DAM. UNIT 2. ACCESS TO DATABASES. PART 1. WORKING WITH RELATIONAL DATABASES

DAM. Acceso a Datos (ADA) (a distancia en inglés)

Unit 2. ACCESS TO DATABASES

Part 1. Working with Relational Databases

Abelardo Martínez
Year 2023-2024

1. Introduction

This unit provides an overview of how object-oriented applications use database management systems (DBMS) to achieve persistence of their instances. **DBMSs are specialised applications for storing structured data**, with the ability to store and retrieve them consistently and efficiently, regardless of the number of simultaneous accesses. It should be noted, however, that the management of DBMSs **requires quite specific technical knowledge** that makes it difficult for non-specialised users to access their stored data.

In order to simplify storage tasks, it is possible to automate them by using DBMSs' own query and administration languages and tools. However, these tools are not sufficient to enable a neophyte user to use them without having to make a significant learning and planning effort to translate their needs into an effective sequence of tasks.

The application development industry uses databases as tools to derive the complex storage tasks to focus efforts on the difficulties of the domain where the applications will be developed, as well as on the access to automation of the required storage tasks so that users can reduce the learning curve of the applications they will use, approaching more intuitive ways according to their specific background knowledge.

In order to co-ordinate programming languages with the potential of DBMSs, the industry has developed a set of specific tools to connect to a given database and send it the sequence of tasks that applications may need during execution.

1.1. Relational and non-relational databases

Despite the fact that during the early years of the history of computing, the persistence of data has passed, from decade to decade, through different paradigms of data representation and storage, the trend of change has been fading with the growth of the **relational paradigm**. It is a simple but very efficient technology that has been able to adapt to the majority of data systems that companies demanded and at a cost affordable enough to take the **absolute hegemony of the market** since the last quarter of the last century.

It is true that the relational model has important limitations when it comes to representing information that is not very structured, or excessively dynamic and complex structures, but in spite of all this, relational database management systems (**RDBMS**) **offer great solidity and maturity**.

In recent years, **non-relational DBMSs** have emerged to complement the limitations of relational systems. For example, non-relational databases allow for **great horizontal scalability, so they can be replicated much more easily and at a much lower cost**. On the other hand, they suffer from other limitations, such as the fact that they do not provide immediate updating of information. They are currently widely used in social networking applications and historical data storage.

1.2. Client-server technology

As relational data theories gained momentum and networks became more popular due to increased efficiency at very competitive prices, database management systems based on client-server technology began to be implemented.

Client-server technology made it possible to isolate the data and the specific access programs from the application development. The main reason for this division was probably to enable remote access to data from any computer connected to the network. The truth, however, is that this fact pushed database systems to develop in isolation and to create specific protocols and languages to be able to communicate remotely with applications running on external computers.
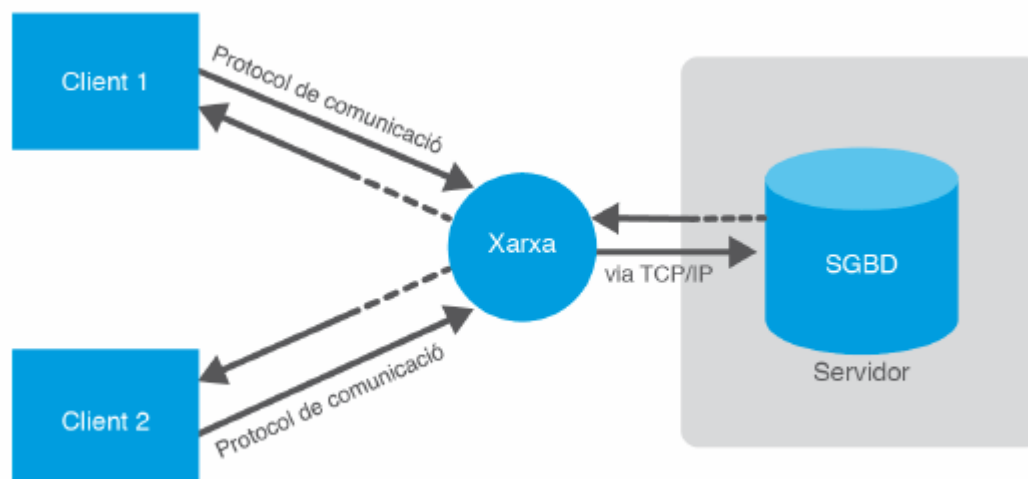

Slowly, the software around databases grew spectacularly trying to respond to a maximum range of demands through highly configurable systems. This is what is known today as **middleware** or the intermediate persistence layer. That is, the set of applications, utilities, libraries, protocols and languages, located on both the server and client sides, which allow remote connection to a database in order to configure it or exploit its data.

1.3. The arrival of standards

Initially, each company developing a DBMS implemented proprietary solutions specific to their system, but soon realised that by working together they could get more out of it and make much faster progress.

Building on relational theory and some early implementations by IBM and Oracle, the data query language called **SQL** was developed. This challenge was certainly a big step forward, **but applications needed APIs with functions that allowed calls to be made from the development language to send queries made with SQL**. That is, a proprietary connection system was still in place, where **each DBMS has its own connection and its own API**.
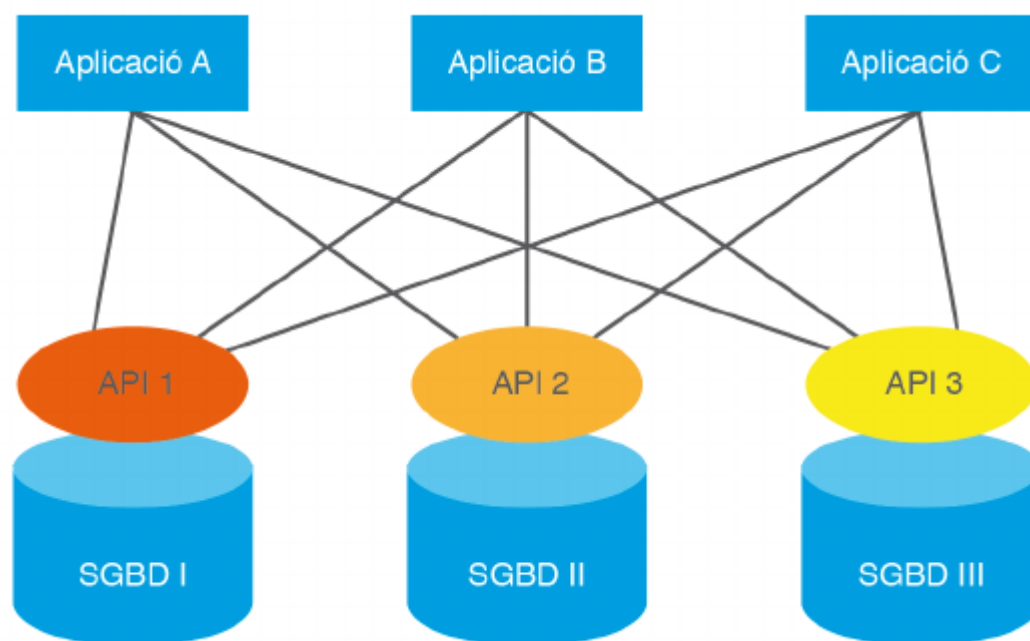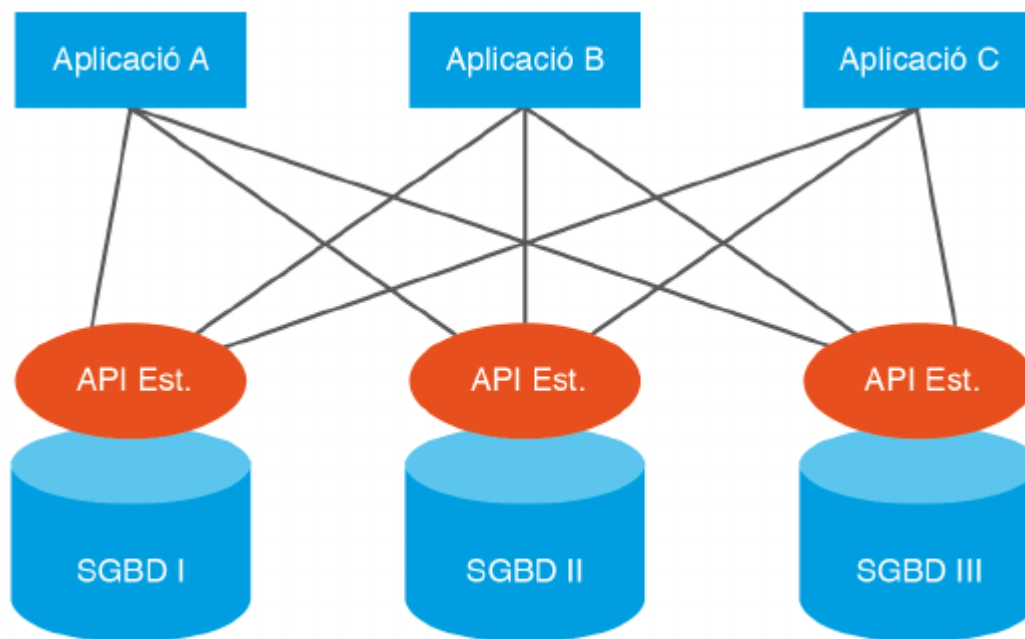
2. Connectors

The history of relational databases has been closely linked to the history of distributed architecture applications and, in particular, client-server architectures. A little more than a quarter of a century ago, application persistence was part of application development, which was conceived as a monolithic whole. Data storage and retrieval was mixed and blurred with data exploitation. Databases often had their own programming languages incorporating specific data access calls and statements.

In reality, this situation did not represent many advantages either for the companies developing the DBMS or for the user companies. The former found that maintaining the development of a programming language was really expensive if they did not want to quickly become obsolete. The user companies, on the other hand, were tied to a programming language that did not always meet all their requirements. In addition, any change of data management system was an impossible task, as it meant having to reprogramme all the company's applications. The development languages had to be decoupled from the DBMS.

Database management systems (DBMSs) of different types have their own specialised languages for dealing with the data they store and the application programs are written in general purpose programming languages, such as Java. In order for these programs to **operate with DBMSs**, mechanisms are needed to allow the application programs to communicate with the databases in these languages. These are implemented in an API and are called **connectors**.

2.1. ODBC

The SQL Access Group, which included prestigious companies in the sector such as Oracle, Informix, Ingres, DEC, Sun and HP, defined a universal API regardless of the development language and the database to be connected.

