

# Dostęp do baz danych przy wykorzystaniu interfejsu JDBC

---

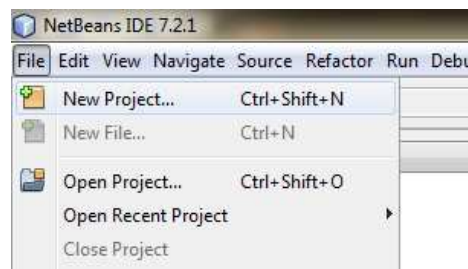
## 1 Wprowadzenie

W opisie zadań wykorzystano środowisko programistyczne *NetBeans* (wersja 7.2.1). Celem ćwiczenia jest zapoznanie studenta z mechanizmem dostępu do bazy danych z poziomu języka Java. Przedstawione zostaną takie elementy jak nawiązywanie połączenia, wykonywanie zapytań prostych i parametryzowanych, obsługa transakcji, uruchamianie procedur w bazie danych itp.

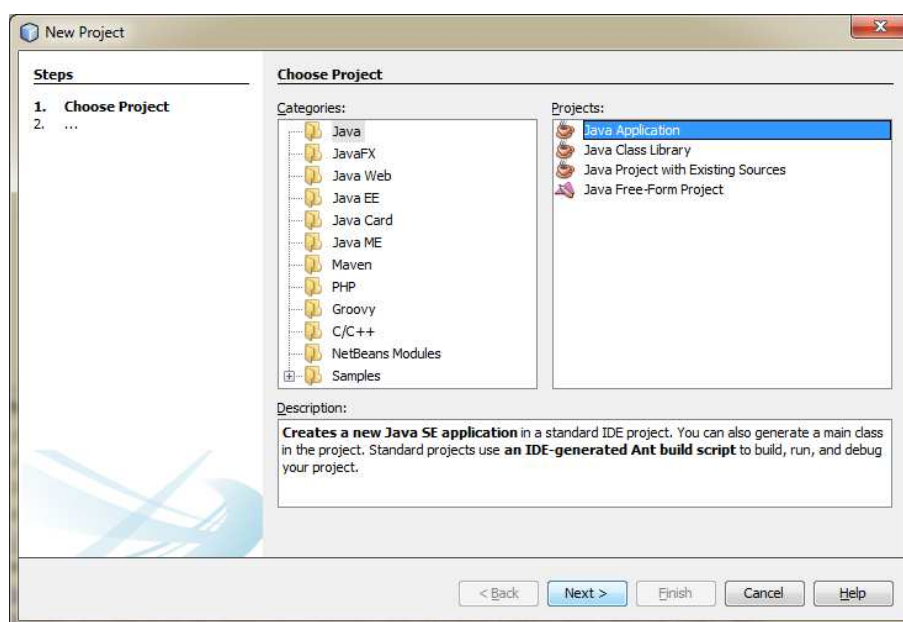
## 2 Utworzenie projektu w środowisku NetBeans

2.1 Uruchom środowisko *NetBeans*.

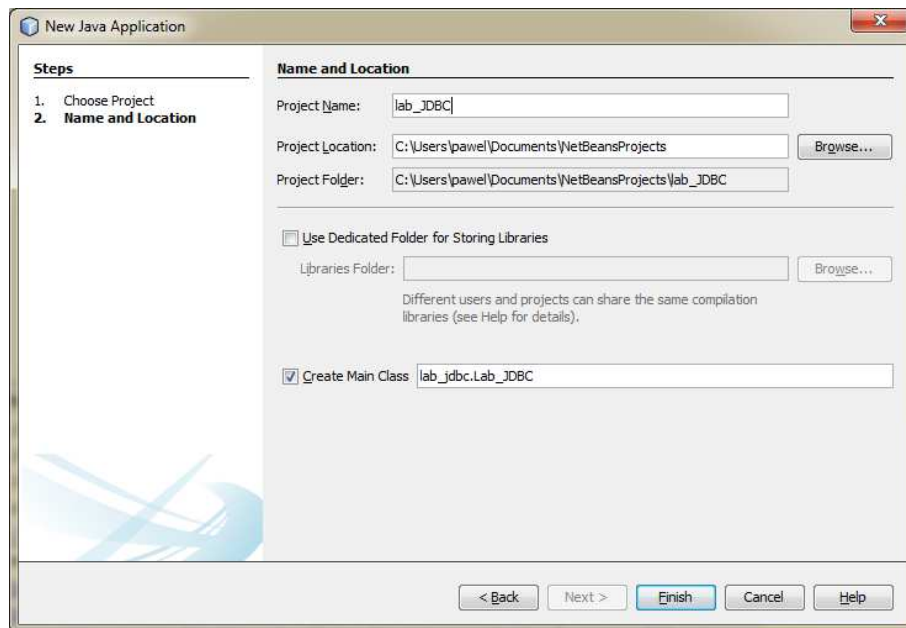
2.2 Utwórz nowy projekt wybierając z górnego menu **File->New Project**.



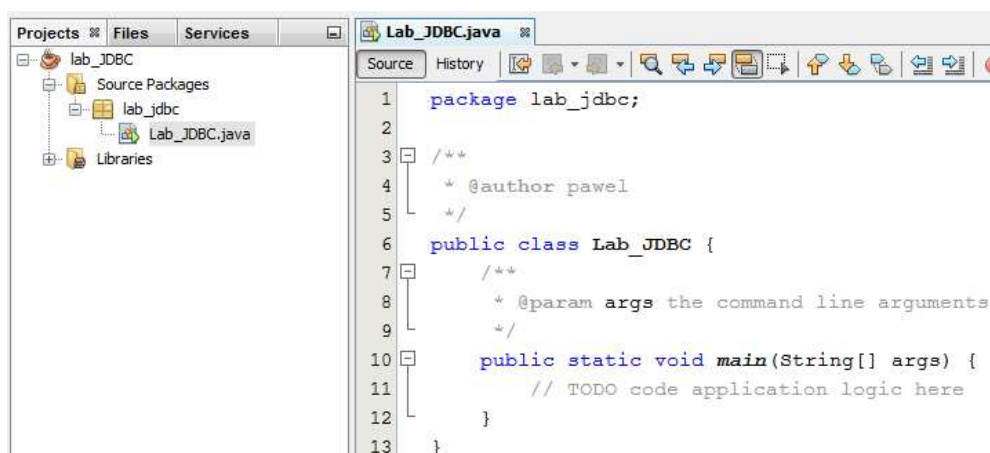
2.3 Wybierz kategorię *Java* i wskaż jako typ projektu *Java Application*. Naciśnij przycisk **Next >**.



