object to start and commit transactions),
- createQuery (prepares a query for retrieving entities that match some criteria).

The EntityManager is obtained from an EntityManagerFactory, which is configured via a persistence.xml file.

One EntityManager instance corresponds (and refers) to one JDBC connection.

3.1.3. Persistence Unit

The *persistence.xml* file must be in a directory (folder) named "META-INF" in the top level source directory of the classes it will apply to. In a standard Eclipse project it should go in the "src" folder beside the top level package folders.

When you make a "JPA Project" in Eclipse, as opposed to a standard Java Project, the META-INF and *persistence.xml* are made in the correct locations for you so you could make one to study the correct layout of folders. Putting the *persistence.xml* file in a wrong place is a common cause of problems for people adding Java Persistence to a standard project, so a "known good" reference can help.

The three key elements of a persistence unit definition are:

- its name
- the database connection parameters
- JPA implementation options

The persistence unit name is referenced when making an EntityManagerFactory object in your Java code.

The database connection is handled by JDBC so the parameters are the expected ones:

- driver class, as a fully qualified path like "com.db-firm.jdbc.Driver"
- connection url
- username
- password

Implementation options vary with each provider (Hibernate, TopLink etc.) but it is common to give the

JPA dialect.

This example uses the HSQL database configured as a standalone server, as provided in the downloadable code examples. In this mode it is possible to use the HSQL Connection Manager application connected to the server while you execute your code in Eclipse for testing.

Source

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
   http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
  version="2.0">

  <!-- unit name -->
  <persistence-unit name="blackbeltunit" transaction-type="RESOURCE_LOCAL">
    <description>using HSQL DB in standalone server mode</description>

    <provider>org.hibernate.ejb.HibernatePersistence</provider> <!-- optional -->

    <properties>

      <!-- database connection -->
      <property name="hibernate.connection.driver_class"  value="org.hsqldb.jdbcDr
      <property name="hibernate.connection.url"  value="jdbc:hsqldb:hsql://localho
      <property name="hibernate.connection.username" value="SA"/>
      <property name="hibernate.connection.password" value=""/>

      <!-- JPA parameters -->
      <property name="hibernate.archive.autodetection"  value="class"/> <!-- optio
      <property name="hibernate.hbm2ddl.auto"  value="create"/>
      <property name="hibernate.dialect"  value="org.hibernate.dialect.HSQLDialect
     </properties>
  </persistence-unit>
</persistence>
```

You will also need to have a hsqldb.jar file on the project's class path.

To use HSQL with a flat file you need change only these parameters:

Source

```
 <properties>
  <property name="javax.persistence.jdbc.url" value="jdbc:hsqldb:file:hsqldb/testd
  <property name="javax.persistence.jdbc.user" value="SA"/>
  <property name="javax.persistence.jdbc.password" value=""/>
  <property name="javax.persistence.jdbc.driver" value="org.hsqldb.jdbc.JDBCDriver
```

and make sure there is a folder "hsqldb" beside the "src" folder in the top level directory of your project.

In this mode HSQL is faster, but can only support one connection at a time, so you will have to close the HSQLDB administration tool before launching your program.

For using PostgreSQL as the database the persistence file could be:

Source

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
   http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
  version="2.0">

  <!-- unit name -->
  <persistence-unit name="blackbeltunit" transaction-type="RESOURCE_LOCAL">
    <description>using a PostgreSQL server</description>

    <provider>org.hibernate.ejb.HibernatePersistence</provider> <!-- optional -->

    <properties>

      <!-- database connection -->
        <property name="hibernate.connection.driver_class" value="org.postgresql.D
        <property name="hibernate.connection.username" value="myuser"/>
        <property name="hibernate.connection.password" value="mypassword"/>
        <property name="hibernate.connection.url" value="jdbc:postgresql://localho

        <property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQL
        <property name="hibernate.hbm2ddl.auto" value="create" />
      </properties>
  </persistence-unit>
</persistence>
```

Another very common database for small scale use is H2. It is versatile, fast and easy to use.

This is the properties section of a persistence.xml configured to use it as a standalone server.

Source

```
<properties>
  <property name="hibernate.archive.autodetection" value="class" />

  <property name="hibernate.connection.url" value="jdbc:h2:tcp://localhost/jbb-jpa
  <property name="hibernate.connection.driver_class" value="org.h2.Driver" />
  <property name="hibernate.connection.username" value="SA" />
  <property name="hibernate.connection.password" value="" />

  <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect" />
  <property name="hibernate.hbm2ddl.auto" value="create" />
</properties>
```

Note that spelling and case are critical in these parameters. A common mistake when using MySQL is to write ... *"hibernate.dialect" value="org.hibernate.dialect.MYSQLDialect"* when the dialect name should be "...dialect.M**y** SQLDialect"

The resulting error messages are about not finding the persistence.xml or other issues, which can be misleading.

3.1.4. JavaDoc

The classes, interfaces and annotations you have used are documented by Sun in the JEE part of the JavaDoc.

3.2. Hibernate setup

Skip this paragraph if you have chosen the easy setup in chapter 1.5.2.

3.2.1. Hibernate libraries and dependencies

Use your temporary directory defined in chapter 1 to receive the downloaded files. The versions change frequently so we mention them here as x.y.z

From hibernate.org, locate the download page. Click on the words "release bundles" in blue. Choose the latest release of Hibernate3. Download the ZIP file into your temporary folder. Unzip it and collect the following jar files from this downloaded zip :

- From the "hibernate-distribution-x.y.z.Final" folder : **hibernate3.jar**

- From the "hibernate-distribution-x.y.z.Final\lib\required" sub-folder : **all the jars (since they are required :))**

Download the JPA 2 API from the JCP website.

- Click the link download page for the final release

- Click the second download button to "download the specification for building an implementation"

- Agree their agreement

- take the *JSR-317 Persistence Final JAR and SCHEMA*

- From the zip file you will need *javax.persistence_2.0.0.jar*

Download the most recent SLF4J from www.slf4j.org.

Extract the slf4j-log4j12-x.y.z.jar file from this zip.

This file is mandatory to enable the Hibernate logging

From the website logging.apache.org/log4j, download the recent log4j.jar

3.2.2. Create a new Eclipse project

Launch Eclipse and create a new project (not a JPA project, but a regular Java project).

- Open File > New > Java Project, name it JpaSimple then click the Finish button
- Import the jar files you just downloaded
- Open the build path configuration: right click on the project, select properties; select “Java Build Path” in the left column, then open the library tab
- Click the Add JARs button, and add all the jars you just imported
- Create the folder "META-INF" in the source folder "src"
- Create the file "persistence.xml" in the folder "META-INF"
- Configure your persistence.xml

Note: You can have this message:

Source

```
log4j:WARN No appenders could be found for logger (org.springframework.context.sup
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info
```

This message occurs because the log4j part is not configured. Do not worry about this warning at this moment, it will be explained later.

3.3. Hello World Exercise

First of all, make sure your Java project contains the necessary jar files for:

- hibernate
- hsqldb

You will now create your own Java code.

Reproduce the example from the TechMap on the Hello World chapter page and make it work on your machine. You will have to create:

- a PRODUCT table,
- a Product class,
- a Main class using the EntityManager,
- a persistence.xml containing your persistence unit.

You may adapt some values to your environment (such as the DB connection values).

Note that the @Id, @GeneratedValue and @Column annotations are covered in a later chapter. Just copy them from the TechMap.

Make the main program work with no exception and check that the database contains the expected data.

3.3.1. FAQs

Here is a short list of questions frequently asked by students at the end of this exercise:

Why are there several versions of the same annotations?

Sometimes, there is a version from the javax.persistence package and another from an org.hibernate... subpackage. Always prefer the javax.persistence one, as that is the standard one. Otherwise, it probably won't work.

Why is the product inserted with no id?

The DB was manually generated in the first chapter's exercise. Hibernate tries to call the identity stored procedure which does not exist on that DB. When Hibernate creates the DB Schema itself, it generates the necessary elements in order to be able to assign id values to newly created entities. This may be a sequence, a stored procedure or something else, depending on the type of database you use and the generator strategy defined in your entities.

To **solve** the problem, let hibernate create/update the schema for you by setting the hbm2ddl.auto property to either CREATE or UPDATE (see Techmap). In some cases (e.g. when switching from a join table to a foreign key column), you may have to drop certain tables manually.

3.4. Javadoc Exercise

To reinforce your understanding of the base JPA classes, please execute the following ...

Locate the JPA JavaDoc. For any official specification JavaDoc, the site java.sun.com is a good start. Look in the Java EE section (not on the Java SE side).

Search the answers to these questions in the JPA JavaDoc (javax.persistence package):

- Which method of the EntityManager interface is used to load an entity (a POJO) from its given primary key ?
- Which method from which interface would you call to delete a row from the DB (delete an entity) ?
- What does the @Transient annotation mean ?

3.5. Database Integration

3.5.1. Show SQL

Some Hibernate options in the persistence.xml file enable the SQL generated by Hibernate to be logged on the console.

Source

```
<property name="hibernate.show_sql" value="true"/>
<property name="hibernate.format_sql" value="true"/>
<property name="hibernate.use_sql_comments" value="true"/>
```

Note that these properties are documented in the official hibernate documentation on Hibernate.org.

In the documentation section, take the documentation named "Hibernate Core for Java", and search (crtl + F in the browser) for a property name as "hibernate.show_sql".

3.5.2. Show SQL - Try it yourself

Add the appropriate lines to your persistence unit in order to:

- show the SQL generated by Hibernate,
- nicely formatted (human readable),
- with comments inserted

Run your project again and analyze the SQL generated by Hibernate. Does it match what you asked JPA/Hibernate to do?

3.5.3. DB Generation

Hibernate can produce the CREATE TABLE statements in SQL to generate your DB, based on your entities. If you have one Product entity, hibernate will create a PRODUCT table for you, according to the mapping information.

Source

```
<property name="hibernate.hbm2ddl.auto" value="create"/>
```

The property hbm2ddl (Hibernate Mapping to Data Definition Language) can be changed to:

- **validate:** does not affect the DB.
- **create:** empties the DB, then creates all the tables when the EntityManagerFactory is opened.
- **create-drop:** drops all the DB's tables when the EntityManagerFactory is opened and again when it is closed.
- **update:** add new fields and tables when the EntityManagerFactory is opened.