In 1992, Microsoft and Simba implemented ODBC (Open Data Base Connectivity), an API based on the SQL Access Group definition, which is integrated into the Windows operating system and allows multiple connectors to be added to several SQL databases in a very simple and transparent way, since the connectors are self-installable and fully configurable from the same operating system tools.
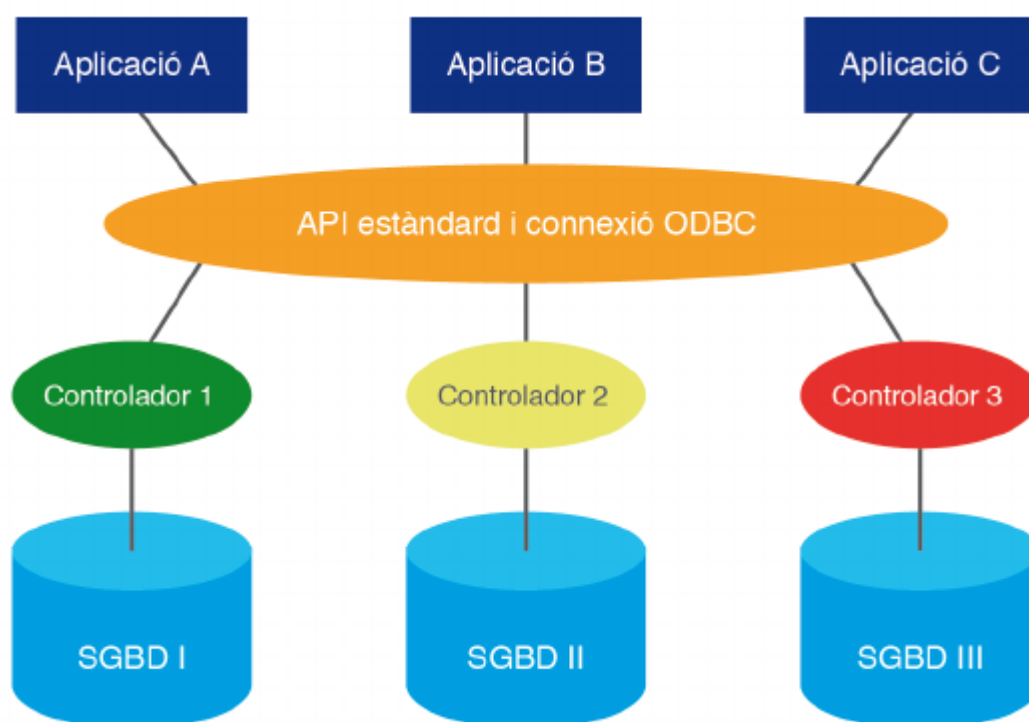
The advent of ODBC was an unprecedented breakthrough on the road to interoperability between databases and programming languages. Most database management system developers incorporated the connectivity drivers into their system utilities and the major programming languages developed specific libraries to support the ODBC API.

Present situation

Currently, ODBC continues to be a suitable initiative for connecting to relational DBMSs. Its development is still led by Microsoft, but there are versions in other operating systems outside the company, such as UNIX/LINUX or MAC. The most popular development languages keep the communication libraries updated with the successive versions that have appeared and most DBMSs have a basic ODBC driver.

Currently, **ODBC** is structured in **three levels**:

1st. Core API. Is the most basic level corresponding to the original specification (based on the SQL Access Group).

2nd. Level 1 API.

3rd. Level 2 API. Both Level 1 and Level 2 add advanced functionalities, such as calls to stored procedures in the system, access security aspects, definition of structured types, etc.

In reality, **ODBC is a low-level specification**, i.e. basic connection-enabling functions that ensure the atomicity of requests, the return of information, the encapsulation of the SQL query language or the retrieval of data obtained in response to a request. The low-level

functionality makes it adaptable to a large number of applications; however, at the expense of a considerable number of lines of code needed to adapt to the logic of each application. This is why other higher-level persistence alternatives have emerged on the basis of ODBC. For example, Microsoft has developed OLE DB or ADO.NET.

2.2. JDBC

Almost simultaneously to ODBC, the company Sun Microsystems, released **JDBC** in 1997, a **database connector API, implemented specifically for use with the Java language**. It is an API quite similar to ODBC in terms of functionality, but adapted to the specificities of Java. That is to say, the functionality is encapsulated in classes (since Java is a totally object-oriented language) and, furthermore, it does not depend on any specific platform, in accordance with the cross-platform characteristic advocated by Java.

This connector will be the API that we will study in detail in this unit, since Java does not have any specific ODBC library. The reasons given by Sun were that ODBC cannot be used directly in Java since it is implemented in C and is not object-oriented. In other words, using ODBC from the Java language would require a whole library to adapt the ODBC API to Java requirements.

Sun Microsystem opted for a solution that allows doing both things at the same time; on the one hand, implementing a Java-specific connector that can communicate directly with any database using drivers in a very similar way to ODBC, and on the other hand, incorporating as standard a special driver that acts as an adapter between the JDBC specification and the ODBC specification. This driver is usually also called JDBC-ODBC bridge. Using this driver you can link any Java application to any ODBC connection.

Currently, the vast majority of DBMSs have JDBC drivers, but in case you have to work with a system that does not, if it has an ODBC driver, you can use the JDBC-ODBC bridge to achieve the connection from Java.

What is JDBC?

JDBC (**J**ava **D**ata**B**ase **C**onnectivity) is the **common interface that Java provides to be able to connect to any RDBMS** (**R**elational **D**ata**B**ase **M**anagement **S**ystem) with this programming language. It provides a complete API (Application Program Interface) to work with Relational Databases in such a way that whatever the engine we connect with, the API will always be the same.

We will simply have to get the Driver corresponding to the engine we want to use, which will depend entirely on it. In spite of that it is not much problem since at the moment we can find a JDBC driver for practically any existing RDBMS.
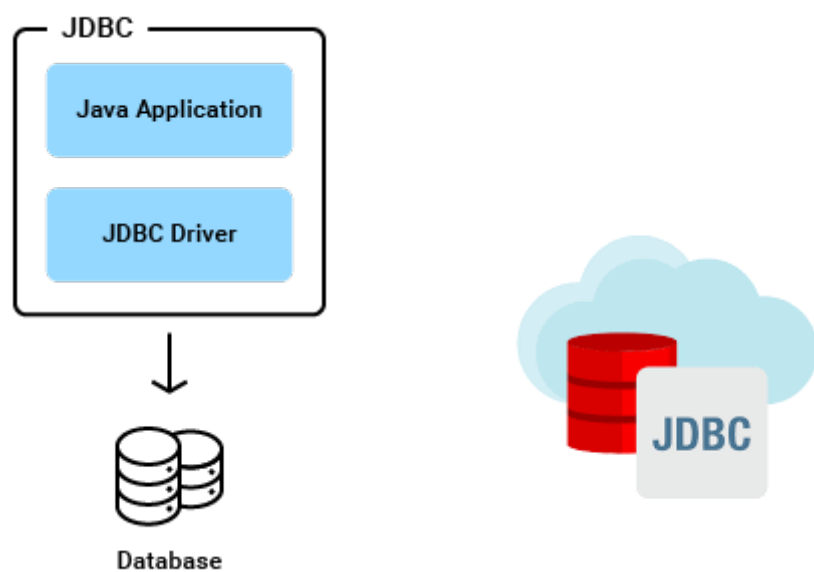
Since the driver is the only thing that depends exclusively on the RDBMS used, it is very easy to write applications whose code can be reused if later on we have to change the database engine or if we want to allow the application to connect to more than one RDBMS so that the user does not have to be tied to the same RDBMS.

2.2.1. JDBC Architecture

Like ODBC, each database management system developer implements its own JDBC drivers. To achieve interoperability between drivers, the JDBC standard library contains a large number of interfaces without the classes that implement them. Each vendor's driver incorporates the classes that implement the JDBC API interfaces.

In this way, the driver used will be completely transparent to the application, i.e. during the development of the application it will not be necessary to know which driver will finally be used for the exploitation of the data, since the application will declare exclusively the JDBC API interfaces.
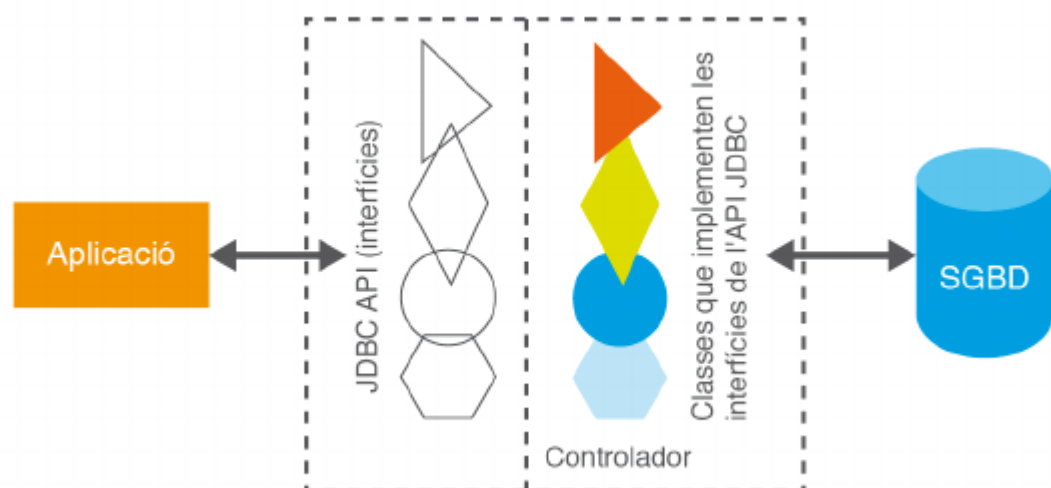
2.2.2. Controller types

JDBC distinguishes between four types of drivers:

1. Type I. **Bridge** drivers such as JDBC-ODBC. They are characterised by the fact that they use a technology outside JDBC and act as an adapter between the JDBC API specifications and the specific technology used. The best known is the JDBC-ODBC bridge driver, but there are others, such as JDBC-OLE DB.

Their main raison d'être is to allow the use of a widespread technology and thus ensure connection to virtually any data source. On the other hand, it must be said that it is necessary for each client to have an ODBC data source management and configuration utility installed (or of the technology used) in addition to the specific ODBC driver of the DBMS, which must be installed and configured. The fact of having to connect using an adapter that acts as a bridge can sometimes cause performance problems and, therefore, it is advisable to use it only as a last alternative.

2. Type II. **Java drivers with partially native API** (Native-API partly Java driver). They are also referred to as simply native. As the name implies, they consist of one part coded in Java and another part using binary libraries installed in the operating system. This type of drivers exist because some data management systems have among their standard utilities connectors specific to the data management system. They are usually proprietary connectors that do not follow any standard, as they tend to predate ODBC or JDBC, but are maintained due to the fact that they are usually highly optimised and efficient.
Using a Java technology called JNI it is possible to implement classes, the methods of which invoke functions from binary libraries installed in the operating system. Type II drivers use this technology to create the JDBC API implementing classes. In some cases it may require an extra installation of certain utilities on the client side, required by the native connector of the managing system.

3. Type III. **Java drivers via network protocol**. This is a driver written entirely in Java that translates JDBC calls to a network protocol against an intermediate server (commonly called Middleware) that may be connected to several DBMSs. This type of driver has the advantage that it uses a protocol independent of the DBMS and therefore the change of data source can be done in a completely transparent way to the clients. This makes it a very flexible system, although, on the other hand, an intermediate server connected to all necessary DBMSs will need to be installed somewhere accessible on the network. This kind of drivers are quite useful when there are a very large number of clients, since DBMS changes will not require any changes to the clients, not even the addition of a new library.

4. Type IV. **Pure Java 100% Java type drivers**. These are also called native protocol drivers. They are drivers written entirely in Java. The calls to the management system are always made through the network protocol used by the management system itself and, therefore, neither native code in the client nor an intermediate server is needed to connect to the data source. It is, therefore, a driver that does not require any type of installation or requirement, which makes it a highly regarded alternative that has recently become more and more popular. In fact, most manufacturers have ended up creating a type IV driver even though they continue to maintain the other types as well.

