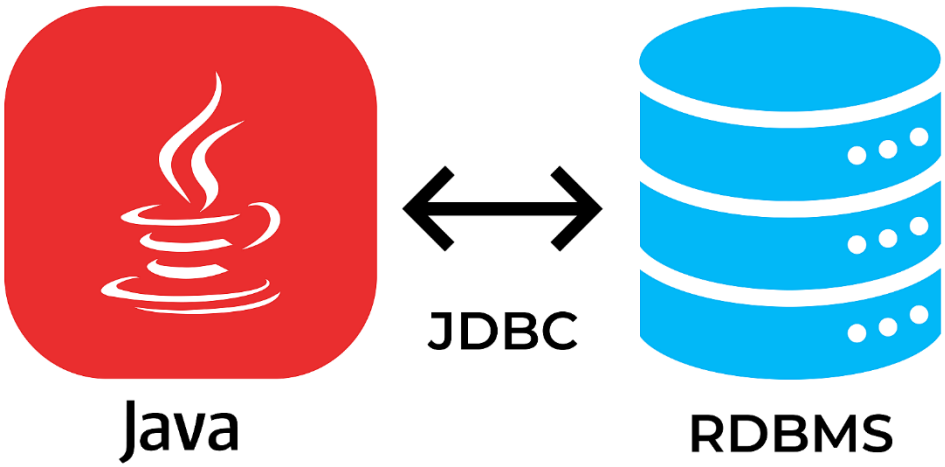


Persistence in Databases.
Connectors.

Content

1. Database connection technologies

In computing, a database is an organized collection of data stored and accessed electronically from a computer system. Where databases are more complex they are often developed using formal design and modeling techniques.

The database management system (DBMS) is the software that interacts with end users, applications, and the database itself to capture and analyze the data. The DBMS software additionally encompasses the core facilities provided to administer the database. The sum total of the database, the DBMS and the associated applications can be referred to as a "database system". Often the term "database" is also used to loosely refer to any of the DBMS, the database system or an application associated with the database.

There are many types of DBMS's. Possibly, one of the most employed are Oracle and Microsoft SQL Server, speaking about the enterprise use. MySQL has also a lot of presence over the servers spreaded along the Internet. And SQLite is widely used for small data embedded in applications. Another well-known manager and that we will be interested in in this text because it has additional advantages, such as being open source, be available for different operating systems and have some support for Object-Oriented Databases, it is PostgreSQL.

There are different ways to connect to database managers, such as:

- Embedded databases, in which the connection between the program and the database is direct. This type of connection allows the highest speed in the communication, but, in case you need to connect to a different database, it will imply to perform a lot of changes in the application, which is not desirable.
- Alternatively, it is preferable to use a generic API (programming interface applications), which allows connecting in the same way to different database managers, as well as sending requests and receiving tuples with the responses in a uniform manner. One of the most widespread standards is ODBC (open database connectivity), commonly used in C, C++, C# and Visual Basic.
- ODBC is unsuitable to be used from Java, because it is based on C language calls (which involves using pointers, which do not exist in Java), and the conversion can cause security risks and lack of robustness. Therefore, in Java an alternative fully based on objects is usually used, called Java Database Connectivity (JDBC).

Oftenly, to use JDBC, you will have to install a "driver" that permits access from the language or tool being used to the desired database manager. When using

Java, these drivers are often called “connectors”. If no JDBC driver exist for your DBMS, but there is an ODBC option, it is possible to use an "ODBC-JDBC bridge".

2. PostgreSQL installation

PostgreSQL, also known as Postgres, is a free and open-source relational database management system (RDBMS) emphasizing extensibility and SQL compliance. It was originally named POSTGRES, referring to its origins as a successor to the Ingres database developed at the University of California, Berkeley. In 1996, the project was renamed to PostgreSQL to reflect its support for SQL. After a review in 2007, the development team decided to keep the name PostgreSQL and the alias Postgres.

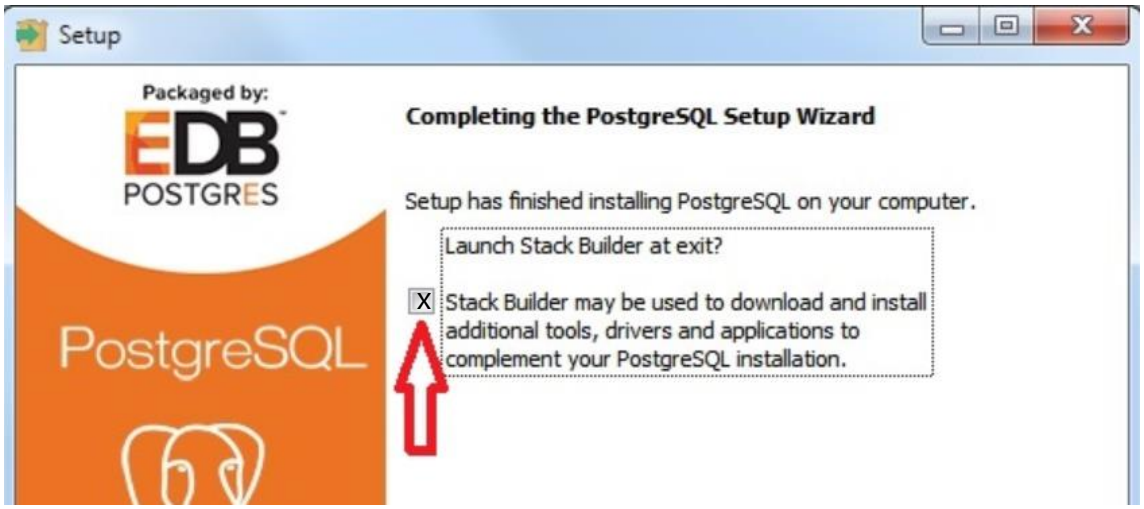
PostgreSQL features transactions with Atomicity, Consistency, Isolation, Durability (ACID) properties, automatically updatable views, materialized views, triggers, foreign keys, and stored procedures. It is designed to handle a range of workloads, from single machines to data warehouses or Web services with many concurrent users. It is the default database for macOS Server and is also available for Windows, Linux, FreeBSD, and OpenBSD.

For this course we will use PostgreSQL as RDBMS, both for being open-source and their multi-platform availability (as Java).

