



# JDBC

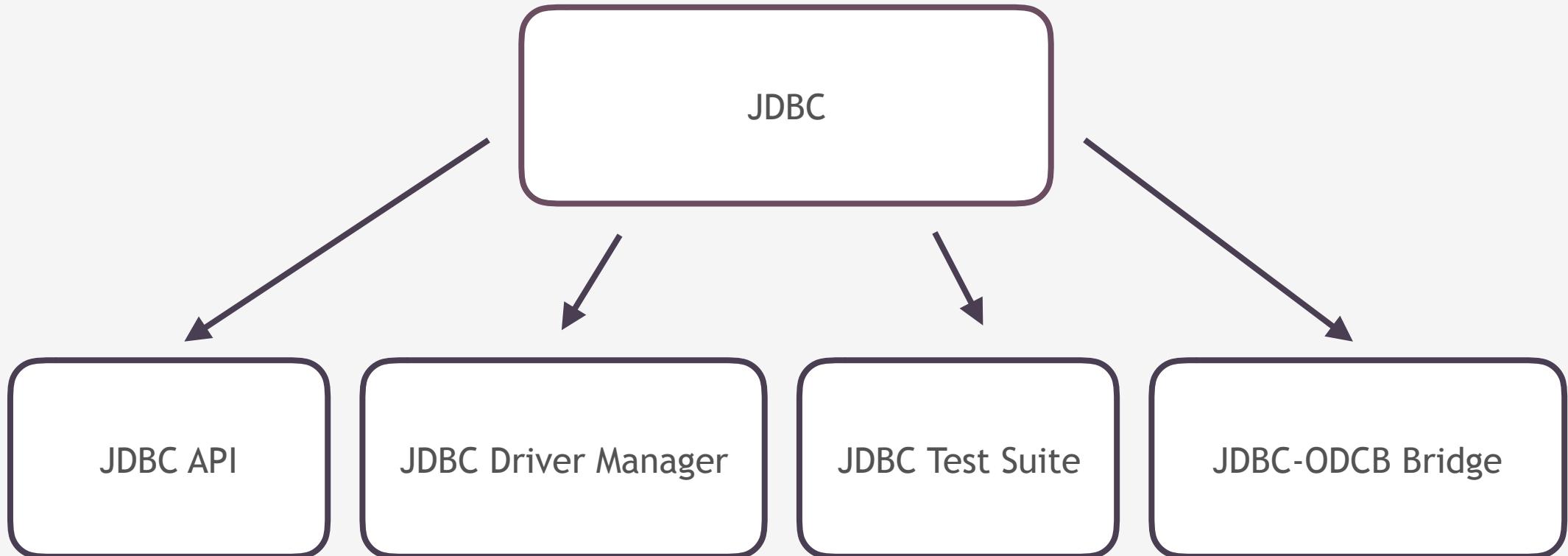
Piotr Brzozowski

SDA



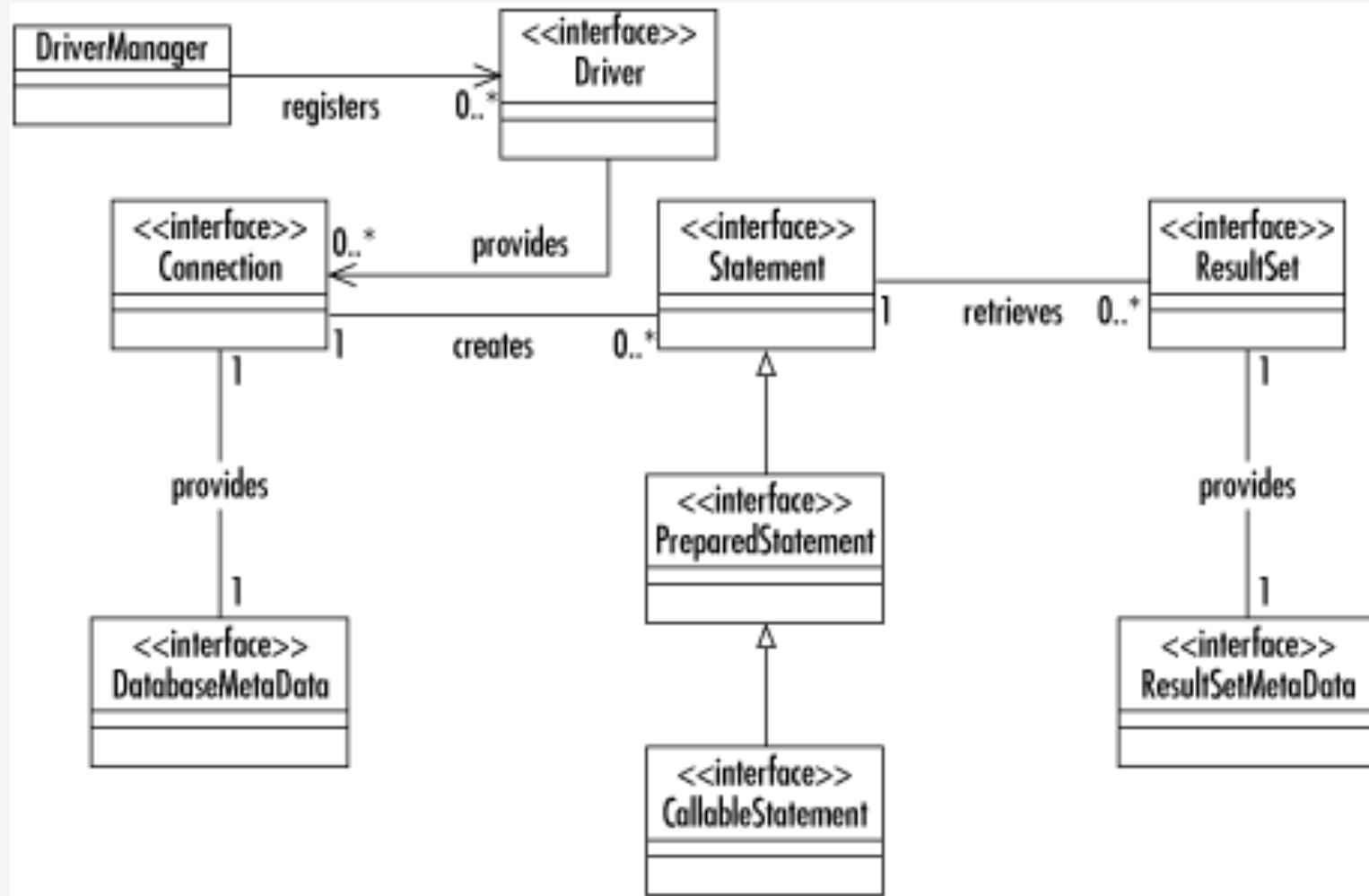
# JDBC - Podstawy

# Komponenty JDBC





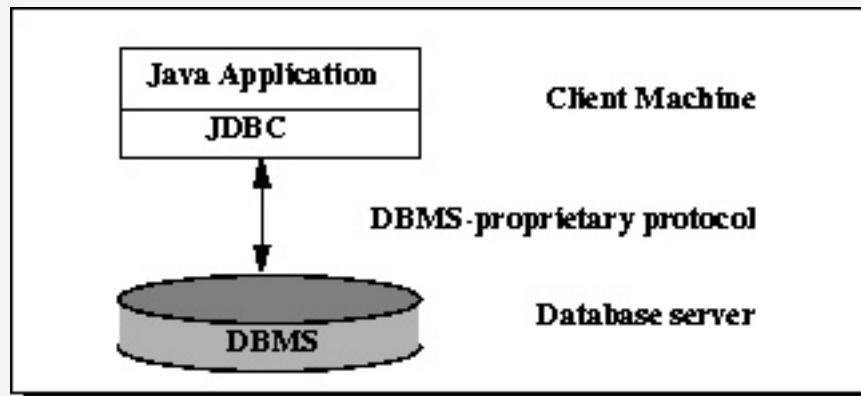
# Podstawowe klasy



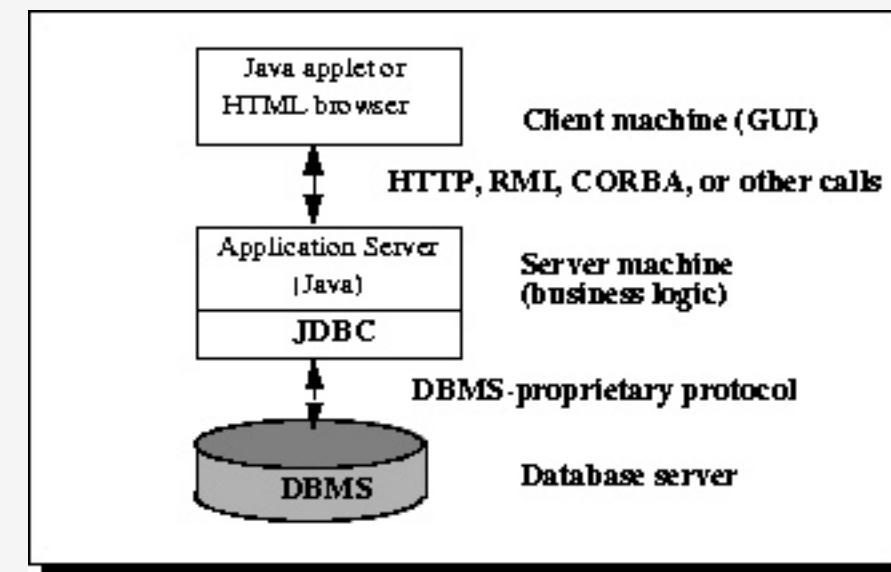


# Architektura JDBC

**dwu-poziomowa**

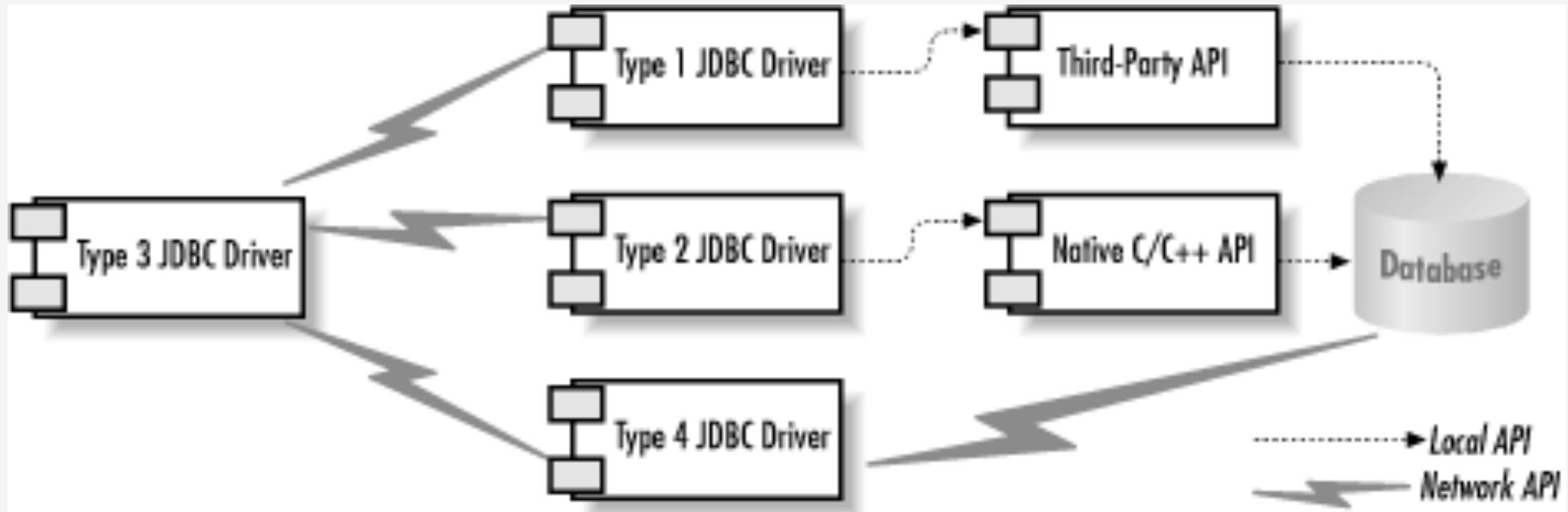


**trój-poziomowa**





# Typy sterowników





# SQLException

```
for (Throwable e : ex) {
    if (e instanceof SQLException) {
        if (!ignoreSQLException(
            ((SQLException)e).
            getSQLState()) == false) {

            e.printStackTrace(System.err);
            System.err.println("SQLState: " +
                ((SQLException)e).getSQLState());

            System.err.println("Error Code: " +
                ((SQLException)e).getErrorCode());

            System.err.println("Message: " + e.getMessage());

            Throwable t = ex.getCause();
            while(t != null) {
                System.out.println("Cause: " + t);
                t = t.getCause();
            }
        }
    }
}
```