3. What is Maven?

> The word **maven** comes from the Yiddish meyvn, meaning "one who understands." But to be a maven you have to more than just understand a topic, you have to know its ins and outs. Often mavens are the people that you turn to as experts in a field. You don't become a maven overnight. That kind of expertise comes with an accumulation of knowledge over the years.
>
> Source: https://www.vocabulary.com/dictionary/maven

3.1. Maven explained

Building a software project typically consists of such tasks as downloading dependencies, putting additional jars on a classpath, compiling source code into binary code, running tests, packaging compiled code into deployable artifacts such as JAR, WAR, and ZIP files, and deploying these artifacts to an application server or repository.

Apache **Maven automates these tasks**, minimizing the risk of human making errors while building the software manually and separating the work of compiling and packaging our code from that of code construction.

The key features of Maven are:

- **Simple project setup that follows best practices**. Maven tries to avoid as much configuration as possible, by supplying project templates (named archetypes).

- **Dependency management**. It includes automatic updating, downloading and validating the compatibility, as well as reporting the dependency closures (known also as transitive dependencies).

- **Isolation between project dependencies and plugins**. With Maven, project dependencies are retrieved from the dependency repositories while any plugin's dependencies are retrieved from the plugin repositories, resulting in fewer conflicts when plugins start to download additional dependencies.

- **Central repository system**. Project dependencies can be loaded from the local file system or public repositories, such as Maven Central.

In order to learn how to install Maven on your system, please check this tutorial: https://www.baeldung.com/install-maven-on-windows-linux-mac

Why is Maven useful?

Source: https://stackabuse.com/search/?q=maven

Maven has been an open source project under Apache since 2003, starting at Sonatype

before that. Given its strong backing and immense popularity, Maven is very stable and feature-rich, providing numerous plugins that can do anything from generate PDF versions of your project's documentation to generating a list of recent changes from your SCM. And all it takes to add this functionality is a small amount of extra XML or an extra command line parameter.

Have a lot of dependencies? No problem. **Maven connects to remote repositories** (or you can set up your own local repos) **and automatically downloads all of the dependencies needed to build your project**.

3.2. How Maven works?

It is based on a central file, **pom.xml - the Project Object Model (POM) written in XML-** where you define everything your project needs. Maven manages the project dependencies, compiles, packages and runs the tests. Through plugins, it allows you to do much more, such as generating Hibernate maps from a database, deploying the application, etc.

The most useful thing about Maven is the handling of dependencies. Before Maven, you had to manually download the jar that you needed in your project and copy them manually in the classpath. It was very tedious. With Maven this is over. You only need to define in your **pom.xml** the dependencies and maven downloads them and adds them to the classpath.

More information on POM file:

https://maven.apache.org/guides/introduction/introduction-to-the-pom.html

3.3. Maven and artifacts

a) Libraries and limitations

The concept of library is a concept that is sometimes limited. For example, we may want to use library A in our project. However, it will not be enough to simply want to use the library A, but we will also need to know what exact version of it we need (A v1, A v2, etc.).
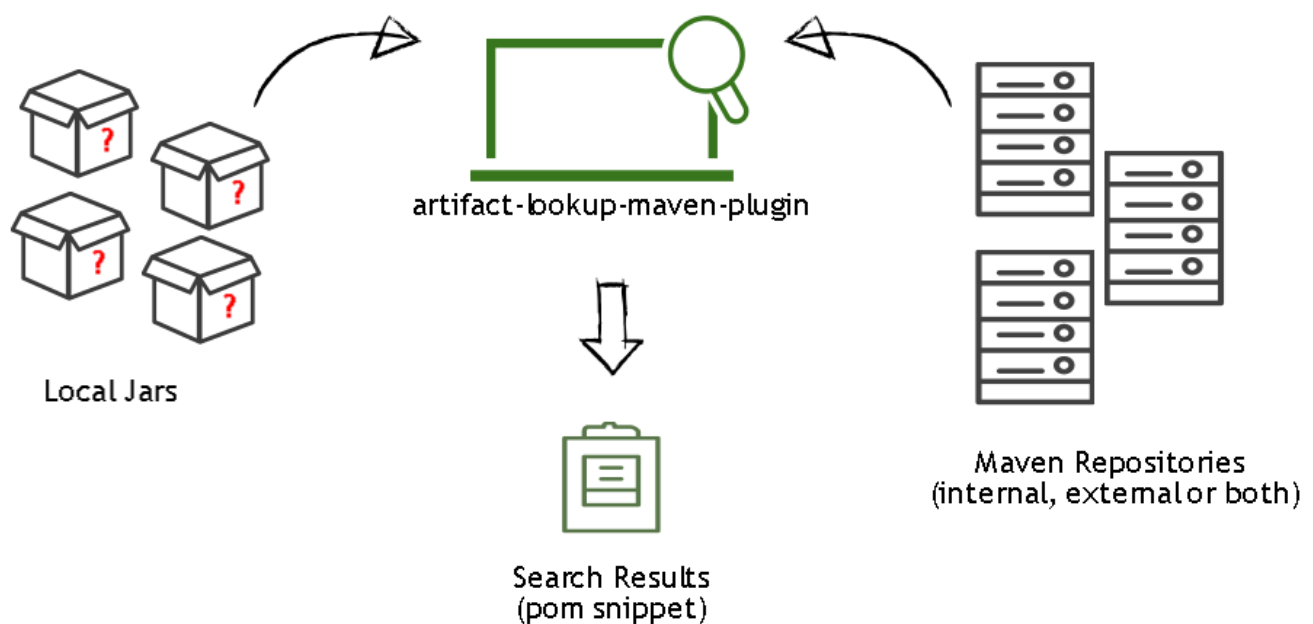
b) Maven solution

Maven solves this problem through the concept of Artifact. An Artifact can be seen as a library with satellites (although it groups more concepts). It contains the library's own classes, but it also includes all the necessary information for its correct management (group, version, dependencies, etc.).

> An artifact is a man-made object that has some kind of cultural significance. If you find a 12th-century vase, it's an artifact of that time. Don't drop it!
>
> Artifact is a combination of two Latin words, arte, meaning "by skill" and factum which means "to make." Usually when you use the word artifact, you are describing something crafted that was used for a particular purpose during a much earlier time.
>
> Source: https://www.vocabulary.com/dictionary/artifact

Local Jars

artifact-lookup-maven-plugin

Search Results
(pom snippet)

Maven Repositories
(internal, external or both)

3.4. Artifacts and POM file

To define an Artifact we need to create a pom.xml file (Project Object Model) which is responsible for storing all the information we have discussed above. This could be an **example**. Just have a look, you don't need to do anything now.

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.genbetadev.proJect1</groupId>
    <artifactId>project1</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>
    <dependencies>
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.17</version>
        </dependency>
    </dependencies>
</project>
```


The file structure can be very complex and may depend on other POMs. In this example we are looking at the simplest possible file. In it we define the name of the Artifact (artifactID) the type of packaging (jar) and also the dependencies it has (log4j). In this way our library is defined in a much clearer way.

3.5. Maven repository and artifacts

Once all the Artifacts we need are correctly defined, Maven provides us with a Repository where to host, maintain and distribute them. Allowing us a correct management of our libraries, projects and dependencies. The use of Maven is nowadays a necessity in any Java/Java EE project of certain entity.
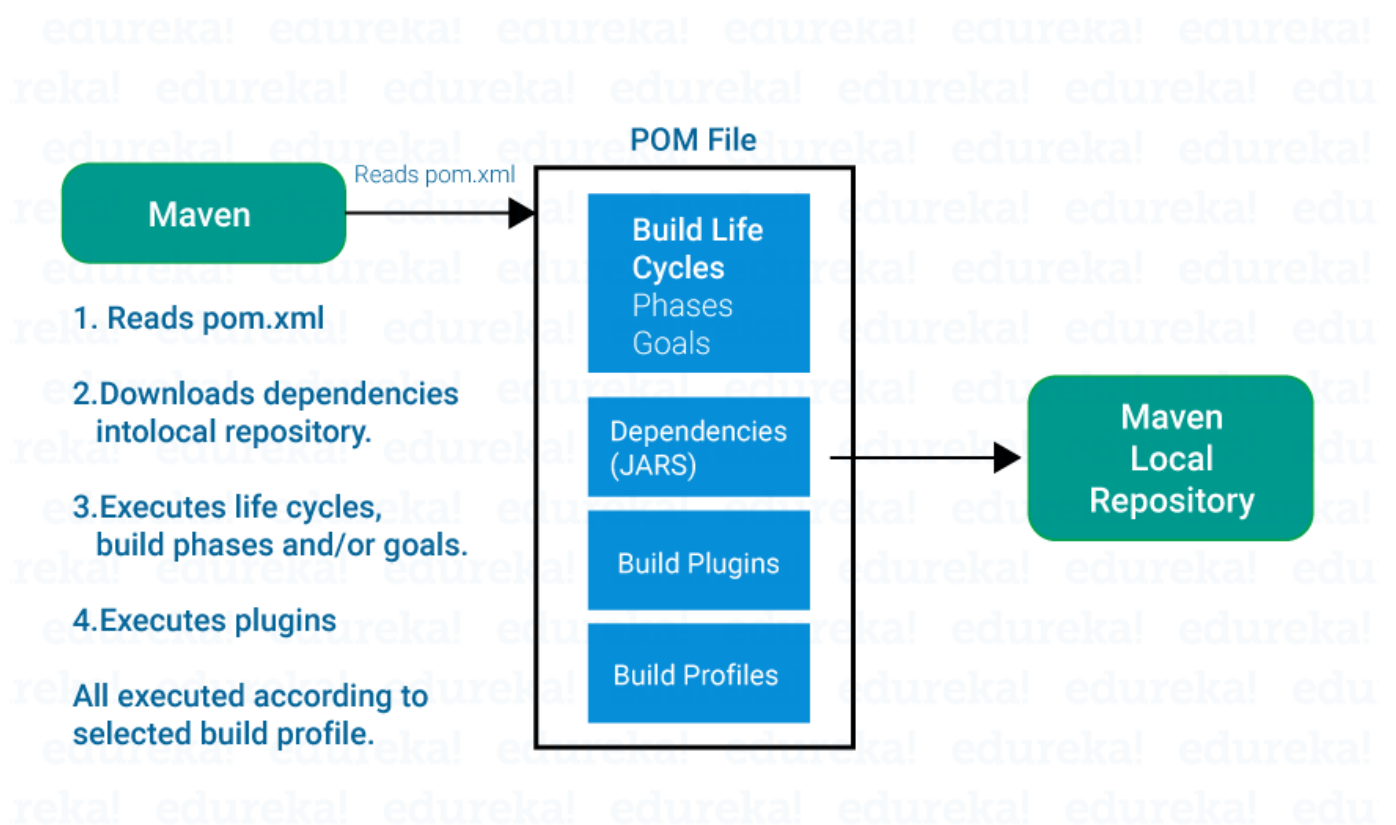
4. Access to RDBMS with Maven

Let's see a complete example of how to access relational databases using Maven.

4.1. Prepare environment and database

We will choose the DBMS, create the necessary structures and populate the database. You can check a ranking of the most used DBMS in the world, updated every month: https://db-engines.com/en/ranking

In this case we choose **MySQL**. The following steps must be followed:

1) Install MySQL

- https://www.digitalocean.com/community/tutorials/how-to-install-mysql-on-ubuntu-20-04-es
- https://dev.mysql.com/doc/mysql-installation-excerpt/8.0/en/

2) Use the provided code ahead to **create a new database** called "**DBCompany**" **and a user** called "**mavenuser**" with all privileges (to make it easier) and password "**ada0486**". You can use either the terminal or the graphical MySQL WorkBench environment.

- Matomo. How do I create a new database and database user in MySQL? https://matomo.org/faq/how-to-install/faq_23484/

3) Within that database, **create a new table** called "**Employee**" with the following fields:

- **TaxID** VARCHAR(9)
- **Firstname** VARCHAR(100)
- **Lastname** VARCHAR(100)
- **Salary** DECIMAL(9,2)

4) Add five random employees to that database/table.

This could be a good example of how to achieve those steps:

```sql
-- Create database https://matomo.org/faq/how-to-install/faq_23484/
CREATE DATABASE DBCompany;


-- Create a user if you are using MySQL 5.7 or MySQL 8 or newer
CREATE USER 'mavenuser'@'localhost' IDENTIFIED WITH mysql_native_password BY '


-- Or if you are using an older version such as MySQL 5.1, 5.5, 5.6:
```

```sql
-- CREATE USER 'mavenuser'@'localhost' IDENTIFIED BY 'ada0486';

GRANT ALL PRIVILEGES ON DBCompany.* TO 'mavenuser'@'localhost';

-- Select the database to use
USE DBCompany;

-- Create the employee's table
CREATE TABLE Employee (
    TaxID        VARCHAR(9),
    Firstname    VARCHAR(100),
    Lastname     VARCHAR(100),
    Salary       DECIMAL(9,2),
    CONSTRAINT emp_tid_pk PRIMARY KEY (TaxID)
);

-- Insert random employees
INSERT INTO Employee (TaxID, Firstname, Lastname, Salary) VALUES ('11111111A',
INSERT INTO Employee (TaxID, Firstname, Lastname, Salary) VALUES ('22222222B',
INSERT INTO Employee (TaxID, Firstname, Lastname, Salary) VALUES ('33333333C',
INSERT INTO Employee (TaxID, Firstname, Lastname, Salary) VALUES ('44444444D',
INSERT INTO Employee (TaxID, Firstname, Lastname, Salary) VALUES ('55555555E',
```

4.2. Create a new Maven Java project

The first step is to open Eclipse, which comes with the integrated Maven environment.

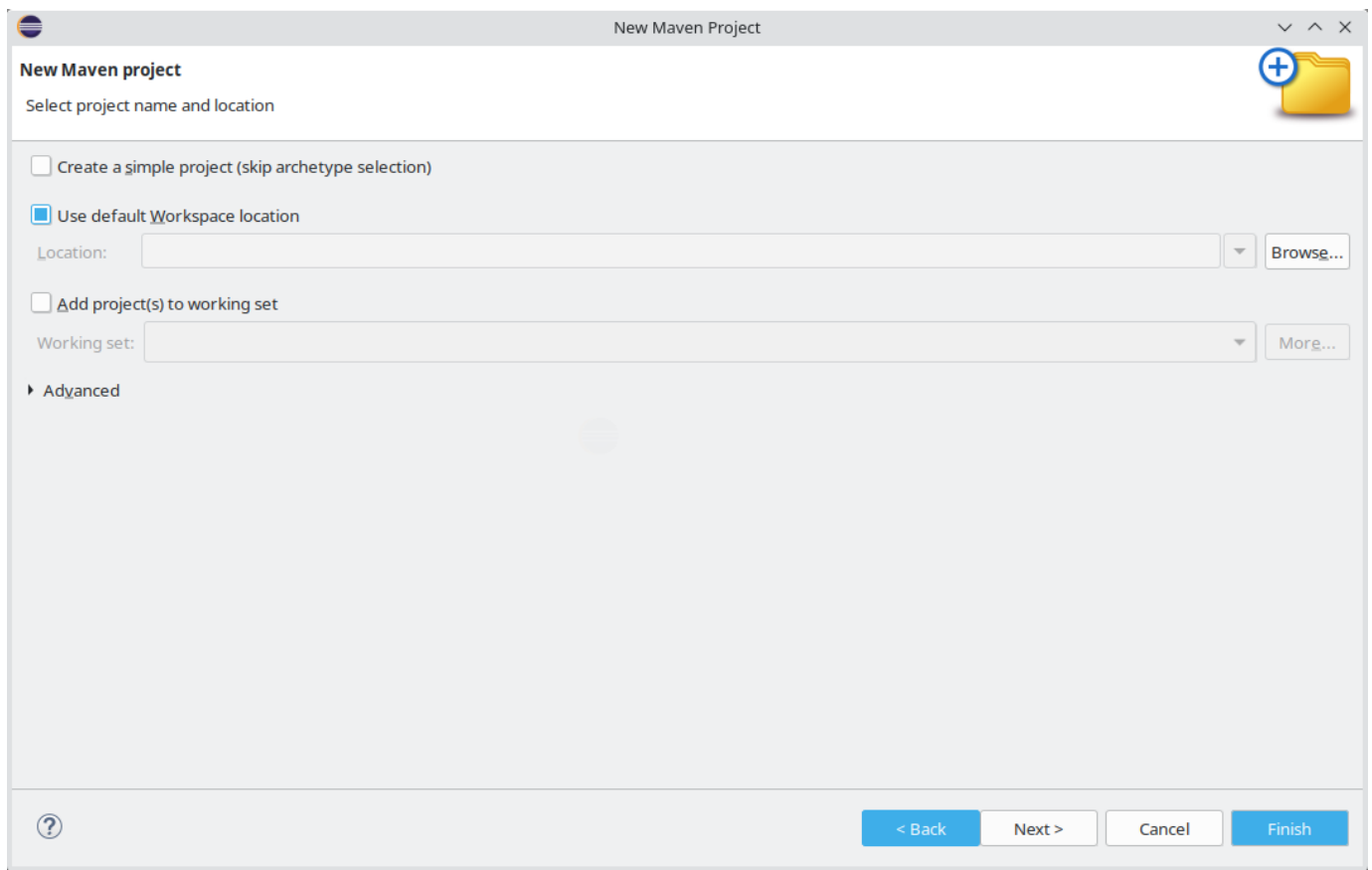- Go to the File menu, option **New** → **Project**.

The dialog box that appears on the screen will display different types of projects.

- Select the **Maven Project** option.
- Click on Next.
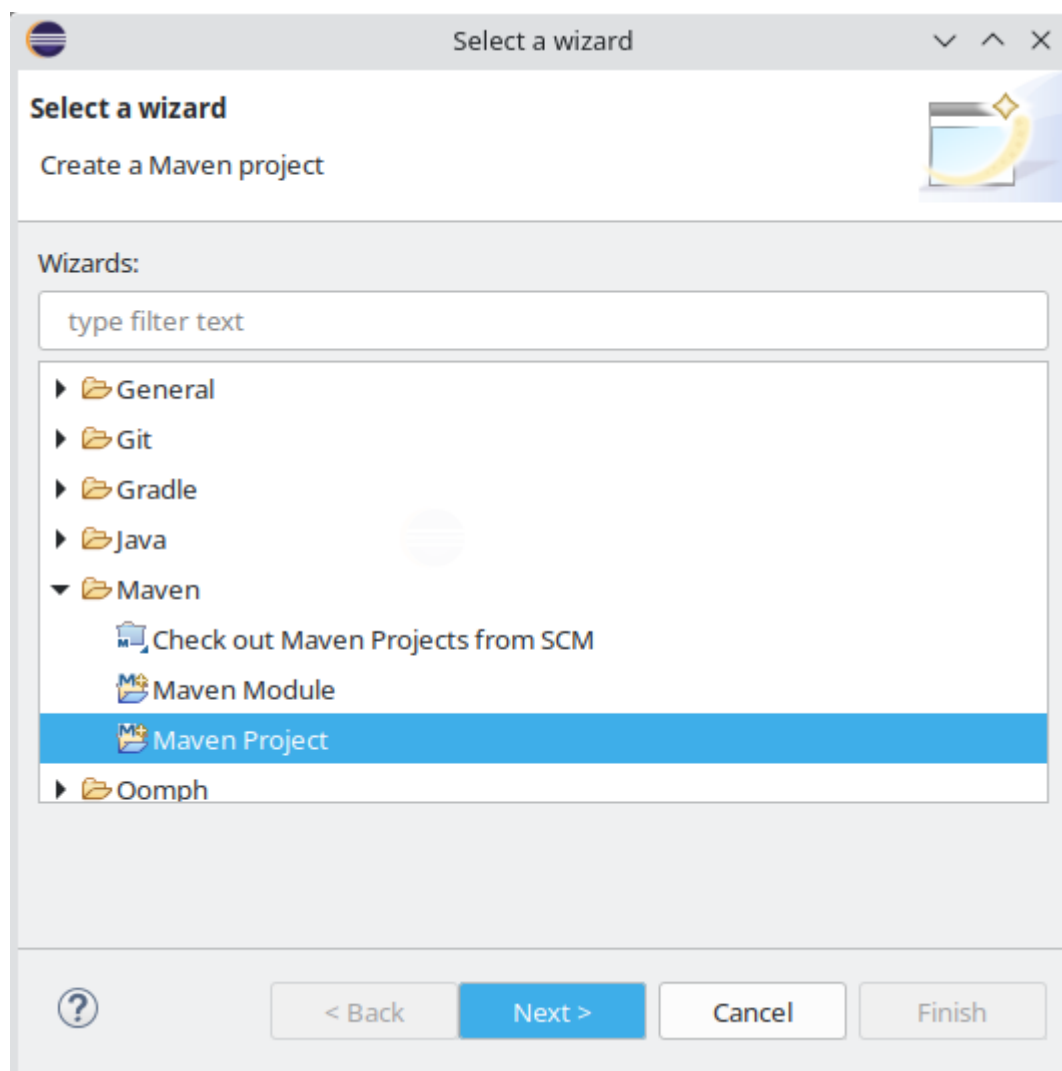
A dialog box will appear. Select the default workspace.

- Click on "Next".
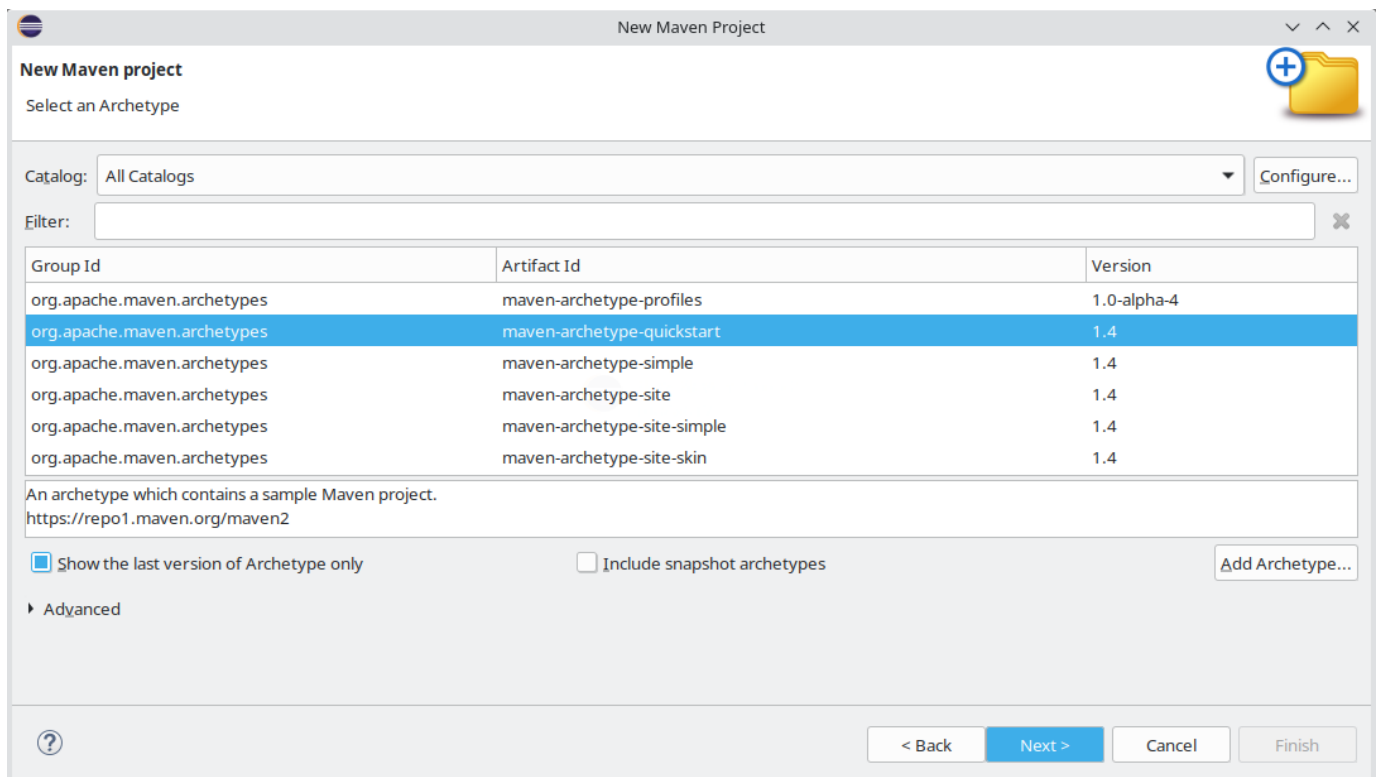
Several Group IDs, Artifact IDs, and Versions will then appear.

- Select a plugin there and click on "Next".

In the next dialog box that appears, you'll complete the following steps:

- Enter the Group ID. "com.simplilearn"

- Enter the Artifact ID. "mavenproject"

- The version will appear on the screen.

- These items can all be modified at a later time if needed. Click on "Finish"
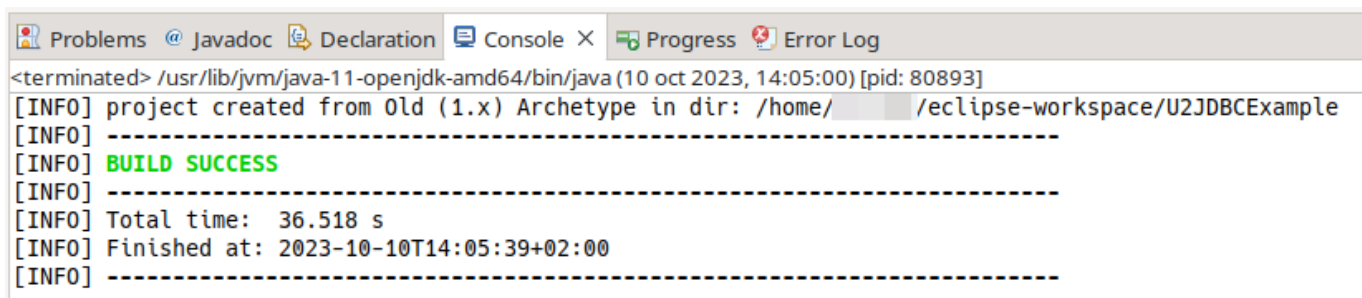
The structure and the necessary dependencies are created below.

- **Type Y** in the terminal.


The project is now created.