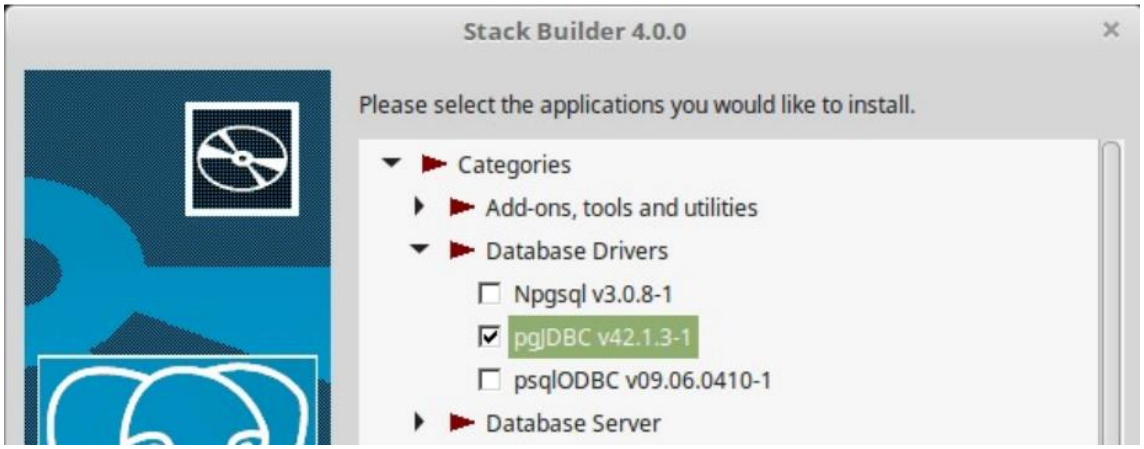
To install PostgreSQL, you can use this tutorial:

https://www.sqlservercentral.com/articles/installing-postgresql-11-on-windows-step-by-step-instruction

Stack Builder installation (and execution) is mandatory to install the JDBC drivers. So, in the final step, we should check the checkbox to launch Stack Builder:



When Stack Builder has been launched, we will select the current PostgreSQL installation (we will have just one) and the JDBC driver to be installed:



We will just accept all the default installation values both for PostgreSQL and for the JDBC driver. All done.

To manage the database, the use of the utility Pgadmin4 is strongly recommended.

2.1. Activities

1. Using the PgAdmin4 utility, create a new database called VTInstitute (Vocational Training Institute). Within this database, create a new table called 'subjects' with three fields:

    - Code: a sequence field (find out what a sequence field is in PostgreSQL).
    - Name: the name of the course, a string with a maximum length of 50 chars.
    - Year: an integer showing in which year is due to be taken the course.

2. Using the Pgadmin Query tool and the SQL INSERT clause, add the following data to the 'subjects' table:

    - 'DATA ACCESS', second year
    - 'DEVELOPMENT ENVIRONMENTS', first year
    - 'DATABASE MANAGEMENT SYSTEMS', first year
    - 'SERVICES AND PROCESSES', second year

3. Create a new database called 'employees'. Download and run the script 'employee-dept.sql'. Check that the script has finished successfully creating two tables with records.

3. Connecting to PostgreSQL via JDBC

The first step needed to interact with a database via JDBC is to connect to that database. The programming involved to establish a JDBC connection is fairly simple. Basically, you will need to do the following:

1. Register JDBC Driver: This step causes the JVM to load the desired driver implementation into memory so it can fulfill your JDBC requests.
2. Database URL Formulation: This is to create a properly formatted address that points to the database to which you wish to connect.
3. Create Connection Object: Finally, code a call to the `DriverManager` object's `getConnection()` method to establish actual database connection.

3.1. Register JDBC Driver

You must register the driver in your program before you use it. Registering the driver is the process by which the PostgreSQL driver's class file is loaded into the memory, so it can be used as an implementation of the JDBC interfaces.

You need to do this registration only once in your program. You can register a driver in one of two ways, using the `Class.forName()` static method, or registering an specific driver.

If you are using JDBC 4.0 or superior, any JDBC 4.0 drivers that are found in your class path are automatically loaded. (However, you must manually load any drivers prior to JDBC 4.0 with the method Class.forName).

3.1.1. Class.forName()

The most common approach to register a driver is to use Java's `Class.forName()` method, to dynamically load the driver's class file into memory, which automatically registers it. This method is preferable because it allows you to make the driver registration configurable and portable.

The following example uses `Class.forName()` to register the PostgreSQL driver.

```
try {
   Class.forName("org.postgresql.Driver")
}
catch (ClassNotFoundException ex) {
   System.out.println("Error: unable to load driver class!");
   System.exit(1);
}
```

3.1.2. Registering a specific driver

The second approach you can use to register a driver, is to use the static `DriverManager.registerDriver()` method. You should use the `registerDriver()` method if you are using a non-JDK compliant JVM, such as the one provided by Microsoft.

3.2. Database URL Formulation

After you've loaded the driver, you can establish a connection using the `DriverManager.getConnection()` method. There are three options to establish a connection using the `getConnection` method:

- getConnection(String url)
- getConnection(String url, Properties prop)
- getConnection(String url, String user, String password)

Each form requires a database URL. A database URL is an address that points to your database. Formulating a database URL is where most of the problems associated with establishing a connection occurs.

The following table lists down the most popular JDBC driver names and database URLs.

RDBMS	JDBC driver name	URL format
PostgreSQL	org.postgresql.Driver	**jdbc:postgresql**://hostname:portNumber/databaseName
SQL Server	com.microsoft.sqlserver.jdbc.SQLServerDriver	**jdbc:sqlserver:**//hostname:portNumber;database=databaseName;
MySQL	com.mysql.jdbc.Driver	**jdbc:mysql:**//hostname/databaseName
ORACLE	oracle.jdbc.driver.OracleDriver	**jdbc:oracle:thin:@**hostname:portNumber:databaseName

3.3. Create Connection Object

We have listed down three different ways in which `DriverManager.getConnection()` method creates a connection object. The most commonly used form requires you to pass a database URL, a *username*, and a *password*, like in the following example:

```
String URL = "jdbc:postgresql://localhost:5432/VTInstitute";
String USER = "postgres";
String PASS = "pwd";
```

```
Connection conn = DriverManager.getConnection(URL, USER, PASS);
```

A second form requires only a database URL. However, in this case, the database URL must include the username and password and has the following form:

```
"jdbc:postgresql://localhost:5432/VTInstitute?user=postgres&password=p
wd"
```

Please note that, depending of your specific DBMS, the URL can be slightly different. For example, if you are using Oracle, then the URL will have this form:

```
jdbc:oracle:driver:username/password@database
```

Finally, the third form requires a database URL and a Properties object:

```
String url = "jdbc:postgresql://localhost:5432/VTInstitute";
Properties props = new Properties();
props.setProperty("user","postgres");
props.setProperty("password","pwd");
Connection conn = DriverManager.getConnection(url, props);
```

3.4. Closing JDBC Connections

At the end of your JDBC program, it is required explicitly to close all the connections to the database to end each database session. However, if you forget, Java's garbage collector will close the connection when it cleans up stale objects.

Relying on the garbage collection, especially in database programming, is a very poor programming practice. You should make a habit of always closing the connection with the `close()` method associated with connection object.