# Zadanie 1 - Konfiguracja środowiska

- uruchom MySQL Workbench
- Stwórz bazę danych o nazwie ***library***



# Zadanie 1 - Konfiguracja środowiska - opcjonalnie

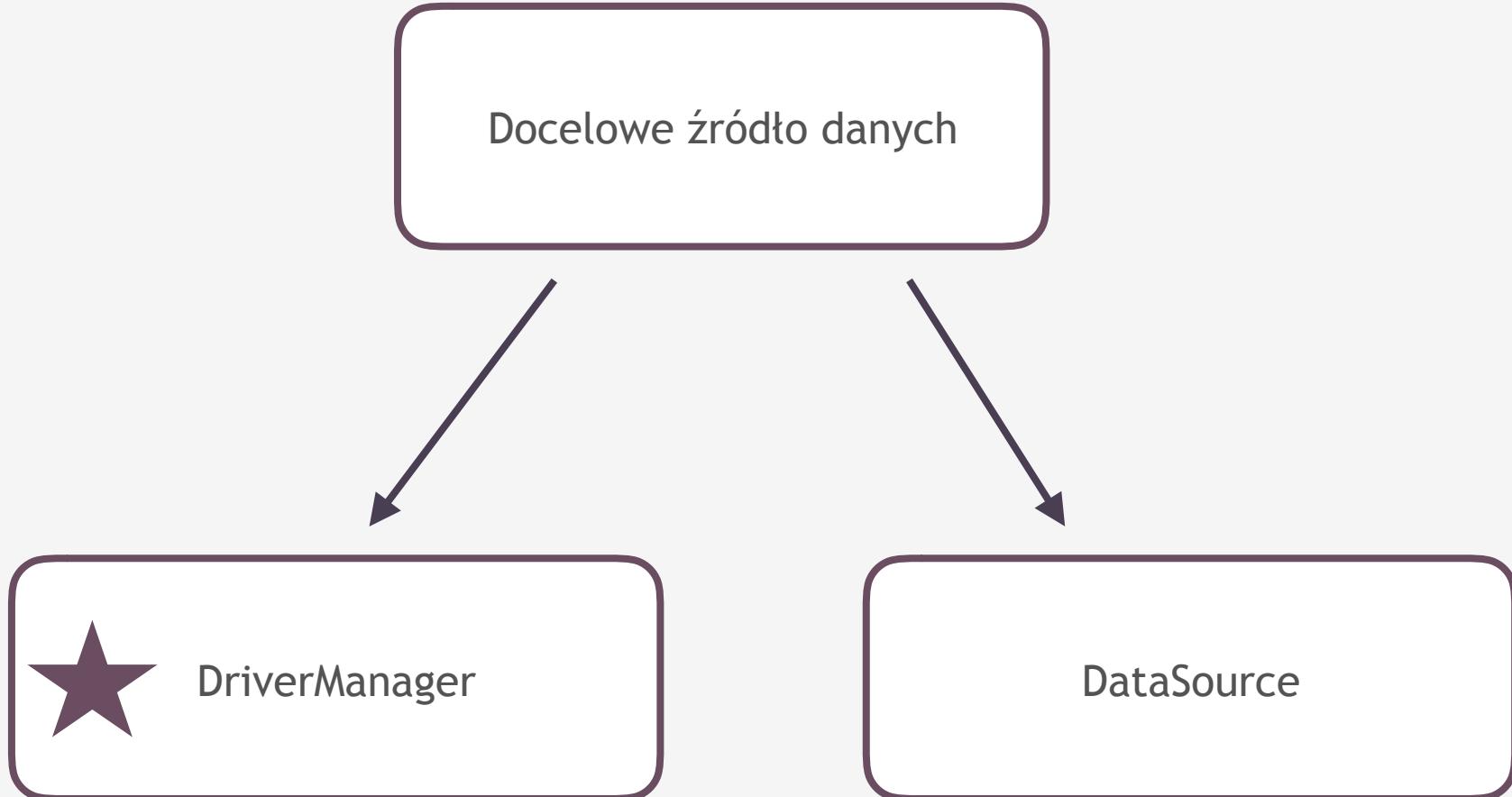
- Wejdź na stronę: <https://www.postgresql.org/download/>
- Pobierz najnowszą wersję PostgreSQL Core Distribution
- Zainstaluj PostgreSQL
- Postępuj zgodnie z krokami instalacyjnymi
- Po poprawnej instalacji uruchom pgAdmin 4
- Stwórz użytkownika zgodnie z następującym wzorem  
**pierwsza litera imienia + nazwisko**, np. pbrzozowski
- Stwórz bazę danych o nazwie ***library***



# Ustanawianie połączenia



# Ustanawianie połączenia





# Podstawowe użycie klasy DriverManager

```
public Connection getConnection() throws SQLException {  
  
    Connection conn = null;  
    Properties connectionProps = new Properties();  
    connectionProps.put("user", this.userName);  
    connectionProps.put("password", this.password);  
  
    if (this.dbms.equals("mysql")) {  
        conn = DriverManager.getConnection(  
            "jdbc:" + this.dbms + "://" +  
            this.serverName +  
            ":" + this.portNumber + "/",  
            connectionProps);  
    } else if (this.dbms.equals("derby")) {  
        conn = DriverManager.getConnection(  
            "jdbc:" + this.dbms + ":" +  
            this.dbName +  
            ";create=true",  
            connectionProps);  
    }  
    System.out.println("Connected to database");  
    return conn;  
}
```



# Potencjalne problemy

- W przypadku niektórych sterowników baz danych należy wcześniej stworzyć nową instancję klasy go reprezentującej
- Problemy zaobserwowane między innymi dla połączeń z MySQL
- Poniższy kod należy wywołać przed pierwszą próbą połączenia (patrz: poprzedni slajd)

```
// Notice, do not import com.mysql.jdbc.*  
// or you will have problems!  
  
public class LoadDriver {  
    public static void main(String[] args) {  
        try {  
            // The newInstance() call is a work  
            around for some  
            // broken Java implementations  
  
            Class.forName("com.mysql.jdbc.Driver").newInstance();  
        } catch (Exception ex) {  
            // handle the error  
        }  
    }  
}
```



# Zadanie 2 - Inicjalizacja projektu bazowego

- Stwórz nowy projekt
- dodaj do pliku pom jedną z następujących konfiguracji:

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.6</version>
</dependency>
```