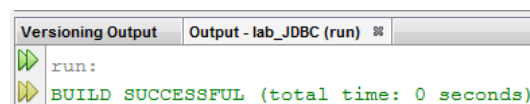
2.4 Wprowadź nazwę projektu np. *lab\_JDBC*. Pozostałe opcje pozostaw bez zmian. Naciśnij przycisk **Finish** aby zakończyć działanie kreatora.



2.5 Efektem działania kreatora jest utworzenie projektu o wskazanej nazwie. W projekcie automatycznie zostanie dodany pakiet *lab\_jdbc* z klasą *Lab\_JDBC*. W tej klasie znajduje się również metoda *main*, która stanowi punkt startowy wykonania programu.



2.6 Utworzony projekt możesz uruchomić wybierając z górnego menu opcję **Run->Run Project** lub używając skrótu klawiszowego **F6**. Przed uruchomieniem nastąpi automatyczna kompilacja programu. Efekty pracy kompilatora oraz programu zostaną wyświetlone w oknie *Output*.



### 3 Nawiązanie połączenia z bazą danych.

3.1 Połączenie z bazą danych nawiązywane jest przy wykorzystaniu klasy *DriverManager*. Konieczne jest podanie URL wskazującego na konkretną instancję bazy danych oraz nazwy użytkownika i

hasła. Ponieważ różne czynniki mogą spowodować brak możliwości ustanowienia połączenia (np. złe hasło, błędny host itp.), należy przechwycić wyjątek *SQLException*. W poniższym przykładzie, po wykryciu wyjątku na stosie, na konsoli zostanie wyświetlony komunikat „nie udało się połączyć z bazą danych”. Pierwszy parametr metody **log** dla obiektu typu *Logger* to poziom błędu. W tym przypadku, opcja *SEVERE* oznacza poważny błąd. Skopiuj poniższy fragment kodu do metody **main**.

```
...
    Connection conn = null;
    Properties connectionProps = new Properties();
    connectionProps.put("user", "login");
    connectionProps.put("password", "haslo");
    try {
        conn = DriverManager.getConnection("
            jdbc:oracle:thin:@//admlab2-main.cs.put.poznan.pl:1521/
            dblab01.cs.put.poznan.pl",
            connectionProps);
        System.out.println("Połączono z bazą danych");
    } catch (SQLException ex) {
        Logger.getLogger(Lab_JDBC.class.getName()).log(Level.SEVERE,
            "nie udało się połączyć z bazą danych", ex);
        System.exit(-1);
    }
...
}
```

3.2 Dodaj przed deklaracją klasy *Lab\_JDBC* następujące polecenia. Służą one do zaimportowania obiektów wykorzystywanych przy komunikacji *JDBC* oraz do logowania zdarzeń.

```
import java.sql.*;
import java.util.Properties;
import java.util.logging.Level;
import java.util.logging.Logger;
```

3.3 Uruchom aplikację i sprawdź efekt jej działania (komunikaty na konsoli).

**UWAGA!** Jednym z możliwych wyników jest następujący błąd:

```
run:
lis 05, 2012 10:02:28 PM lab_jdbc.Lab_JDBC main
SEVERE: nie udało się połączyć z bazą danych
java.sql.SQLException: No suitable driver found for
    jdbc:oracle:thin:@//admlab2-min.cs.put.poznan.pl:1521/dblab01
    at java.sql.DriverManager.getConnection(DriverManager.java:604)
    at java.sql.DriverManager.getConnection(DriverManager.java:190)
    at lab_jdbc.Lab_JDBC.main(Lab_JDBC.java:22)

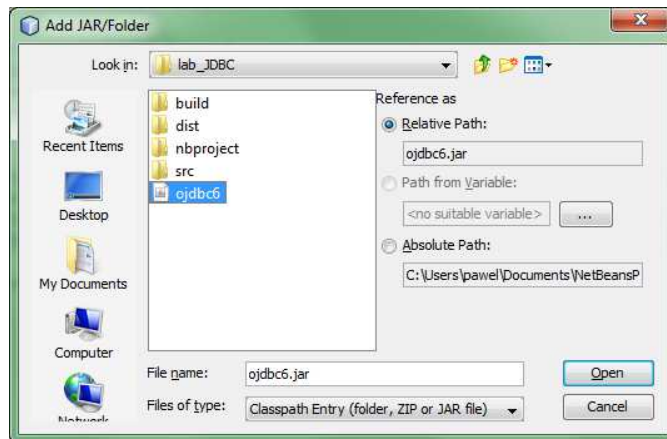
Java Result: -1
BUILD SUCCESSFUL (total time: 0 seconds)
```

Komunikat ten oznacza, że nie udało się odnaleźć sterownika do bazy danych. Może to wynikać np. z tego, że plik sterownika nie jest dostępny na ścieżce wskazywanej przez zmienną *CLASSPATH*. W środowisku *NetBeans* można dodać do projektu biblioteki (w tym przypadku plik sterownika *JDBC*), które będą dostępne dla aplikacji. W tym celu:

- (a) Kliknij prawym przyciskiem myszy na węzeł *Libraries* w drzewie projektu.



