

Tutorial – JdbcTemplate

Objectives

By the end of this tutorial document you will be able to do the following:

- Create a Spring Boot starter project
- Autowire the JdbcTemplate bean
- Run SQL queries using JdbcTemplate
- Map table / result set rows to Java objects

Overview

We will use the Spring framework to access data in a relational database using its JdbcTemplate. We will also make use of Spring Boot, which accelerates and reduces the amount of configuration required when setting up a new project.

We will perform queries and updates using JdbcTemplate on a small MySQL database containing artists and movements. We will also look at how we can map from the columns in database tables to the properties in domain objects.

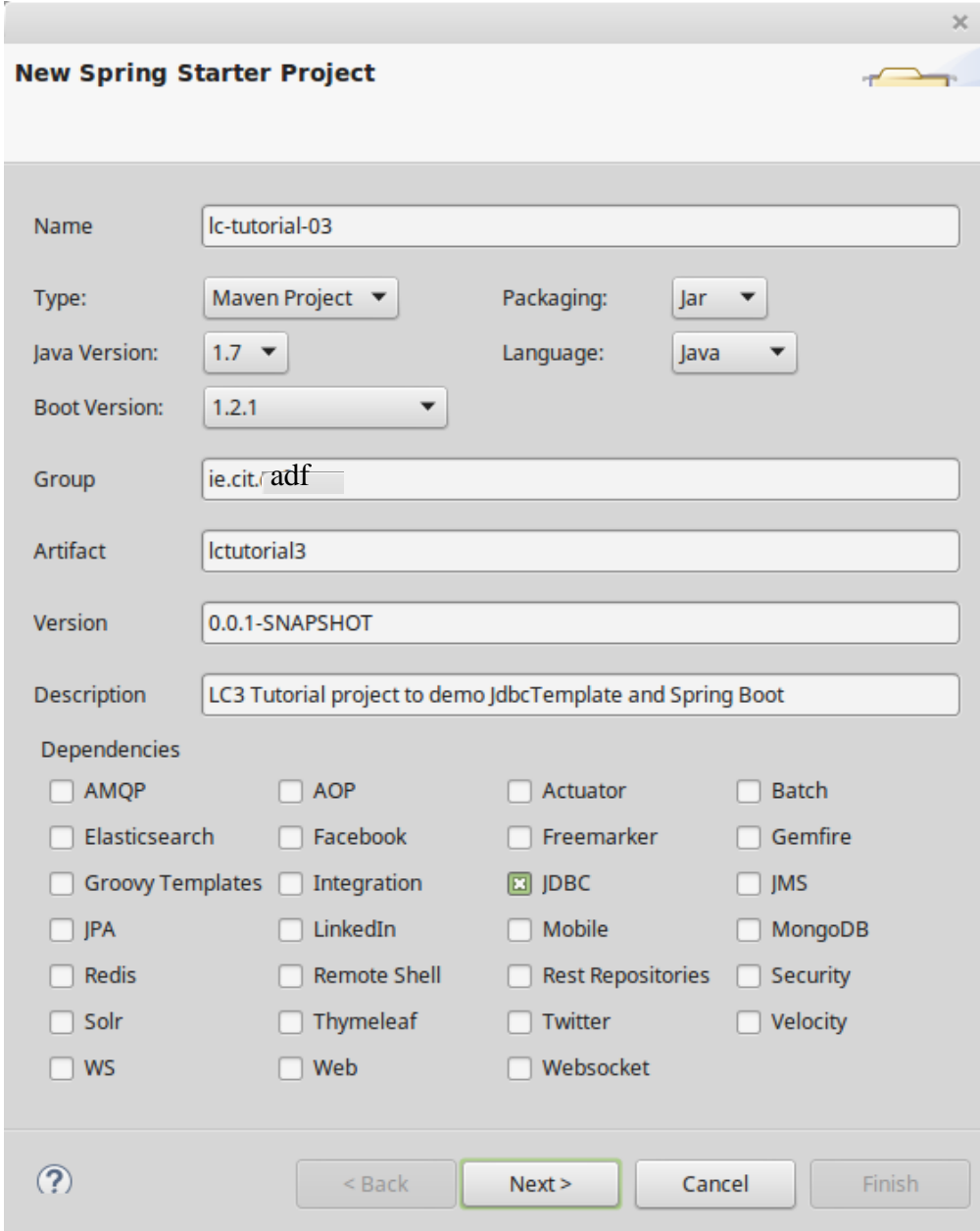
NOTE: This tutorial refers to Spring Boot 1.2.1. The current stable release is 1.2.6. Note also that the screens for creating a Spring Starter Project will differ slightly with the current version of the STS plugin.