3.5.3.1. Usage during the exercises

The create option is convenient because it re-creates your DB every time to retest your application. When your application is ended, your DB is still there should you need to inspect it with SQL. You'll probably use this option when your application instantiates and persists some entities at startup to populate the DB with some test data.

The create-drop option is not very convenient, because you cannot inspect your data after your application completes, because all the tables have been dropped.

The update option keeps your existing DB and create/alter tables to make room for entities/attributes that you would have created since the last update of the DB schema. It’s convenient to keep your test data, while having a DB schema adapted for your modified entities.

3.5.3.2. Usage during real-world development.

Most real development only uses the validate option.

You probably work with a test DB full of test data. It's typically a copy of the production DB (an older version of your application is in production). It makes the create and the create-drop options unusable.

Don't use update, because it does not drop tables/fields for removed entities/attributes.

In fact, you will write (with the help of hibernate, don't worry), SQL scripts that will migrate your DB to the next version of the schema structure. That SQL script will create tables, alter tables to add/remove fields, and probably initialize new data. The SchemaExport tool from Hibernate will help you to generate SQL between two versions of your java code.

3.5.4. Log4j

Hibernate uses the log4j library to log messages.

Place a file named log4j.properties in the root of your source ('src') folder.

If you want to display the SQL commands for table creation, use the level 'DEBUG' else use 'ERROR'.

In the hereunder sample, the messages will be written to the console with logging at 'Error' level:

Source

```
### Console appender(used in dev)
#log4j.appender.Console=org.apache.log4j.ConsoleAppender
#log4j.appender.Console.layout=org.apache.log4j.PatternLayout
#log4j.appender.Console.layout.ConversionPattern=%-5p %d{dd MMM yyyy HH\:mm\:ss} [

# initialize root logger and call it "Console"
#  change here to the desired log level ("ERROR", "DEBUG", etc.).
log4j.rootLogger=ERROR, Console
# add a ConsoleAppender to the logger Console
log4j.appender.Console=org.apache.log4j.ConsoleAppender
# set set that layout to be SimpleLayout
log4j.appender.Console.layout=org.apache.log4j.SimpleLayout
```

In the level 'DEBUG', you should see more informations, and also the 'create table':

Source

```
.....
DEBUG -
   create table BankAccount (
       id bigint generated by default as identity (start with 1),
       balance double not null,
       number varchar(255),
       primary key (id)
   )
```

```
INFO - schema export complete
DEBUG - Checking 0 named HQL queries
DEBUG - Checking 0 named SQL queries
.....
```

Without this configuration file, you will have the warnings mentionned in chapter 'Hibernate setup', §3.2.2 :

Source

```
log4j:WARN No appenders could be found for logger (org.springframework.context.sup
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info
```

For more information, see http://logging.apache.org/log4j/.

3.5.5. DB Generation - Try it yourself

In your application, set the DB generation value to “create-drop”.

Execute your application. When it’s completed, look in your DB, the tables should be gone (with HSQLDB, hit ctrl + R to refresh the DB tree in the DataManager GUI).

Change the value back to “create”, run your application again and check the DB. Your tables should remain after your program’s completion. Similarly to the show SQL part, you'll find a very limited explanation in the official Hibernate documentation. Search for "create-drop" in the single html file.

Hint: create-drop does not drop the table!?
You must explicitly close the EntityManagerFactory to trigger the drop.

Hint: some tables are not dropped!?
Only the tables corresponding to mapped JPA entities will be dropped by Hibernate, because Hibernate created them.

3.6. Exam and Resources

Close this chapter by taking the following JPA - Hello World test.

Review the exam objectives (defining what you are supposed to be able to do) before taking it. Should you feel uncomfortable with any topic, it’s a good time to review.

Read the following articles if the summary and diagram aren't clear enough, or you want more in-depth information.

- Taking JPA for a Test Drive
  Oracle is a very valuable contributor in the JPA space. This very good article is an in depth overview of the technology (going much further than what we have done here in this chapter).
- Master the New Persistence Paradigm with JPA - DevX
  Another nice introduction

- Understanding JPA, Part 1: The object-oriented paradigm of data persistence - JavaWorld
  High quality introduction to JPA.
- Gentle Introduction to JPA
  Very good text explaining an simple example. Don't worry about the "Query" part yet, we'll cover it later.
- JavaBeat JPA Tutorial
  Good resource that probably goes too far at this point.
- RoseIndia JPA Tutorial
  If you don't mind invasive Google ads, you may find some additional information.
- Sun JEE Tutorial / JPA section.
  This is a quality resource. Don't read too far, because it covers theory we still have to see. We suggest:
  - Requirements for Entity Classes (in the Entity part)
  - Persistent Fields and Properties in Entity Classes
- GlassFish Java Persistence Example
  A good example kept simple. You don't have to download/install anything, just read the text explaining the code fragments.

4. JPA Concepts

This chapter is quite theoretical. Until now, we focused on making JPA/Hibernate work. It's time to put some background and perspective to JPA. Why would you want to use JPA anyway?

4.1. DB History

Old mainframe flat DBs had no field structure and were not relational. Files had the notion of lines (records) and programs had to cut the records themselves into “fields”. The responsibility of knowing the structure in fields was not in the DB.

Relational DBs were a big step forward and many mainframe applications have been ported to (mainframe) applications using a RDB. The RDB knows the field structure and field types, as numeric, string and date. They also enforced relationship constraints (foreign key – primary key) between tables.

Because RDBs had the notion of structure, SQL (1974) could be invented as a standard way to read/write data.

OO programs representing data in memory as objects had mismatches with RDBs. OODBs can directly persist memory objects having the notion of inheritance and the notion of “pointer” to other objects. This concept of “DB for graph structured objects” has been invented in the mid-1970s and commercialized since the 1990s, but never approached the adoption success of RDBs.

RDBs are de-facto standard and default choice for any new typical development project.

4.2. RDB Data in Your Program

Typically in a mainframe Cobol application, the structure of the data in RAM is identical to the structure in the RDB.

In the 1990’s, many applications had a client-server architecture, with Windows clients and an RDB server. The client languages (Visual Basic, PowerBuilder, Delphi,…) implemented the RAD (Rapid Application Development) principle. It mainly took the form of GUI widgets (listboxes, input fields, radio buttons,…) that were aware of the DB structure. It

was common to associate a SQL select statement to a listbox, for example. This approach is particularly efficient if your RDB structure is stable and your application mainly performs CRUD (Create Read Update Delete) operations (basically, editing the RDB's data).

Applications with a lot of business logic often manipulate the data in RAM, outside of the GUI's simple CRUD. With OO languages, you create a domain model, which is a set of classes representing the data. The main challenge is to populate the objects in RAM with records from the RDB and vice-versa.

4.3. Paradigm Mismatch

Your data lives in your RDB and in your OO objects. But these two structures don't work in the same way:

- **Granularity:** The domain model in Java (your entities), usually contains more (and smaller) classes than your DB contains tables. Your data model in Java is finer grained. For example, a Customer object can reference an Address object. In the DB, they might be combined in the same CUSTOMER table containing fields from the Address object.
- **Subtypes:** Inheritance might be used in your Java domain model. For example, the CompanyCustomer and the IndividualCustomer classes could have a common Customer ancestor, with common attributes. This notion does not exist in RDB.
- **Identity:** In memory, objects are identified by their reference which can be compared with the == operator. The .equals() method can also be used to check the equality of two distinct objects. In a RDB, records are identified by their primary key.
- **Associations:** An object can refer to another object via a “pointer”. A DB record can refer to another record via a foreign key.
- **Navigation:** It is possible to navigate the object graph by following pointers (references). In the OO model, some associations can be unidirectional. In RDBs and SQL, either multiple selects, either joins need to be performed to navigate between records. RDBs have no real notion of directionality.

4.4. Manual Solution to the Mismatch

Early in the (21st) century, a Java programmer typically had to write JDBC code with SQL statements. For example, the ProductDao class contained the code to read products from the RDB. The ProductDao.findUnsoldProducts(int days) method contained an SQL select statement to find the products with no sales the last n days. Then the method had a JDBC result set with the selected records. Finally, that data had to be copied into newly instantiated Product objects.

Let’s see how this kind of manual JDBC coding handled the various paradigm mismatch problems described above:

- **Granularity & Subtype:** This was no big deal. One ResultSet row can be transferred into one or two bound objects. An easy solution is to design/adapt the OO domain model to closely match the RDB structure (one class per business table).
- **Identity:** This was usually badly or not managed by the manual JDBC code. For example, a method can return the products with their associated suppliers (instance of the Supplier class). Another method just called after the first, would return the top 5 providers. If some providers were returned by both queries, there would be 2 instances of the same provider in memory.
- **Association:** The mapping code had to manually create a (sub)graph of object from a ResultSet of a join SQL select. For example, a method returning the products unsold for the last n days, with the product’s supplier, would return a list of product with a supplier attached to each product. One SQL select joining the PRODUCT and SUPPLIER table would retrieve the needed data.
- **Navigation:** That’s probably the main problem, unsolved by manual coding. Our ProductDao.findUnsoldProducts(int days) method, could be called from multiple places in our program. One of the places may need to get the products and the supplier of each product. If our method only returns the

product, that caller has to call SupplierDao.findById(long id) for each provider. If we talk about 100 products, that makes 1 SQL statement executed to find the list of product + 100 SQL statements to find the suppliers (known as the “n+1 problem” explained in details in the course *JPA/Hibernate Lifecycle*).

A solution would be to create 2 methods in ProductDao:

- findUnsoldProducts(int days)
- findUnsoldProductsWithSupplier(int days)

This leads to the multiplication of DAO methods (with different fetching plans).

Another problem is the interpretation of data: if a business method having a reference to a product, sees that product.getSupplier() == null, what does it mean ?

- There is no supplier in the DB for that product?
- Or the supplier has not been loaded in memory for that product?

If you start solving that kind of problem, you probably start writing an O/R mapping framework as Hibernate.

4.5. JPA Solution to the Mismatch

JPA brought the following solutions:

- **Granularity:** JPA provides the notion of “Embeddable”. Some classes (as Address) can be embeddable instead of entities. They are bound to an entity (as Customer).
- **Subtypes:** The developer can select one of the three strategies, according to the number of RDB tables he has for storing the data of the OO hierarchy.
- **Identity:** JPA provides a way of defining auto-generation for the ID column of your tables. Check out the @GeneratedValue annotation if you want to know more
- **Association:** JPA will automatically transform foreign-primary key relationship into pointers relationships. It will automatically perform the necessary SQL joins when needed.
- **Navigation:** With lazy loading, JPA will transparently perform the SQL queries to populate the object graph as you navigate it. It reduces the amount of DAO method you write, as much of the data is gathered through navigation.

