# Notatki - Database intro

## Spis treści



# Czym są bazy danych? Co to i po co to?

Strzelając na początku definicją z Wikipedii (angielskiej):

> In computing, a database is an organized collection of data stored and accessed electronically from a computer system.

Czyli baza danych jest takim tworem/programem/systemem, który pozwala nam przechowywać dane. Możemy odczytywać te dane, zapisywać, edytować oraz usuwać.

Dzięki tym operacjom powstał taki skrót w programowaniu jak **CRUD** - **create**, **read**, **update**, **delete**.

Dane w bazach danych są w pewien sposób zorganizowane, mogą być podzielone na kolumny i wiersze, wiersze mogą mieć swoje indeksy, wszystko to po to aby łatwiej móc takimi danymi zarządzać. Oczywiście nie jest to jedyny możliwy sposób organizacji danych, ale na początek będziemy rozmawiać o organizacji w wiersze i kolumny.

## DBMS

Skrót **DBMS** rozwija się jako `Database Management System` - czyli oprogramowanie, które służy jako interface pomiędzy użytkownikiem a bazą danych. Inaczej mówiąc jest to oprogramowanie, które umożliwia użytkownikom definiowanie, tworzenie, utrzymywanie i kontrolowanie dostępu do bazy danych. Nazwa takiego systemu świadczy też o tym o jakim serwerze bazy danych rozmawiamy. System zarządzania baz danych pozwala nam tworzyć konkretne bazy danych oraz na nich operować.

W dalszej części wypisane zostały różne systemy zarządzania bazami danych.

## Database server

Jeżeli chcemy być bardzo dokładni to wyróżnimy też stwierdzenie serwera bazy danych. Zanim natomiast przejdziemy do wyjaśnienia tego terminu, powiedzmy sobie czym w ogóle jest serwer. A jego definicja jest taka (Oczywiście angielska Wikipedia 😀).

> In computing, a server is a piece of computer hardware or software (computer program) that provides functionality for other programs or devices, called "clients".

Czyli skoro serwer służy do pełnienia jakiejś funkcji dla innych programów lub urządzeń, to serwer bazodanowy jest takim serwerem, który dostarcza nam funkcjonalność bazy danych. Dlaczego tak mieszam?

Nazwa konkretnego **DBMS** świadczy też o tym o jakim serwerze bazy danych rozmawiamy - pomimo, że teoretycznie **DBMS** i serwer bazy danych to są inne pojęcia. Ale jak w praktyce usłyszysz nazwy takie jak np. **PostgreSQL**, **MySQL** lub **Oracle Database** to będą one często oznaczały serwer baz danych i system zarządzania bazami danych.

# A czym są te dane?

Imiona, nazwiska, produkty zakupione w internecie, ceny produktów, historie zakupów, daty zakupów, ilość zakupionych produktów i tak dalej... To wszystko to są dane. Jeżeli zapisywalibyśmy takie dane na kartkach, w pewnym momencie zacząłby być potrzebny jakiś system organizacji takich informacji, bo chyba ciężko by było zapisywać te dane na kartkach, a następnie przeszukiwać przykładowo 800 kartek w poszukiwaniu jakiejś informacji.

Naturalnie wypłynęłaby wtedy kwestia takiej organizacji kartek, żeby móc pewne informacje szybko znaleźć. Podzielilibyśmy te kartki na grupy, trzymali w koszulkach, w segregatorach, koszulki byłyby oznaczone karteczkami samoprzylepnymi, moglibyśmy posortować grupy alfabetycznie i tak dalej.

# Relacyjne bazy danych

Na potrzeby tego warsztatu skupimy się tylko na relacyjnych bazach danych (ze względu na to jak często są spotykane w praktyce). Relacyjna baza danych (**Relational database**) prezentuje dane w strukturze tabel. Każda tabela składa się z wierszy oraz kolumn. Możesz to sobie wyobrazić w formie pliku w excelu. Możemy w nim stworzyć tabelki, które będą miały określoną ilość kolumn, do tego będziemy wypełniać te tabelki kolejnymi wierszami, w których będą znajdowały się nasze dane.