(b) Z dostępnych opcji wybierz **Add JAR/Folder...**



(c) Wskaż plik sterownika (w konfiguracji używanej na zajęciach jest to *ojdbc6.jar*). Naciśnij przycisk **Open**.

(d) Ponownie uruchom aplikację. Poprawny wynik działania powinien wyglądać następująco:

```
run:
Połączono z bazą danych
BUILD SUCCESSFUL (total time: 0 seconds)
```

**UWAGA!** Jeżeli używasz sterownika *JDBC* w wersji niższej niż 4, konieczne jest dodanie (na początku) następującego kodu, który rejestruje sterownik:

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

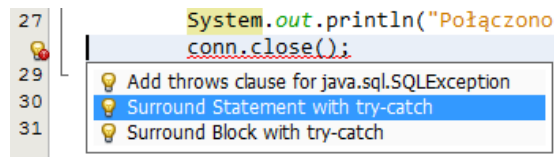
W przypadku opisywanego rozwiązania, sterownik *odbc6.jar* obsługuje mechanizm *JDBC* w wersji 4 i nie należy umieszczać w/w kodu.

3.4 Jako ostatni rozkaz metody *main* dodaj polecenie rozłączenia się z bazą danych. Zauważ, że środowisko programistyczne podkreśliło dodany fragment kodu. Jest to spowodowane koniecznością obsługi wyjątków, które mogą się pojawić przy operacjach wywoływanych na bazie danych.

```
27 |         System.out.println("Połączono z bazą danych.");
28 |         conn.close();
29 |     }
}
```

3.5 Kliknij lewym przyciskiem myszy na ikonkę żarówki po lewej stronie pola tekstowego.

```
27 System.out.println("Połączono");
28 conn.close();
29
30
31
```



3.6 Wybierz opcję *Surround Statement with try-catch*. Obejrzyj wygenerowany kod.

```
27 System.out.println("Połączono z bazą danych");
28 try {
29     conn.close();
30 } catch (SQLException ex) {
31     Logger.getLogger(Lab_JDBC.class.getName()).log(Level.SEVERE, null, ex);
32 }
```

3.7 Dodaj na końcu metody **main** informację o odłączeniu się od bazy danych. Możesz to zrobić używając metody **println**. (Wskazówka dla użytkowników *NetBeans*: napisz *sout* i naciśnij klawisz tabulatora, automatycznie zostanie wygenerowany kod `System.out.println("")`, który można uzupełnić o własny parametr będący tekstem przeznaczonym do wyświetlenia na konsoli).

## 4 Odczyt danych

4.1 Przed wykonaniem polecenia *SQL* należy utworzyć obiekt implementujący interfejs *Statement*. Można tego dokonać poprzez wywołanie bezparametrowej metody **createStatement** na obiekcie połączenia do bazy danych.

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

4.2 Do odczytu danych z bazy danych wykorzystywany jest obiekt implementujący interfejs *ResultSet*. Możesz go uzyskać poprzez wywołanie metody **executeQuery** dostarczanej przez interfejs *Statement*. Parametr metody to treść zapytania *SQL*. Nie wstawiaj średnika na końcu zapytania.

```
ResultSet rs;
rs = stmt.executeQuery("treść zapytania SQL");
```

4.3 Interfejs *ResultSet* dostarcza metody, które mogą być wykorzystane do odczytania wyniku wykonania zapytania. Wśród tych metod można znaleźć m.in. bezparametrową metodę **next**, która przesuwa wskaźnik na kolejną krotkę ze zbioru wyników. Po utworzeniu obiektu implementującego interfejs *ResultSet*, wskaźnik ten znajduje się przed pierwszą krotką ze zbioru wyników. Metoda zwraca wartość *true* jeśli udało się przejść do kolejnej krotki, w przeciwnym wypadku zwracana jest wartość *false*.

```
boolean w = rs.next();
```

4.4 Interfejs *ResultSet* dostarcza także metody do pobrania wartości z aktualnie wskazywanej krotki. Są to, m.in., **getInt**, **getString**, **getFloat**. Każda z tych metod przyjmuje jako parametr jedną z wartości: indeks atrybutu z klauzuli *SELECT* lub nazwę atrybutu z klauzuli *SELECT*. Typ zwracanej wartości zależy od wybranej metody.

```
int x = rs.getInt(4);
String s = rs.getString(1);
```

Pełną listę metod możesz uzyskać czytając dokumentację *API JDBC* lub poprzez wykorzystanie podpowiedzi kontekstowych w środowisku *NetBeans*. Wpisz „*rs.get*” a na ekranie pojawi się lista dostępnych metod interfejsu *ResultSet* zaczynających się od słowa *get*.

The screenshot shows an IDE with a search for `rs.get`. A list of methods is displayed, with `getMetaData()` selected. Below the list, the documentation for `getMetaData()` is shown:

```

public ResultSetMetaData getMetaData() throws
    SQLException

Retrieves the number, types and properties of this ResultSet object's
columns.

Returns:
    the description of this ResultSet object's columns

Throws:
    SQLException - if a database access error occurs or this method
    is called on a closed result set

```

4.5 Ostatnim krokiem jest zwolnienie zajmowanych zasobów. Zasoby są zwalniane w odwrotnej kolejności niż były tworzone. Do zwalniania zasobów służą metody **close** interfejsów *ResultSet* i *Statement*. Jeżeli chcesz wykonywać kolejne zapytania zamknij tylko obiekt implementujący interfejs *ResultSet* i ponownie wykonaj punkty 4.2 – 4.4, a na końcu wywołaj metodę **close** na obiekcie implementującym interfejs *Statement*.

```
rs.close();
stmt.close();
```