Setting up your project

Eclipse / STS recently introduced support for Spring Boot projects. The following steps are all that is required to set up a JDBC enabled project. You might be surprised at just how straightforward it is.

Select *File* → *New* → *Spring Starter Project*.

Enter the following details:

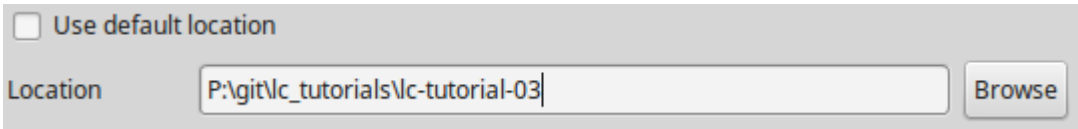


The image shows the 'New Spring Starter Project' dialog box in an IDE. It contains the following fields and options:

- Name:** lc-tutorial-03
- Type:** Maven Project (dropdown)
- Packaging:** Jar (dropdown)
- Java Version:** 1.7 (dropdown)
- Language:** Java (dropdown)
- Boot Version:** 1.2.1 (dropdown)
- Group:** ie.cit.adf
- Artifact:** lctutorial3
- Version:** 0.0.1-SNAPSHOT
- Description:** LC3 Tutorial project to demo JdbcTemplate and Spring Boot
- Dependencies:** A grid of checkboxes for various dependencies. The 'JDBC' checkbox is checked.

At the bottom, there are four buttons: '?', '< Back', 'Next >' (highlighted with a green border), 'Cancel', and 'Finish'.

Click *Next* and on the next screen you may change the default location to one within a git repository folder (assuming you want to add your files to GitHub) and add on the name of the project. E.g:

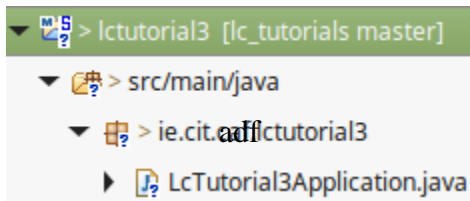


The image shows a dialog box for selecting the project location. It has a checkbox labeled 'Use default location' which is unchecked. Below it, there is a text field labeled 'Location' containing the path 'P:\git\lc_tutorials\lc-tutorial-03'. To the right of the text field is a 'Browse' button.

Click *Finish*.

You now have a Spring Boot project created. Expand the project and expand `src/main/java`. You may or may not need to rename the demo package (right-click on it and select *Refactor* > *Rename...*) to `ie.cit.awd.lctutorial3`.

You'll see something like this:



Open the auto-generated application file.

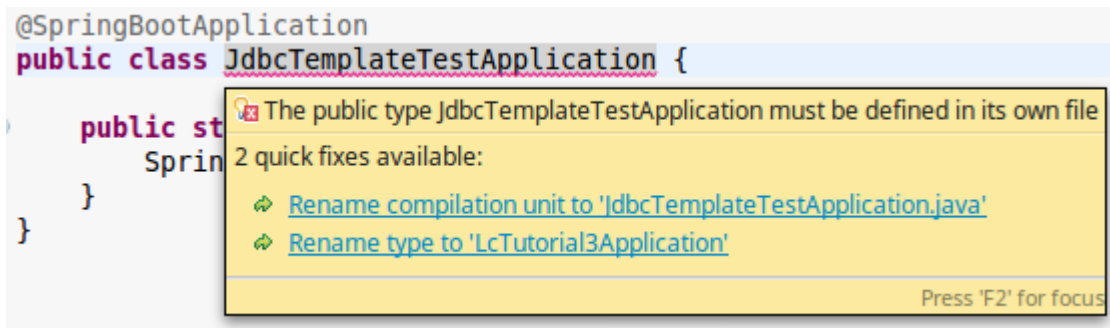
You'll see something like this:

```
@SpringBootApplication
public class LcTutorial3Application {

    public static void main(String[] args) {
        SpringApplication.run(LcTutorial3Application.class, args);
    }
}
```

The annotation `@SpringBootApplication` tells the JDK and the Spring framework that this is a Spring Boot application. This causes a lot of things to happen out of our sight when we build and run the application. For instance, it does a lot of auto-configuration by discovering classes, configuration properties, and more.