4.6. JPA's History

EJB version 1 introduced entity beans as one of the first Java Object-Relational Mapping (ORM) solutions. It was created top-down by Sun and was unusable due to complexity. Several third-party solutions appeared to meet the need for a persistence solution, like Java Data Objects (JDO) and Hibernate (pre v3.2).

JPA was first mentioned in the EJB3 specification released with JEE v5 in 2006. This was an attempt to merge concepts from the third-party solutions (Toplink, JDO, Hibernate etc.) and EJB v2 Container Managed Persistence (CMP) to create a better way to do ORM. A significant advantage of JPA was that it did not require the use of an EJB entity bean container. This made JPA *much* more usable than EJB v1 or v2 persistence models.

The Java community seized on JPA as a standard and new versions of Hibernate, Toplink Essentials, among others, moved to deliver improved ORM tools. These tools have been widely accepted and have steadily improved, with spin-offs to .Net technology (nHibernate) and other OO development platforms.

In December 2009 JPA version 2 was approved, now as a stand-alone specification separate from EJB. It introduced

more features from the third-party ORM tools that could not be included in JPA 1 due to lack of consensus. JPA 2.0 has spawned new products, or versions, from DataNucleus (formerly JPOX), EclipseLink (formerly Oracle TopLink), JBoss Hibernate, OpenJPA, and IBM.

4.7. Exam and Resources

Now that you know it all, it's time for a little quiz. If you don't feel ready yet, get a little more study time in.

Further Reading

On the web, Wikipedia is a good start:

- O/R Mapping
- Impedence Mismatch

Ted Neward has written an extensive and popular article on the mismatch: The Vietnam of Computer Science.

5. Entity

During this chapter, we'll dig into the most used mapping options. Relationships are covered in a later chapter.

The following resources will help you out with this chapter's material:

- Oracle TopLink JPA Reference ... even though we use Hibernate rather than TopLink, this is a good resource, as TopLink is the reference implementation.
- JPA JavaDoc

5.1. Implementation

The following sections describe the usage of the @Entity, @Table, @Column, @Id, @GeneratedValue annotations.

5.1.1. Entity Example

Source

```
@Entity
@Table(name="PRODUCTS")
public class Product {
   @Id  @GeneratedValue
   private Long id;
   @Column(name="PR_LABEL")
   private String label;
   //… getters/setters/hashcode/equals …
}
```

5.1.2. @Entity

The @Entity annotation is mandatory for each JPA entity.

When the EntityManagerFactory is created, the Java packages specified in the persistence unit are scanned. JPA searches the classes with the @Entity annotation to register them and check the other mapping annotations.

5.1.3. Constructor

The entity class must contain a public or protected (not private) no-argument constructor to enable JPA/Hibernate to instantiate the entity.

In the example above, it will be generated automatically by Java because there is no other constructor defined.

Implementing the Serializable interface is not required unless your entities will be used in an application that will run over more than one virtual machine (like a clustered environment).

5.1.4. @Table

Use this annotation to change the table name. Without it, the class name is taken as table name. You can also specify an RDB schema name.

It is especially useful, when:

- Your DBA imposes table names (short, or with DBA conventions) that would not be nice looking in Java
- Your domain class name is an SQL (or DB) reserved word. A typical example is Order (which is used in "ORDER BY" SQL clauses).

Source

```
@Entity
@Table(name="ORDER_DUMB_POSTIX");
class Order {
  ...
}
```

5.1.5. @Column

Used to change the column name, which is the attribute name by default. It can be placed directly on attributes or on getters, but it’s more modern on attributes.

You can also define some constraint of the corresponding DB field:

- nullable = true/false (default to true)
- unique = false/true (default to false)

These constraints could be specified in your DB structure, but omitted in your Java code. It's convenient to put them also in your Java code as reminder for the programmer, and in case you ask Hibernate to generate (part of) your DB structure.

Source

```
@Column(name="barcode", nullable=false, unique=true)
```

5.1.6. @Id

5.1.6.1. Mandatory

Each entity must have at least one field annotated with @Id. It's the corresponding primary key in the DB table.

The primary key field must be one of the following Java language types:

- Java primitive types
- Java primitive wrapper types
- java.lang.String
- java.util.Date (the temporal type should be DATE)
- java.sql.Date

5.1.6.2. Surrogate key

We recommend using a surrogate key (instead of a business key) called id and of type Long.

Source

```
@Id  @GeneratedValue
Long id;
```

If you are not sure about the difference between a business key and a surrogate key, these two articles offer high quality discussions comparing these two strategies:

- Choosing a Primary Key: Natural or Surrogate? - Scott Ambler
- Wikipedia

From the Java point of view, we clearly favor surrogate keys, unless you have a legacy reason.

This lets you write more generic (and less) code. For example, finding entities by primary key can be handled by a base DAO, or equals and hashcode methods can delegate to a specialised class.

5.1.6.3. Null

When the entity instance has not yet been inserted in the table, the id is null. If the entity has already been inserted in the DB (is read from the DB for example), then the id should not be null.

If we had defined the id as a primitive *long* (instead of a wrapper *Long*), the id could never been null. In such case the convention is that a long id == 0 means no id. But it's more clear/explicit with Long/null.

5.1.6.4. @GeneratedValue

With the annotation @GeneratedValue, JPA will automatically assign a value when you persist the entity for the first time. By default, this value is a number taken from a DB sequence, common to all the tables of your application.

If the id is generated, then you don't need a setter (setId() method) because Hibernate will directly assign the private attribute (doesn't need the setter to do that), and you don't want your business code the change the id.

With business keys, you have to assign a value yourself to new entities before you call EntityManager.persist(entity).

A warning to PostgreSQL users

PostgreSQL defaults to a sequence to generate its primary keys. If you are having problems with auto-generated primary key values, modify @GeneratedValue:

Source

```
@GeneratedValue(strategy=GenerationType.SEQUENCE)
```

In certain cases (name collisions, reserved words...), you may also have to use the generator attribute to specify a name for the sequence that will be created. Otherwise, the JPA provider will give the sequence a default name.

5.2. Exercise

Do not hesitate to experiment with any annotation you like or think you might need the advanced features of on your project. This exercise should familiarise you with the usage of business and surrogate keys.

Define an entity with a business key instead of a surrogate key. You probably don't want to do that in a project unless you are forced to do so, but let's do it here to understand keys better.

- Create the entity: a class in your domain model. For example, a User that has a name and an e-mail. The eMail field is the business key.
- Create a new instance of User and fill the name, but not the e-mail. Try to persist it. Why do you get an exception ?
- Fix your code in the main method (don't change the User class) to make it work.

Hint: In some DB product (e.g. postgresql), "user" is a reserved word and cannot be used as a table name, it is also true for "Order". A possible solution is to use the plural words.

Optional Exercise

Use a composite business key. For example, in the User class from the previous exercise, the key could be firstName and lastName together, instead of the e-mail address.

Hint: Interface implemented by the business key class. The business key class should implement Serializable.

5.3. Exam

Once you feel confident (at the basic level of course), take this delicious (basic) exam.

6. Relationships

This chapter covers simple foreign-primary keys relationships, which are most common in typical applications. The relationship diagram gathers basic and more advanced elements into one diagram. Refer to this diagram and try to identify each element while working through this chapter. Look at the three first examples (at the top of the diagram): Dog, Car, House. The other example are explained in the *JPA/Hibernate Mapping* course.

Click to download

6.1. Unidirectional Relationships

6.1.1. ManyToOne

Source

```
public class Invoice {
   @ManyToOne
   Customer customerWhoPays;
   ....
}
```

Many Invoices can refer the same customer in this example. In the DB structure, the Invoice table has a foreign key to the customer table. This foreign key field is named the JoinColumn in JPA. By default there is only one JoinColumn, which name is the same as the attribute name (assumed Customer's primary key is only composed of one field).

6.1.2. OneToMany

Source

```
public class Supplier {
   @OneToMany
   @JoinColumn
   Set<Product> products = new HashSet<Product>();
   ....
}
```

One supplier provides many products (and one product is provided by only one supplier). This relation is also represented by a foreign key, but in this case, in the other (Product) table. JPA maps this relationship to a JoinColumn named supplier (in the PRODUCT table).

Note, that @JoinColumn is optional, but, if you don't specify it, Hibernate creates a join table between the tables (between supplier and product), which is probably not what you want in a one-to-many relationship.

6.1.3. OneToOne

Use the @OneToOne annotation for one-to-one relationship. It is very similar to the @ManyToOne case.

6.1.4. JoinColumn

You can change the name of the JoinColumn with the @JoinColumn annotation.

This is the DB column that stores the foreign key.

Source

```
public class Invoice {
   @ManyToOne
   @JoinColumn(name="customer_id")
   Customer customerWhoPays;
   ....
}
```

Activity: The diagram shows a Person-Dog and a Person-Car relationships. Which cases above (Invoice-Customer and Supplier-Product) these relationships match?

6.2. Unidirectional Relationship Exercise

Implement the following entities of the TechMap diagram, with their relationships:

- Person
- Dog
- Car

Ignore the other entities of the diagram (House, Supermarket, FootballTeam,…). In the Person class, ignore the addSubordinate and addSuperMarket methods that are on the UML diagram.

Write code that creates and persists the same entities/records as show in the diagram's DB tables.

Hint: In that code, first persist a Dog, before persisting its bound Person.

After exercise note: To find dogs having John as owner, you either need to define the other direction of the relationship (Dog.owner), or use a join query that we have not seen yet.

6.3. Bidirectional Relationships

6.3.1. mappedBy

For bidirectional associations, you define its characteristics (such as the @JoinColumn) on the owning side. On the other side, you specify the "mappedBy" attribute. With the mappedBy attribute, you specify the bidirectionality of the association to JPA (vs two unidirectional associations).