Oprócz baz relacyjnych występują też nierelacyjne bazy danych, które przechowują dane w innych strukturach niż tabelki, natomiast na ten moment nie poruszamy tego tematu. Takie rodzaje baz danych często nazywane są NoSQL.

# A co z tym JDBC?

**Java Database Connectivity** - bo takie jest rozwinięcie tego skrótu. JDBC to API, które zapewnia nam podstawowe klasy i metody, pozwalające na podłączenie się do bazy danych przy wykorzystaniu Javy.

# Jeszcze jakieś ważne skróty?

**SQL** - **Structured Query Language** - język zapytań pozwalający na zapytanie bazy o dane, które nas interesują. Pozwala również na dodawanie, modyfikowanie oraz usuwanie danych. Będziemy go używać w kolejnych przykładach.

**JPA** - **Java Persistence API** - zanim wyjaśnię czym jest JPA, to zaczniemy od czegoś innego. W praktyce bardzo często nie używa się JDBC, tylko warstwę abstrakcji wyżej. Na czym polega ta warstwa wyżej? W skrócie chodzi o to, że JDBC jest mechanizmem, który pozwala nam komunikować się z bazami danych. Natomiast dużo rzeczy musi wtedy być robione ręcznie (zobaczysz niedługo). Aby to ułatwić/obejść/przyspieszyć powstało coś takiego jak JPA, o którym będziemy rozmawiać później. Natomiast JDBC trzeba poruszyć, żeby zrozumieć w jaki sposób Java komunikuje się z bazami danych. Pierwszym przykładem, który będzie dosyć denerwujący jest to jak dane z tabelki przemapować na obiekty.

**RDBMS** - **Relational Database Management System** - do poprzedniego skrótu DBMS dołożyliśmy jeszcze kwestię relacyjności wynikającej z koncepcji relacyjnych baz danych i w ten sposób powstał ten skrót.

# Popularne DBMS

Na rynku dostępnych jest dużo popularnych DMBS. Jak zaczniesz szukać w Internecie to często będą pojawiały się takie nazwy jak PostgreSQL, MySQL, Oracle Database lub MariaDB. Podałem nazwy 4 różnych produktów, ale nie przejmuj się, nie musisz znać wszystkich od podszewki. Założenie jest takie, że bazy SQL mają wspierać język SQL, a my jako programiści powinniśmy umieć się SQLem posługiwać, żeby móc z tymi bazami gadać. Jeżeli potrafimy operować językiem zapytań baz danych (SQL) to z każdą z nich powinniśmy być w stanie porozmawiać ☺. Jednakże trzeba pamiętać, że w niektórych przypadkach, składnia tego języka w zależności od tego z której bazy danych korzystamy może się nieznacznie różnić, ale to zaczniesz zauważać już w praktyce.

# Jak przechowywane są dane

W tabelach. Wydaje mi się, że ten sposób dla człowieka jest dosyć naturalny. W przypadku baz danych, o których będziemy tutaj rozmawiać, dane są przechowywane w formie tabelarycznej. Przykładowo, jak może wyglądać tabela USERS reprezentująca użytkowników jakiegoś systemu:

| Id | Name | Surname | City |
|---|---|---|---|
| 1 | Stefan | Romański | Warszawa |
| 2 | Roman | Stefański | Szczecin |
| 3 | Aleksandra | Iksińska | Zakopane |
| 4 | Stefania | Stefańska | Wrocław |
| 5 | Aleksander | Romanowski | Poznań |
| 6 | Anna | Zajavkowa | Gdynia |

## Kolumna

Kolumna zawiera w sobie informacje określonego typu i jej przeznaczeniem jest przetrzymywanie tylko danego typu informacji. Oznacza to, że w kolumnie `City` nie będziemy wpisywać nazwisk i odwrotnie.

## Wiersz

Wiersz jest pojedynczym wpisem w tabeli. W tabeli powyżej mamy 6 wierszy, gdzie każdy z nich reprezentuje inny wpis. Oznacza to, że każdy wiersz reprezentuje inną osobę. Wiersz może być też określany słowem **wpis**. W angielskim możemy spotkać takie słowo jak **record** albo **entry**. Tak jak w Mapach, pamiętasz `Map.Entry`?