```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>42.2.2</version>
</dependency>
```



# Zadanie 2 - Inicjalizacja projektu bazowego

- zwróć szczególną uwagę na budowę adresu URL dla bazy danych PostgreSQL mając na uwadze różnicę pomiędzy np. MySQL:

jdbc:mysql://[host][,failoverhost...]

[:port]/[database]

[?propertyName1][=propertyValue1]

[&propertyName2][=propertyValue2]...

jdbc:postgresql://<database\_host>:<port>/<database\_name>



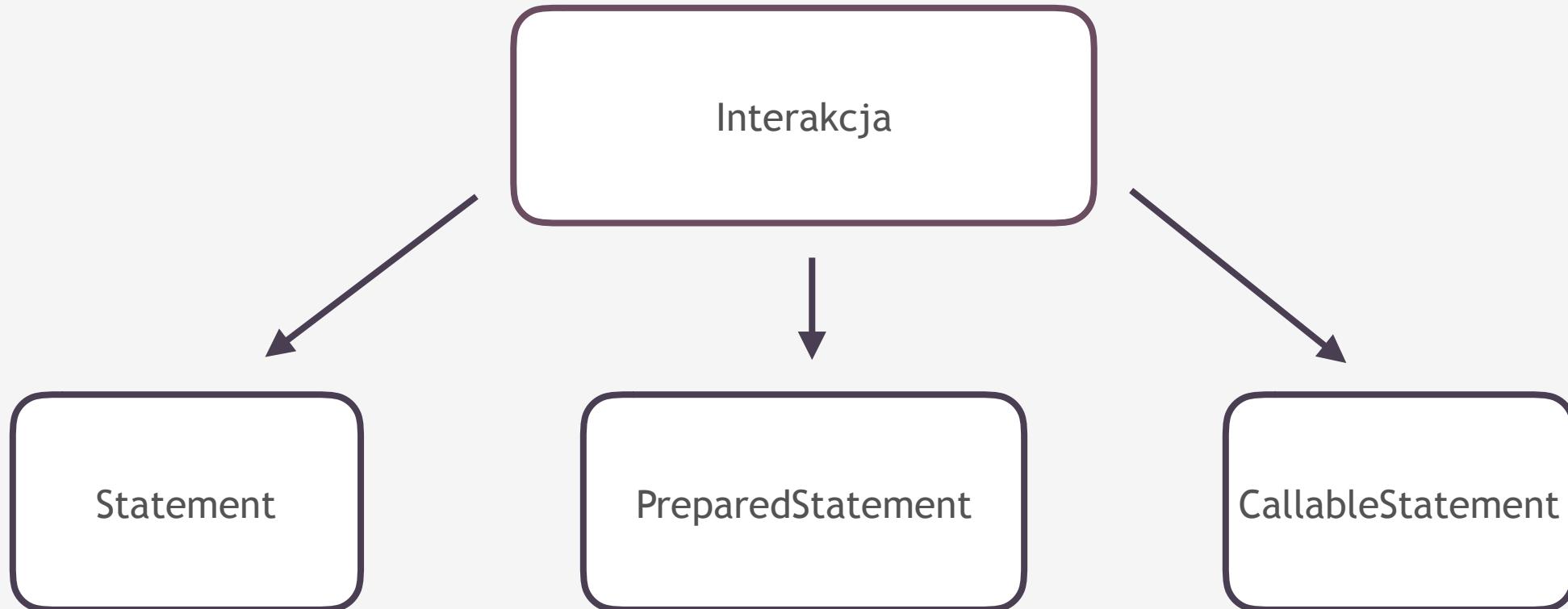
# Zadanie 3 - Ustanawianie połączenia

- dokonaj implementacji klasy w oparciu o interfejs **JDBCService**
  - implementacja powinna znajdować się pakiecie **service**
  - nazwa implementacji powinna odzwierciedlać faktyczną funkcjonalność serwisu, np. **PostgreSQLService, MySQLService**
  - dodaj funkcjonalność dla pierwszej metody interfejsu **connect(Config config)**
  - dodaj funkcjonalność dla drugiej metody interfejsu **disconnect()**, która będzie odpowiedzialna za zamykanie połączenia
- stwórz obiekt **Config** w oparciu o takie dane jak: **url, nazwa użytkownika, hasło**
- stwórz instancję **JDBCService** i wywołaj metodę **connect(Config config)**
- zamknij połączenie po wykonanej operacji
- sprawdź zwrócony rezultat



# Interakcja z bazą danych

# Rodzaje interakcji





# Statement

- Wykorzystywany najczęściej w przypadkach ogólnego przeznaczenia
- Przydatny w przypadku statycznych wywołań w czasie działania aplikacji
- Posiada trzy rodzaje wywołań
  - boolean execute (String SQL)
  - int executeUpdate(String SQL)
  - ResultSet executeQuery(String SQL)

```
Statement stmt = null;
try {
    stmt = conn.createStatement( );
    ...
}
catch (SQLException e) {
    ...
}
finally {
    stmt.close();
}
```



# PreparedStatement

- Wykorzystywany najczęściej w przypadkach gdy zapytanie wykonywane jest wielokrotnie w różnych konfiguracjach
- PreparedStatement zawiera w sobie prekomplilowane zapytanie SQL w odróżnieniu od klasycznego Statement

```
PreparedStatement updateSales = null;
String updateString =
    "update " + dbName + ".COFFEES " +
    "set SALES = ? where COF_NAME = ?";

try {
    updateSales = con.prepareStatement(updateString);

    for (Map.Entry<String, Integer> e : salesForWeek.entrySet()) {
        updateSales.setInt(1, e.getValue().intValue());
        updateSales.setString(2, e.getKey());
        updateSales.executeUpdate();
        con.commit();
    }
} catch (SQLException e) {
    JDBCUtilities.printSQLException(e);
} finally {
    if (updateSales != null) {
        updateSales.close();
    }
}
```



# CallableStatement

- Wykorzystywane do wykonywania procedur na bazie danych