#### Konieczność jawnego zamykania obiektów.

Kluczowym pojęciem, pomagającym w zrozumieniu konieczności zamykania odpowiednich obiektów, jest termin „kursor”. Kursorem nazywamy obszar pamięci operacyjnej na serwerze SZBD (System Zarządzania Bazą Danych) zawierający przetwarzany przez SZBD zbiór krotek (np. wyniki zapytania, krotki modyfikowane przez polecenia DML). Sposób dostępu do kursora zależy od języka programowania. W języku Java każdy obiekt implementujący interfejs *ResultSet* zapewnia interfejs pozwalający na odczytywanie wyników zwracanych przez kursor zapytania w SZBD. Kursor jest rezerwowany przez obiekt implementujący interfejs *Statement*, w momencie jego utworzenia. Zamknięcie tego obiektu za pomocą metody *close* zwalnia w SZBD zajęty przez niego kursor. Maksymalna liczba używanych równocześnie kursorów jest ograniczona. Jeżeli obiekty typu *Statement* nie będą zamykane, w krótkim czasie dojdzie do zajęcia wszystkich dostępnych kursorów, przez co nie będzie możliwe wykonywanie poleceń przez SZBD. W ogólności możliwe jest równoczesne wykonywanie wielu poleceń SQL, jednak dla każdego z tych poleceń należy utworzyć osobny obiekt implementujący interfejs *Statement*, gdyż na każdy taki obiekt przypada jeden kursor, a zatem może na niego przypadać tylko jeden obiekt implementujący interfejs *ResultSet*.

4.6 Poniżej znajduje się kod odpowiedzialny za odczyt wartości trzech atrybutów relacji *Pracownicy* z bazy danych (*id\_prac*, *nazwisko*, *placa\_pod*). Dopisz do programu pokazany kod poprzez umieszczenie go po poleceniu otwierającym połączenie do bazy danych.

Zwróć uwagę na wielopoziomowy schemat obsługi błędów. Takie rozwiązanie zapewnia, że wystąpienie błędu przy którymkolwiek z poleceń bazodanowych zostanie wykryte i obsłużone przez aplikację (w przykładzie pominięto kod związany z reakcją na zaistniałe błędy – możesz go dopisać np. umieszczając polecenia wypisujące na konsoli odpowiednie komunikaty).

W kolejnych ćwiczeniach kod związany z wykrywaniem błędów i zwalnianiem zasobów nie będzie prezentowany w przykładach (co nie oznacza, że należy taki kod pomijać w programie).

```
Statement stmt = null;
ResultSet rs = null;
try {
    stmt = conn.createStatement();
    rs = stmt.executeQuery("select id_prac, nazwisko, placa_pod " +
        "from pracownicy");

    while (rs.next()) {
        System.out.println(rs.getInt(1) + " " + rs.getString(2) + " " +
            rs.getFloat(3));
    }
} catch (SQLException ex) {
    System.out.println("Błąd wykonania polecenia" + ex.toString());
} finally {
    if (rs != null) {
        try {
            rs.close();
        } catch (SQLException e) { /* kod obsługi */ }
    }
    if (stmt != null) {
        try {
            stmt.close();
        } catch (SQLException e) { /* kod obsługi */ }
    }
}
```

4.7 Uruchom aplikację i sprawdź czy format wyniku zgadza się z poniższym.

```
Połączono z bazą danych
100 WEGLARZ 2595.0
110 BLAZEWICZ 2025.0
120 SLOWINSKI 1605.0
130 BRZEZINSKI 1440.0
140 MORZY 1245.0
150 KROLIKOWSKI 968.25
160 KOSZLAJDA 885.0
170 JEZIERSKI 659.55
190 MATYSIAK 556.5
180 MAREK 615.3
200 ZAKRZEWICZ 312.0
210 BIALY 375.0
220 KONOPKA 720.0
230 HAPKE 720.0
Odłączono od bazy danych
```

4.8 Użyj funkcji *LPAD/RPAD* języka SQL do “wyrównania” wyników zapytania. Wykorzystaj metody dostępu do wartości krotek w wersji z nazwami zamiast identyfikatorów atrybutów z klauzuli *SELECT*. Ponownie uruchom aplikację.



```

Połączono z bazą danych
100 WEGLARZ      2595.0
110 BŁAZEWICZ   2025.0
120 SŁOWINSKI   1605.0
130 BRZEZINSKI  1440.0
140 MORZY        1245.0
150 KROLIKOWSKI 968.25
160 KOSZLAJDA   885.0
170 JEZIERSKI   659.55
190 MATYSIAK    556.5
180 MAREK        615.3
200 ZAKRZEWICZ  312.0
210 BIAŁY        375.0
220 KONOPKA     720.0
230 HAPKE        720.0
Odłączono od bazy danych

```

### Zadanie 1

Napisz program, który wyświetla następujące informacje:

Zatrudniono X pracowników, w tym:

$X_1$  w zespole  $Y_1$ ,

$X_2$  w zespole  $Y_2$ ,

...