```
Problems  @ Javadoc  Declaration  Console ×  Progress  Error Log
<terminated> /usr/lib/jvm/java-11-openjdk-amd64/bin/java (10 oct 2023, 14:05:00) [pid: 80893]
[INFO] project created from Old (1.x) Archetype in dir: /home/        /eclipse-workspace/U2JDBCExample
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  36.518 s
[INFO] Finished at: 2023-10-10T14:05:39+02:00
[INFO] ------------------------------------------------------------------------
```

Next, we select **Project** → **Clean** on our project so the necessary libraries and files have been downloaded correctly. The .jar file is the one that we can distribute in different computers, which contains our project. This is how to get there with Eclipse:
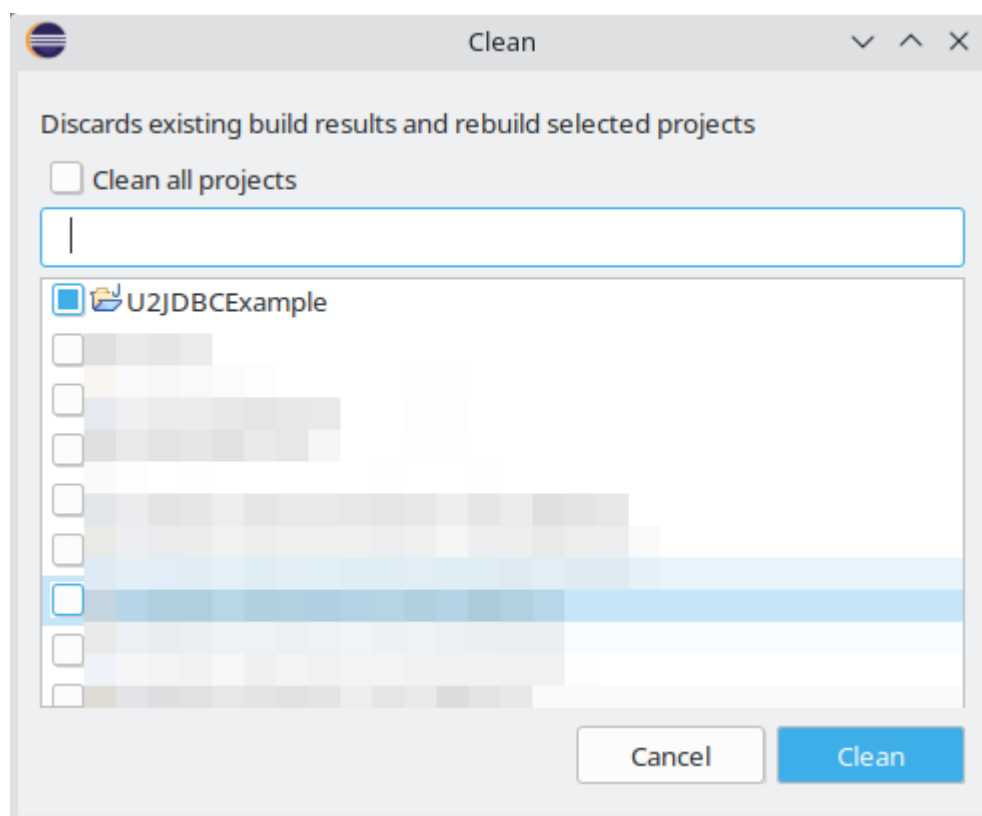
- Open the **pom.xml** file.

You can see all the basic information that you have entered on the screen, such as the Artifact ID, Group ID, etc. You can see the junit dependencies have been added. This process takes place by default in Eclipse.

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.or
```

```xml
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.or
<modelVersion>4.0.0</modelVersion>

<groupId>com.simplilearn</groupId>
<artifactId>U2JDBCExample</artifactId>
<version>0.0.1-SNAPSHOT</version>

<name>U2JDBCExample</name>
<!-- FIXME change it to the project's website -->
<url>http://www.example.com</url>

<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.source>1.7</maven.compiler.source>
  <maven.compiler.target>1.7</maven.compiler.target>
</properties>

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

4.3. Download database library. POM file

Now, we have to go to project folder and set the RDBMS library. In order to do that, we have to:

1. Decide which RDBMS to use. We've chosen MySQL.

2. Look for dependencies:

   ○ https://mvnrepository.com/artifact/mysql/mysql-connector-java

   ○ Also, you can browse on Internet for more information. For example, you can type "pom file MySQL dependencies": https://dev.mysql.com/doc/connectors/en/connector-j-installing-maven.html

3. Check the MySQL version you have installed

   ○ https://phoenixnap.com/kb/how-to-check-mysql-version

   ○ Or type in a terminal:

```
mysql -V
```

1. Copy and paste the dependency code and add it to your pom.xml file inside a new node called <dependencies>, setting the proper version inside <version> node. Your POM file should be like this in Eclipse.

```xml
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.or
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.or
   <modelVersion>4.0.0</modelVersion>

   <groupId>com.simplilearn</groupId>
   <artifactId>U2JDBCExample</artifactId>
   <version>0.0.1-SNAPSHOT</version>

   <name>U2JDBCExample</name>
   <!-- FIXME change it to the project's website -->
   <url>http://www.example.com</url>

   <properties>
     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
     <maven.compiler.source>1.7</maven.compiler.source>
     <maven.compiler.target>1.7</maven.compiler.target>
   </properties>

   <dependencies>
     <dependency>
       <groupId>junit</groupId>
       <artifactId>junit</artifactId>
       <version>4.11</version>
       <scope>test</scope>
     </dependency>
     <!-- https://mvnrepository.com/artifact/com.mysql/mysql-connector-j -->
     <dependency>
         <groupId>com.mysql</groupId>
         <artifactId>mysql-connector-j</artifactId>
         <version>8.0.33</version>
```