To ensure that a connection is closed, you could provide a `finally` block in your code. A `finally` block always executes, regardless of an exception occurs or not. Therefore, the proper way to establish and close a connection would look like this:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConnectionTest {
     public static void main(String[] args)
        throws ClassNotFoundException, SQLException {
          Connection conn = null;
          try {
              Class.forName("org.postgresql.Driver");
              String url =  "jdbc:postgresql://localhost:5432/" +
                            "VTInstitute";
```

```
            String user = "postgres";
            String password = "pwd";
            conn = DriverManager.getConnection(url, user, password);
            System.out.println("Connection established!!!");
        }
        catch (ClassNotFoundException e) {
            System.out.println("Cannot load PostgreSQL Driver!!");
        }
        catch (SQLException e) {
            System.out.println("Connection couldn't be established!");
        }
        finally {
            if ( conn != null )
                conn.close();
         }
    }
}
```

4. Statements

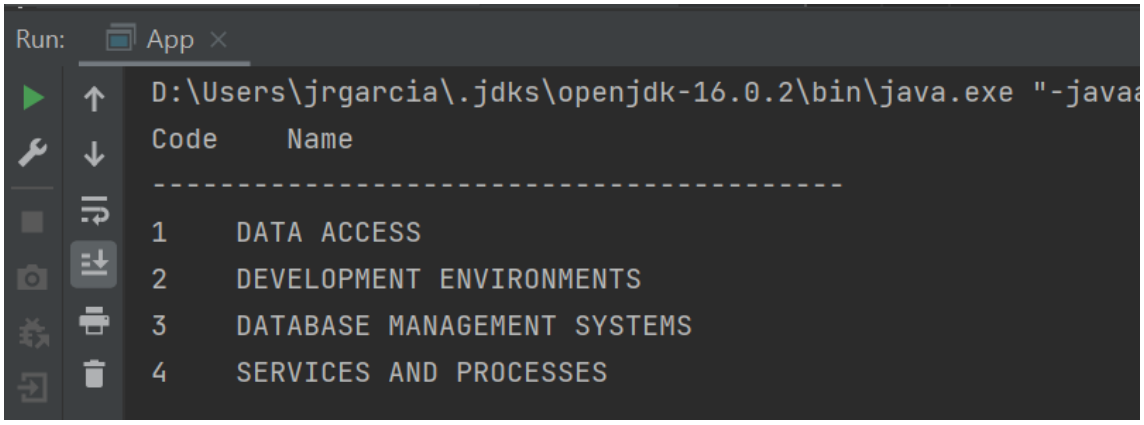
Once a connection is obtained we can interact with the database. The JDBC `Statement` interface define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database. It also defines methods that help bridging data type differences between Java and SQL data types used in a database.

The first step to create an statement is to open a connection to the desired database. Once made, statements can be created from the connection object:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
public class ConsultaBD1 {
      public static void main(String[] args)
        throws ClassNotFoundException, SQLException {
           Class.forName("org.postgresql.Driver");
           String url = "jdbc:postgresql://localhost:5432/VTInstitute";
           String user = "postgres";
           String password = "pwd";
           Connection con = DriverManager.getConnection(url, user,
password);
           Statement statement = con.createStatement();
           String SQLsentence = "SELECT * FROM subjects ORDER BY code";
           ResultSet rs = statement.executeQuery(SQLsentence);
```

```
            System.out.println("Code" + "\t" + "Name");
            System.out.println("----------------------------------------
-");
            while (rs.next()) {
            System.out.println(rs.getString(1)     +     "\t     "     +
rs.getString(2));
            }
            rs.close();
            con.close();
      }
}
```

The execution of this application should return something similar to this:



As you can see in this example, we use the method *executeQuery* to retrieve data from the database through a *ResultSet* object. This is tipically associated to the SELECT SQL statement. For other DML sentences, like INSERT, UPDATE or DELETE, that don't return a list of rows but alter the information in the database, we will better use *executeUpdate*. There is a third option, that is to use the execute method. Here below you will find a detailed explanation of the three methods:

- **boolean execute (String SQL)**: Returns a boolean value of true if a `ResultSet` object can be retrieved; otherwise, it returns false. Use this method to execute SQL DDL statements or when you need to use truly dynamic SQL.
- **int executeUpdate (String SQL)**: Returns the number of rows affected by the execution of the SQL statement. Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.
- **ResultSet executeQuery (String SQL)**: Returns a `ResultSet` object. Use this method when you expect to get a result set, as you would with a SELECT statement.

4.1. Activities

1. Create a Maven project with IntelliJ IDEA to run the previous application. Verify that you obtain the desired result. You will have to add the PostgreSQL dependency to the POM.XML file in order to properly run the application. Modify the application to show also the year in which every course is taken. Finally, add an error control block (try … catch …).
2. Using the method `executeUpdate`, modify the previous application to insert a new subject into the 'subjects' table (for example, 'Markup Languages', first year). You should print out on the screen the value returned by the method.
3. Using either `execute` or `executeUpdate`, alter the table 'subjects' to add a new field, 'Hours', an integer showing the yearly hours of the subject.

5. Closing resources

5.1. Apache common DBUtils

As you have probably noticed in both the example in page 9 and activity 4.1.1, to release the resources of every object we use to connect to the database (`connection`, `statement`, `resultset`) can be quite cumbersome. In order to avoid a *`finally`* block looking like this:

```
Connection conn = null;
Statement st = null;
ResultSet rs = null;
try {
 // Code
 ...
} catch (SQLException ex) {
 // Error control block
 ...
}
finally {
 if (rs != null) {
 try {
      rs.close();
 }
 catch (SQLException e) { /* ignored */}
 }
 if (st != null) {
 try {
      st.close();
 }
```

```
 catch (SQLException e) { /* ignored */}
 }
 if (conn != null) {
 try {
      conn.close();
 }
 catch (SQLException e) { /* ignored */}
 }
}
```

We can include the *DBUtils* library in our project (by adding the corresponding reference to the POM.XML file) and replace this bunch of code for just the following three lines:

```
finally {
      DbUtils.closeQuietly(rs);
      DbUtils.closeQuietly(st);
      DbUtils.closeQuietly(conn);
}
```

5.2. Try-with-resources

Starting from JAVA 7, it is possible to declare a *try-with-resources* block, that is, a *try* statement that declares one or more resources. A resource is an object that must be closed after the program is finished with it. The *try-with-resources* statement ensures that each resource is closed at the end of the try statement. Any object that implements *java.lang.AutoCloseable*, which includes all objects which implement *java.io.Closeable*, can be used as a resource.

So the code in page 9 could be rewritten as follows:

```
static final String url =  "jdbc:postgresql://localhost:5432/" +
                           "VTInstitute";
static final String user = "postgres";
static final String password = "pwd";

     public static void main(String[] args)
        throws ClassNotFoundException, SQLException {
          try {
              Class.forName("org.postgresql.Driver");
          }
          catch (ClassNotFoundException e) {
              System.out.println("Cannot load PostgreSQL Driver!!");
          }
```

```
            try (Connection conn = DriverManager.getConnection(url, user,
password)) {
                System.out.println("Connection established!!!");
            }
            catch (SQLException e) {
                System.out.println("Connection couldn't be established!");
            }
        }
```

Please note that we have had to convert the variables *url*, *user* and *password* into constants in order to take advantage of this structure.