## 5 Przewijalne zbiory wyników

5.1 Od wersji 2.0 *JDBC* wprowadzono możliwość przeglądania zbioru wyników w dowolnej kolejności (o ile pozwala na to sam *SZBD*). Interfejs *ResultSet* definiuje następujące metody związane z przewijalnym zbiorem wyników:

<b>absolute(n)</b>	ustawia wskaźnik na n-tej krotce wyniku zapytania. Jeżeli n jest ujemne (-n), to pozycja wskaźnika jest liczona od końca zbioru wynikowego (n-ta krotka od końca)
<b>relative(n)</b>	przesuwa wskaźnik o n krotek względem aktualnej pozycji. Wartość n może być zarówno dodatnia (przesunięcie do przodu), jak i ujemna (cofnięcie się)
<b>beforeFirst</b>	przesuwa wskaźnik na pozycję przed pierwszą krotką
<b>afterLast</b>	przesuwa wskaźnik na pozycję za ostatnią krotką
<b>first</b>	przesuwa wskaźnik na pierwszą krotkę
<b>last</b>	przesuwa wskaźnik na ostatnią krotkę
<b>next</b>	znane z poprzednich slajdów, przesuwa wskaźnik na następną krotkę i zwraca prawdę, jeśli nie znalazł się on za ostatnią krotką
<b>previous</b>	odwrotność <b>next</b> , przesuwa wskaźnik na poprzednią krotkę i zwraca prawdę, jeśli nie znalazł się on przed pierwszą krotką
<b>isAfterLast</b>	zwraca <i>true</i> , jeśli wskaźnik znajduje się za ostatnią krotką i <i>false</i> w przeciwnym wypadku
<b>isBeforeFirst</b>	zwraca <i>true</i> , jeśli wskaźnik znajduje się przed pierwszą krotką i <i>false</i> w przeciwnym wypadku



<b>isFirst</b>	zwraca <i>true</i> jeśli wskaźnik znajduje się na pierwszej krotce i <i>false</i> w przeciwnym wypadku
<b>isLast</b>	zwraca <i>true</i> jeśli wskaźnik znajduje się na ostatniej krotce i <i>false</i> w przeciwnym wypadku

## 5.2 Wykonaj dowolne zapytanie odczytujące krotki z bazy danych w odwrotnej kolejności.

```
rs.afterLast();
while (rs.previous()) {
    System.out.println(rs.getString(2));
}
```

Próba wykonania takiego polecenia skończy się błędem

```
java.sql.SQLException: Niepoprawna operacja dla zestawu wyników "tylko do przesłania" : afterLast
```

Używanie opisanych metod jest możliwe tylko wtedy, gdy poinformujemy sterownik (a tym samym *SZBD*), że chcemy korzystać z przewijalnych zbiorów wyników. Można tego dokonać poprzez utworzenie obiektu implementującego interfejs *Statement*, za pomocą innej wersji metody **createStatement**, która przyjmuje dwa parametry typu liczbowego. Konkretnie wartości są reprezentowane przez zdefiniowane w *JDBC* stałe. Pierwszy parametr określa dopuszczalny sposób przeglądania wyników zapytania, a drugi określa czy zbiór wyników można modyfikować. Pierwszy parametr może przyjmować następujące wartości:

<b>ResultSet.TYPE_SCROLL_FORWARD</b>	klasyczny rodzaj wyników zapytania, wyniki mogą być przeglądane jedynie sekwencyjnie, „do przodu”
<b>ResultSet.TYPE_SCROLL_INSENSITIVE</b>	wyniki mogą być przeglądane w dowolny sposób, ale nie odzwierciedlają zmian wykonanych przez innych użytkowników na relacjach, do których odnosi się zapytanie
<b>ResultSet.TYPE_SCROLL_SENSITIVE</b>	wyniki mogą być przeglądane w dowolny sposób i odzwierciedlają zmiany wykonane przez innych użytkowników

Drugi parametr może przyjąć jedną z dwóch wartości:

<b>ResultSet.CONCUR_READ_ONLY</b>	zbiór wyników nie można modyfikować
<b>ResultSet.CONCUR_UPDATABLE</b>	zbiór wyników można modyfikować

### Modyfikowalne zbiory wyników.

Ponieważ wynik zapytania jest relacją, to przy pewnych spełnionych warunkach można go przetwarzać w celu modyfikowania relacji bazowych. W *JDBC 2.0* wprowadzono funkcjonalność pozwalającą na modyfikowanie wyniku zapytania dostępnego poprzez obiekt implementujący interfejs *ResultSet*. Wszystkie wymagania, które zapytanie musi spełnić, aby była możliwość modyfikowania wyników, odpowiadają wymaganiom dla zapytań dotyczących modyfikowalnych perspektyw. Dodatkowo, aby utworzony w wyniku wykonania zapytania obiekt pozwalał na modyfikację danych, obiekt implementujący interfejs *Statement* musi być utworzony z parametrem **ResultSet.CONCUR\_UPDATABLE**.

Kiedy wskaźnik jest ustawiony na krotce, którą chcemy zmodyfikować, należy użyć metod

**updateXXX** (gdzie XXX oznacza typ wartości), za pomocą których można modyfikować wartości w wyniku. Metody **updateXXX** posiadają dwa parametry: pierwszy określa który atrybut ma zostać zmodyfikowany (nazwa albo numer wystąpienia w klauzuli *SELECT*), drugi określa nową wartość atrybutu. Aby wprowadzone zmiany zostały zapisane do bazy danych, należy je zatwierdzić za pomocą metody **updateRow** interfejsu *ResultSet*. Alternatywnie można je wycofać za pomocą metody **cancelRowUpdates** interfejsu *ResultSet*. Przesunięcie wskaźnika na inną krotkę w zbiorze wynikowym również powoduje wycofanie zmian w krotce. Aby wstawić nową krotkę do relacji wynikowej, należy przesunąć wskaźnik na specjalną, wirtualną krotkę, która jest wypełniona samymi wartościami *NULL*. Można to wykonać za pomocą metody **moveToInsertRow** interfejsu *ResultSet*. Następnie, za pomocą metod **updateXXX**, należy wypełnić wartości poszczególnych atrybutów. Należy pamiętać o podaniu przynajmniej wszystkich wartości obowiązkowych, gdyż jeżeli tego nie zrobimy, zostanie zgłoszony wyjątek. Kiedy wszystkie wartości zostaną wypełnione, krotkę można wstawić za pomocą metody **insertRow** interfejsu *ResultSet*. Metoda ta wstawia do bazy danych nową krotkę i przesuwa wskaźnik na nowy, wirtualny, wiersz do wstawiania krotek. Aby przywrócić położenie wskaźnika do pozycji, na której się znajdował przed aktywowaniem metody **moveToInsertRow**, należy użyć metody **moveToCurrentRow**. Usuwanie krotek polega na przesunięciu wskaźnika na krotkę, którą chcemy usunąć, a następnie aktywowaniu bezparametrowej metody **deleteRow** interfejsu *ResultSet*.