Source

```
public class Customer {
   @OneToMany(mappedBy="customerWhoPays")
   Set invoices
   ....
}
```

Source

```
public class Invoice {
   @ManyToOne
   @JoinColumn
   Customer customerWhoPays;

   @ManyToOne
   @JoinColumn
   Customer customerWhoReceives;
   ....
```

```
}
```

The value mappedBy="customerWhoPays" refers to the attribute named Invoice.customerWhoPays. JPA infers that the other entity is an Invoice thanks to the Set's parameterized type. In this way, we can distinguish between two or three ManyToOne relations to the entity Customer (customerWhoPays and customerWhoReceives, for example).

6.3.2. Where is the foreign key?

In JPA, a bidirectional association is always owned by the @ManyToOne side, not the @OneToMany side. So the mappedBy attribute is always on the @OneToMany side and the @JoinColumn should be on the @ManyToOne side.

Source

```
public class Customer {
   @OneToMany
   @JoinColumn   NOOOO -- EXCEPTION AT START UP.
   Set invoices
   ....
}
```

Some JPA engines (such as Hibernate) allow workarounds to bypass the limitation, but this often results in unnecessary updates statements.

This is not true for unidirectional associations (as Person->Dog): you have no choice where to put @JoinColumn, as the association is not defined in the other side. The case of Person->Dog on the diagram is interesting: even if the *@JoinColumn(name="PERSON_ID")* is defined in the Person class, the foreign key is always in the many table (DOGS).

This is also not true for @OneToOne as there is no "many" side, the entity where you put @JoinColumn defines on which table the foreign key is created.

6.3.3. Keeping both directions in sync.

When you set one side of the association, JPA does not set the other side. You have to do it yourself:

Source

```
customer.getInvoices().add(invoice);
invoice.setCustomerWhoPays(customer);
```

If you don't, here's what will happen:

Source

```
invoice.setCustomerWhoPays(customer);
entityManager.persist(invoice);
customer.getInvoices().size() // this is 0 because the Invoice was never added to
```

In other words, the in-memory state will be out of synch with the DB state.

6.4. Bidirectional Relationships Exercise

In the previous exercise, you implemented the Person entity described in the TechMap diagram. Add the bidirectional House entity.

Ignore the other entities of the diagram (Supermarket, FootballTeam,…)

Write code that creates and persists the same entities/records as show in the diagram's DB tables.

Optional

When it works, try to exchange the @JoinColumn and the mappedby between Person and House. Now Person contains the mappedby attribute. House contains the @JoinColumn attribute.

It should not compile, because the @ManyToOne annotation has no mappedBy attribute. But you could do the exchange for @ManyToMany and @OneToOne.

6.5. Recap video & Exam

Please have a look at this video that summarizes the main relationships you can establish between your entities. Remark: this video is hibernate related and uses old-style xml mapping.

When done, close the chapter with this small test.

7. Lifecycle

This section describes the various states of an entity and the related EntityManager methods.

7.1. Entity States

Basically there are 4 states an entity can have : New, Managed, Detached and Removed. Here is a description of the 4 states:

- **New (or transient)** entity is new if it has just been created (typically by using the new operator), it is thus not associated with an EntityManager (yet). It has no representation in the database and no primary key value.
- **Managed** entity is a pojo with a persistent identity that is currently associated with an EntityManager. It means a Managed instance has a primary key value set as its database identifier. Transient instances might be made Managed by either calling persist() on the EntityManager or they might even be objects that became persistent when a reference was created from another persistent object already associated with a persistence manager.
- **Detached** entity is a pojo instance having an identity that is no longer associated with an EntityManager. Entity Detachment typically happens because the persistence context was closed or the pojo instance was evicted (this is explained later). When the EntityManager is closed the associated persistent objects still exist in the JVM memory but are detached from any persistence context.
- **Removed** entity is an instance having a corresponding record in the database associated with a persistence context and scheduled for deletion from the database. When the associated EntityManager is closed, corresponding SQL DELETE statements will be executed.

The notion of eviction (i.e. objects being cleared from the cache) is beyond the scope of this course. You can read this article if you want to know more.

7.2. EntityManager methods

The EntityManager API is quite rich, here is a summary of the most commonly used methods. Take a look at the below diagram then read the method description.

- `public void persist (Object entity)` turns a New instance into a Managed one, meaning that on the next EntityManager flush or commit the newly persisted instances will be inserted into the database. When called on an instance that is already persisted the call is ignored.
- `public void remove (Object entity)` turns a Managed entity into a Removed entity. When a New or Removed entity is passed as argument nothing happens. If a Detached entity is given, it throws an exception.
- `public void refresh (Object entity)` takes data from the database and puts it into the entity. Giving a New or Removed entity as argument will do nothing. If a non-Managed entity is given, an exception is thrown.
- `public Object merge (Object entity)` takes a Detached entity and returns a Managed. The returned entity copies its state from the entity passed in as argument. An UPDATE statement will be issued when the persistence context is flushed. In this sense, merge() is the opposite of refresh(). It is important to note that the argument entity does not become managed! Therefore, that reference should no longer be used. If a removed entity is passed in, an exception is thrown.
- `public boolean contains(Object entity)` returns a boolean saying whether the given entity belongs to the EntityManager or not.
- `T find(Class entityClass, Object primaryKey)` based on the type of entity and a PK value it returns the corresponding Managed entity (or null if no entity with that primary key exists).

Note: The behaviors described in the side table correspond to the JPA specification document. Some JPA engine don't exactly follow these; e.g. when persisting a Removed entity Hibernate will throw an exception instead of making it Managed again as specified in the JPA spec.

7.3. Handling Entity Equality

Entity equality is a little more complicated than ordinary object equality. We'll first quickly look at object equality in Java, then at the JPA-specific equality.

7.3.1. Object Equality

== returns true if and only if the two instances are the same in memory.

The equals() method (inherited from Object) offers a more flexible comparison strategy. Classes can override it and return true or false depending on the state of the object. The contract your implementation must follow are described in the Object#equals(Object) JavaDoc.

Important: whenever you override equals(), you must also override hashcode() and return a value calculated from the same state as equals(). See the JavaDoc for the full contract.

Here is an example implementation of equals() and hashCode() methods:

Source

```
public class BankAccount {

 private int number;
 private double balance;
 private String name;

 public boolean equals(Object obj) {
   if (this == obj)
     return true;

   if (!(obj instanceof BankAccount))
     return false;

   BankAccount bankAccount = (BankAccount) obj;

   return number == bankAccount.getNumber()
           && balance == bankAccount.getBalance()
           & name.equals(bankAccount.getName());
 }

 public int hashCode () {
   int hash = 1;
   int multiplier = 7;

   hash = hash * multiplier + number;
   hash = hash * multiplier + Double.valueOf(balance).hashCode();
   hash = hash * multiplier + (name == null ? 0 : name.hashCode());

   return hash;
 }

 public BankAccount(String name) {
```

```
    this.name = name;
  }

  public int getNumber() {
    return number;
  }

  public double getBalance() {
    return balance;
  }

  public String getName() {
    return name;
  }
}
```

7.3.2. Entity Equality

The EntityManager controls object identity within a Persistence Context. This means that it keeps a cache of all the objects it manages. Therefore, if you call EntityManager.find() on an entity that has already been loaded, you will always get the same instance back. In other words:

Source

```
em.find(BankAccount.class, 1L) == em.find(BankAccount.class, 1L)
```

However, if you wish to compare detached entities to managed ones, you must implement equals() and hashcode(). There are a few possible ways of doing this, several of them bad.

7.3.2.1. Generated Key

An easy way to compare entities would be to see if their primary keys match.

However, if these keys are generated by the DB, the following could happen:

Source

```
BankAccount myAccount = new BankAccount("me");
BankAccount myAccountToo = new BankAccount("me");

assert(myAccount.equals(myAccountToo));

em.persist(myAccount);

assert(!myAccount.equals(myAccountToo));
```

The equality of two objects would depend on whether or not it has been persisted!

7.3.2.2. Business Key

Another possibility is to use a business key. This is not the same a generated id. A business key is the functional, real-world way in which two records can be distinguished from each other. In other words, a business key is composed of the sub-set of fields that make a particular record unique. The best business keys are good candidates for being a primary key.

Good business key candidates:

- Immutable fields
- Fields that are constrained to have unique values
- Elements that your application's clients use to distinguish records. For example, if each TransferDetail has a clientNumber and a transferDate those could form a good business key
- The value of the primary key of associated entities, if you know it will never change during this entity's lifetime. In other words, if the associated entity is this entity's owner.

7.3.2.3. Choice

If your primary keys are generated, you have to choose a strategy for implementing your equals/hashcode methods (based on the generated key, or on a business key?)

This leads to a long and very technical discussion that is covered in the *JPA/Hibernate Lifecycle course.*

7.4. Summary Exercise

The goal of this exercise is to let you test some EntityManager methods. The steps to follow are easy:

- Open your JPA project in order to manipulate an Entity (e.g. the Product entity)
- Create a new class called TestLifecycle containing a main method
- Write code to fill the missing cells in the below table

7.5. Exam and References

If you want to know more about Entity Lifecycle you can optionally read the following reference:

- EntityManager API Documentation

When finished take the related JavaBlackBelt exam.

8. Fetching Strategies

In a typical business application all DB tables and all entities are indirectly bound by relationships, in one graph.

The upper part of this diagram shows simple Customer-Invoice-InvoiceLine-Product-Supplier relationships. Below you find a schematic in-memory representation of an object graph and the order in which object will be loaded (if no lazy-loading mechanism is present) if we fetch the Invoice i003. Numbers on the arrows indicate when the objects will be loaded.

8.1. Understanding the Lazy Loading principle

Loading is a mechanism that solves the "how deep do you fetch entities?" problem. To cut a long story short: when you fetch a Customer from the database, do you want to fetch all the bound entities (list of Orders, ...), their relationships and so on, or stop at a given level?

8.1.1. The Lazy/Eager Principle

Using **Lazy-loading** when you load one entity in memory, you generally don't want to fetch the whole DB, so the entity's relationships will not be loaded. However, to avoid null pointers when you navigate the object graph, JPA will automatically load the target if it is not yet in the JVM's memory.

For example, when you load an Invoice, you can configure JPA to not load its associated customer. But when you access one of the customer's attributes (e.g. her name), JPA will transparently load the customer (by executing a SQL select statement).

Source

```
invoice.getCustomerWhoPays().getName();
```

Eager-loading refers to loading both entities together. In our example, it means that the customer bound to the invoice is loaded along with the invoice (probably in a single query with a SQL join).