5.3. Activities

1. Rewrite the activities in point 4.1 to properly close resources.

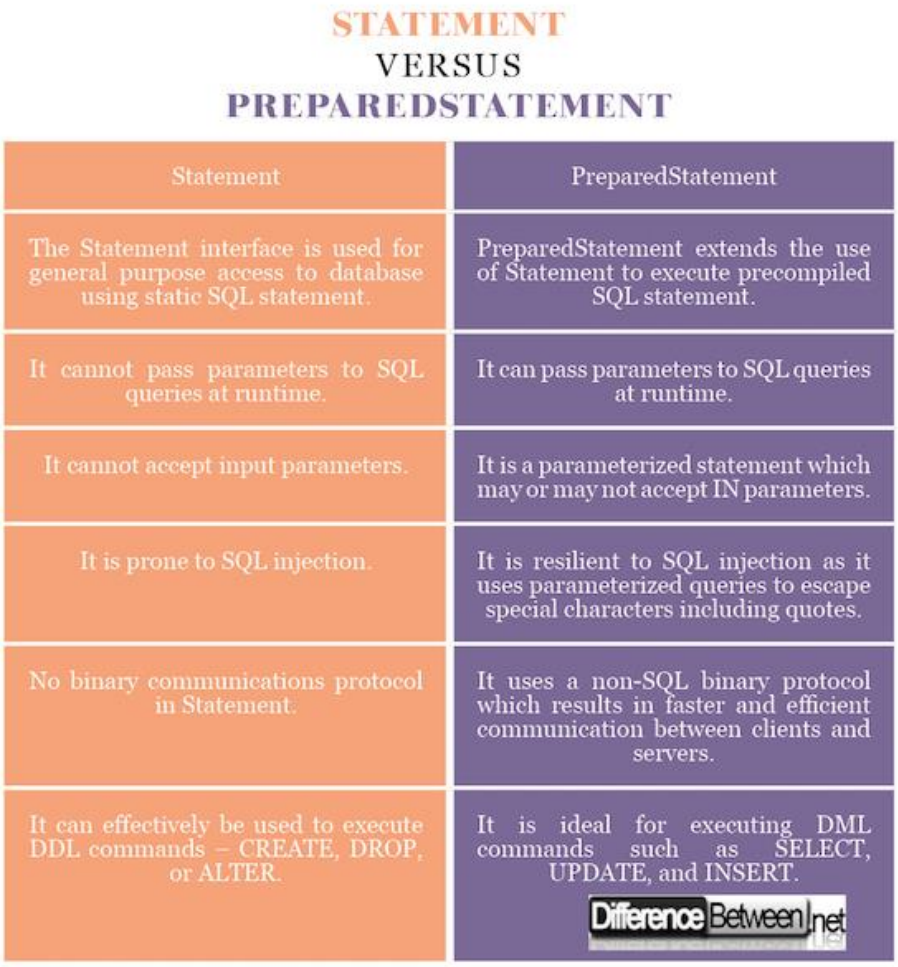
6. Prepared Statements

Java JDBC Prepared statements are pre-compiled SQL statements. Precompiled SQL is useful if the same SQL is to be executed repeatedly, for example, in a loop. Prepared statements in java only save you time if you expect to execute the same SQL over again. Every java sql prepared statement is compiled at some point. To use a java preparedstatements, you must first create a object by calling the *Connection.prepareStatement()* method. JDBC PreparedStatements are useful especially in situations where you can use a for loop or while loop to set a parameter to a succession of values. If you want to execute a Statement object many times, it normally reduces execution time to use a PreparedStatement object instead.

The syntax is straightforward: just insert question marks for any parameters that you'll be substituting before you send the SQL to the database. You need to supply values to be used in place of the question mark placeholders (if there are any) before you can execute a PreparedStatement object. You do this by calling one of the setXXX methods defined in the PreparedStatement class. There is a setXXX method for each primitive type declared in the Java programming language.

```
PreparedStatement pstmt = conn.prepareStatement("Insert into subjects
(Name, Year) values( ?, ? )");
pstmt.setString( 1, "Markup Languages" );
pstmt.setInt( 2, 1 );
```

In the following table you can see the differences between Statement and PreparedStatement. It's particularly important the fact that PreparedStatement is more robust against SQL injection attacks.



Another advantage of the use of `PreparedStatement` is the ability the ability to insert non-standard data types into the SQL statement itself, such as *`Timestamp`*, *`InputStream`*, and many more. For example, we can use a `PreparedStatement` to add a cover photo to our book record using the *`.setBinaryStream()`* method:

```
ps = connection.prepareStatement("INSERT INTO books (cover_photo) VALUES
(?)");
ps.setBinaryStream( 1, book.getPhoto() );
ps.executeUpdate();
```

In the following table you can find the more important methods of PreparedStatement. Check the oracle official documentation for the complete method list.

Method	Description

`public void setInt (int paramIndex, int value)`	sets the integer value to the given parameter index.
`public void setString (int paramIndex, String value)`	sets the String value to the given parameter index.
`public void setFloat (int paramIndex, float value)`	sets the float value to the given parameter index.
`public void setDouble (int paramIndex, double value)`	sets the double value to the given parameter index.
`public int executeUpdate()`	executes the query. It is used for create, drop, insert, update, delete etc.
`public ResultSet executeQuery()`	executes the select query. It returns an instance of ResultSet.

6.1. Activities

1. Rewrite activity 4.1.2 to use a PreparedStatement. Instead of introducing one subject at a time, allow the user to introduce as many subjects as desired, using a loop, ending with a breaking signal (a null subject code, or any other valid signal).
2. Using a PreparedStatement, create a new table called 'courses', with two columns, code (a serial) and name (a varchar of length 90). Introduce a pair of valid values, like 'Multiplatform app development' and 'Web development'.
3. Add also a new column to 'subjects' called 'course'. It will be a foreign key referencing the column 'code' in the table 'courses'. To know how to write a foreign key in PostgreSQL, check this link. Keep in mind that, before creating the foreign key constraint, you will have to update the column 'subjects.course' to have valid values (the code of one of the courses in table 'course').

7. CallableStatement

7.1. Stored Procedures

As many other DBMS's, like Oracle and (with some limitations) MySQL, PostgreSQL allows the user to define their own procedures and functions using its particular programming language, Pl/pgSQL. In Pl/pgSQL, the difference between a procedure and a function is that the second one is due to return a

value, either a set of rows or a single value. The basic format for a Pl/pgSQL function will be the following:

```
CREATE [OR REPLACE] FUNCTION function_name (arguments)
RETURNS
      return_datatype
LANGUAGE 'plpgsql'
AS $BODY$
DECLARE
      declaration;
      [...]
BEGIN
      < function_body >
      [...]
RETURN { value | NEXT };
END;
$BODY$;
```

The format of a procedure is even simpler:

```
CREATE [OR REPLACE] PROCEDURE procedure_name (arguments)
LANGUAGE 'plpgsql'
AS $BODY$
DECLARE
      declaration;
      [...]
BEGIN
      < function_body >
      [...]
END;
$BODY$;
```

Both procedures and functions are usually called 'Stored Procedures' and can be invoked from other programming languages (like JAVA). Actually, this is the recommended way to work, in order to avoid SQL injection attacks.