## Wartości w wierszach mogą być NULL

Tak samo jak referencja może nie wskazywać na żaden obiekt i wtedy mamy w Javie zapis:

```
String value = null;
```

Tak samo w przypadku danych w bazach danych, informacja w danej kolumnie może być `NULL`, czyli brak informacji. Tak jak w przypadku Javy, należy pamiętać, że `null` nie oznacza pustego Stringa tylko brak informacji - tak samo jest tutaj. Jeżeli w wierszu, w kolumnie numerycznej (bo kolumny mają określone typy danych, do których przejdziemy) określimy, że spodziewamy się danych numerycznych, to `null` oznacza brak danych, a jakąś wartość domyślną, np. `0`.

# Integralność danych

Możemy poszukać definicji w Internetach "What is data integrity" i cytując angielską Wikipedię:

> Data integrity is the maintenance of, and the assurance of, data accuracy and consistency over its entire life-cycle and is a critical aspect to the design, implementation, and usage of any system that stores, processes, or retrieves data.

Co to oznacza w praktyce? Powstały pewne zasady, które znajdziesz poniżej, które pomagają nam utrzymać porządek w naszych danych. Przestrzegając tych zasad dążymy do tego, żeby nasze dane były kompletne i spójne, co jest krytycznym aspektem w przypadku przechowywania jakichkolwiek danych.

Bo wyobraźmy sobie, że organizujemy sobie dane wypisując je na karteczkach w segregatorach. Przykładowo nie miałoby żadnego sensu posiadanie 5 identycznych karteczek z identyczną zapisaną informacją. Albo drugi przykład, jeżeli karteczka `A` mówiłaby, że jakaś informacja jest zapisana na karteczce `B`, a karteczka `B` wylądowałaby w koszu, to nasze dane są niekompletne. Trzymając się zasad wypisanych poniżej dążymy do zachowania integralności naszych danych.

# Integralność encji (Entity integrity)

W jednej tabeli nie możemy mieć dwóch identycznych wierszy. Czyli, nie możemy mieć takiej sytuacji:

| Id | Name | Surname | City |
|---|---|---|---|
| 1 | Stefan | Romański | Warszawa |
| 1 | Stefan | Romański | Warszawa |

Ale taką już jak najbardziej:

| Id | Name | Surname | City |
|---|---|---|---|
| 1 | Stefan | Romański | Warszawa |
| 2 | Stefan | Romański | Warszawa |

Zwróć uwagę, że w drugim przypadku są to już inne wiersze, bo w kolumnie `Id` wartości są różne.

# Integralność domeny (Domain integrity)

Kolumny mają określone typy, tak samo jak w Javie, żeby `Integer` nie przetrzymywał wartości `String` itp. Dzięki temu narzucamy jaki typ danych może być przechowywany w danej kolumnie.

# Więzy integralności (Referential integrity)

Tabele mogą mieć wzajemne relacje (dlatego nazywa się to bazami relacyjnymi), to znaczy, że wartości w kolumnie mogą odnosić się do wartości w innych tabelach i być z nimi powiązane. Jeszcze sobie o tym później powiemy. Natomiast kwestia polega na tym, że przykładowo nie możemy usunąć z tabeli wiersza, jeżeli wartości w tym wierszu odnoszą się do wpisów w innych tabelach, które nie zostały usunięte.

# Integralność narzucona przez użytkownika (User-defined integrity)

Tutaj dodam, że chodzi o reguły narzucane przez osobę konfigurującą tabele w bazach danych, a nie o użytkownika końcowego, czyli np. klienta banku. Oprócz tego, że możemy powiedzieć, że dane mają być typu tekstowego, to możemy narzucić wymóg, że w danej kolumnie możemy zapisać tylko wyrazy o długości 10 znaków.

# Constraints

Czyli przymus, ograniczenie? Nie wiem jak to przetłumaczyć sensownie na polski, bo w praktyce mówi się o konstrejntach (chyba dobrze napisałem ☺).

Definiując kolumny w bazie danych mamy możliwość narzucenia pewnych reguł, które muszą zostać spełnione przez dane, które chcemy w danej tabeli zapisać, aby w ogóle móc je zapisać.