Rename the application class to `JdbcTemplateTestApplication` (either refactor as before, or type in the new name in the code – both locations – and hover over the new name to select *Rename compilation unit...* as below).



We are now going to make life easier when running the application in the console or using the command prompt. Make the highlighted changes:

```
@SpringBootApplication
public class JdbcTemplateTestApplication implements CommandLineRunner {

    @Override
    public void run(String... args) throws Exception {
        // TODO Auto-generated method stub
    }

    public static void main(String[] args) {
        SpringApplication.run(JdbcTemplateTestApplication.class, args);
    }
}
```

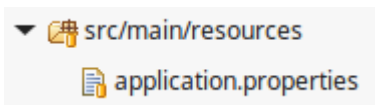
```
}  
}
```

We no longer need to worry about declaring variables outside the methods as static variables or additional methods as static methods.

If you try to run the application at this stage, it will actually fail because it is expecting to be told what type of database we are using – remember, we ticked the JDBC box when creating the project.

There are a number of ways to configure the database connection to use. Perhaps the best approach is to externalise the configuration in either a .properties file or a .yml file. Both achieve the same result, but are structured slightly differently.

Under src/main/resources, you will find that an empty application.properties file was created.



Add the following to it:

```
spring.datasource.url=jdbc:mysql://localhost/lc_tutorials  
spring.datasource.username=root  
spring.datasource.password=
```

The prefix spring.datasource is important. Spring Boot will be looking for properties that begin with that (as well as other prefixes for other things). This tutorial assumes you are using a local MySQL installation. For example, on the vDesktop, you have access to a local MySQL server via XAMPP. You may change the username and password if you wish, but the above will work for the default XAMPP MySQL setup.

Add the following dependency to pom.xml (just below the spring-boot-starter-test dependency):

```
<dependency>  
  <groupId>mysql</groupId>  
  <artifactId>mysql-connector-java</artifactId>  
</dependency>
```

Create the database lc_tutorials and run the following SQL into it:

```
CREATE TABLE `artists` (  
  `id` int(11) NOT NULL AUTO_INCREMENT,  
  `fullName` varchar(100) NOT NULL,  
  `gender` varchar(10) NOT NULL,  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB AUTO_INCREMENT=6 DEFAULT CHARSET=latin1;  
  
INSERT INTO `artists` VALUES (1,'Chadwick, Lynn','female'), (2,'Ono,
```

```
Yoko', 'female'), (3, 'Opie, Julian', 'male'), (4, 'Etty, William', 'male'), (5, 'Wallis, Henry', 'male');
```

```
CREATE TABLE `movements` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(100) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=5 DEFAULT CHARSET=latin1;
```

```
INSERT INTO `movements` VALUES (1, 'Geometry of Fear'), (2, 'Kinetic Art'), (3, 'New British Sculpture'), (4, 'Romanticism');
```

```
CREATE TABLE `artist_movements` (
  `artist_id` int(11) NOT NULL,
  `movement_id` int(11) NOT NULL,
  PRIMARY KEY (`artist_id`, `movement_id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

```
INSERT INTO `artist_movements` VALUES (1, 1), (1, 2), (3, 3), (4, 4), (5, 4);
```

Now we are ready to code with JdbcTemplate!

The JdbcTemplate bean

A Spring bean associated with the class JdbcTemplate is already there in the Spring context waiting to be referenced and used. And it already has a DataSource object within it to connect to the database. Spring Boot handled the creation of the DataSource object, and its injection into the JdbcTemplate bean for us.

Add the highlighted code:

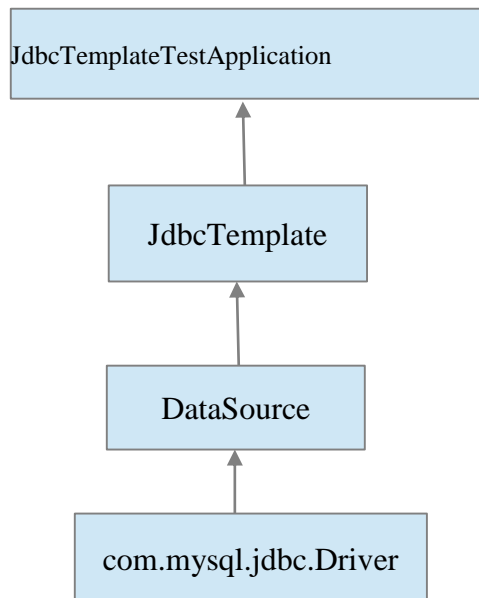
```
@SpringBootApplication
public class JdbcTemplateTestApplication implements CommandLineRunner {