For example, we can define a stored procedure to know the number of subjects in every academic year. It will look like that:

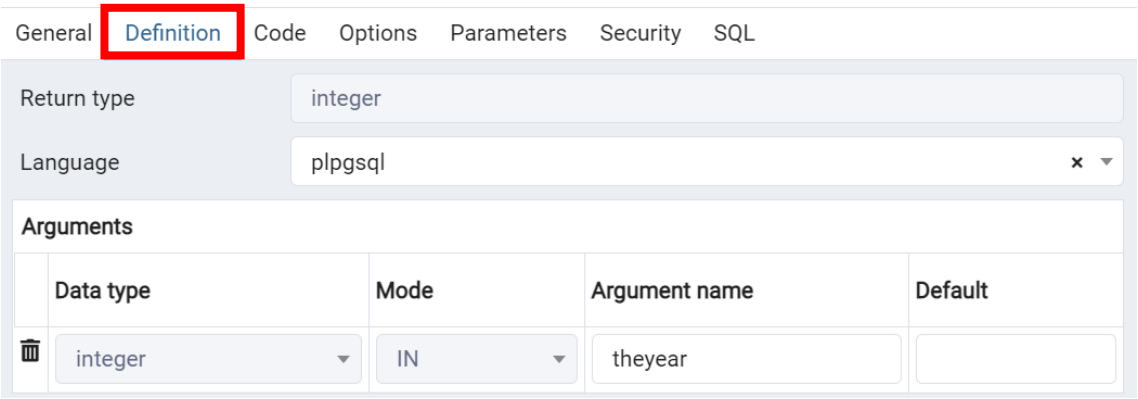
```
CREATE OR REPLACE FUNCTION public.subjectsinyear( theyear integer )
    RETURNS integer
    LANGUAGE 'plpgsql'
AS $BODY$
DECLARE
 amount INTEGER;
BEGIN
 SELECT COUNT(*) INTO amount FROM subjects WHERE subjects.year=theyear;
 RETURN amount;
```

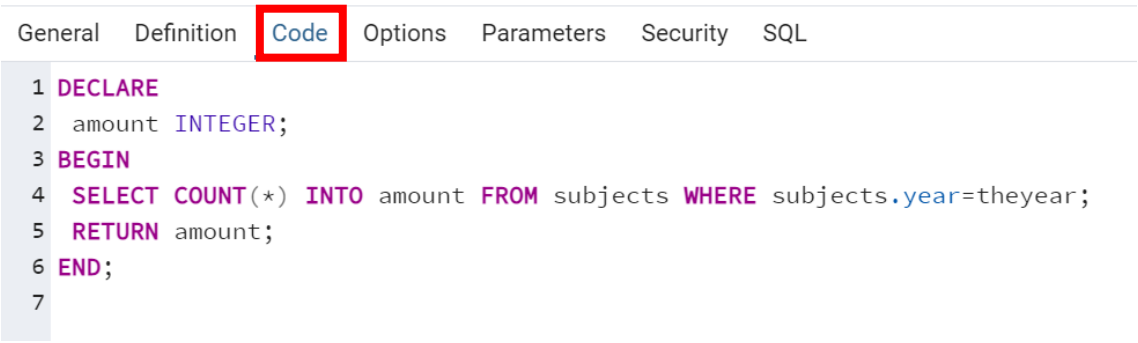
```
END;
$BODY$;
```

You can invoke this function from PgAdmin4 using the Query Tool like that:

```
Select subjectsinyear(1);
```

Keep in mind that, if you define the function using the PgAdmin4 assistant, you will have to split these definition in two sections, header (definition) and body (code):





7.2. Retrieving data

7.2.1. SetOf

Stored procedures are very likely to return a huge amount of data, usually records. In this case, the most usual return type used is the *SetOf* type. For example, suppose that we need to know the subjects in every academic year (not the number of then). We can achieve this with the following stored procedure:

```
CREATE OR REPLACE FUNCTION listofsubjects( theyear integer )
    RETURNS SETOF subjects
    LANGUAGE 'plpgsql'
AS $BODY$
DECLARE
 mysubject subjects;
```

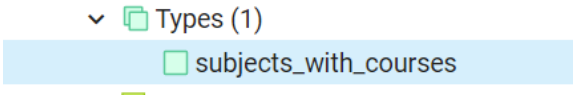
```
BEGIN
 FOR mysubject IN SELECT * FROM subjects WHERE year=theyear
 LOOP
   RETURN NEXT mysubject;
 END LOOP;
END;
$BODY$;
```

7.2.2. Create Type

Instead of returning a whole record, we can just return the data needed, similar to creating a view. To do that, we can use CREATE TYPE to mix data from different tables or just select the fields needed. In our example, if we just need the name of the subject and the course, we can create the following data type before creating the procedure:

```
CREATE TYPE subjects_with_courses AS (
      name VARCHAR(50),
      course VARCHAR(90)
);
```

You can also create this type using the PgAdmin4 assistant:



Once created, you can use the type in a stored procedure as an ordinary record:

```
CREATE OR REPLACE FUNCTION public.subjects_and_courses(
      theyear integer)
    RETURNS SETOF subjects_with_courses
    LANGUAGE 'plpgsql'
AS $BODY$
DECLARE
 mysubject subjects_with_courses;
BEGIN
 FOR mysubject IN
      SELECT subjects.name, courses.name
      FROM subjects INNER JOIN courses ON subjects.course=courses.code
      WHERE year=theyear
   LOOP
   RETURN NEXT mysubject;
 END LOOP;
END;
$BODY$;
```

7.2.3. Table definition

The last option is to directly define the structure inside the stored procedure, without previously creating the data type. This can look like that:

```
CREATE OR REPLACE FUNCTION public.subjects_and_courses_2 (
      theyear integer)
    RETURNS TABLE( name VARCHAR(50), course VARCHAR(90) )
    LANGUAGE 'plpgsql'
AS $BODY$
DECLARE
 mysubject RECORD;
BEGIN
 FOR mysubject IN
      SELECT subjects.name AS sname, courses.name AS scourse
      FROM subjects INNER JOIN courses ON subjects.course=courses.code
      WHERE year=theyear
   LOOP
   name := mysubject.sname;
   course := mysubject.scourse;
   RETURN NEXT;
 END LOOP;
END;
$BODY$;
```

7.3. Calling an stored procedure from JAVA

Once we have defined our stored procedures, we can invoke them from JAVA using the CallableStatement interface.

```
import java.sql.*;

public class App
{
    static final String url =
         "jdbc:postgresql://localhost:5432/VTInstitute";
    static final String user = "postgres";
    static final String password = "pwd";

    public static void main( String[] args )
         throws ClassNotFoundException, SQLException
    {
        try ( Connection conn = DriverManager.getConnection(url, user,
password) ) {
            CallableStatement statement = conn.prepareCall( "{call
listofsubjects(1)}" );
            ResultSet rs = statement.executeQuery();
            System.out.println("Code" + "\t" + "Name");
```

```
            System.out.println("-----------------------------------
---");
            while (rs.next()) {
                System.out.println(rs.getString(1) + "\t "
                        + rs.getString(2));
            }
        }
        catch (SQLException e) {
            e.printStackTrace();
        }
}
```

There is a slight difference depending on if we are calling a function or a procedure. To call a procedure, the braces in the call must be removed:

```
CallableStatement statement = conn.prepareCall( "call doSomething()" );
```

7.4. Activities

1. Using the 'employees' database, create a new stored procedure to list all the employees on a specific job.
2. Create another procedure to return employees belonging to a specific department.
3. Create a procedure to list all the employees which name matches a pattern using wildcards, for example, the pattern "a%" will return "ALLEN", "ADAMS".
4. Create a JAVA application that permits to call any of the previous procedures.