Constrainty mogą być zarówno na poziomie kolumny jak i całej tabeli. Poniżej wypiszemy sobie najczęściej używane:

- **NOT NULL** - constraint mówiący, że w danej kolumnie nie możemy zapisać `NULL`.
- **UNIQUE** - constraint mówiący, że w danej kolumnie dane nie mogą w żaden sposób się powtarzać, muszą być unikalne. Jeżeli dodalibyśmy `UNIQUE` constraint na kolumnie z imionami (patrz tabele wyżej), to nie moglibyśmy w tej samej tabeli zapisać dwóch `Zbyszków`, bo moglibyśmy mieć w kolumnie z imionami tylko jedno imię `Zbyszek`.
- **DEFAULT** - constraint narzucający wartość domyślną, jeżeli sami takiej nie podaliśmy. Czyli jeżeli podczas zapisywania danych w bazie, w konkretnej kolumnie, w której ustawiono wartość `DEFAULT` nie określimy wartości to zapisana zostanie ta `DEFAULT`.
- **CHECK** - constraint sprawdzający czy dane, które chcemy zapisać w danej kolumnie są zgodne z określonym przez nas warunkiem. Czyli przykładowo, czy zapisany tekst jest w formacie `JSON` (wiem, że jeszcze o tym nie rozmawialiśmy, ale podaję to tylko jako przykład).
- **PRIMARY KEY** (Klucz główny) - constraint jednoznacznie identyfikujący unikalny rekord/wiersz w bazie danych, coś jak pesel. Będziemy o tym rozmawiać.
- **FOREIGN KEY** (Klucz obcy) - constraint jednoznacznie identyfikujący odniesienie do wpisu w innej tabeli. Będziemy o tym rozmawiać.

# Notatki - SQL podstawy - cz.1

## Spis treści



## Instalacja PostgreSQL

Zanim przejdziemy do zabawy językiem zapytań `SQL` musimy mieć gdzie się bawić. W tym celu zainstalujemy `PostgreSQL`.


*Obraz 1. PostgreSQL logo. Źródło: https://icon-library.com/*

Link do pliku instalacyjnego (Windows) Installer.

Pobieramy plik z rozszerzeniem `.exe` i instalujemy przechodząc przez kolejne kroki. Pamiętajmy, żeby podać hasło, które będziemy pamiętać, żeby potem nie bawić się w resetowanie ☺. `Locale` możemy ustawić na Polskę, wiemy już co to `Locale` jeżeli idziemy z kolejnością warsztatów. Reszta ustawień może pozostać domyślna, port również zostawiamy `5432`. Na końcu może pojawić się pytanie o `StackBuilder`, które odznaczamy i zrobione.

Teraz aby uruchomić panel do zarządzania należy poszukać aplikacji `pgAdmin`. U mnie jest to `pgAdmin 4`. Jeżeli `pgAdmin` nie chce się poprawnie uruchomić po instalacji można spróbować albo wykonać restart komputera albo uruchomić aplikację jako administrator.

`pgAdmin` może też poprosić o ustawienie **master password**. Z racji, że nasza baza danych nie będzie bazą produkcyjną dla 1mln klientów, możemy ustawić te same hasło co wcześniej przy instalacji. Trzeba też pamiętać, że to co zostanie pokazane w obrębie przedstawionych materiałów ma charakter edukacyjny, w praktyce w systemach produkcyjnych konfiguracja będzie inna i dostosowana do konkretnego przypadku.

# Własna baza danych

## Jak dostać się do serwera baz danych

Bardzo przyjemnie ogląda się to co teraz dzieje się na GUI `pgAdmina`, natomiast w praktyce często jest tak, że dużo operacji musi być wykonywane z konsoli. Wynika to z tego, że jeżeli chcemy podłączyć się do serwera bazodanowego, który zapewnia nam bazę danych dla naszej aplikacji, to serwer taki udostępnia przykładowo tylko dostęp z poziomu terminala, a nie daje nam dostępu przez GUI. Z tego powodu, uruchom proszę terminal ☺.

Na Windowsie można wpisać CMD w pasku wyszukiwania dostępnym pod ikoną 'Windows' na klawiaturze, albo z poziomu Intellij możemy poszukać Terminal na dole ekranu, lub użyć skrótu `ALT + F12`.