```
        </dependency>
      </dependencies>
    </project>
```

5. Now go to **Project → Clean** and check the JAR files your IDE will download for you. **If the jar file is not generated, delete the whole project (with the files) and go back to step 1**.
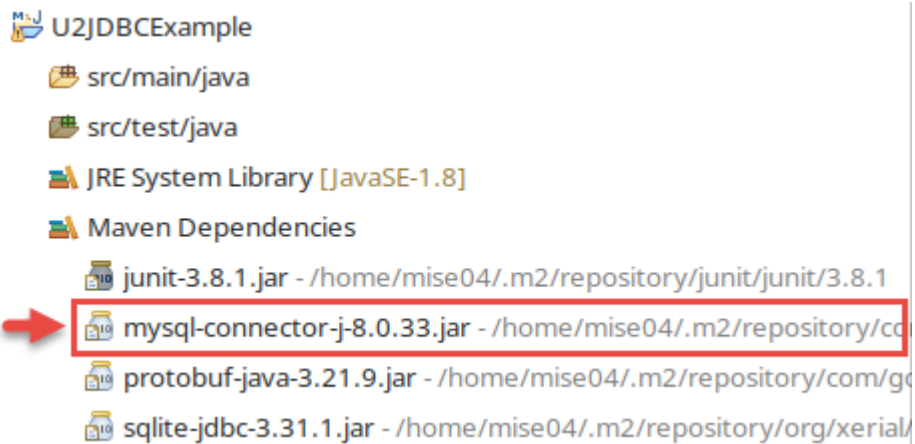
4.4. Establishing a connection

There are several ways to connect to a RDBMS. We'll use DriverManager class explained here:

https://dev.mysql.com/doc/connector-j/8.0/en/connector-j-usagenotes-connect-drivermanager.html

It should be remembered that the JDBC API, apart from some specific classes, is mostly composed of interfaces that the driver implements by giving them the appropriate functionality. To ensure interoperability, applications will never reference the specific classes of any driver but the standard JDBC API interfaces.

To achieve this, **JDBC objects can never be directly instantiated by the application with a new statement**, but are created indirectly by calling a method of an existing class or object that instantiates it internally before returning it to the application. For example, the Connection interface must be instantiated from the static method getConnection of the DriverManager class.

Connection represents a connection to the database, a communication channel between the application and the DBMS. Connection objects will maintain the ability to communicate with the management system as long as they remain open. That is, from the time they are created until they are closed using the close method.

The Connection object is fully linked to a data source, so when requesting the connection, the source must be specified following the JDBC protocol and indicating the url of the data, and if necessary the user and password.

The url will follow the JDBC protocol, always starting with the word jdbc followed by a colon. The rest will depend on the type of driver used, the host where the DBMS is hosted, the port it uses to listen to requests and the name of the database or schema we want to work with.

DriverManager is a very special **JDBC standard API class**, since its mission is to intercede between the application and the JDBC driver or drivers (if there is more than one). From the connection url to a data source, the DriverManager class is able to locate among all the drivers that it has registered the appropriate driver, which allows a connection to be made using the specified url. That is to say, thanks to the DriverManager we don't need to know the name of the main class of the driver (class that has to implement the java.sql.Driver interface of the standard JDBC API).
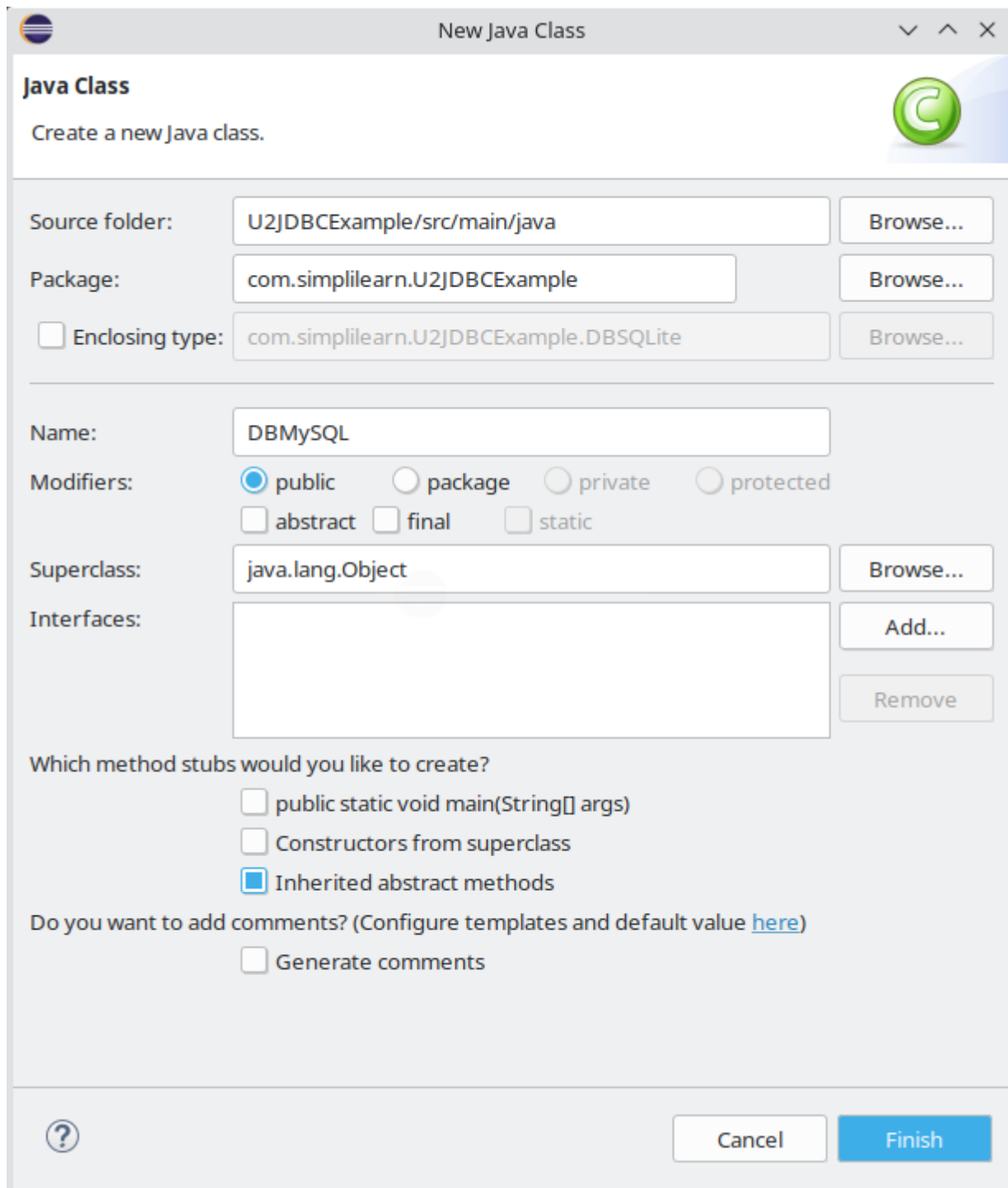
The **getConnection** method of the DriverManager, besides searching and locating the corresponding driver, will obtain a Connection of the selected driver and will return it active to the application.

This method may throw exceptions and the compiler will force us to protect the code by including it in a try-catch statement.

Finally, before starting to work with SQL statements, it **is important to close all open connections before leaving the application**, because if the connection is not closed, the management system will keep the connection in memory, wasting resources. Connection objects have the methods **isClosed** and **close** methods to manage the closing. By invoking the first one, we will know if the connection is still operative or if it has already been closed. The second is the method that performs the closure.

4.4.1. Add the connection string towards our RDBMS

- First, we need to create a new class. For example:

- Second, we have to build our connection string for MYSQL (where **DBNAME** corresponds to the database name):

```
URL="

jdbc:mysql://localhost:3306/DBNAME

?useSSL=false

&useTimezone=true

&serverTimezone=UTC

&allowPublicKeyRetrieval=true

";
```

- Third, we implement the code:

```java
import java.sql.*;

/**
 * =======================================================================
 * Example of accessing a MySQL relational database with JDBC
 * @author Abelardo Martínez
 * =======================================================================
 */

public class DBMySQL {

    /**
     * -----------------------------------------
     * GLOBAL VARIABLES AND CONSTANTS
```

```java
     * -----------------------------------------
     */
    //DB constants
    static final String DBUSER = "mavenuser"; //user
    static final String DBPASSWORD = "ada0486"; //password
    //string to connect to MySQL
    static final String URL = "jdbc:mysql://localhost:3306/DBCompany?useSSL=fa

    /**
     * ------------------
     * MAIN PROGRAMME
     * ------------------
     */
    public static void main( String[] stArgs )
    {
        try {
            //establish the connection to DBCompany
            Connection cnDBCompany = DriverManager.getConnection(URL, DBUSER,

            cnDBCompany.close(); //close the connection to the DB
        } catch (SQLException sqle) {
            sqle.printStackTrace(System.out);
        }
    }
}
```

4.5. Execute SQL statements

For writing SQL statements, JDBC provides for **Statement** objects. These are objects instantiated by Connection, which can send SQL statements to the connected management system to be executed by invoking the executeQuery or executeUpdate method.

- The **executeQuery** method, which we will see later, is used to execute statements that are expected to return data, i.e. queries.

- The **executeUpdate** method, on the other hand, is specifically for statements that modify the connected database but are not expected to return any data.

SQL queries

Queries are the SQL statements that allow us to retrieve, in whole or in part, the data stored in the database. In this case, we are retrieving data from the only table we have.

With Statements objects we can also manage SQL queries, but the executeQuery method must be invoked, because this method returns the set of data corresponding to the query performed. The data is returned using a ResultSet object. Therefore, the execution of queries will look something like the following:

```
ResultSet rs = statement.executeQuery(sentenceSQL);
```

The **ResultSet object contains the result of the query organised by rows**. All rows can be visited one at a time by calling the next method, since each invocation of next will advance to the next row. Immediately after an execution, the returned ResultSet is positioned just before the first row, so to access the first row you have to invoke next once. When the rows run out the next method will return false.

From each row, the value of its columns can be accessed using the diversity of methods available depending on the type of data to be returned and passing as a parameter the number of the column we want to obtain. The name of the methods follows a fairly simple pattern: we will use get as a prefix and the name of the type as a suffix. So, if we wanted to retrieve the second column, knowing that it is a String data type, we would have to invoke:

```
rs.getString(2);
```

It has to be taken into account that columns start counting from the value 1 (not zero). Most DBMSs support the possibility of passing the column name as a parameter, but not being able to guarantee a correct operation in any system, it is normally always chosen the numeric parameter.

Finally, it should be noted that **ResultSet objects must also be closed in the same way as Statements or connections**. Note, however, that **ResultSets are the first ones to be closed**.

4.5.1. Querying table data

Execute the query

We perform the query through the **Statement interface**.

1. To obtain a Statement object we call the createStatement() method of a valid Connection object.

2. The Statement object has the executeQuery() method that executes a query in the database. This method receives as parameter a String that must contain a SQL statement.

3. The result is collected in a ResultSet object, this object is a sort of array or list that includes all the data returned by the DBMS.

Traversing the result

Once we have obtained the results of the query and we have stored them in our ResultSet we only have to traverse them and treat them as we see fit.

- ResultSet has implemented a pointer that points to the first element of the list.

- By means of the next() method, the pointer advances to the next element.
  - The next() method, in addition to advancing to the next record, returns true if there are more records and false if it has reached the end.
  - The getString() and getDouble() methods are used to obtain the values of the different columns. These methods receive the name of the column as a parameter.

Release resources

Finally, we must release all the resources used to ensure the correct execution of the programme.

Connection and Statement instances store, in memory, a lot of information related to the executions performed. In addition, while they remain active, they keep in the DBMS an important set of open resources, destined to serve efficiently the requests of the clients. Closing these objects frees up both client and server resources.

In fact, Statements are objects intimately linked to the Connection object that instantiated them and if they are not specifically closed, they will remain active as long as the connection remains active, even after the variable that referenced them has disappeared.

Moreover, the complexity of these objects means that even though the connection has been

closed, Statements objects that have not been expressly closed will remain in memory longer than previously closed objects, since the Java garbage collector will have to do more checks to ensure that it no longer has any internal or external dependencies and can be removed. That is why it is recommended to always close it manually using the close operation. Closing Statement objects ensures immediate release of resources and removal of dependencies.

If in the same method we have to close a Statement object and the connection that instantiated it, we will have to close first the Statement and then the Connection instance. If we do it the other way round, when we try to close the Statement we will get an exception of type SQLException, since the closing of the connection would have made it inaccessible.

Example

We connect to the database and get all rows:

```java
import java.sql.*;

/**
 * ============================================================================
 * Example of accessing a MySQL relational database with JDBC
 * @author Abelardo Martínez
 * ============================================================================
 */

public class DBMySQL {

    /**
     * ----------------------------------------
     * GLOBAL VARIABLES AND CONSTANTS
     * ----------------------------------------
     */
    //DB constants
    static final String DBUSER = "mavenuser"; //user
    static final String DBPASSWORD = "ada0486"; //password
    //string to connect to MySQL
    static final String URL = "jdbc:mysql://localhost:3306/DBCompany?useSSL=fa
```