Note that while an entity's relationships may be lazily loaded, it's fields are eagerly loaded. This means that when you access the invoice's customer's name, all of the customer's other fields (date of birth, customer number, etc.) are loaded as well. This cuts down on the number of SQL statements that are executed, while minimizing the amount of data in memory.

8.1.2. Default

By default, all

- *ToOne relationships are eager,
- *ToMany relationships are lazy.

The following fetch attribute is NOT optional (if you want the relationship to be lazy):

Source

```
public class Invoice {
  @ManyToOne(fetch=FetchType.LAZY)
  Customer customerWhoPays;
  ...
```

On the contrary, the fetch attribute below is optional (the default value is the same). We decide to always load an InvoiceLine and the associated Product together:

Source

```
public class InvoiceLine {
  @ManyToOne(fetch=FetchType.EAGER)
  Product product;
  …
```

8.1.3. When to use eager?

In the example above (Customer-Invoice-InvoiceLine-Product-Supplier), if all the relationships were eager, then loading one customer would load:

- all its invoices
- all the invoices' invoice lines
- all the products referenced by those invoice lines
- the products' suppliers
- and worse: if the Supplier-Product relationship is bidirectional, it loads all the other products of these suppliers

The impact on the performance and memory (RAM) footprint could be disastrous (huge, time consuming queries to the DB).

On the other hand, if all relationships are lazy, then the amount of selects can be higher than necessary, hurting performance. For example, when we load an invoice and loop over its invoice lines

Source

```
@Entity
public class Invoice {
  ....
  @ManyToOne(fetch=FetchType.LAZY)
  Customer customer;
  ....
}
```

Source

```
List<Invoice> allInvoices = (List<Invoice>)entityManager
  .createQuery("select inv from Invoice inv").getResultList();

for (Invoice invoice : allInvoices) {
   System.out.println("Customer Name = " + invoice.getCustomer().getName());
}
```

a SQL select would be sent at each step of the loop, for the customer of each invoice.

On the contrary, if the relationship Invoice-Customer is eager, then all the customers are already in RAM when starting the loop. But they are not always brought with only one (bigger) select when loading the invoices. This difficult topic is explained in details in the *JPA/Hibernate Lifecycle* course.

Our advice is to use the default values unless you detect a performance problem. For the rest, it depends on your business logic and requirements.

8.1.4. More references

More references on the topic can be found there:

- Wikipedia - A generic explanation
- Javalobby - an interesting article that goes beyond this course

8.2. Hibernate Uses Proxies

8.2.1. Where's the Magic?

Hibernate implements the lazy loading feature through proxies.

When Hibernate loads the invoice, it knows nothing about the bound customer. It creates a fake empty customer (named the proxy). It’s not an instance of the Customer class, it’s an instance of a Customer's sub-class created and compiled at runtime by Hibernate (yes, they can do that!).

When you request a customer's attribute (e.g. invoice.getCustomer().getName()), you call the getName() method on the proxy class with Hibernate code in it.

Hibernate knows that it hasn’t loaded that customer yet and sends a SQL statement to the DB to get it. The real Customer (the "target", in technical terms) is instantiated and filled in with data from the DB. Finally, Hibernate returns the value of the name of that customer.

8.2.2. Lazy-loading gotcha

When a lazily-loaded entity becomes detached (cf. the lifecycle chapter), its relationships may not have been loaded, leading to exceptions being thrown if that relationship is dereferenced.

8.3. Summary Diagram

Review this summary diagram before the exercise.

Click to download

8.4. Exercise

The goal of this exercise is to implement a **OneToMany** relationship as Invoice-InvoiceLine, with lazy loading. To help you understand what happen behind the scene make sure you turn the hibernate property to show the generated SQL to true.

Steps

- Setup
  - Open your JPA project and build 2 new entities Invoice and InvoiceLine having a **OneToMany** relationship.
  - Run a main class to fill your database with Invoices and related InvoiceLines. Create at least 1 Invoice with 3 InvoiceLines.
- Testing relationship Laziness
  - Create a new TestLazy executable class : It is important to use a new class or at least a new EntityManager instance otherwise the first-level cache will be used and no SQL statements will be executed.
  - Write code that loads a *master* element (an *Invoice*)
  - Retrieve the collection of InvoiceLine
  - Extract the Iterator and loop over the *detail* elements (over the *InvoiceLines* of that *Invoice*), for the purpose of the exercise use a while loop (or a pre-java 5 for loop)
  - Put breakpoints after having fetched the collection and also one in the loop, prior you access (and print to the console) the detail element's attribute (as invoiceLine.getQuantity()).
  - Execute the loop step by step with the debugger and see how many SQL statements are issued
  - Inspect the local variables ?
    - What is the type of the collection retrieved from the Invoice ?
    - Browse the attributes, to see them change
- Testing eager fetching
  - Change the relationship fetch to eager, and re-execute your program. When are the SQL statements sent to the DB this time?

9. Query Language

Most of your work as JPA programmer job is to create/annotate entities and to write query methods in DAOs.

DAOs are essentially composed of queries.

9.1. Syntax Basics

JPA-QL (Query Language) is similar to, and more powerful than SQL.

Source

```
select c from Customer c where c.city = 'Brussels'
```

This selects all the customers in Brussels.

9.1.1. Java Names

Your select strings use the names of the Java class and of their properties. In the example above, Customer is the class name and city is an attribute name of Customer. The corresponding DB table and field may have completely different names.

After renaming Java classes/attributes, it's very common to get exceptions when executing queries, because the names in the query strings have not been changed yet by the developer. Also, entity, property and parameter names in queries are case-sensitive. Unit-testing your queries can be very helpful to catch this kind of mistake.

9.1.2. API

Use the EntityManager to create queries.

Expanded form:

Source

```
Query query = entityManager.createQuery("select c from Customer c where c.city = '
List result = query.getResultList();
```

Chained form (more common):

Source

```
List result = entityManager
 .createQuery("select c from Customer c where c.city = 'Brussels'")
 .getResultList();
```

Note that getResultList() returns a raw List, so you'll get a compiler warning when you convert it to a parameterized List.

When you expect only one result, use the getSingleResult() method:

Source

```
Customer result = (Customer) entityManager
 .createQuery("select c from Customer c where c.vat='1234567890'")
 .getSingleResult();
```

Be aware that if the query returns more than one result, an exception will be thrown!

9.1.3. Parameters

It's very common to place named parameters in the query:

Source

```
String cityName = "Brussels";
List result = entityManager
 .createQuery("select c from Customer c where c.city = :cityName")
 .setParameter("cityName", cityName)
 .getResultList();
```

This allows the caching of queries, which improves performance.

9.1.4. Short Form

When you query for a whole entity or a list of whole entities, you can use the short form:

Source

```
from Customer c where c.city = 'Brussels'
```

which equals

Source

```
select c from Customer c where c.city = 'Brussels'
```

but they are not equivalent if you add a join clause to the select. For that reason, it is preferable to always use the full form, even if some books use the short form in examples.

9.1.5. Implicit Joins

Queries can navigate across entities. The generated SQL will include join clause(s).

The easiest form is the implicit join, but it is only possible in the “to one” direction (ManyToOne and OneToOne):

Source

```
select inv from Invoice inv where inv.customerWhoPays.city = 'Brussels'
```

This selects the invoices paid by customers in Brussels. “inv.customerWhoPays” is possible because Invoice.customerWhoPays attribute/relationship is @ManyToOne.

The other way (one to many) using an explicit join (other syntax) will be seen in the *More Query Language* chapter.

9.2. Exercise

In the previous exercise, you’ve implemented the Person, Dog, Car and House entities from the "Entity Relationships" techMap diagram.

1. Write a method that takes a first name as parameter and finds (and returns) all the Persons having that first name.
2. Write a query to retrieve the persons using the car having the plate number “XSD-456”.
3. Try to execute a (wrong) query that uses an implicit join in the wrong direction. Find the houses, which have a resident having “John” as first name, with a where clause like: house.residents.firstName = “John”. Identify the error message you get from Hibernate.
   Don't worry, it cannot work, and you will see how to make it work later in this course.

9.3. Exam

You are now probably ready for this query language quiz.

10. DAO

In this section you will discover what Data Access Object classes are useful for and how the Spring framework helps us with EntityManager creation.

If the typical structure of a Java application is unclear to you, read this presentation of the relationship between UI, service, DAO and Domain Model.

10.1. DAO Basics

10.1.1. What are DAOs?

Data Access Objects (DAO) are classes responsible for hiding the persistence mechanism from the rest of the application. They typically contains methods such as (for a ProductDao):

- `findProductByNumber(String partNumber)` that returns a Product entity
- `getAllProducts()` that returns all Product entities
- ...

10.1.2. Where is my EntityManager?

Typically you need one DB connection per thread. If you run a batch program or a fat client, connected to a DB, you probably have only one thread and one connection. That's how you've done in this course so far.

But in multi-threaded applications, managing EntityManager manually is painful.

10.1.3. More about DAOs

If you want to read more about DAOs feel free to review the following references:

- InfoQ.com - http://www.infoq.com/news/2007/09/jpa-dao : Has JPA killed the DAO ?
- Wikipedia - http://en.wikipedia.org/wiki/Data_Access_Object : An overview of the general principle
- Sun.com - http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html : The official Java EE Design Pattern description

10.2. Spring shows up ...

To simplify the way we write DAO and handle EntityManager lifecycle we will add Spring in our application.

10.2.1. @PersistenceContext

Fortunately, the Spring framework can manage the lifecycle of the EntityManager for us.

In a DAO class managed by Spring, an attribute can be linked to an EntityManager instance. If we annotate it with @PersistenceContext, a standard annotation provided by the JPA spec, Spring will automatically inject a valid EntityManager into it when it creates the DAO object instance. In fact, em is a reference to a shared EntityManager, provided by Spring. The method em.createQuery(…) finds the real EntityManager somewhere (in a ThreadLocal object actually) bound to the DB connection, and calls the createQuery method on that one. This allows the EntityManager injected by Spring in your singleton DAO to be used in a multi-threaded application.

Source

```
@Repository
public class ProductDao {
   @PersistenceContext EntityManager em;

   public Product findProductByCode(String prodCode){
       return
           em.createQuery("from Product p where p.code = :code").setParameter("cod
               .getSingleResult();
   }
}
```

10.2.2. ApplicationContext

To make it work, we need some base Spring configuration.