Będziemy korzystać z komendy `psql`, natomiast jeżeli wpiszesz teraz komendę `psql` to dostaniesz informację, że nie rozpoznano takiej komendy. Mamy 2 opcje, albo dodajemy ścieżkę z plikiem `psql.exe` do zmiennej `Path`, albo znajdujemy folder, w którym mamy zainstalowany `PostgreSQL` i znajdujemy tam folder `bin`. Następnie w terminalu wykonujemy komendę (lokalizacja jest przykładowa, Twoja może się różnić):

```
cd C:/Program Files/PostgreSQL/<nr_wersji>/bin
```

Mamy teraz dostęp do komendy `psql`, wykonajmy zatem komendę:

```
psql -U postgres
```

Oznacza ona próbę połączenia się z serwerem baz danych `PostgreSQL` przy wykorzystaniu użytkownika `postgtres`. Ustawialiśmy hasło temu użytkownikowi na etapie instalacji `PostgreSQL`. Zostaniemy poproszeni o hasło podane podczas instalacji. Na ekranie pojawi się:

```
postgres=#
```

Zwróć uwagę, że znika ścieżka systemu Windows z której próbowaliśmy się dostać. Oznacza to, że udało nam się połączyć do serwera bazy danych. Aby wydostać się z serwera wystarczy napisać `quit`.

Możemy w tym momencie wylistować utworzonych użytkowników za pomocą komendy:

```
\du
```

I zobaczymy, że mamy dostępnego tylko użytkownika `postgres`. Na ten moment nam to wystarczy.

Możemy teraz wylistować dostępne bazy danych za pomocą komendy:

```
\l
```

Aby pobrać trochę więcej informacji, możemy wpisać:

```
\l+
```

Widzimy teraz domyślnie utworzoną bazę danych `postgres`, ale zaraz będziemy tworzyć swoją. Aby zobaczyć dostępne bazy danych, możemy równie dobrze napisać:

```
SELECT datname FROM pg_database;
```

# Tworzenie własnej bazy danych

Wystarczy, że wpiszemy komendę:

```
CREATE DATABASE zajavka;
```

Możemy sprawdzić, czy udało nam się utworzyć bazę danych stosując ponownie komendę:

```
\l+
```

Taka baza danych będzie też teraz widoczna w `pgAdminie`. Na ikonie `servers` klikamy prawym przyciskiem myszki, wybieramy **Refresh** i po rozwinięciu **Databases** zobaczymy poprzednią bazę danych oraz bazę danych `zajavka`.

Jeżeli na którymś etapie będziemy szukać odpowiedzi na nurtujące nas kwestie, polecam dokumentację. Możemy tutaj znaleźć opis komend, które stosujemy, przykładowo komenda CREATE DATABASE.

Wykorzystaliśmy najprostszy wariant komendy, gdzie bardzo duża ilość ustawień została wybrana domyślnie, ale na ten moment nie przejmujmy się tym. Prawdopodobnie zaczniesz się tego uczyć dopiero jak będzie Ci to potrzebne w praktyce w pracy. Nikt też nie uczy się tych wszystkich opcji na pamięć (przynajmniej ja nie znam), od tego jest dokumentacja, także uspokajam 😉.

# Usuwanie bazy danych

Taką bazę danych możemy też usunąć stosując komendę DROP DATABASE:

```
DROP DATABASE zajavka;
```

Jeżeli otrzymasz informację zwrotną, mówiącą, że jeszcze ktoś korzysta z tej bazy danych, sprawdź czy nie robi tego przypadkiem `pgAdmin`.

# Notatki - SQL podstawy - cz.2

## Spis treści



## SQL i podstawowa składnia

Przejdziemy do omówienia składni języka SQL. Na potrzeby przykładów będziemy używać PostgreSQL. Wspominam o tym z tego powodu, że wspomniałem wcześniej, że konkretne DBMS mogą różnić się pewnymi niuansami i tak samo typy danych jak i składania zapytań, które są obsługiwane przez konkretne DBMS mogą się nieznacznie różnić. Nie przejmuj się tym, w pracy jak mamy taką sytuację to zwyczajnie trzeba googlować ☺.