```
CallableStatement cstmt = null;
try {
    String SQL = "{call getEmpName (?, ?)}";
    cstmt = conn.prepareCall (SQL);
    ...
}
catch (SQLException e) {
    ...
}
finally {
    ...
}
```



# Zadanie 4 - Tworzenie tabel

- w pakiecie model znajdują się dwie klasy: **Book** i **Author** odzwierciedlające stan tabeli
  - uzupełnij klasy **Book** i **Author** o brakujące pola
  - dostosuj konstruktor do przyjmowania nowych parametrów
  - w modelu stwórz pola typu: **public static final String ...** zawierające nazwy kolumn każdej z tabel
  - wygeneruj metodę **toString()** dla każdego z modelu
- stwórz klasę implementującą interfejs **Executor**, która każdorazowo dla wywołania utworzy połączenie do bazy danych i stworzy obiekt Statement
- w klasie **BooksManager** i **AuthorsManager** dodaj implementacje dla:  
**getCreateTableQuery()** w oparciu o klauzule niezbędną do tworzenia tabel
- w metodzie **createRepository()** wykorzystaj klasę **Executora** do stworzenia tabeli
- wywołaj niezbędne operacje w metodzie main i za pomocą pgAdmin 4 sprawdź czy tabele zostały utworzone



# Zadanie 5 - wprowadzanie danych

- w klasie BooksManager i AuthorsManager dodaj implementacje metody: ***getInsertQuery(Book object)*** w oparciu o klauzule niezbędną do dodawania nowych wierszy do tabel - wykorzystaj przykład z poprzednich slajdów