We use annotations to declare our beans (as the DAOs). Spring has to scan the java packages blackbelt.*

Source

```
<context:annotation-config />
<context:component-scan base-package="blackbelt" />
```

The name/id “entityManagerFactory” is well known by Spring. If we declare a bean with that name, Spring will use that bean to create an EntityManager when needed.

Source

```
<bean id="entityManagerFactory"
     class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
   <property name="persistenceUnitName" value="BlackBeltUnit" />
</bean>
```

To use the @Transactional Spring annotation (not shown in the code above) we tell Spring to use a transactionManager (well-known bean name) bound with our JPA EntityManagerFactory. We also tell Spring to look at annotations for transaction demarcation.

Source

```
<!-- Transaction mgr for a single JPA EntityManagerFactory (alternative to JTA) --
<bean id="transactionManager"
     class="org.springframework.orm.jpa.JpaTransactionManager"
     p:entityManagerFactory-ref="entityManagerFactory"/>
<tx:annotation-driven/>
```

10.3. Exercise

This exercise will let you build your first DAO using Spring. We provide a base project with the required libraries to make you start faster.

- Setup the project
  - Download JpaWithSpringExercise.zip. Import it as an existing eclipse project. It contains the necessary libraries and configuration files.
  - Note : Do not worry about errors related to the missing BankAccountDao class, you'll have to write it in a minute.
  - Review the content of the project to make sure you understand the purpose of each class and configuration file.
- Build the Dao class
  - Create a new class called BankAccountDao in the blackbelt.dao package
  - Add 2 business methods
    - public BankAccount findByNumber(String number)
    - public List findAccountsWithBalanceGreaterThan(double balance)
  - The 2 methods must use an injected EntityManager.

Hint: Some RDBMS use specific reserved words for types, functions, queries ... for instance NUMBER is a reserved word in Oracle (i.e. BankAccount.number).

11. More Query Language

Now that you have seen simple queries, it's time to move on to some of JPA QL's more advanced features.

11.1. Explicit Joins

11.1.1. Syntax

Source

```
select p from Person p
        join p.dogs d
        where d.color = 'brown'
```

Selects the persons having (at least) one brown dog.

The main difference with a SQL join is that you don't need to specify the foreign key / primary key equality. You identified the keys in the relation mapping already (@Id and @JoinColumn annotations). For JPA/Hibernate, identifying the relation (p.dogs) is enough.

JPA/Hibernate will produce an SQL statement with a join, similar to this:

Source

```
SELECT p.firstName, p.lastName, ... FROM Person p
  JOIN Dog d ON d.person_id = p.id
  WHERE d.color = 'brown'
```

Explicit joins are mandatory if you want to exploit *ToMany relationships, for example in the where clause as above (d.color = ...).

You can join multiple relations:

Source

```
select p from Person p
        join p.dogs d
        join d.doctor.complains comp
        where d.color = 'brown'
          and comp.date > :dateParam
```

Selects the persons having (at least) one brown dog. And that brown dog's doctor must have complained after the specified date.

11.1.2. When Are You Forced to Use Explicit Joins?

There are two cases when you have no other choice but to use an explicit join (instead of an implicit join).

- -ToMany relationships
- Unidirectional associations in the wrong direction

The example above with Person.dogs, illustrates the first case. This is wrong:

Source

```
select p from Person where p.dogs.color = 'brown'        wrong
```

We have seen the solution in the section above:

Source

```
select p from Person p
   join p.dogs d
   where d.color = 'brown'
```

For the second case, we need a -ToOne relationship in the wrong direction, for example Person.sharedCar (on the same diagram). We'd like to select the cars having a person which firstName is 'John'.

Source

```
select c from Car c where c.persons.firstName = 'John'        wrong
```

This is wrong because:

- The class *Car* has no attribute *persons*. The relationship unidirectional and only the other direction is implemented: Person has an attribute referring a Car.
- Even if Car.person**s** did exist it would be a -ToMany relationship requiring a explicit join.

This is the correct query:

Source

```
select c from Car c, Person p
   where p.sharedCar = c
     and p.firstName = 'John'
```

As in SQL, *from Car c, Person p* is a kind of join. We must explicitly tell which car belongs to which person *p.sharedCar = c* using the other direction of the relation.

This is another way to write the same query:

Source

```
select c from Person p
   join p.sharedCar c
   where p.firstName = 'John'
```

And another:

Source

```
select p.sharedCar p from Person p
   where p.firstName = 'John'
```

11.2. Aggregation Functions

You can use the aggregation functions typically found in SQL.

select **count(d)** from Dog d

Selects the amount of dogs in the table.

Source

```
Long count = (Long) em.createQuery("select count(d) from Dog d").getSingleResult()
```

Supported aggregate functions: count, min, max, avg, sum.

11.3. Further Reading

For more information, you may read the following resources

- A description of all the aggregate functions
- A full join syntax reference

11.4. Practice

11.4.1. Exercise

Execute the following steps

1. Correct the query from the Simple QL exercise to find the houses with a resident named "John"
2. Write a query that gets the last names of people who have a brown dog.
3. Count the number of supporters a particular team has.

11.4.2. Exam

You may now take the chapter quiz

12. Project Setup Summary

This chapter is optional. It provides you a summary with a checklist of operations and steps of the implementation of the project during this course.

12.1. Overview

The schema shows the libraries hierarchical tree of a JPA project with Spring, copied from an Eclipse project that uses Spring and JPA/Hibernate. We intend here to point out the steps to create a simple JPA project and to improve it for usage of Spring, while referring to the chapters where these steps are described.

First of all create a classical Java project with a main class.

12.2. Hibernate Setup

- Create a META-INF sub-folder with a persistence.xml file (2). An example is given in the chapter “Hello world“. Mind the level of the META-INF folder.
- Create a “log4j.properties” (4), see chapter Database integration
- From the hibernate.org you will download the following files (see chapter 3.2 for a detailed modus operandi)
    - the library for JPa hibernate (5)
    - its dependencies , including the logging jars (6)
- Add the correct JDBC driver according to your DB choice (7)
- Add the library of JPA specification (8)
- In the persistence.xml file, customize the properties according to the chosen db driver and its dialect, and the url of the DB. You can introduce parameters to display SQL messages, or not. See a complete example in the chapter 3.1.3.
- Mind the value of the hbm2ddl parameter. The schemas and tables need not exist before the exercise: they will be created by Hibernate if you specify another value than ‘validate’. But if they exist, they might be dropped at the beginning of the next exercise with the value"create", or at the end of the exercise with"create/drop".

By this time we have set up the environment needed to run a simple java project : JPASimple.

Review: In your Eclipse package explorer you must have 2 files: persistence.xml and log4j.properties, not on the same level, and 11 jars.

The schema has one more jar because we mention 2 JDBC drivers when only 1 is mandatory (because you must at least access one database).

12.3. Spring setup

- In the course chapter DAO (10.3) you download the JpaWithSpringStart.zip file. This will provide you with the library for Spring(9) and with the Spring dependencies(10).
- Add a dao class in your source application (1). This source is downloaded during the exercise of §3 in the Chapter DAO.
- In the src of your package, add an applicationContext.xml file .Customize the base package name and the persistence unit name into your own values according to the persistence file.

Here, we have some examples of Application Context annotations : @Service, @Repository.

In the BankAccount class we have an example of @Entity definition.

See Javadoc, package javax.persistence for more information on this topic.

You are now ready to develop your application with JPA/Hibernate and, eventually, Spring.

13. Exceptions

During the course and your exercises, you'll meet exceptions. In this chapter, we'll see the most commons of them you might encounter.

13.1. Class Diagram

13.2. BatchUpdateException

An exception that you will soon or later encounter is the *BatchUpdateException*. Unfortunately, the exception itself doesn't give you enough details to see what's happening. Let's see how to solve it below.

13.2.1. When does it happen ?

According to the JavaDoc, the BatchUpdateException is an exception thrown when an error occurs during a batch update operation...So, what does it mean?

It happens when Hibernate does not detect a problem and gives the DB a SQL statement that cannot be executed.

13.2.2. Stack trace

This is a typical BatchUpdateException stacktrace. As you can see, the last cause exception (bottom) does not show the real problem.

Source

```
javax.persistence.RollbackException: Error while committing the transaction
at org.hibernate.ejb.TransactionImpl.commit(TransactionImpl.java:93)
at blackbelt.main.Main.main(Main.java:50)
Caused by: javax.persistence.PersistenceException: org.hibernate.exception.SQLGram
Could not execute JDBC batch update
at org.hibernate.ejb.AbstractEntityManagerImpl.convert(AbstractEntityManagerImpl.j
at org.hibernate.ejb.AbstractEntityManagerImpl.convert(AbstractEntityManagerImpl.j
at org.hibernate.ejb.TransactionImpl.commit(TransactionImpl.java:81)
... 1 more
Caused by: org.hibernate.exception.SQLGrammarException: Could not execute JDBC bat
at org.hibernate.exception.SQLStateConverter.convert(SQLStateConverter.java:92)
at org.hibernate.exception.JDBCExceptionHelper.convert(JDBCExceptionHelper.java:66
at org.hibernate.jdbc.AbstractBatcher.executeBatch(AbstractBatcher.java:275)
at org.hibernate.jdbc.AbstractBatcher.prepareStatement(AbstractBatcher.java:114)
at org.hibernate.jdbc.AbstractBatcher.prepareStatement(AbstractBatcher.java:109)
... 10 more
Caused by: java.sql.BatchUpdateException: The element of batch 0 insert into Invoi
values ('join first invoice', '1') has been cancelled. Call getNextException to kn
at org.postgresql.core.v3.QueryExecutorImpl.processResults(QueryExecutorImpl.java:
at org.postgresql.core.v3.QueryExecutorImpl.execute(QueryExecutorImpl.java:407)
at org.postgresql.jdbc2.AbstractJdbc2Statement.executeBatch(AbstractJdbc2Statement
at org.hibernate.jdbc.BatchingBatcher.doExecuteBatch(BatchingBatcher.java:70)
at org.hibernate.jdbc.AbstractBatcher.executeBatch(AbstractBatcher.java:268)
... 16 more
```

13.2.3. Using getNextException method

The BatchUpdateException does not contain the cause returned by the DB.