## Typy danych

Zacznijmy od tego jakiego rodzaju dane możemy zapisywać w konkretnych kolumnach w tabelach. Od razu też zaznaczę, że poruszone poniżej typy nie są wszystkimi możliwymi.

### Boolean

| nazwa | rozmiar | zakres | komentarz |
|---|---|---|---|
| boolean | 1 byte | false lub true | boolean może przetrzymywać wartości true lub false, w kolumnie może też być zapisany null |

### Typy numeryczne

| nazwa | rozmiar | zakres | komentarz |
|---|---|---|---|
| smallint | 2 bajty | -32768 do 32767 | integer o małym zakresie danych |
| integer | 4 bajty | -2147483648 do 2147483647 | najczęściej używany |

| nazwa | rozmiar | zakres | komentarz |
|---|---|---|---|
| bigint | 8 bajtów | -9223372036854775808 do 9223372036854775807 | integer o dużym zakresie danych |
| real | 4 bajty | Dokładność 6 cyfr dziesiętnych | Cytując dokumentację: "The data types real and double precision are inexact, variable-precision numeric types". Czyli nie używać tego typu do pieniędzy, gdyż jest on niedokładny. Pamiętasz double i float z Javy vs BigDecimal? |
| numeric(p,s) | zmienny | Do 131072 cyfr przed przecinkiem oraz do 16383 cyfr po przecinku | 'p' oznacza ilość cyfr, a 's' ilość miejsc po przecinku, czyli numeric(5,2) oznacza 5 cyfr, z czego 2 są po przecinku. Ten typ powinien być używany do kalkulacji pieniężnych |

## Typy tekstowe/znakowe

| nazwa | komentarz |
|---|---|
| char(n) | Ciąg znaków o długości określonej jako 'n'. Jeżeli wstawimy ciąg znaków, który jest krótszy niż 'n', reszta zawartości zostanie dopełniona spacjami. W przypadku próby zapisania dłuższego tekstu dostaniemy błąd |
| varchar(n) | Ciąg znaków o długości maksymalnej określonej jako 'n'. Jeżeli ciąg znaków jest krótszy niż 'n', nie zostanie on dopełniony spacjami. W przypadku próby zapisania dłuższego tekstu dostaniemy błąd |
| text | Tekst o zmiennej długości, teoretycznie może mieć nieograniczoną długość |

## Daty i czasy

| nazwa | komentarz |
|---|---|
| DATE | przechowuje tylko datę |
| TIME | przechowuje tylko czas |
| TIMESTAMP | przechowuje oba datę i czas |
| TIMESTAMP WITH TIME ZONE | przechowuje oba datę i czas z uwzględnieniem strefy czasowej |

# Tworzenie tabeli

Jeżeli stworzyliśmy już swoją bazę danych, możemy teraz stworzyć w niej tabelki. Zanim natomiast zaczniemy tworzyć tabelki musimy wiedzieć, co tak na prawdę chcemy stworzyć ☺. Załóżmy, że będziemy tworzyć aplikację, w której będziemy przetrzymywać informacje o wypłatach naszych pracowników razem z datą zatrudnienia pracownika. Potrzebna nam zatem będzie tabela w stylu:

| ID | NAME | SURNAME | AGE | SALARY | DATE_OF_EMPLOYMENT |
|---|---|---|---|---|---|
| 1 | Aleksander | Wypłata | 33 | 8791.12 | 2018-03-12 |
| 2 | Roman | Pomidorowy | 43 | 7612.12 | 2012-01-01 |
| 3 | Anna | Rosół | 38 | 5728.90 | 2015-07-18 |
| 4 | Urszula | Nowak | 39 | 3817.21 | 2014-12-15 |
| 5 | Stefan | Romański | 38 | 9201.23 | 2020-07-14 |
| 6 | Jolanta | Kowalska | 27 | 6521.22 | 2012-06-04 |

Zanim w ogóle będziemy mogli dodać dane do takiej tabeli to musimy ją stworzyć i określić jaką taka tabela może mieć strukturę. Pamiętasz, że powinniśmy określić jakiego typu dane mogą się znaleźć w każdej kolumnie? Albo, że możemy narzucić aby w danej kolumnie były tylko unikalne wartości?