    @Autowired
    JdbcTemplate jdbcTemplate;

    @Override
    public void run(String... args) throws Exception {
```

We can now reference the object jdbcTemplate anywhere in the JdbcTemplateTestApplication class.

We can represent the dependency injection that occurs like this:



And yet, the only Java code we added was in the application class. Spring Boot automatically discovered by looking at the Jar files the project is using and the properties in application.properties that the dependency injection hierarchy above is what should happen.

Now we can run some queries.

Using JdbcTemplate

1. Querying for Maps

We will add a number of methods to the application class, each demonstrating a feature of JdbcTemplate.

Add the following method:

```
public void query01() {  
    // Query for a list of maps with key-value pairs  
    // The hard way!!!  
  
    System.out.println("\nQuery 1 (List all artists using resultset Map)\n-----");  
  
    String sql = "SELECT * FROM artists";  
    List<Map<String, Object>> resultSet = jdbcTemplate.queryForList(sql);  
  
    for (Map<String, Object> row : resultSet) {  
        System.out.println("Name: " + row.get("fullName"));  
        System.out.println("ID: " + row.get("id"));  
        System.out.println("Gender: " + row.get("gender") + "\n");  
    }  
}
```

The type `List<Map<String, Object>>` might look a bit confusing. If we break it down, we have a map within a list. A map is just like a lookup table with key-value pairs. In the case of `Map<String, Object>` the keys are Strings and the value can be any type of Object. All classes ultimately inherit from the base Object class, so we can store any type of object as an instance of Object and turn it back into the original class type later. In this example, each row returned from the query will be in the form of a map, so there could be integers, strings, booleans, and so on.

Having a List of `Map<String, Object>` allows us to store many rows in the object variable `resultSet`. We can represent the `resultSet` data structure like this:

```
{'id':'1', 'fullName':'Chadwick, Lynn', 'gender':'female'},
{'id':'2', 'fullName':'Ono, Yoko', 'gender':'female' },
{'id':'3', 'fullName':'Opie, Julian', 'gender':'male' },
{'id':'4', 'fullName':'Etty, William', 'gender':'male' },
{'id':'5', 'fullName':'Wallis, Henry', 'gender':'male' }
```

We can loop through a list, of course. The following code loops through the elements of `resultSet`, naming each element `row`.

```
for (Map<String, Object> row : resultSet) {
    System.out.println("Name: " + row.get("fullName"));
    System.out.println("ID: " + row.get("id"));
    System.out.println("Gender: " + row.get("gender"));
}
```

The highlighted code shows how we retrieve a column value from a row. The key (or column name) is “fullName”.

Now let's make use of the method. Add a call to `query01()` in the run method.

```
@Override
public void run(String... arg0) throws Exception {
    query01();
}
```

Run the program. Amongst a lot of log output, you should see the following:

Query 1 (List all artists using resultset Map)

Name: Chadwick, Lynn
ID: 1
Gender: female

Name: Ono, Yoko
ID: 2
Gender: female

Name: Opie, Julian
ID: 3
Gender: male

Name: Etty, William
ID: 4

Gender: male
Name: Wallis, Henry
ID: 5
Gender: male

Once you get your head around the list of maps concept, getting results back from a query and looping through them is straightforward. But there is rather a lot of code, getting each column value individually. Wouldn't it be nice if we could just query for a list of artists and receive back the already-populated Artist objects?

2. Querying for objects

Obviously, if we want to retrieve Artist objects, we need an Artist class. Create a new subpackage ie.cit.awd.lctutorial3.domain (right-click on the existing package and select New->Package; add .domain to the end of the package name). Add the following class in the new subpackage:

```
public class Artist {  
  
    private int id;  
  
    private String name;  
  
    private String gender;  
  
    @Override  
    public String toString() {  
        return "Artist [id=" + id + ", name=" + name +  
            ", gender=" + gender + "]";  
    }  
}
```

Note: getters and setters are omitted for brevity. You can add them quickly using:

Source->Generate Getters and Setters... (Select All and OK).

To have rows automatically converted to objects, we need to first write a utility class that maps rows to objects. We can implement the interface RowMapper to do that.

Create a new subpackage called ie.cit.awd.rowmapper.

Add the following class:

```
public class ArtistRowMapper implements RowMapper<Artist> {  
  
    @Override  
    public Artist mapRow(ResultSet rs, int index) throws SQLException {  
        Artist artist = new Artist();  
  
        artist.setId(rs.getInt("id"));  
        artist.setName(rs.getString("fullName"));  
        artist.setGender(rs.getString("gender"));  
    }  
}
```



```
        return artist;
    }
}
```

Note: you need to get used to adding the imports. You can hover over unknown classes and select the appropriate import, or with the cursor inside the name of the unknown class, hit Ctrl+Space.

You should end up with the imports listed in **Appendix A**. See if the imports you generate match the ones in **Appendix A**.

So what does the following mean?

```
implements RowMapper<Artist>
```

We *implement* interfaces. Interfaces are Java specifications. They tell us about how classes that implement the interfaces **must** interact with the outside environment. If an interface specifies a method, a class that implements **must** implement that and all other methods in the interface.

Have a look at the RowMapper interface Javadoc at:

<http://docs.spring.io/spring-framework/docs/2.5.6/api/org/springframework/jdbc/core/RowMapper.html>

It contains the following method detail:

mapRow

Object **mapRow**(ResultSet rs,
int rowNum)
throws SQLException

Implementations must implement this method to map each row of data in the ResultSet. This method should not call `next ()` on the ResultSet; it is only supposed to map values of the current row.

Parameters:

rs - the ResultSet to map (pre-initialized for the current row)

rowNum - the number of the current row

Returns:

the result object for the current row

Throws:

SQLException - if a SQLException is encountered getting column values (that is, there's no need to catch SQLException)

There is only one method specified, so this is the only one we are obliged to implement. In our ArtistRowMapper class, we implement mapRow according to the specification above, adding our own code to create an Artist object and set the properties of the object using the corresponding column values in the ResultSet.

Note: we never actually call the mapRow method ourselves (though we could with the right parameters). The framework handles the calling of the method for us. This is another example of *Inversion of Control*.

Back in the application class, add the following method:

```
public void query02() {
    // Query for a list of objects - automatic mapping from row to object using RowMapper class
    // Using parameterised "prepared statements" reduces the risk of a SQL inject attack

    System.out.println("\nQuery 2 (List male artists using RowMapper)\n-----");

    String sql = "SELECT * FROM artists WHERE gender = ?";
    List<Artist> artists = jdbcTemplate.query(sql, new Object[] { "male" }, new ArtistRowMapper());

    for (Artist artist : artists) {
        System.out.println(artist.toString());
    }
}
```

Note how when we call query, we pass a new ArtistRowMapper object with it. This will ensure that mapping from table and columns to object and properties will occur.

The middle parameter, `new Object[] { "male" }`, passes an array of objects (usually String or Integer or Float) to the query method. These are used to replace the question marks (?) in the SQL. This is an example of where an SQL statement becomes a Prepared Statement, which is good for avoiding SQL injection attacks that try to replace parameters with bogus, insecure SQL.

If the SQL statement had `WHERE gender = ? AND fullName = ?`, then we would pass `new Object[] { "male", "Opie, Julian" }` as the second parameter, for example.

Now I can get a list of Artist objects and iterate through them to print their details.

In the run method, add a call to `query02()` and run the program.

The output should look like this:

```
Query 2 (List male artists using RowMapper)
-----
Artist [id=3, name=Opie, Julian, gender=male]
Artist [id=4, name=Ett, William, gender=male]
Artist [id=5, name=Wallis, Henry, gender=male]
```

3. Querying for a single object

Add the following method to the application class and call it in the run method.