The short code below will allow you to get the “origin” exception returned by the DB

Source

```
public static void printBatchUpdateException(Throwable throwable, PrintStream out)
   Throwable cause = getCauseException(throwable);

   while (cause != null) {
       if (cause instanceof BatchUpdateException) {
           BatchUpdateException bue = (BatchUpdateException)cause;
           out.println();
           out.println("XXXXXXXXXXXXXXXX NEXT from BatchUpdateException");
           bue.getNextException().printStackTrace(out);
       }
       cause = getCauseException(cause);
   }
}

protected static Throwable getCauseException(Throwable t) {
```

```
    if (t instanceof Exception) {
        return t.getCause();
    } else {
        return null;
    }
}
```

You can see the console result of the getNextException Method Below.

Source

```
org.hibernate.exception.SQLGrammarException: Could not execute JDBC batch update
at org.hibernate.exception.SQLStateConverter.convert(SQLStateConverter.java:92)
at org.hibernate.exception.JDBCExceptionHelper.convert(JDBCExceptionHelper.java:66
at org.hibernate.transaction.JDBCTransaction.commit(JDBCTransaction.java:133)
at org.hibernate.ejb.TransactionImpl.commit(TransactionImpl.java:76)
at blackbeltfactory.main.Main.main(Main.java:75)
... 14 more

Caused by: java.sql.BatchUpdateException: The element of batch 0 insert into Invoi
values ('join first invoice', '1') has been cancelled. Call getNextException to kn
at org.postgresql.core.v3.QueryExecutorImpl.processResults(QueryExecutorImpl.java:
at org.hibernate.jdbc.BatchingBatcher.doExecuteBatch(BatchingBatcher.java:70)
at org.hibernate.jdbc.AbstractBatcher.executeBatch(AbstractBatcher.java:268)
... 19 more

XXXXXXXXXXXXXXXXX NEXT from BatchUpdateException
org.postgresql.util.PSQLException: ERROR: the relationship « invoiceline » does no
 Position : 13
at org.postgresql.core.v3.QueryExecutorImpl.receiveErrorResponse(QueryExecutorImpl
at org.postgresql.core.v3.QueryExecutorImpl.processResults(QueryExecutorImpl.java:
at org.hibernate.impl.SessionImpl.flush(SessionImpl.java:1216)
at org.hibernate.impl.SessionImpl.managedFlush(SessionImpl.java:383)
at org.hibernate.transaction.JDBCTransaction.commit(JDBCTransaction.java:133)
at org.hibernate.ejb.TransactionImpl.commit(TransactionImpl.java:76)
at blackbeltfactory.main.Main.main(Main.java:75
... 20 more
```

13.3. Common exceptions

Below we will list common exceptions you may encounter during this course and why they happen.

13.3.1. NullPointerException

This exception occurs when an application attempts to use null in a case where an object is required.

Take an example in which you want to find a Person record with the ID number is 2, using the find method.

Source

```
public static void main(String[] args)  {
   // EntityManagerFactory, EntityManager,...
   Person thePerson=em.find(Person.class, 2L);
   thePerson.getFirstName(); // NullPointerException
}
```

The find method returns null because hibernate hasn't found the record requested, then if you call the method "getFirstName" on "thePerson", a NullPointerException will be thrown because the pointer on this object is null.

13.3.2. IllegalArgumentException

This exception is thrown to indicate that a method has been passed an illegal or inappropriate argument.

Source

```
Class Person {
String firstName;
...
}

public static void main(String[] args)  {
   // EntityManagerFactory, EntityManager, ...

   em.createQuery( "select p FROM Person p WHERE p.firstname =:firstname" )
       .setParameter( "firstname", firstname )    // will throw IllegalArgumentExc
       .getResultList();
}
```

In this case, we get an exception because p.firstname should be p.first**N**ame.

JP-QL requests are case sensitive and you **must** take care of spelling of instance variables.

13.3.3. ClassNotFoundException

This exception occurs when an application tries to load in a class, but no definition for the class with the specified name could be found. (No **.class** file in the class path)

This simply means that you've forgotten a *.jar file in your build path.

Let's take the example in which we have forgotten to put in the build path the jar file "commons-collections-3.1.jar".

We will get, of course, a ClassNotFound exception because this jar contains classes needed by hibernate to work.

Source

```
Caused by: java.lang.ClassNotFoundException: org.apache.commons.collections.map.LR
at java.net.URLClassLoader$1.run(Unknown Source)
at java.security.AccessController.doPrivileged(Native Method)
at java.net.URLClassLoader.findClass(Unknown Source)
at java.lang.ClassLoader.loadClass(Unknown Source)
at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
at java.lang.ClassLoader.loadClass(Unknown Source)
... 47 more
```

So, to avoid this exception, check your build path and ensure that you have all jars needed.

13.3.4. ClassCastException

This exception is thrown to indicate that the code has attempted to cast an object to a subclass of which it isn't an instance.

For example, the following code will generates this exception.

Source

```
public static void main(String[] args)  {
   // EntityManagerFactory, EntityManager, ...
   Person pers = (Person)em.createQuery("select p,d FROM Person p join p.dogs d wh
   .setParameter("firstname",firstname).getSingleResult();
}
```

"select p from ..." would work fine. But "select p,d from ..." returns a 2 entities array which cannot be casted to a Person.

13.3.5. NoSuchMethodException

This exception is thrown when a particular method isn't found.

You can see on the example above, that **ejb3-persistence.jar** and **javax.persistence.jar** have same classes

but not same implementations.

Take care with these ones; the only difference between them is their version (ejb3 is on v1.0 and javax on v2.0)

but they are same persistence provider!

The latest version of Hibernate (v3.6) uses javax, so if you try to use ejb3, you will get the NoSuchMethodException.

An easy way to avoid this kind of problems is to use the latest and stable version of jars.

13.3.6. SqlException

This exception provides information on a database access error or other errors, which are:

- A string describing the error.
- A “SQLstate”.
- An integer error code that specific to each vendor (It’s returned by the database).
- A chain to next Exception.

It is thrown when you write a wrong SQL query when you use JDBC. Note that the BatchUpdateException we have seen

above is an SQLException.

Another case where this exception can occur is if you have declared a wrong dialect in the persistence.xml when you use JPA/Hibernate.

Source

```
public static void main(String[] args)  {
  ...
  rs = stmt.executeQuery("SELECT * FRM Person");  //  ERROR : forget “O” for “FROM
  while (rs.next()) {
System.out.println(rs.getString("firstname"));
  }
}
```

13.3.7. QuerySyntaxException

This exception inherits QueryException, and this one is thrown when there is a syntax error in the HQL.

Source

```
public static void main(String[] args)  {

  List result = em.createQuery ("select p from Personn p where p.firstName =:first
  setParameter("firstname ", "Noe")).getResultList();

  for(Product produit:result) {
      System.out.println(produit);
  }
}
```

In this case, the exception is thrown because “Personn” is an incorrect spelling for the class name.

The class name must be correctly spelled in the request, it has to be called “Person”.

13.3.8. PersistenceException

This exception is thrown when there's an error between the persistence.xml file and the creation of your EntityManagerFactory.

Source

```
<!—persistence.xml-->

<persistence-unit name="myPersist">
   <properties>
   ...
   </properties>
</persistence-unit>
```

You can see in the following code that the persistence-unit name in the **xml file** and in the **createEntityManagerFactory method** are different.

Source

```
public static void main(String[] args)  {
  EntityManagerFactory emf=Persistence.createEntityManagerFactory("myPersistence")
  EntityManager em=emf.createEntityManager();
  ...
}
```

The name of these ones must be the same! If it isn't the case, you will get the exception.

13.3.9. NoSuchBeanDefinitionException

This exception is thrown when a the Spring applicationContext is asked for a bean instance name for which Spring cannot find a definition.

So, you **must** write your class name with the first letter in lowercase, the method getBean is case sensitive.

Source

```
public static void main(String[] args)  {
ApplicationContext applicationContext = new ClassPathXmlApplicationContext("applic
ProductStoreService pss1 = (ProductStoreService)applicationContext.getBean(" YourS
}
```

You can see below the correct spelling of the getBean method parameter, with the first letter in lowercase.

Source

```
public static void main(String[] args)  {
ApplicationContext applicationContext = new ClassPathXmlApplicationContext("applic
ProductStoreService pss1 = (ProductStoreService)applicationContext.getBean("yourSe
}
```

13.3.10. TransientObjectException

This exception occurs when you persist an object which references another object that is transient

(With no cascading rule, it will be covered in the course JPA-Hibernate / Lifecycle).

It commonly happens when you are creating an entire graph of new objects but you haven't explicitly saved all of them.

Source

```
public static void main(String[] args)  {
   em.getTransaction().begin();
   Person john = new Person("John","Rizzo");
   Group theGroup = new Group("TheGroup");
   theGroup.getPersons().add(john);
   em.persist(theGroup);
   em.flush(); // TransientObjectException
   em.persist(john);
   em.getTransaction().commit();
}
```

You get an exception in the code above because you've forced Hibernate to save(persist) "theGroup" (flush method) before having saved the Person object. The correct way to avoid this exception is show below.

Source

```
public static void main(String[] args)  {
   em.getTransaction().begin();
   Person john = new Person("John","Rizzo");
   Group theGroup = new Group("TheGroup");
   theGroup.getPersons().add(john);
   em.persist(john);
   em.persist(theGroup); // OK
   em.getTransaction().commit();
}
```

13.3.11. BeanCreationException

This exception occurs when BeanFactory encounters an error when attempting to create a bean from a bean definition.

Source

```
<!—applicationContext.xml-->

<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
<property name="persistenceUnitName" value="myPersist" />
</bean>
```

You can see on the examples that the persistence-unit name in **applicationContext.xml** and in **persistence.xml** is different.So, you will get of course the BeanCreationException.

Source

```
<!—persistence.xml-->

<persistence-unit name="myPersistence">
...
</persistence>
```

The course of the BeanCreationException tells you why the bean could not be instanciated.

13.3.12. AnnotationException

This exception occurs when you have forgot to write or if you have wrong used an hibernate annotation.

Source

```
@Entity
Class Person {

... // forgot the @Id annotation
       Long id;

       @OneToMany
       Set <Dogs> dogSet = new HashSet<Dogs>();
}
```

You can see below an example of the AnnotationException error message you will get in the console.