8. Transactions

A transaction, in the context of a database, is a logical unit that is independently executed for data retrieval or updates. A transaction is a single unit of work. If a transaction is successful, all of the data modifications made during the transaction are committed and become a permanent part of the database. If a transaction encounters errors and must be canceled or rolled back, then all of the data modifications are erased.

In relational databases, database transactions must be atomic, consistent, isolated and durable -summarized as the ACID acronym- in order to ensure accuracy, completeness, and data integrity.

- **Atomicity**: This property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none. There

must be no state in a database where a transaction is left partially completed. States should be defined either before the execution of the transaction or after the execution/abortion/failure of the transaction.

- **Consistency**: The database must remain in a consistent state after any transaction. No transaction should have any adverse effect on the data residing in the database. If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.
- **Durability**: The database should be durable enough to hold all its latest updates even if the system fails or restarts. If a transaction updates a chunk of data in a database and commits, then the database will hold the modified data. If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system springs back into action.
- **Isolation**: In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.

Let's take an example of a simple transaction. Suppose a bank employee transfers 500€ from A's account (starting amount = 1000€) to B's account (0€). This very simple and small transaction involves several low-level tasks:

2. Check if A's account balance >= 500€
3. A's account balance = A's account balance – 500€
4. B's account balance = B's account balance + 500€

If an error occurred between step 2 and 3, then A's will have a balance of 500€ and B's a balance of 0€, that is, the 500€ have been lost, and probably neither A or B will be happy. This is the kind of problems that are avoided by defining transactions. The three previous steps would make a transaction.

A DBMS supporting transactions is called 'transactional'. The basic structure of a transaction would be similar to this:

```
Start transaction
Execute individual operations
If ( everything_ok )
      Apply_changes;
Else
      Revert_changes;
```

In SQL, we confirm changes with the keyword *COMMIT* and we cancel changes with *ROLLBACK*. In JAVA we use the same keywords to define a transaction.

Suppose the following example: we are going to add a new course to the table courses and, once done, a new subject to the table subjects, referencing the new course. Obviously, if an error occurred adding the course, we cannot add the subject, because there is no course to refer to. The code would be something like that:

```
try ( Connection conn = DriverManager.getConnection(url, user, password)
) {
     conn.setAutoCommit(false);
     try {
          PreparedStatement st = conn.prepareStatement("INSERT INTO
courses(name) VALUES(?)", Statement.RETURN_GENERATED_KEYS);
          st.setString(1, "Computer Systems Administration");
          st.executeUpdate();

          ResultSet keys = st.getGeneratedKeys();
          keys.next();
          int courseCode = keys.getInt(1);

          PreparedStatement st2 = conn.prepareStatement("INSERT INTO
subjects(name, year, course) VALUES( ?, ?, ? )");
          st2.setString( 1, "CyberSecurity" );
          st2.setString( 2, 2 );
          st2.setInteger( 3, courseCode );
          st2.executeUpdate();

          System.out.println("Transaction finished successfully!!");
          conn.commit();
          }
     catch( SQLException ex ) {
          System.err.println( ex.getMessage() );
          conn.rollback();
     }
}
```

Note that we use the methods *commit* and *rollback* from *connection* to confirm or not the transaction. Also, there is a new feature about *PreparedStatement*, the use of *Statement.RETURN_GENERATED_KEYS*. This option allows us to obtain the keys generated during the insert operation (usually, serials).

8.1. Activities

1. Using the Employee database, create a JAVA application to insert a new employee in a department. The department can exist or not, in this case it should be created. This insertion must be transactional, that is, if the

department is not successfully created, we must revert the insertion of the employee.

9. ORDBMS

An object–relational database (ORD), or object–relational database management system (ORDBMS), is a database management system (DBMS) similar to a relational database, but with an object-oriented database model: objects, classes and inheritance are directly supported in database schemas and in the query language. In addition, just as with pure relational systems, it supports extension of the data model with custom data types and methods.

An object–relational database can be said to provide a middle ground between relational databases and object-oriented databases. In object–relational databases, the approach is essentially that of relational databases: the data resides in the database and is manipulated collectively with queries in a query language; at the other extreme are OODBMSes in which the database is essentially a persistent object store for software written in an object-oriented programming language, with a programming API for storing and retrieving objects, and little or no specific support for querying.

Some of the most popular RDBMS that support object-oriented features are ORACLE, PostgreSQL and Informix.

10. PostgreSQL as ORDBMS

PostgreSQL supports stored procedures written in entirely procedural languages like PL/PGSQL or Perl without loaded modules, and more object-oriented languages like Python or Java, often through third party modules. To be sure you can't write a graphical interface inside PostgreSQL, and it would not be a good idea to write additional network servers, such as web servers, directly inside the database. However, the environment allows you to create sophisticated interfaces for managing and transforming your data. Because it is a platform in a box the various components need to be understood as different and yet interoperable. In fact, the primary concerns of object-oriented programming are all supported by PostgreSQL, but this is done in a way that is almost, but not quite, entirely unlike traditional object oriented programming. For this reason, the "object-relational" label tends to be a frequent source of confusion.

Data storage in PostgreSQL is entirely relational, although this can be degraded using types which are not atomic, such as arrays, XML or JSON. In object-oriented

terms, every relation is a class, but not every class is a relation. Operations are performed on sets of objects (an object being a row), and new row structures can be created ad-hoc. PostgreSQL is, however, a strictly typed environment and so in many cases, polymorphism requires some work.

10.1. Data Abstraction and Encapsulation

The relational model itself provides some tools for data abstraction and encapsulation, and these features are taken to quite some length in PostgreSQL. Taken together these are very powerful tools and allow for things like calculated fields to be simulated in relations and even indexed for high performance.

Views are the primary tool here. With views, you can create an API for your data which is abstracted from the physical storage. Using the rules system, you can redirect inserts, updates, and deletes from the view into underlying relations, preferably using user defined functions. Being relations, views are also classes and methods. Views cannot simply be inherited and workarounds cause many hidden gotchas.

A second important tool here is the ability to define what appear to be calculated fields using stored procedures. If I create a table called "employee" with three fields (first_name, middle_name, last_name) among others, and create a function called "name" which accepts a single employee argument and concatenates these together as "last_name, first_name middle_name" then if I submit a query which says:

```
select e.name from employee e;
```

it will transform this into:

```
select name(e) from employee e;
```

This gives you a way to do calculated fields in PostgreSQL without resorting to views. Note that these can be done on views as well because views are relations. These are not real fields though. Without the relation reference, it will not do the transformation (so SELECT name from employee will not have the same effect).

10.2. Polymorphism

PostgreSQL is very extensible in terms of all sorts of aspects of the database. Not only types can be created and defined –as we saw in unit 3, point 7.2.2-, but also operators can be defined or overloaded.