- w metodzie ***add(Book object)*** wykorzystaj klasę ***Executora*** do dodania nowych elementów do tabel
- wywołaj niezbędne operacje w metodzie main i za pomocą pgAdmin 4 sprawdź czy dane zostały dodane



# Zadanie 6 - aktualizowanie danych

- dokonaj implementacji kolejnej metody interfejsu **Executor**, która utworzy połączenie oraz obiekt typu **PreparedStatement** i przekaże go do akcji - wykorzystaj przykład z poprzednich slajdów
- w klasie BooksRepository i AuthorsRepository dodaj implementacje metody: **getUpdateQuery()** w oparciu o klauzule niezbędną do aktualizacji pojedynczego wiersza w wskazanej tabeli
- w metodzie **update(Book object)** wykorzystaj klasę **Executora** do aktualizacji elementów w tabeli
- wywołaj niezbędne operacje w metodzie main i za pomocą pgAdmin 4 sprawdź czy dane zostały zaktualizowane

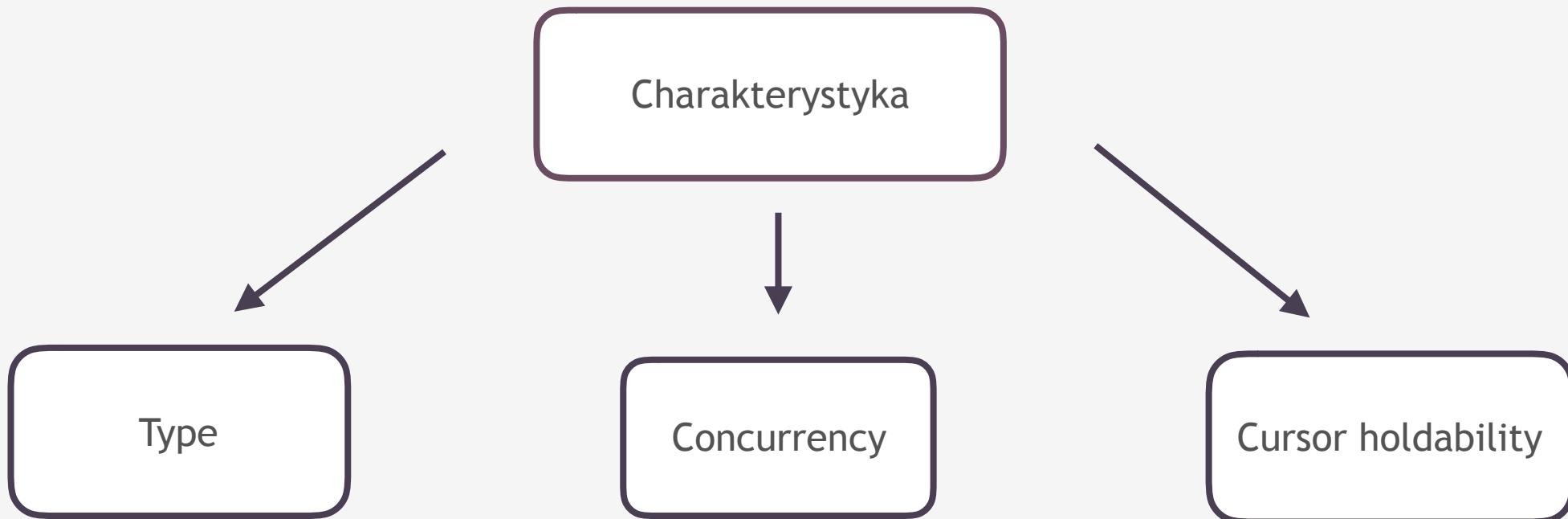


# Zwracane typy danych



# ResultSet

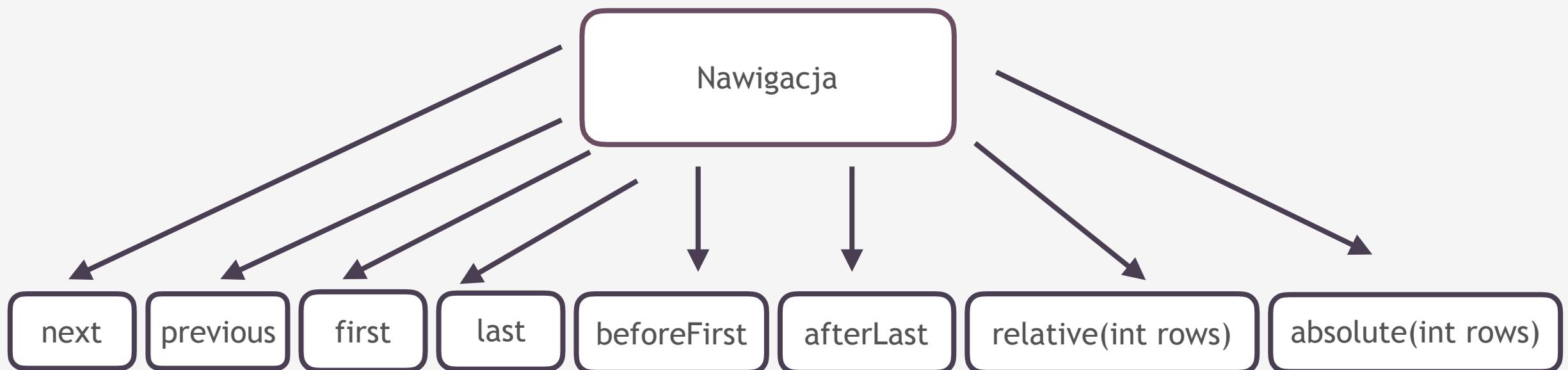
- Interfejs dostarczający metod do pobierania i manipulacji danych
- Obiekt ResultSet może posiadać różną funkcjonalność i charakterystykę





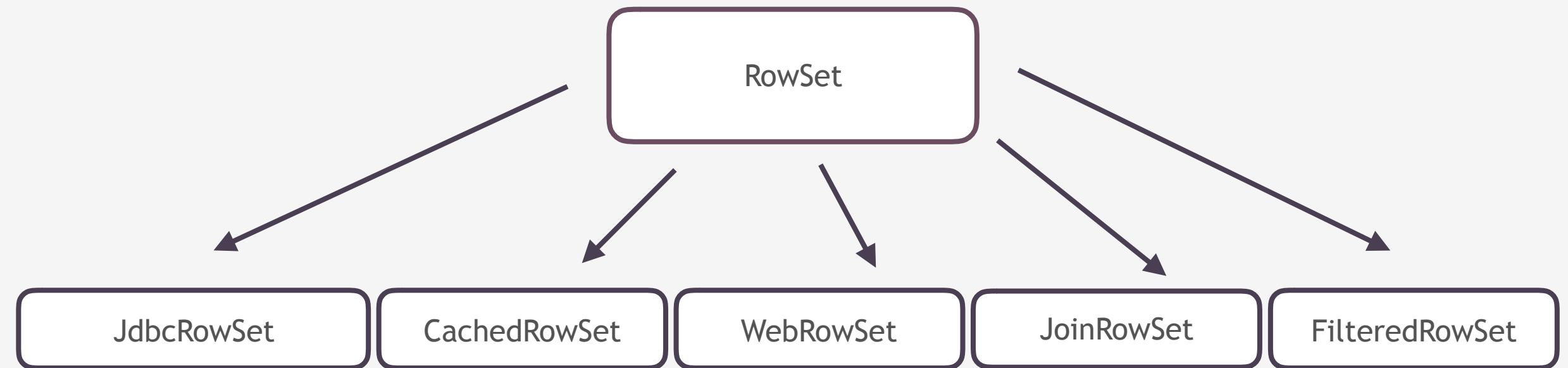
# Cursor

- dostęp do danych ResultSet możliwy jest za pomocą Cursora, który wskazuje na pojedynczy wiersz w tabeli
- w przypadku tworzenia ResultSet należy pamiętać, że Cursor znajduje się na pozycji przed pierwszym wierszem





# Obiekty typu RowSet





# ResultSet w praktyce