5.3 Wykorzystaj poniższy fragment kodu do modyfikacji programu. Ponownie uruchom aplikację.

```
stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,  
                            ResultSet.CONCUR_READ_ONLY);
```

### Zadanie 2 Modyfikacja danych

Napisz program, który wykonuje zapytanie odnajdujące wszystkich pracowników zatrudnionych na etacie ASYSTENT i sortuje ich malejąco według pensji, a następnie wyświetla asystenta, który zarabia najmniej, trzeciego najmniej zarabiającego asystenta i przedostatniego asystenta w rankingu najmniej zarabiających asystentów (użyj do tego celu kursorów przewijanych).

## 6 Modyfikacja danych, zarządzanie transakcjami

6.1 Wykonywanie aktualizacji danych jest bardzo podobne do wykonywania zapytań. Metodą przeznaczoną do tego celu jest metoda **executeUpdate** interfejsu *Statement*. Metoda ta przyjmuje jako parametr łańcuch zawierający polecenie *SQL-DML* i zwraca liczbę utworzonych, zmodyfikowanych bądź usuniętych krotek (w zależności od typu zapytania). Wprowadź do programu poniższy kod i przeanalizuj uzyskane wyniki. Pamiętaj o utworzeniu obiektu implementującego interfejs *Statement* oraz o zamknięciu tego obiektu po wykonaniu zapytań.

```
int changes;  
  
changes = stmt.executeUpdate("INSERT INTO pracownicy(id_prac,nazwisko) "  
                             + "VALUES(5,'Marecki')");  
System.out.println("Wstawiono " + changes + " krotek.");  
  
changes = stmt.executeUpdate("UPDATE pracownicy SET "  
                             + "placa_pod=placa_pod*1.5");  
System.out.println("Zmodyfikowano " + changes + " krotek.");
```

```
changes = stmt.executeUpdate("DELETE FROM pracownicy WHERE id_prac=5");
System.out.println("Usunieto " + changes + " krotek.");
```

#### Wykonywanie poleceń DDL.

Metoda `executeUpdate` pozwala również na wykonywanie zapytań *DDL*. Jediną różnicą w stosunku do wykorzystania tej metody do poleceń *DML* jest to, iż wartość zwracana przez metodę `executeUpdate` nie ma tutaj znaczenia i można ją zignorować.

#### Zadanie 3

Dane są następujące tablice opisujące zmiany personalne:

```
int [] zwolnienia={150, 200, 230};
String [] zatrudnienia={"Kandefer", "Rygiel", "Boczar"};
```

Tablica *zwolnienia* zawiera identyfikatory pracowników, których należy zwolnić, a tablica *zatrudnienia* – nazwiska pracowników, których należy zatrudnić.

Napisz program, który wykona w bazie danych zmiany opisane w tablicach. W celu generowania kluczy dla nowych pracowników utwórz sekwencję (niekoniecznie z poziomu własnego programu).

6.2 Realizując wcześniejsze ćwiczenia można zauważyć, że wprowadzone zmiany są utrwalane w bazie danych. Wiemy również, że wszystkie polecenia w *SZBD Oracle* są realizowane w ramach transakcji i tylko zmiany wprowadzone przez zatwierdzone transakcje są utrwalane w bazie danych. Oznacza to, że wykonywane wcześniej polecenia były automatycznie utrwalane przez niejawnie zatwierdzenia transakcji przez *JDBC*. W *JDBC* w ramach jednego połączenia z bazą danych, można równocześnie realizować tylko jedną transakcję. Dodatkowo, każde nawiązane połączenie jest domyślnie skonfigurowane tak, że każde wykonywane w jego ramach polecenie *SQL* jest automatycznie zatwierdzane (jest osobną transakcją). Aby tego uniknąć, należy wykorzystać metodę `setAutoCommit` dostępną dla obiektu połączenia. Metoda ta przyjmuje jeden parametr typu *boolean*, którego wartość jest równa *true* lub *false* w zależności od tego czy chcemy włączyć czy wyłączyć automatyczne zatwierdzanie transakcji (uwaga: umieść wywołanie metody zaraz po utworzeniu obiektu połączenia). Do jawnego zatwierdzania lub wycofywania transakcji wykorzystuje się metody `commit` i `rollback` dostępne dla obiektu połączenia.

Przetestuj działanie poniższego kodu, sprawdź np. z poziomu *iSQLPlus'a* czy wprowadzone modyfikacje zostały utrwalone w bazie danych.

```
conn.setAutoCommit(false);
stmt = conn.createStatement();
stmt.executeUpdate("INSERT INTO pracownicy(id_prac,nazwisko)"
+ "VALUES(200, 'Nowacki')");
conn.rollback();
```

6.3 Usuń polecenie `conn.rollback()` i ponownie wykonaj program. Co zauważyłaś/eś?

#### Zadanie 4

Napisz program wykonujący następujące czynności:

- Wyłącz automatyczne zatwierdzanie transakcji.
- Wyświetl wszystkie etaty.
- Wstaw nowy etat do tabeli ETATY.
- Ponownie wyświetl wszystkie etaty.
- Wycofaj transakcję.