```
public void query03() {
    // Query for a specific object - automatic mapping from row to object using RowMapper class

    System.out.println("\nQuery 3 (Print artist with id 1 - uses RowMapper)\n-----");

    String sql = "SELECT * FROM artists WHERE id = ?";
    Artist artist = jdbcTemplate.queryForObject(sql, new Object[] { 1 }, new ArtistRowMapper());

    System.out.println(artist.toString());
}
```

This requires little explanation after the second query. Instead of a list, we are getting a single Artist object.

The output should look like this:

Query 3 (Print artist with id 1 - uses RowMapper)

Artist [id=1, name=Chadwick, Lynn, gender=female]

4. Querying for aggregate function results or specific columns

Add the following method:

```
public void query04() {
    // Query for specific column values

    System.out.println("\nQuery 4 (Specfic columns)\n-----");

    // Old version (now deprecated)
    String sql = "SELECT count(*) FROM artists";
    int artistCount = jdbcTemplate.queryForInt(sql);
    System.out.printf("Number of artists: %d\n", artistCount);

    // New way
    sql = "SELECT count(*) FROM movements";
    int movementCount = jdbcTemplate.queryForObject(sql, Integer.class);
    System.out.printf("Number of movements: %d\n", movementCount);

    // Getting a map of values
    sql = "SELECT fullName, gender FROM artists WHERE id = 1";
    Map<String, Object> map = jdbcTemplate.queryForMap(sql);
    System.out.printf("Name: %s, Gender: %s\n", map.get("fullName"),
        map.get("gender"));
}
```

Note the use of the `queryByInt` method above. Eclipse will strike a line through it to indicate the method is deprecated. That means it can still be used, but future support is not guaranteed. There will usually be a new alternative to switch to in those situations. We can instead `queryForObject` and pass in the class type, in this case `Integer.class`. To get a `VARCHAR` / `CHAR` result back, use `String.class`.

The final call is to `queryForMap`, which gives us access to specific columns mentioned in the query.

Add a call to `query04()` in *run*. The output of the program will have the following added:

```
Query 4 (Specfic columns)
-----
Number of artists: 5
Number of movements: 4
Name: Chadwick, Lynn, Gender: female
```

5. Using a join query

Where the use of JDBC starts to become problematic is when we need to join tables, retrieving column values from multiple tables.

Let's add a domain class for movements in `ie.cit.awd.lctutorial3.domain`:

```
public class Movement {  
  
    private int id;  
  
    private String name;  
  
    @Override  
    public String toString() {  
        return "Movement [id=" + id + ", name=" + name + "];"  
    }  
  
    public int getId() {  
        return id;  
    }  
  
    public void setId(int id) {  
        this.id = id;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

We need to add the associated `RowMapper` in `ie.cit.awd.lctutorial3.rowmapper`:

```
public class MovementRowMapper implements RowMapper<Movement> {  
  
    @Override  
    public Movement mapRow(ResultSet rs, int index) throws SQLException {  
        Movement movement = new Movement();  
  
        movement.setId(rs.getInt("id"));  
        movement.setName(rs.getString("name"));  
  
        return movement;  
    }  
}
```

With our next demo method, we want to print out an artist and his/her movements. So it makes sense to add a list of movements to the `Artist` class. Add the following lines of code to the `Artist` class:

```
private List<Movement> movements;  
  
public List<Movement> getMovements() {  
    return movements;  
}
```

```
public void setMovements(List<Movement> movements) {
    this.movements = movements;
}
```

We need to update the Artist class's toString method:

```
@Override
public String toString() {
    String out = "Artist [id=" + id + ", name=" + getName() + ", gender="
        + gender + ", movements=";
    for (Movement m : movements) {
        out += m.toString() + ",";
    }
    out += "]]";
    return out;
}
```

But what if the movement list was empty? E.g. all we wanted was the artist's name, so we never bothered to populate movements. The toString method would generate a NullPointerException and crash our program. Add the following constructor to prevent that happening:

```
public Artist() {
    movements = Collections.<Movement>emptyList();
}
```

An empty list is different than null. When we try to loop through it, the loop will just do nothing rather than crash.

Now we have the infrastructure in place to join artists and their movements. Add the following to the program class:

```
public void query05() {
    // Specific artist and the artist's movements (many-to-many)
    // This is an inefficient version with 2 separate queries

    System.out.println("\nQuery 4 (Print artist and movements - 2 queries)\n-----");

    String sql = "SELECT * FROM artists WHERE id = ?";
    Artist artist = jdbcTemplate.queryForObject(sql, new Object[] { 1 }, new
ArtistRowMapper());

    sql = "SELECT m.* FROM movements m, artist_movements am WHERE m.id =
am.movement_id AND am.artist_id = ?";
    List<Movement> movements = jdbcTemplate.query(sql, new Object[] { 1 }, new
MovementRowMapper());

    artist.setMovements(movements);

    System.out.println(artist.toString());
}
```

The result of calling this method should be (Lynn Chadwick is in two movements):

Query 5 (Print artist and movements - 2 queries)

```
-----
Artist [id=1, name=Chadwick, Lynn, gender=female, movements=[Movement [id=1,
name=Geometry of Fear],Movement [id=2, name=Kinetic Art],]]
```

However, this is a particularly inefficient way of doing it. There are two separate queries, one to get the artist, the next to get the artist's movements. That's double the resources and double the network latency from application to database. Doing it in a single query would be much more efficient.