Aby stworzyć strukturę tabeli musimy wykorzystać instrukcje **DDL** (**Data Definition Language**). W przykładzie, którego potrzebujemy, instrukcja taka może wyglądać w taki sposób:

```
CREATE TABLE EMPLOYEES(
    ID                  INT           NOT NULL,
    NAME                VARCHAR(20)   NOT NULL,
    SURNAME             VARCHAR(20)   NOT NULL,
    AGE                 INT,
    SALARY              NUMERIC(7, 2) NOT NULL,
    DATE_OF_EMPLOYMENT DATE,
    PRIMARY KEY (ID)
);
```

Pojawiło się stwierdzenie `PRIMARY KEY (ID)`. W tłumaczeniu na nasz jest to **klucz główny**, który jest 'zakładany' na kolumnie `ID`. Oznacza to, że wartość w tej kolumnie ma być unikalna dla każdego zapisywanego w tabeli wiersza. **Primary key** oznacza, że dzięki tej wartości możemy unikalnie identyfikować 2 wiersze, które mogą mieć dokładnie te same zawartości w innych kolumnach.

Teoretycznie są bazy danych, które pozwolą nam stworzyć tabelę bez klucza głównego, ale w praktyce tworzy się go zawsze, aby poprawnie odróżnić od siebie rekordy.

Możemy w tym momencie spróbować zobaczyć definicję nowo utworzonej tabeli, ale zanim jednak to zrobimy, upewnijmy się, że uruchomiliśmy/wskazaliśmy odpowiednią bazę danych:

```
\c zajavka
```

Teraz możemy spróbować odczytać informacje o utworzonym schemacie wykorzystując komendę:

```
\d EMPLOYEES;
```

Lub

```
\d+ EMPLOYEES;
```

Możemy również wykorzystać kwerendę SQL określaną jako `SELECT` (niedługo będziemy dużo tego pisać).

```
SELECT *
FROM information_schema.columns
WHERE table_name = 'EMPLOYEES'
   OR table_name = 'employees';
```

Zaznaczam, że na tym etapie w naszej tabelce nie ma jeszcze żadnych danych, jedynie stworzyliśmy schemat/ramkę, w której możemy te dane zapisać.

# Usuwanie tabeli

Skoro tabela została utworzona to możemy ją również usunąć ☺.

```
DROP TABLE EMPLOYEES;
```

# Edytowanie struktury tabeli

I skoro możemy tabelę zarówno utworzyć jak i usunąć to możemy również aktualizować jej definicję. Należy jednak pamiętać, że jeżeli w danej tabeli mamy już zapisane dane to trzeba uważać z aktualizacją jej schematu. Analogicznie do tabelki w excelu, jeżeli usuniemy całą kolumnę z danymi to dane z tej kolumny też stracimy.

Aby zmienić strukturę tabeli należy wykorzystać komendę `ALTER TABLE`:

```
ALTER TABLE EMPLOYEES
ALTER COLUMN SURNAME DROP NOT NULL;
```

Kod wyżej zmienia kolumnę `SURNAME` z `NOT NULL` na akceptującą `NULL`.

# Notatki - SQL podstawy - cz.3

# Spis treści



# SQL i podstawowa składnia

## Wkładanie danych do tabelek

Wkładanie zostało użyte od słowa INSERT bo to jest słówko kluczowe, które służy do wypełnienia tabel danymi.

```
INSERT INTO EMPLOYEES (ID, NAME, SURNAME, AGE, SALARY, DATE_OF_EMPLOYMENT)
    VALUES (1, 'Aleksander', 'Wypłata', 33, 8791.12, '2018-03-12');
INSERT INTO EMPLOYEES (ID, NAME, SURNAME, AGE, SALARY, DATE_OF_EMPLOYMENT)
    VALUES (2, 'Roman', 'Pomidorowy', 43, 7612.12, '2012-01-01');
INSERT INTO EMPLOYEES (ID, NAME, SURNAME, AGE, SALARY, DATE_OF_EMPLOYMENT)
    VALUES (3, 'Anna', 'Rosół', 38, 5728.90, '2015-07-18');
INSERT INTO EMPLOYEES (ID, NAME, SURNAME, AGE, SALARY, DATE_OF_EMPLOYMENT)
    VALUES (4, 'Urszula', 'Nowak', 39, 3817.21, '2014-12-15');
INSERT INTO EMPLOYEES (ID, NAME, SURNAME, AGE, SALARY, DATE_OF_EMPLOYMENT)
    VALUES (5, 'Stefan', 'Romański', 38, 9201.23, '2020-07-14');
INSERT INTO EMPLOYEES (ID, NAME, SURNAME, AGE, SALARY, DATE_OF_EMPLOYMENT)
    VALUES (6, 'Jolanta', 'Kowalska', 27, 6521.22, '2012-06-04');
```