```java
    /**
     * -----------------
     * MAIN PROGRAMME
     * -----------------
     */
    public static void main( String[] stArgs )
    {
        try {
            //establish the connection to DBCompany
            Connection cnDBCompany = DriverManager.getConnection(URL, DBUSER,
            Statement staSQLquery = cnDBCompany.createStatement();
            /*
             * -------------------------
             * SQL sentences. Definition
             * -------------------------
             */
            //select all the records
            String stSQLSelect = "SELECT * FROM Employee";
            /*
             * -------------------------
             * SQL sentences. Execution
             * -------------------------
             */
            //retrieve data and display it on screen. SQL select statement
            ResultSet rsEmployee = staSQLquery.executeQuery(stSQLSelect);
            while (rsEmployee.next()) {
                System.out.println("TaxID:" + rsEmployee.getString("TaxID"));
                System.out.println("First name: " + rsEmployee.getString("Firs
                System.out.println("Last name: " + rsEmployee.getString("Lastn
                System.out.println("Salary: " + rsEmployee.getBigDecimal("Sala
            }
            //release resources
            rsEmployee.close(); //close the ResultSet
            staSQLquery.close(); //close the statement
            cnDBCompany.close(); //close the connection to the DB
        } catch (SQLException sqle) {
            sqle.printStackTrace(System.out);
```

```
            }
        }
}
```

And this could be an output:

```
TaxID:11111111A
First name: José
Last name: Salcedo López
Salary: 1279.90
TaxID:22222222B
First name: Juan
Last name: De la Fuente Arqueros
Salary: 1100.73
TaxID:33333333C
First name: Antonio
Last name: Bosch Jericó
Salary: 1051.45
TaxID:44444444D
First name: Ana
Last name: Sanchís Torres
Salary: 1300.02
TaxID:55555555E
First name: Isabel
Last name: Martí Navarro
Salary: 1051.45
```

5. Executing DML sentences

In SQL, DML statements are those that allow us to work with the data in a database.

- **SELECT**. To get data from a database.

- **INSERT**. To insert data into a table.

- **UPDATE**. To modify existing data within a table.

- **DELETE**. To delete all the records in the table; it does not delete the spaces assigned to the records.

To execute any of these statements we must use objects:

- **Statement** → Allows you to execute SQL statements without parameters.

- **PreparedStatement** → Execute SQL statements with parameters.

5.1. The Statement interface

The **Statement interface** allows the creation of Statement objects. As it is an interface, we cannot directly instantiate this type of objects but we must use the createStatement() method of the Connection class. The main methods of this interface are:

Method	Description
ResultSet executeQuery(String query)	Used to execute SQL statements that retrieve data. Returns a ResultSet object with the retrieved data.
int executeUpdate(String Query)	Used to execute manipulation statements such as Insert, Update and Delete. Returns an integer indicating the number of records that have been affected.
boolean execute(String Query)	It can be used to execute any SQL query. In case the statement returns a ResultSet, the execute() method returns true and we must retrieve the ResultSet object through the getResultSet() method. Otherwise (any other type of statement) it will return false and we must use the getUpdateCount() method to retrieve the returned value.

5.2. The PreparedStatement interface

It is very common to use variables within a SQL statement:

- Values to insert, update or delete.

- Filters in selection queries.

- Etc.

For this type of queries with variable part we must make use of the so-called Prepared Statements. The PreparedStatement interface will allow us to create **SQL placeholders** that represent the data that will be added later (Variables). These placeholders are represented by question marks (**?**). For example:

```
String stSQLInsert = "INSERT INTO Employee (TaxID, Firstname, Lastname, Salary
```

Each Placeholder has an index, where 1 is the first element found in the chain. Before executing a PreparedStatement it is necessary to assign values to each of the placeholders. The advantage of this over traditional statements is the fact that you can prepare queries (precompile) only once and have the possibility to execute them as many times as necessary with different input values. They offer great flexibility.

The main PreparedStatement **methods** are:

Method	Description
ResultSet executeQuery()	Similar to its counterpart in the Statement interface.
int executeUpdate()	Similar to its counterpart in the Statement interface.
boolean execute()	Similar to its counterpart in the Statement interface.

In addition, we have a number of methods for assigning values to each of the placeholders.

Method	SQL type
void setString(int index, String value)	VARCHAR
void setBoolean(int index, boolean value)	BIT
void setByte(int index, byte value)	TINYINT
void setShort(int index, short value)	SMALLINT
void setInt(int index, int value)	INTEGER
void setLong(int index, int value)	BIGINT
void setFloat(int index, float value)	FLOAT
void setDouble(int index, double value)	DOUBLE
void setBytes(int index, byte[ ] value)	VARBINARY
void setDate(int index, Date value)	DATE
void setTime(int index, Time value)	TIME

To assign NULL values to the placeholders we must use:

- setNull(index, int typeSQL), where typeSQL is any of the types defined in java.sql.Types.

```java
String stSQLInsert = "INSERT INTO Employee (TaxID, Firstname, Lastname, Salary
PreparedStatement pstaSQLquery = cnDBCompany.createStatement(stSQLInsert);
pstaSQLquery.setString(1, "11111111A");
pstaSQLquery.setString(2, "José");
pstaSQLquery.setString(3, "Salcedo López");
pstaSQLquery.setFloat(4, 1279.90f);
int iQueryvalue = pstaSQLquery.executeUpdate();
System.out.println(iQueryvalue);
```

5.3. The ResultSet class

Through an object of the ResultSet class we can:

- **Collect the values returned by a selection query**.

- **Access the value of any column** of the current tuple through its position or its name.

- Get general information about the query (such as the number of columns, their type, etc.) through the **getMetada()** method.

To access any of the columns of the current tuple we will use the getXXX() methods:

Method	Java type
getString(int numberCol) getString(String nameCol)	String
getBoolean(int numberCol) getBoolean(String nameCol)	boolean
getByte(int numberCol) getByte(String nameCol)	byte
getShort(int numberCol) getShort(String nameCol)	short
getInt(int numberCol) getInt(String nameCol)	int
getLong(int numberCol) getLong(String nameCol)	long
getFloat(int numberCol) getFloat(String nameCol)	float
getDouble(int numberCol) getDouble(String nameCol)	double
getBytes(int numberCol) getBytes(String nameCol)	byte[ ]
getDate(int numberCol) getDate(String nameCol)	Date
getTime(int numberCol) getTime(String nameCol)	Time
getTimeStamp(int numberCol) getTimeStamp(String nameCol)	Timestamp

To traverse a ResultSet we will use the next() method as we have seen in previous

examples.

```java
while (rsEmployee.next()) {
        System.out.println("TaxID:" + rsEmployee.getString("TaxID"));
        System.out.println("First name: " + rsEmployee.getString("Firstname"));
        System.out.println("Last name: " + rsEmployee.getString("Lastname"));
        System.out.println("Salary: " + rsEmployee.getBigDecimal("Salary"));
}
```

6. Executing DDL sentences

Although the bulk of the operations that are carried out from a client application against a database are data manipulation operations, there may be cases in which we are forced to carry out definition operations.

- We have just installed an application on a new computer and we need to create, in an automated way, a local database for its operation (the most common case).

- After a software update, the structure of the DB has changed and we need to update it from our Java application.

- We do not know the structure of the DB and we want to make dynamic queries on it.

Statements that do not return data

Although SQL is essentially a query language, it also includes a number of imperative commands that allow requests to be made to change the internal structures of the DBMS where the data will be stored (instructions known as **DDL** or **D**ata **D**efinition **L**anguage), grant permissions to existing users or create new ones (a subset of instructions known as **DCL** or **D**ata **C**ontrol **L**anguage) or modify the stored data using the insert, update and delete instructions.

Although these are very different statements, from the point of view of communication with the DBMS they behave in a very similar way, following the pattern below:

1. Instantiation from an active connection.

2. Execution of an SQL statement passed as a parameter to the executeUpdate method.

3. Closing of the instantiated Statement object.

Traditional DDL statements are still SQL statements and can therefore be executed in the way we have already studied through the Statement/PreparedStatement interface.

6.1. Definition and modification of structures

Through this mechanism we could create databases and their corresponding structures (tables, restrictions, views, indexes, etc.). It is a very common practice when we are going to use a local DB, and it is currently widely used in applications for mobile devices, local management applications, web content managers, etc.

Traditional DDL statements are still SQL statements and can therefore be executed in the way we have already studied through the Statement/PreparedStatement interface.

We can make use of the IF NOT EXISTS modifier within our SQL code to ensure that the database is created before starting the CRUD processes of our application.

```java
//drop table Employee if exists
String stSQLDrop = "DROP TABLE IF EXISTS Employee";
//create table Employee
String stSQLCreate = "CREATE TABLE Employee ("
        + "TaxID      VARCHAR(9), "
        + "Firstname    VARCHAR(100), "
        + "Lastname     VARCHAR(100), "
        + "Salary       DECIMAL(9,2), "
        + "CONSTRAINT emp_tid_pk PRIMARY KEY (TaxID))";

String stSQLInsert = "INSERT INTO Employee (TaxID, Firstname, Lastname, Salary
        + "('11111111A', 'José', 'Salcedo López', 1279.90),"
            + "('22222222B', 'Juan', 'De la Fuente Arqueros', 1100.73),"
            + "('33333333C', 'Antonio', 'Bosch Jericó', 1051.45),"
            + "('44444444D', 'Ana', 'Sanchís Torres', 1300.02),"
            + "('55555555E', 'Isabel', 'Martí Navarro', 1051.45)";
//delete first employee
String stSQLDelete1 = "DELETE FROM Employee WHERE TaxID='11111111A'";
//select all the records
String stSQLSelect = "SELECT * FROM Employee";
//SQL statements and operations
staSQLquery.executeUpdate(stSQLDrop);
```

```java
System.out.println("Dropped table (if exists) in given database...");
staSQLquery.executeUpdate(stSQLCreate);
System.out.println("Created table in given database...");
staSQLquery.executeUpdate(stSQLInsert);
System.out.println("Populated table with several records in given database...'
staSQLquery.executeUpdate(stSQLDelete1);
System.out.println("Deleted first record table in given database...");
```

6.2. Full example

Complete example of a Maven project for accessing a MySQL database:

```java
import java.sql.*;

/**
 * ========================================================================
 * Example of accessing a MySQL relational database with JDBC
 * @author Abelardo Martínez
 * ========================================================================
 */

public class DBMySQL {

    /**
     * ----------------------------------------
     * GLOBAL VARIABLES AND CONSTANTS
     * ----------------------------------------
     */
    //DB constants
    static final String DBUSER = "mavenuser"; //user
    static final String DBPASSWORD = "ada0486"; //password
    //string to connect to MySQL
    static final String URL = "jdbc:mysql://localhost:3306/DBCompany?useSSL=fa

    /**
     * ------------------
     * MAIN PROGRAMME
     * ------------------
     */
    public static void main( String[] stArgs )
    {
```