A more important polymorphism feature is the ability to cast one data type as another. Casts can be implicit or explicit. Implicit casts, which have largely been removed from many areas of PostgreSQL, allow for PostgreSQL to cast data types when necessary to find functions or operators that are applicable. Implicit casting can be dangerous because it can lead to unexpected behavior because minor errors can lead to unexpected results. '2012-05-31' is not 2012-05-31. The latter is an integer expression that reduces to 1976. If you create an implicit cast that turns an integer into a date being the first of the year, the lack of quoting will insert incorrect dates into your database without raising an error ('1976-01-01' instead of the intended '2012-05-31'). Implicit casts can still have some uses.

10.2.1. Domains

A domain is a user-defined data type that is based on another underlying type. Optionally, it can have constraints that restrict its valid values to a subset of what the underlying type would allow. Otherwise it behaves like the underlying type. For example, any operator or function that can be applied to the underlying type will work on the domain type. The underlying type can be any built-in or user-defined base type, enum type, array type, composite type, range type, or another domain.

For example, we could create a domain over integers that accepts only positive integers:

```
CREATE DOMAIN posint AS integer CHECK (VALUE > 0);
CREATE TABLE mytable (id posint);
INSERT INTO mytable VALUES(1);   -- works
INSERT INTO mytable VALUES(-1);  -- fails
```

When an operator or function of the underlying type is applied to a domain value, the domain is automatically down-cast to the underlying type. Thus, for example, the result of mytable.id - 1 is considered to be of type integer not posint. We could write (mytable.id - 1)::posint to cast the result back to posint, causing the domain's constraints to be rechecked. In this case, that would result in an error if the expression had been applied to an id value of 1. Assigning a value of the underlying type to a field or variable of the domain type is allowed without writing an explicit cast, but the domain's constraints will be checked.

10.2.2. Composite types

A *composite type* represents the structure of a row or record; it is essentially just a list of field names and their data types. PostgreSQL allows composite types to be used in many of the same ways that simple types can be used. For example, a column of a table can be declared to be of a composite type.

Here are two simple examples of defining composite types:

```
CREATE TYPE complex AS (
    r       double precision,
    i       double precision
);

CREATE TYPE inventory_item AS (
    name            text,
    supplier_id     integer,
    price           numeric
);
```

To access a field of a composite column, one writes a dot and the field name, much like selecting a field from a table name. In fact, it's so much like selecting from a table name that you often have to use parentheses to keep from confusing the parser. For example, you might try to select some subfields from our *on_hand* example table with something like:

```
SELECT item.name FROM on_hand WHERE item.price > 9.99;
```

This will not work since the name item is taken to be a table name, not a column name of on_hand, per SQL syntax rules. You must write it like this:

```
SELECT (item).name FROM on_hand WHERE (item).price > 9.99;
```

or if you need to use the table name as well (for instance in a multitable query), like this:

```
SELECT (on_hand.item).name FROM on_hand WHERE (on_hand.item).price > 9.99;
```

10.2.3. Enumerated types

Enumerated (enum) types are data types that comprise a static, ordered set of values. They are equivalent to the *enum* types supported in a number of programming languages. An example of an enum type might be the days of the week, or a set of status values for a piece of data.

Enum types are created using the CREATE TYPE command, for example:

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

Once created, the enum type can be used in table and function definitions much like any other type:

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

```
CREATE TABLE person (
    name text,
    current_mood mood
);
INSERT INTO person VALUES ('Moe', 'happy');
SELECT * FROM person WHERE current_mood = 'happy';
```

The ordering of the values in an enum type is the order in which the values were listed when the type was created. All standard comparison operators and related aggregate functions are supported for enums.

10.2.4. Arrays

Theoretically, a relational database should store just one value per attribute (that is, a column). Despite that, PostgreSQL (and other DBMSs that support object-oriented features) allows the use of arrays.

To illustrate the use of array types, we create this table:

```
CREATE TABLE notes
(
   noteDate DATE,
   lines TEXT[]
);
```

We can add values to this table with both these two options:

```
INSERT INTO notes VALUES ('2021-09-08', '{"Course beginning", "Prepare a PPT file"}');
```

Or, alternatively:

```
INSERT INTO notes VALUES ('2021-09-08', ARRAY["Course beginning", "Prepare a PPT file"]);
```

To retrieve information from an array, we can obtain the whole array or just an element (or elements), keeping in mind that the first element is labeled with the number 1:

```
SELECT noteDate, lines FROM notes;
SELECT noteDate, lines[1] FROM notes;
SELECT noteDate, lines[1:2] FROM notes;
```

You can also use the function *array_length()* to know the size of the array.

When modifying arrays, you can modify the whole array or just an element (or elements):

```
UPDATE notes SET lines = '{"Course beginning", "Prepare a PPT file"}';
UPDATE notes SET lines[2] = 'Prepare a PPT file';
```

10.3. Inheritance

In PostgreSQL tables can inherit from other tables. Their methods are inherited but implicit casts are not chained, nor are their indexes inherited. This allows you develop object inheritance hierarchies in PostgreSQL. Multiple inheritance is possible, unlike any other ORDBMS (Oracle, DB2, and Informix all support single inheritance).

Table inheritance is an advanced concept and has many undesired issues. On the whole it is probably best to work with table inheritance first in areas where it is more typically used, such as table partitioning, and later look at it in terms of object-relational capabilities.

Let's start with an example: suppose we are trying to build a data model for cities. Each state has many cities, but only one capital. We want to be able to quickly retrieve the capital city for any particular state. This can be done by creating two tables, one for state capitals and one for cities that are not capitals. However, what happens when we want to ask for data about a city, regardless of whether it is a capital or not? The inheritance feature can help to resolve this problem. We define the capitals table so that it inherits from cities:

```
CREATE TABLE cities (
    code            int   PRIMARY KEY,
    name            text,
    population      float,
    elevation       int     -- in feet
);
CREATE TABLE capitals (
    state           char(2)
) INHERITS (cities);
```

When launching a query, you must be aware that queries on the 'parent' table return rows from both the parent table and every inherited table. So, if you want to launch a query on cities that are not capitals, you should write it with the *ONLY* keyword:

```
SELECT name, elevation
    FROM ONLY cities
    WHERE elevation > 500;
```

Another thing that must be kept in mind is that indexes are not inherited, so, in our example, it would be possible to do the following:

```
INSERT INTO capitals VALUES(100, 'Alicante', 300000, 0);
INSERT INTO capitals VALUES(100, 'Murcia, 700000, 150);
```