6. Using a ResultSetExtractor

In the previous example, we had to use `ArtistRowMapper` to first map to an `Artist` object, then `MovementRowMapper` to map to a list of movements. Then we set the movements list so we could print out the artist including movements.

An alternative to the `RowMapper` is a `ResultSetExtractor`. This allows us to create any type of object we want from the query results. In this case, we want to be able to construct an `Artist` that includes the list of movements already populated.

With the `RowMappers` we created public classes. For this example, we will use an anonymous inner class to do the work for us. If we do not intend to use the same logic again, we can use this technique.

Add this method to the program class:

```
public void query06() {
    // Specific artist and the artist's movements (many-to-many)
    // More efficient version with a single join query and
    // a ResultSetExtractor.... but it can get messy the more
    // complicated the join queries! The solution? ORM.

    System.out.println("\nQuery 6 (Print artist and movements - 1 join query)\n-----
    -----");

    String sql = "SELECT a.id as artistid, a.fullName, a.gender, m.id as movementid,
m.name " +
                "FROM artists a, movements m, artist_movements am " +
                "WHERE m.id = am.movement_id AND a.id = am.artist_id AND
am.artist_id = ?";
    Artist artist = jdbcTemplate.query(sql, new Object[] { 1 },
        new ResultSetExtractor<Artist>() {

            @Override
            public Artist extractData(ResultSet rs)
                throws SQLException, DataAccessException {

                ;
                Artist artist = null;
                List<Movement> movements = new ArrayList<>();

                while (rs.next()) {
                    if (artist == null) {
                        artist = new Artist();
                        artist.setId(rs.getInt("artistid"));
                        artist.setName(rs.getString("fullName"));
                    }
                }
            }
        });
}
```

```

        artist.setGender(rs.getString("gender"));
    }
    Movement movement = new Movement();
    movement.setId(rs.getInt("movementid"));
    movement.setName(rs.getString("name"));
    movements.add(movement);
}

artist.setMovements(movements);
return artist;
}
}

);

System.out.println(artist.toString());
}

```

Call the method in *run* and run the program. The output should be the same as the previous version.

There's a lot to absorb there!

In the SQL, we are joining artists to movements via artist_movements. When we call `jdbcTemplate.query(...)` we pass in a new (unnamed) instance of a `ResultSetExtractor`. The `ResultSetExtractor` is an interface that requires the implementation of one method, `extractData`. Because of the query we execute, we are getting back column values from two tables, artists and movements. We need to do some work to create only one Artist object with potentially multiple movements. As we loop through the rows returned in the `ResultSet` `rs`, we only want to create the Artist object once, so the first time around the artist object variable will be null, so we populate it with a new Artist object. Now it won't be null for iterations 2 and above.

For each iteration we will create a new Movement object and add it to a list of Movements. When we have looped through all rows, we will add the list of movements, if any, to the Artist object using `setMovements`.

One thing we can say about the above... it isn't straightforward! For more complex joins, the coding becomes even more complex. This is one strong argument in favour of JPA and Object-Relational Mappers (ORMs), which we will see in a couple of weeks and which can simplify things a lot at the expense of a minor performance hit.

Exercise

This might be a tough one, but if you follow the patterns above, you should be able to do it.

We have been looking at things from the perspective of the artist. What about from the perspective of the movement?

Update the movement class so that for a given movement, we can get a list of artists. This includes adding the list variable and its getter and setter, as well as updating `toString`. You can try out versions of queries 5 and / or 6 – getting the movement, then the artists; or returning a Movement object from a `ResultSetExtractor` with a list of artists populated.

Tutorial code

All of the above code is available in the folder lc-tutorial-3 at:

https://github.com/dlarkinc/lc_tutorials

Next time

We add some structure with a layered architecture that includes service and DAO classes.

Appendix A

```
import java.sql.ResultSet;  
import java.sql.SQLException;  
import org.springframework.jdbc.core.RowMapper;
```