```java
try {
    //establish the connection to DBCompany
    Connection cnDBCompany = DriverManager.getConnection(URL, DBUSER,
    Statement staSQLquery = cnDBCompany.createStatement();
    /*
     * --------------------------
     * SQL sentences. Definition
     * --------------------------
     */
    //drop table Employee if exists
    String stSQLDrop = "DROP TABLE IF EXISTS Employee";
    //create table Employee
    String stSQLCreate = "CREATE TABLE Employee ("
            + "TaxID        VARCHAR(9), "
            + "Firstname    VARCHAR(100), "
            + "Lastname     VARCHAR(100), "
            + "Salary       DECIMAL(9,2), "
            + "CONSTRAINT emp_tid_pk PRIMARY KEY (TaxID))";
    //insert employees
    String stSQLInsert = "INSERT INTO Employee (TaxID, Firstname, Last
            + "('11111111A', 'José', 'Salcedo López', 1279.90),"
            + "('22222222B', 'Juan', 'De la Fuente Arqueros', 1100.73)
            + "('33333333C', 'Antonio', 'Bosch Jericó', 1051.45),"
            + "('44444444D', 'Ana', 'Sanchís Torres', 1300.02),"
            + "('55555555E', 'Isabel', 'Martí Navarro', 1051.45)";
    //delete first employee
    String stSQLDeleteFirst = "DELETE FROM Employee WHERE TaxID='11111
    //select all the records
    String stSQLSelect = "SELECT * FROM Employee";
    /*
     * --------------------------
     * SQL sentences. Execution
     * --------------------------
     */
    //SQL statements and operations
    staSQLquery.executeUpdate(stSQLDrop);
    System.out.println("Dropped table (if exists) in given database...
```

```java
            staSQLquery.executeUpdate(stSQLCreate);
            System.out.println("Created table in given database...");
            staSQLquery.executeUpdate(stSQLInsert);
            System.out.println("Populated table with several records in given
            staSQLquery.executeUpdate(stSQLDeleteFirst);
            System.out.println("Deleted first record table in given database..
            //retrieve data and display it on screen. SQL select statement
            ResultSet rsEmployee = staSQLquery.executeQuery(stSQLSelect);
            while (rsEmployee.next()) {
                System.out.println("TaxID:" + rsEmployee.getString("TaxID"));
                System.out.println("First name: " + rsEmployee.getString("Firs
                System.out.println("Last name: " + rsEmployee.getString("Lastn
                System.out.println("Salary: " + rsEmployee.getBigDecimal("Sala
            }
            //release resources
            rsEmployee.close(); //close the ResultSet
            staSQLquery.close(); //close the statement
            cnDBCompany.close(); //close the connection to the DB
        } catch (SQLException sqle) {
            sqle.printStackTrace(System.out);
        }
    }
}
```
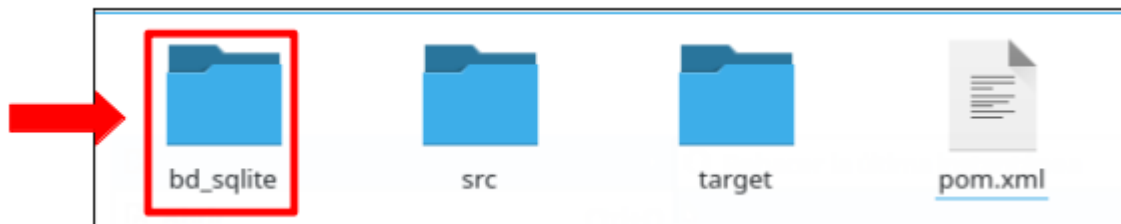
7. Switching to another RDBMS

What about using another RDBMS? It's as easy as:

1. Add its dependencies to the POM file.

2. Change the connection string.

3. THAT'S ALL! No JAR to download, no path to change… nothing to set! **Maven does it all**.

7.1. Go easier with SQLite

First of all, we have to install SQLite. If you want to make it simpler, you don't need to install SQLite. Just download a database example from here: https://www.sqlitetutorial.net /sqlite-sample-database/. Place the .db file inside your project folder.

Let's try with SQLite following these steps:

1. Create a new class called DBSQLite.

2. Look for dependencies:

- https://mvnrepository.com/artifact/org.xerial/sqlite-jdbc
- Also, you can browse on Internet for more information. For example, you can type "pom file SQLite dependencies".

3. Check the SQLite version you have installed.
   - https://database.guide/check-sqlite-version/

4. Copy and paste the dependency code and add it to your pom.xml file inside a new node called <dependencies>, setting the proper version inside <version> node.

5. Your POM file should be like this:

```xml
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.or
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.or
   <modelVersion>4.0.0</modelVersion>

   <groupId>com.simplilearn</groupId>
   <artifactId>U2JDBCExample</artifactId>
   <version>0.0.1-SNAPSHOT</version>

   <name>U2JDBCExample</name>
   <!-- FIXME change it to the project's website -->
   <url>http://www.example.com</url>
```

```xml
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.source>1.7</maven.compiler.source>
  <maven.compiler.target>1.7</maven.compiler.target>
</properties>

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
  <!-- https://mvnrepository.com/artifact/com.mysql/mysql-connector-j -->
  <dependency>
      <groupId>com.mysql</groupId>
      <artifactId>mysql-connector-j</artifactId>
      <version>8.0.33</version>
  </dependency>
  <!-- https://mvnrepository.com/artifact/org.xerial/sqlite-jdbc -->
  <dependency>
      <groupId>org.xerial</groupId>
      <artifactId>sqlite-jdbc</artifactId>
      <version>3.31.1</version>
  </dependency>
</dependencies>
```
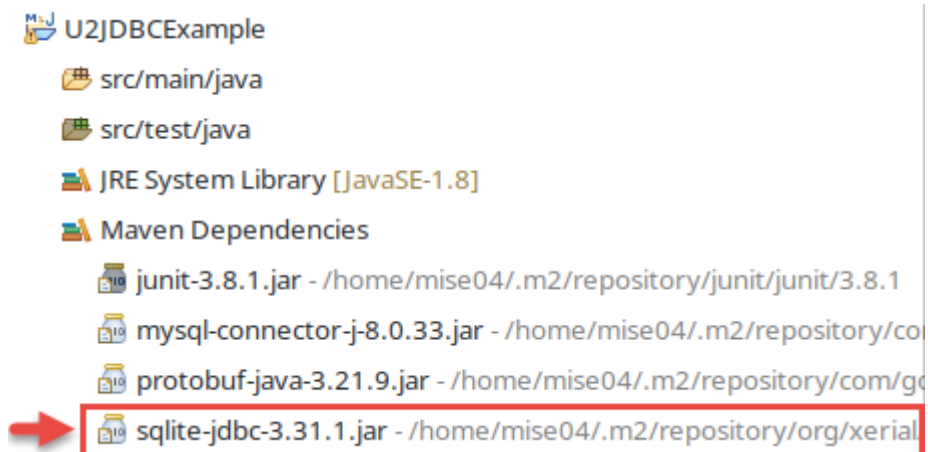
6. Now go to **Project → Clean** and check the JAR files your IDE will download for you.

At this point, you just have to change the connection string and the authentication details at getConnection. In our case, there's no need to set a user and password. Here's what we got:

```java
import java.sql.*;

/**
 * ============================================================================
 * Example of accessing a MySQL relational database with JDBC
 * @author Abelardo Martínez. Based and modified from https://www.sqlitetutori
 * ============================================================================
 */

public class DBSQLite {

    /**
     * ---------------------------------------
     * GLOBAL VARIABLES AND CONSTANTS
     * ---------------------------------------
     */
    //DB constants
    //string to connect to MySQL
    static final String URL = "jdbc:sqlite:bd_sqlite\\chinook.db";


    /**
     * ------------------
```

```java
 * MAIN PROGRAMME
 * -----------------
 */
public static void main(String[] stArgs) {

    try {
        //establish the connection to chinook.db
        Connection cnDBChinook = DriverManager.getConnection(URL);
        System.out.println("Connection to database has been established.")
        Statement staSQLquery = cnDBChinook.createStatement();
        /*
         * --------------------------
         * SQL sentences. Definition
         * --------------------------
         */
        //drop table STUDENTS if exists
        String stSQLDrop = "DROP TABLE IF EXISTS STUDENTS";
        //create table STUDENTS
        String stSQLCreate = "CREATE TABLE STUDENTS "
                + "(studentid INTEGER not NULL, "
                + " first VARCHAR(255), "
                + " last VARCHAR(255), "
                + " age INTEGER, "
                + " PRIMARY KEY ( studentid ))";
        //insert students
        String stSQLInsert = "INSERT INTO STUDENTS (studentid, first, last
                + "(1, 'PETER', 'KENIX', 38),"
                + "(2, 'XENA' ,'HARRIS', 28)";
        //delete first student
        String stSQLDeleteFirst = "DELETE FROM STUDENTS WHERE studentid=1'
        //select all the records
        String stSQLSelect = "SELECT studentid, first, last, age FROM STUD
        /*
         * --------------------------
         * SQL sentences. Execution
         * --------------------------
         */
```

```java
                //SQL statements and operations
                staSQLquery.executeUpdate(stSQLDrop);
                System.out.println("Droped table (if exists) in given database...'
                staSQLquery.executeUpdate(stSQLCreate);
                System.out.println("Created table in given database...");
                staSQLquery.executeUpdate(stSQLInsert);
                System.out.println("Populated table with several records in given
                staSQLquery.executeUpdate(stSQLDeleteFirst);
                System.out.println("Deleted record table in given database...");
                //retrieve data and display it on screen. SQL select statement
                ResultSet rsStudent = staSQLquery.executeQuery(stSQLSelect);
                while (rsStudent.next()) {
                    System.out.println("Student Id:" + rsStudent.getInt("studentic
                    System.out.println("First: " + rsStudent.getString("first"));
                    System.out.println("Last: " + rsStudent.getString("last"));
                    System.out.println("Age: " + rsStudent.getInt("age"));
                }
                //release resources
                rsStudent.close(); //close the ResultSet
                staSQLquery.close(); //close the statement
                cnDBChinook.close(); //close the connection to the DB
            } catch (SQLException sqle) {
                sqle.printStackTrace(System.out);
            }
        }


    }
```

DAM. Unit 2. ACCESS TO DATABASES. Part 1. Working with Relational Databases

8. Bibliography

⊘Sources

- How to Create a Maven Project in Eclipse. https://www.simplilearn.com/tutorials/maven-tutorial/maven-project-in-eclipse

- JDBC - Create Table Example. https://www.tutorialspoint.com/jdbc/jdbc-create-tables.htm

- Josep Cañellas Bornas, Isidre Guixà Miranda. Accés a dades. Desenvolupament d'aplicacions multiplataforma. Creative Commons. Departament d'Ensenyament, Institut Obert de Catalunya. Dipòsit legal: B. 29430-2013. https://ioc.xtec.cat/educacio/recursos

- Alberto Oliva Molina. Acceso a datos. UD 2. Manejo de conectores. IES Tubalcaín. Tarazona (Zaragoza, España).

- W3Schools. MySQL RDBMS. https://www.w3schools.com/mysql/mysql_rdbms.asp

- Connecting to MySQL Using the JDBC DriverManager Interface. https://dev.mysql.com/doc/connector-j/8.1/en/connector-j-usagenotes-connect-drivermanager.html

- Tutorials point. JDBC - Insert Records Example. https://www.tutorialspoint.com/jdbc/jdbc-insert-records.htm

- SQLite Tutorial. SQLite Sample Database. https://www.sqlitetutorial.net/sqlite-sample-database/