Source

```
Caused by: org.hibernate.AnnotationException: No identifier specified for entity:
at org.hibernate.cfg.InheritanceState.determineDefaultAccessType(InheritanceState.
at org.hibernate.cfg.InheritanceState.getElementsToProcess(InheritanceState.java:2
at org.hibernate.cfg.Configuration$MetadataSourceQueue.processAnnotatedClassesQueu
at org.hibernate.cfg.Configuration$MetadataSourceQueue.processMetadata(Configurati
at org.hibernate.ejb.Ejb3Configuration.configure(Ejb3Configuration.java:362)
... 11 more
```

While you haven't written the annotation correctly, the entities won't be persisted and the tables won't be created in

the database,

So you should take care of writing all the annotations needed by hibernate like:

@Entity, @Id, @JoinColumn, @GeneratedValue, @OneToMany, @ManyToOne, ...

14. Conclusion & Exam

After all those chapters, exercises and exams, you should have a good grasp of the JPA basics: setting up, mapping, querying and integrating with Spring.

If you have access to the Workshop, do it now. And if you don't have access to the Workshop, check it out - it's a very interesting and useful complement to the course.

The final step is the course's global exam.

Further resources

Parleys has a lot of very useful JPA videos. Writing JPA Applications discusses some more advanced topics and things to be aware of.

Next Step

There are, however, a number of more advanced topics to be explored, some of them are covered in our advanced courses:

- Advanced mapping with inheritance strategies, embedded objects
- Handling legacy databases
- Entity callbacks and listeners
- Advanced querying
- Optimistic and pessimistic locking
- Hibernate caching strategies
- Performance tuning
- Hibernate validation

15. Exercise Solutions

15.1. reminder

Before coding the solution, make sure you've put the "log4j.properties" file into the "src" folder. After that, verify you've made a "META-INF" folder with "persistence.xml" in the "src" folder.

Now verify you haven't forgotten any jars.

15.2. solutions

15.2.1. fetching strategy

15.2.1.1. Invoice

Source

```
import java.util.HashSet;
import java.util.Set;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToMany;

@Entity
public class Invoice {

@Id @GeneratedValue
protected Long id;

protected String date;

protected Double amount;

@OneToMany(mappedBy="inv")
//By default, the fetching strategy is LAZY
protected Set lines=new HashSet();

public Invoice(){}

public String getDate() { return date; }
public void setDate(String date) { this.date = date; }
public Double getAmount() { return amount; }
public void setAmount(Double amount) { this.amount = amount; }
public Long getId() { return id; }

public Set getInvoiceLines(){ return this.lines; }

public boolean equals(Object other){
 boolean rslt=true;
 if(this==other) rslt=false;
 if(!(other instanceof Invoice)) rslt=false;
 Invoice i=(Invoice)other;
 if(!this.getId().equals(i.getId())) rslt=false;
 return rslt;
}

public int hashCode(){ return this.getId().hashCode(); }
}
```

15.2.1.2. InvoiceLine

Source

```
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;

@Entity
public class InvoiceLine {

@Id  @GeneratedValue
protected Long id;

protected int qty;

protected double price;

@ManyToOne
@JoinColumn(name="invoice_id")
protected Invoice inv;

public InvoiceLine(){}

public int getQty() { return qty; }
public void setQty(int qty) { this.qty = qty; }
public double getPrice() { return price; }
public void setPrice(double price) { this.price = price; }
public Invoice getInv() { return inv; }
public void setInv(Invoice inv) { this.inv = inv; }
public Long getId() { return id; }

public boolean equals(Object other){
 boolean rslt=true;
 if(this==other) rslt=false;
 if(!(other instanceof InvoiceLine)) rslt=false;
 InvoiceLine invLine=(InvoiceLine)other;
 if(!this.getId().equals(invLine.getId())) rslt=false;
 return rslt;
}

public int hashCode(){ return this.getId().hashCode(); }

}
```

15.2.1.3. Main

Source

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

public class Main {

/**
 * @param args
 */


@SuppressWarnings("unchecked")
public static void main(String[] args) {
 EntityManagerFactory emf=Persistence.createEntityManagerFactory("BlackBeltUnit");
 EntityManager em=emf.createEntityManager();
 em.getTransaction().begin();
 //BEGINNING OF TRANSACTION WINDOW

 Invoice inv=new Invoice();
 inv.setDate("01/01/2009");
 em.persist(inv);
 inv.setAmount(10.99+10+10);
 InvoiceLine invl1=new InvoiceLine();
 invl1.setQty(1);
 invl1.setPrice(10.99);
 InvoiceLine invl2=new InvoiceLine();
 invl2.setQty(2);
 invl2.setPrice(5);
 InvoiceLine invl3=new InvoiceLine();
 invl3.setQty(1);
 invl3.setPrice(10);
 em.persist(invl1);
 inv.getInvoiceLines().add(invl1);
 invl1.setInv(inv);
 em.persist(invl2);
 inv.getInvoiceLines().add(invl2);
 invl2.setInv(inv);
 em.persist(invl3);
 inv.getInvoiceLines().add(invl3);
 invl3.setInv(inv);

 //ENDING OF TRANSACTION WINDOW
 em.getTransaction().commit();
 em.close();
 emf.close();
}

}
```

15.2.1.4. Fetching Strategy

to test both strategies, you should make a secondary main.

Source

```
import java.util.Iterator;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import relationship.exercise2.Invoice;
import relationship.exercise2.InvoiceLine;

public class FetchingStrategy {

/**
 * @param args
 */



public static void main(String[] args) {

 EntityManagerFactory emf=Persistence.createEntityManagerFactory("BlackBeltUnit");
 EntityManager em=emf.createEntityManager();
 em.getTransaction().begin();
 //BEGINNING TRANSACTION WINDOW
 //load an invoice
 Invoice inv = em.find(Invoice.class, new Long(93));
 //retrieve her invoice's line
 //loop
 Iterator it=inv.getInvoiceLines().iterator();
 int i=0;
 InvoiceLine il;
 System.out.println("\n\n************************");
 System.out.println("***** I N V O I C E *****");
 System.out.println("************************");
 while(it.hasNext()){
  il=(InvoiceLine)it.next();
  System.out.println(" art "+ i++ +" : "+il.getQty()+"X"+il.getPrice()+" ");
 }
 System.out.println("************************");
 System.out.println(" AMOUNT : "+inv.getAmount());
 System.out.println("************************");

 //ENDING OF TRANSACTION WINDOW
 em.getTransaction().commit();
 em.close();
 emf.close();

}

}
```

15.2.1.4.1. Lazy loading

As you can see the comment in the Invoice java class, the default fetching strategy is "LAZY".

Now, see the Invoice java class with an explicit fetching strategy.

Source

```
import java.util.HashSet;
import java.util.Set;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToMany;

@Entity
public class Invoice {

@Id @GeneratedValue
protected Long id;

protected String date;

protected Double amount;

@OneToMany(mappedBy="inv", fetch=FetchType.LAZY)
//By default, the fetching strategy is LAZY
protected Set lines=new HashSet();

public Invoice(){}

public String getDate() { return date; }
public void setDate(String date) { this.date = date; }
public Double getAmount() { return amount; }
public void setAmount(Double amount) { this.amount = amount; }
public Long getId() { return id; }

public Set getInvoiceLines(){ return this.lines; }

public boolean equals(Object other){
 boolean rslt=true;
 if(this==other) rslt=false;
 if(!(other instanceof Invoice)) rslt=false;
 Invoice i=(Invoice)other;
 if(!this.getId().equals(i.getId())) rslt=false;
 return rslt;
}

public int hashCode(){ return this.getId().hashCode(); }
}
```

with this configuration, look how many queries hibernate executes :

Source

```
Hibernate:
    /* load relationship.exercise2.Invoice */ select
        invoice0_.id as id2_0_,
        invoice0_.amount as amount2_0_,
        invoice0_.date as date2_0_
    from
        Invoice invoice0_
    where
        invoice0_.id=?
Hibernate:
    /* load one-to-many relationship.exercise2.Invoice.lines */ select
        lines0_.invoice_id as invoice4_1_,
        lines0_.id as id1_,
        lines0_.id as id5_0_,
        lines0_.invoice_id as invoice4_5_0_,
        lines0_.price as price5_0_,
        lines0_.qty as qty5_0_
    from
        InvoiceLine lines0_
    where
        lines0_.invoice_id=?
```

15.2.1.4.2. Eager loading

finally, Invoice class with eager strategy

Source

```
import java.util.HashSet;
import java.util.Set;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToMany;

@Entity
public class Invoice {

@Id @GeneratedValue
protected Long id;

protected String date;

protected Double amount;

@OneToMany(mappedBy="inv", fetch=FetchType.EAGER)
//By default, the fetching strategy is LAZY
protected Set lines=new HashSet();

public Invoice(){}

public String getDate() { return date; }
public void setDate(String date) { this.date = date; }
public Double getAmount() { return amount; }
public void setAmount(Double amount) { this.amount = amount; }
```

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

public Set getInvoiceLines(){ return this.lines; }

public boolean equals(Object other){
 boolean rslt=true;
 if(this==other) rslt=false;
 if(!(other instanceof Invoice)) rslt=false;
 Invoice i=(Invoice)other;
 if(!this.getId().equals(i.getId())) rslt=false;
 return rslt;
}

public int hashCode(){ return this.getId().hashCode(); }
}
```

this is how hibernate works :

Source

```
Hibernate:
   /* load relationship.exercise2.Invoice */ select
       invoice0_.id as id2_1_,
       invoice0_.amount as amount2_1_,
       invoice0_.date as date2_1_,
       lines1_.invoice_id as invoice4_3_,
       lines1_.id as id3_,
       lines1_.id as id5_0_,
       lines1_.invoice_id as invoice4_5_0_,
       lines1_.price as price5_0_,
       lines1_.qty as qty5_0_
   from
       Invoice invoice0_
   left outer join
       InvoiceLine lines1_
           on invoice0_.id=lines1_.invoice_id
   where
       invoice0_.id=?
```

15.3. DAO

Source

```
package blackbelt.dao;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import blackbelt.domain.BankAccount;


@Repository
```

```
@Transactional
public class BankAccountDao {

    @PersistenceContext
    private EntityManager em;


    public BankAccount findByNumber(String number) {
        return (BankAccount) em
            .createQuery("select ba from BankAccount ba where ba.number = :number)"
            .setParameter("number", number)
            .getSingleResult();
    }

    public void persist(BankAccount bankAccount) {
        em.persist(bankAccount);
    }

    public BankAccount merge(BankAccount bankAccount) {
        return em.merge(bankAccount);
    }

}
```