```
Statement stmt = null;
String query =
    "select COF_NAME, SUP_ID, PRICE, " +
    "SALES, TOTAL " +
    "from " + dbName + ".COFFEES";

try {
    stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery(query);
    while (rs.next()) {
        String coffeeName = rs.getString("COF_NAME");
        int supplierID = rs.getInt("SUP_ID");
        float price = rs.getFloat("PRICE");
        int sales = rs.getInt("SALES");
        int total = rs.getInt("TOTAL");
        System.out.println(coffeeName + "\t" + supplierID +
                           "\t" + price + "\t" + sales +
                           "\t" + total);
    }
} catch (SQLException e) {
    JDBCUtilities.printSQLException(e);
} finally {
    if (stmt != null) { stmt.close(); }
}
```



# Zadanie 7 - wyświetlanie danych

- w pakiecie parser stwórz dwie klasy ***BookParser*** i ***AuthorParser*** wykorzystujące interfejs ***DataParser<T extends BaseModel>***
- dokonaj implementacji metody interfejsu niezbędnej za przekształcanie danych typu ***ResultSet*** na listę określonego typu obiektów - wykorzystaj pomocniczy przykład z poprzednich slajdów
- w klasie ***BooksManager*** i ***AuthorsManager*** dodaj implementacje metody: ***getSelectQuery()*** w oparciu o klauzule niezbędną do wyświetlania wszystkich danych
- w metodzie ***list()*** wykorzystaj klasę ***Executora*** do dodania zwrócenia wszystkich elementów w tabeli
- w oparciu o poprzednie przykłady dodaj funkcjonalność dla metody: ***ResultSet executeQuery(Action action)*** realizującą zwracanie danych, wykorzystaj pomocniczo ***CachedRowSet***
- wywołaj niezbędne operacje w metodzie main i sprawdź czy wszystkie dane zostały wyświetlane



# Transakcje



# Transakcje

- wykorzystywane w przypadku gdy nie chcemy by jedno zapytanie sql zmieniało stan bazy danych do momentu aż pozostałe, powiązane zapytania nie zostaną wykonane poprawnie
- do wykonania złożonych transakcji niezbędne jest wyłączenie możliwości auto commitów:

```
con.setAutoCommit(false);
```

```
String updateString =
    "update " + dbName + ".COFFEES " +
    "set SALES = ? where COF_NAME = ?";
String updateStatement =
    "update " + dbName + ".COFFEES " +
    "set TOTAL = TOTAL + ? " +
    "where COF_NAME = ?";
try {
    con.setAutoCommit(false);
    updateSales = con.prepareStatement(updateString);
    updateTotal = con.prepareStatement(updateStatement);
    for (Map.Entry<String, Integer> e : salesForWeek.entrySet()) {
        updateSales.setInt(1, e.getValue().intValue());
        updateSales.setString(2, e.getKey());
        updateSales.executeUpdate();
        updateTotal.setInt(1, e.getValue().intValue());
        updateTotal.setString(2, e.getKey());
        updateTotal.executeUpdate();
        con.commit();
    }
} catch (SQLException e) {
    con.rollback();
} finally {
    //close connection
    con.setAutoCommit(true);
}
```

# Ustawianie i przywracanie stanu do SavePointów



```
con.setAutoCommit(false);
String query =
    "SELECT COF_NAME, PRICE FROM COFFEES " +
    "WHERE COF_NAME = '" + coffeeName + "'";

try {
    Savepoint save1 = con.setSavepoint();
    getPrice = con.createStatement(
        ResultSet.TYPE_SCROLL_INSENSITIVE,
        ResultSet.CONCUR_READ_ONLY);
    updatePrice = con.createStatement();

    if (!getPrice.execute(query)) {
        System.out.println(
            "Could not find entry " +
            "for coffee named " +
            coffeeName);
    } else {
        rs = getPrice.getResultSet();
        rs.first();
        float oldPrice = rs.getFloat("PRICE");
        float newPrice = oldPrice + (oldPrice * priceModifier);

        updatePrice.executeUpdate(
            "UPDATE COFFEES SET PRICE = " +
            newPrice +
            " WHERE COF_NAME = '" +
            coffeeName + "'");
    }
}
```

```
CoffeesTable.viewTable(con);
if (newPrice > maximumPrice) {
    System.out.println(
        "\nThe new price, " +
        newPrice +
        ", is greater than the " +
        "maximum price, " +
        maximumPrice +
        ". Rolling back the " +
        "transaction...");
    con.rollback(save1);

    System.out.println(
        "\nCOFFEES table " +
        "after rollback:");

    CoffeesTable.viewTable(con);
}
con.commit();
}
} catch (SQLException e) {
    JDBCUtilities.printSQLException(e);
} finally {
    if (getPrice != null) { getPrice.close(); }
    if (updatePrice != null) {
        updatePrice.close();
    }
    con.setAutoCommit(true);
}
```



# Zadanie 8 - wykonywanie transakcji

- w oparciu o poprzednie przykłady i strukturę aplikacji opracuj własny mechanizm wykonywania transakcji zaprezentowany na poprzednich slajdach