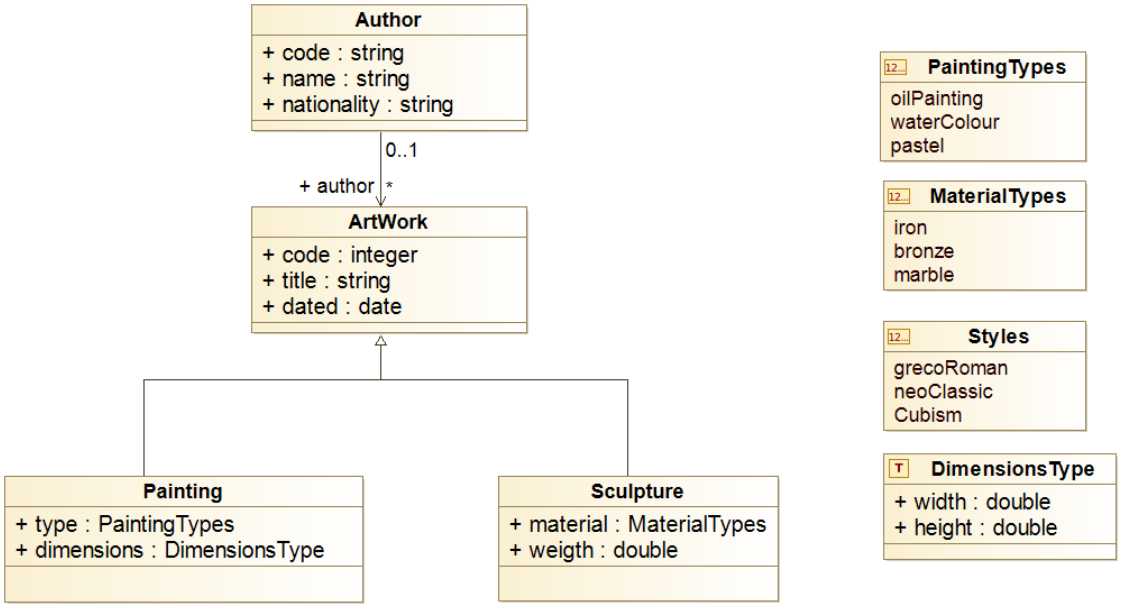
That is, the *code* field (the primary key) would have a duplicated value in the *capitals* table. To avoid this, the primary key must be explicitly added:

```
ALTER TABLE capitals ADD CONSTRAINT pk_capitals PRIMARY KEY (code);
```

You should also explicitly add whatever restriction included in the parent table, such as *UNIQUE* or *FOREIGN KEY* restrictions.

10.4. Activities

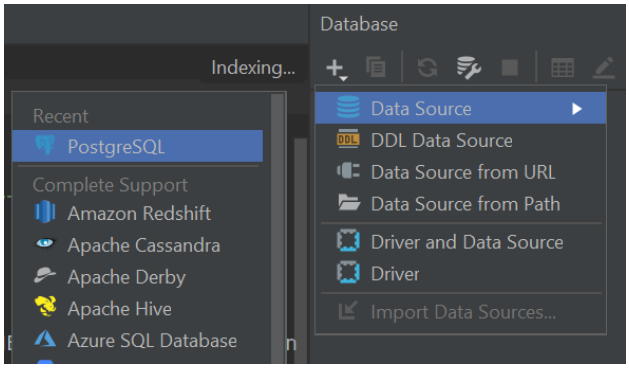
1. Modify the tables *dept* and *employee* from the *Employees* database. Think what fields are suitable to be changed, and what changes should be made.
2. Do the same with the *VTInstitute* database.
3. Given the diagram below, make the corresponding PostgreSQL database. Take in consideration the following restrictions:
    a. *Author* code is a string formed by the initials of the author and the year of birth.
    b. *ArtWork* code must be a sequence.
    c. *Width*, *height* and *weight* are always values greater than zero.
    d. Keep in mind that *DimensionsType* is a composite type.



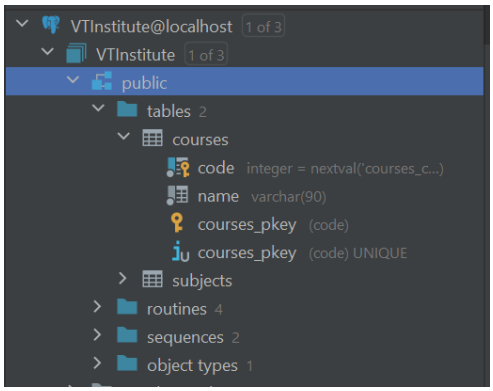
11. To Learn More

12.1. Managing DBMSs from inside INTELLIJ IDEA

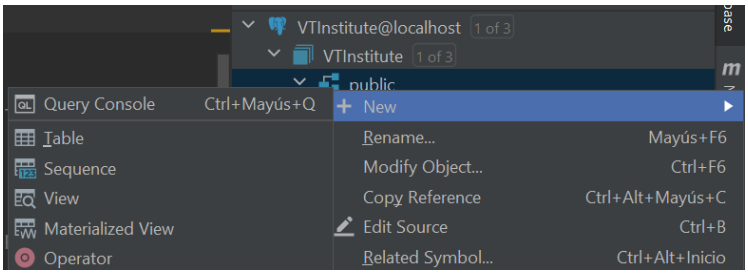
Although PostgreSQL offers a very good tool to manage databases (PgAdmin), it is also possible to make most of the work we have to do with a DBMS without leaving INTELLIJ IDEA, using the 'Database' tool window. With this window, it is possible to establish a connection to almost any of the most popular DBMSs today.



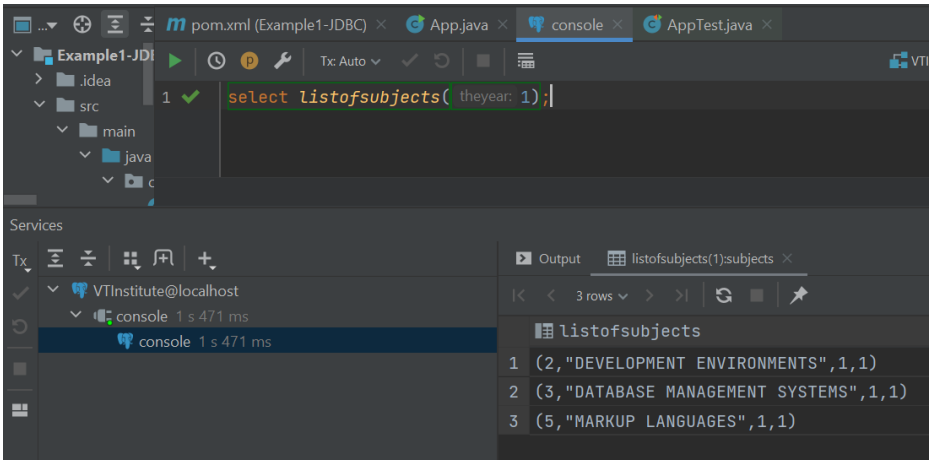
You simply add the connection pressing the '+' button, choosing the type of data source to connect to, and giving the connection parameters. Once done, the connection (schemas, tables, ...) will appear in the database window.



From this window, we can make basically the same work we do with PgAdmin4, like creating tables or stored procedures, for example.



We also have a SQL console to directly type any order we need to send to the DBMS.



You can have more information about the database tool window in the following link:

Working with DBMSs from inside INTELLIJ IDEA

12. Bibliography

PostgreSQL: Documentation

JDBC Oracle Official Guide

The PostgreSQL™ JDBC Interface

PostgreSQL JDBC error codes

Very good tutorial about JDBC, with examples

To learn more about ResultSets