- Ponownie wyświetl wszystkie etaty.
- Wstaw nowy etat do tabeli ETATY.
- Zatwierdź transakcję.
- Ponownie wyświetl wszystkie etaty

## 7 Polecenia prekompilowane (przygotowane)

7.1 Jednym z rozwiązań, którego celem jest przede wszystkim zwiększenie wydajności pracy z SZBD są tzw. polecenia prekompilowane zwane również poleceniami przygotowanymi. Najczęściej występują one w kontekście parametryzowanych zapytań, chociaż mogą dotyczyć także zwykłych, nieparametryzowanych poleceń *SQL*. Polecenia przygotowane są kompilowane przez bazę danych tylko jeden raz, a następnie mogą być wykonywane wielokrotnie.

Polecenia przygotowane są reprezentowane przez interfejs *PreparedStatement*. Obiekt implementujący ten interfejs zwracany jest przez metodę **prepareStatement** obiektu połączenia do bazy danych.

```
PreparedStatement pstmt;  
pstmt = conn.prepareStatement("select nazwisko from pracownicy");
```

7.2 Wykonanie przygotowanego zapytania dla polecenia *SELECT* jest bardzo podobne do standardowego zapytania omówionego wcześniej. Różnicę stanowi konieczność wywołania metody (bezparametrowej) **executeQuery** na obiekcie reprezentującym prekompilowane polecenie. Wykorzystaj kod z tego i poprzedniego punktu i sprawdź efekt działania prekompilowanego polecenia. Pamiętaj o zwalnianiu zasobów.

```
ResultSet rs = pstmt.executeQuery();  
while (rs.next()) {  
    System.out.println(rs.getString("NAZWISKO"));  
}
```

7.3 W rzeczywistych zastosowaniach duża liczba aplikacji wykorzystujących *SZBD* wykonuje jedynie niewielką liczbę różnych poleceń *SQL*, w których zmieniają się jedynie dane. Jeżeli taka aplikacja za każdym razem przesyła do *SZBD* polecenie *SQL* w postaci łańcucha, to polecenie to musi zostać przeanalizowane pod kątem składni, zoptymalizowane i skompilowane. Czas wykonania tych operacji może stanowić nawet kilkadziesiąt procent czasu realizacji całego zapytania. Problem ten można rozwiązać tworząc prekompilowane polecenia, w którym można zmieniać jedynie pewne parametry (nazywane zmiennymi wiązania). W odróżnieniu od przygotowanych poleceń bez parametrów, w poleceniach parametryzowanych, w treści polecenia *SQL*, stosuje się znaki zapytania w miejscach które zmieniają się pomiędzy wywołaniami tego polecenia. Poniżej znajduje się przykład tworzący parametryzowane polecenie prekompilowane. Łatwo zauważyć, że jest to zapytanie odczytujące nazwisko pracownika, którego identyfikator zostanie podany w klauzuli *WHERE*. W wyniku działania metody **prepareStatement** zapytanie zostanie wysłane do *SZBD* i skompilowane, oraz zostanie zarezerwowany kursor na potrzeby późniejszego jego wykonania. Dodatkową zaletą prekompilowanych poleceń jest to, iż programy napisane z ich użyciem są odporne na ataki typu *SQL Injection*.

```
PreparedStatement pstmt = conn.prepareStatement(  
    "SELECT nazwisko FROM pracownicy WHERE id_prac=?");
```

7.4 Do przypisania konkretnej wartości dla znaku zapytania wykorzystuje się odpowiednią metodę **setXXX** interfejsu *PreparedStatement*, gdzie *XXX* oznacza typ wartości zapisywanej do zapytania. Każda taka metoda ma dwa parametry, z których pierwszy to numer wystąpienia znaku zapytania

(wystąpienia są liczone od 1), natomiast drugi parametr to podstawiana wartość. Zakładając, że chcemy odczytać nazwiska najpierw pracownika o numerze 120 a potem o numerze 150 powinniśmy wprowadzić następujący kod:

```
pstmt.setInt(1, 120);
rs = pstmt.executeQuery();
while (rs.next()) {...}
rs.close();

pstmt.setInt(1, 150);
rs = pstmt.executeQuery();
while (rs.next()) {...}
rs.close();

pstmt.close();
```

7.5 Polecenia prekompilowane mogą dotyczyć również poleceń modyfikujących dane. W takim przypadku jedyną różnicą jest użycie metody **executeUpdate** definiowanej przez interfejs *PreparedStatement* zamiast metody **executeQuery**. Pozostałe metody tzn. **prepareStatement**, **setXXX** pozostają bez zmian.

**Zadanie 5**

Dane są następujące tablice opisujące nowych pracowników:

```
String [] nazwiska={"Woźniak", "Dąbrowski", "Kozłowski"};
int [] placy={1300, 1700, 1500};
String [] etaty={"ASYSTENT", "PROFESOR", "ADIUNKT"};
```

Kolejne pozycje tych tablic opisują różne atrybuty nowych pracowników. Wstaw nowych pracowników do relacji PRACOWNICY wykorzystując mechanizm przygotowanych zapytań. Ponownie wyświetl wszystkie etaty.

## 8 Przetwarzanie wsadowe

8.1 OD wersji 2.0 *JDBC* możliwe jest wsadowe wykonywanie poleceń aktualizacji. Polega ono na przygotowywaniu zbioru zapytań do wykonania przy pojedynczej komunikacji z *SZBD*. Do realizacji takiego przetwarzania wsadowego wykorzystywane są prekompilowane polecenia parametryzowane. Na początku należy utworzyć nowe polecenie parametryzowane. W przykładzie jest to polecenie modyfikacji płacy podstawowej pracownika o wskazanym identyfikatorze. Parametrami są współczynnik podwyżki (lub obniżki) oraz identyfikator pracownika.

```
PreparedStatement pstmt;
pstmt = conn.prepareStatement("UPDATE pracownicy SET "
+ "placa_pod=placa_pod * ? WHERE id_prac = ? ");
```

8.2 Kolejnym krokiem jest przygotowanie zestawu parametrów dla których zostaną wykonane polecenia. Wartości dla znaków zapytania ustawiane są za pomocą omówionych wcześniej metod **setXXX**. Dodatkowo, po każdym ustawieniu pary parametrów konieczne jest dodanie ich do zbioru, czyli tzw. wsadu. Służy do tego bezparametrowa metoda **addBatch** interfejsu *PreparedStatement*.

```
pstmt.setFloat(1, new Float(1));
pstmt.setInt(2, 130);
```

```
pstmt.addBatch();

pstmt.setFloat(1, new Float(0));
pstmt.setInt(2, 0);
pstmt.addBatch();
```

Gdy wszystkie parametry zostaną ustawione można wykonać zbiór zapytań. Służy do tego metoda **executeBatch** interfejsu *PreparedStatement*.

```
pstmt.executeBatch();
```

Rezultatem opisywanej metody jest tablica zawierająca wartości całkowitoliczbowe (typ *int*). Każdy z elementów wskazuje rezultat wykonania odpowiedniego polecenia wchodzącego w skład zbioru zapytań. Możliwe wartości i ich interpretacja zostały przedstawione w poniższej tabelce.

<b>Wartość większa lub równa zero</b>	Poprawne wykonanie polecenia + liczba krotek zmodyfikowanych
<b>Statement.SUCCESS_NO_INFO</b>	Poprawne wykonanie polecenia, ale nie jest znana liczba zmodyfikowanych krotek
<b>Statement.EXECUTE_FAILED</b>	Polecenie nie zostało poprawnie wykonane (dostępne tylko dla sterowników, które kontynuują wykonywanie poleceń po zajściu błędu).

Usunięcie utworzonego zbioru poleceń jest możliwe przy użyciu metody **clearBatch** interfejsu *PreparedStatement*.

### Zadanie 6

Spróbuj wstawić po 2000 pracowników: (1) wykonując sekwencyjnie polecenia wstawienia rekordu do relacji oraz (2) łącząc wszystkie polecenia w jedno zadanie wsadowe. Całe zadanie wykonaj w ramach jednej transakcji. Wykonaj pomiary czasu potrzebnego do wykonania jednego i drugiego wstawiania.

Podpowiedź: przybliżony czas (w nanosekundach) można zmierzyć za pomocą statycznej metody **nanoTime** klasy *System*:

```
long start = System.nanoTime();
// kod, dla którego mierzymy czas wykonania
long czas = System.nanoTime() - start;
```

## 9 Procedury i funkcje składowane

9.1 Z poziomu aplikacji języka Java można wywoływać procedury składowane w bazie danych. Służy do tego obiekt interfejsu *CallableStatement*. Taki obiekt można uzyskać przez wywołanie metody **prepareCall** obiektu reprezentującego połączenie do bazy danych. Parametrem tej metody jest łańcuch znaków o określonej strukturze *{call nazwaMetody(x,y,...)}* gdzie *x* i *y* to parametry procedury, które mogą być obsługiwane na zasadzie analogicznej do parametryzowanych zapytań (zapisujemy je wówczas jako znaki zapytania). Do ustawiania konkretnych wartości parametrów służą metody **setXXX**, gdzie **XXX** to nazwa typu danych. Wykonanie polecenia, a w rezultacie wywołanie procedury, odbywa się przy użyciu metody **execute** interfejsu *CallableStatement*. Poniżej zaprezentowano przykład wywołania procedury *WstawZespol*. Utwórz procedurę *WstawZespol* w bazie danych. Procedura przyjmuje trzy parametry: nr zespołu, nazwę oraz adres.

Jedynym działaniem procedury polega na wstawieniu zespołu do bazy danych. Przetestuj działanie poniższego kodu.

```
CallableStatement stmt = conn.prepareStatement("{call WstawZespól(?,?,?)}");  
  
stmt.setInt(1, 60);  
stmt.setString(2, "NOWY ZESPÓŁ");  
stmt.setString(3, "PIOTROWO 3A");  
  
stmt.execute();  
stmt.close();
```

9.2 Funkcje w bazie danych różnią się tym od procedur, że zwracają wynik. Obsługa funkcji przy wykorzystaniu JDBC jest bardzo podobna do wykonywania procedur. Różnice dotyczą łańcucha tekstowego przekazywanego do metody **prepareCall** obiektu połączenia do bazy danych oraz możliwości użycia metody **getXXX** (XXX reprezentuje typ danych wyniku). Dodatkowo istnieje konieczność zarejestrowania parametru wyjściowego przez wywołanie metody **registerOutParameter** interfejsu *CallableStatement*. Pierwszy parametr tej metody to numer parametru wyjściowego, drugi parametr to stała liczbowa, która reprezentuje typ danych parametru wyjściowego. Przykład wywołania funkcji *PoliczPrac*, która przyjmuje jako parametr nazwę zespołu i zwraca liczbę pracowników tego zespołu, został przedstawiony poniżej.

```
CallableStatement stmt = conn.prepareStatement("{? = call PoliczPrac(?)}");  
  
stmt.setString(2, "ALGORYTMY");  
stmt.registerOutParameter(1, Types.INTEGER);  
  
stmt.execute();  
int vLiczbaPracownikow = stmt.getInt(1);  
stmt.close();
```

### Zadanie 7

Przenieś operację zmiany wielkości liter w nazwisku danego pracownika do funkcji składowanej w bazie danych. Funkcja ma mieć dwa parametry:

- identyfikator pracownika, który będzie podlegał zmianom (parametr wejściowy)
- nazwisko pracownika po dokonaniu zmian (parametr wyjściowy).

Wartością zwrótną funkcji ma być wartość 1 w przypadku pomyślnej zmiany oraz 0 w sytuacji, gdy przekazany do funkcji identyfikator pracownika nie jest poprawny. Napisz program, który wykorzysta zbudowaną przez Ciebie funkcję.