Zwróć uwagę, że wpisujemy `INSERT INTO`, później podajemy nazwę tabelki, później w nawiasach podajemy nazwy kolumn, dla których będziemy 'wkładać' dane, a później dodajemy wartości jakie mają się znaleźć w konkretnych wypisanych kolumnach.

# Odczytywanie danych z tabel

Dopiero teraz jak mamy już tabelę uzupełnioną danymi to będzie miało jakikolwiek sens żeby próbować te dane odczytać.

```
SELECT * FROM EMPLOYEES;
```

- **SELECT** - tak jak nazwa mówi, komenda mówi o pobieraniu danych,
- ***** - gwiazda oznacza, że mamy pobrać dane ze wszystkich kolumn w tabeli. Możemy również określić jakie konkretnie kolumny mamy zwrócić w rezultacie oddzielając je przecinkiem,
- **FROM** - określa z jakiej tabeli będziemy pobierać dane,

Natomiast jeżeli chcemy wyświetlić dane z konkretnych kolumn, możemy napisać takie query:

```
SELECT
       ID,
       NAME,
       SURNAME
FROM EMPLOYEES;
```

## Aliasy

Możemy też przy tym nadać tym kolumnom aliasy. Oznacza to, że tylko w widoku, który wyświetlimy, kolumny te mogą nazywać się inaczej, ale nie zmienia to nic w samej tabeli.

```
SELECT
       ID AS MY_ID,
       NAME AS MY_NAME,
       SURNAME AS MY_SURNAME
FROM EMPLOYEES;
```

## Where

Teraz jest o tyle prosto, że mamy 6 wierszy w naszej tabelce, więc możemy pokazać wszystkie wiersze, ale w praktyce w bazach danych są przetrzymywane tysiące wierszy. Możemy zatem mieć potrzebę aby pobrać tylko dane osób, które nazywają się `Roman`.

```
SELECT *
FROM EMPLOYEES
WHERE NAME = 'Roman';
```

W ten sposób pobierzemy z bazy tylko rekordy, dla których imię ma wartość `Roman`. Zwróć uwagę, że mamy pojedyncze `=`, a nie `==`.

## Łączenie warunków

Warunki podane w `WHERE` możemy ze sobą łączyć za pomocą operatorów `AND` lub `OR`. W takim przypadku zadziała to tak jak z operatorami logicznymi w Javie. Przykładowo:

```
SELECT *
FROM EMPLOYEES
WHERE NAME = 'Roman' AND SURNAME = 'Pomidorowy';
```

Ewentualnie:

```
SELECT *
FROM EMPLOYEES
WHERE NAME = 'Roman' OR NAME = 'Anna';
```

## Operatory

W przypadku operatorów, poruszę również najczęściej używane, nie będą to wszystkie możliwe 😉.

### Operatory arytmetyczne

Przykładowe operatory arytmetyczne:

| Słownie | Operator | Opis |
|---|---|---|
| dodawanie | + | Dodaje do siebie wartości użyte z operatorem |
| odejmowanie | - | Odejmuje od siebie wartości użyte z operatorem |
| mnożenie | * | Mnoży przez siebie wartości użyte z operatorem |
| dzielenie | / | Dzieli przez siebie wartości użyte z operatorem |
| modulo | % | Dzieli lewy operand przez prawy i zwraca resztę z dzielenia |

Przykład użycia:

```
SELECT
       NAME,
       AGE % 10 AS AGE_MOD
FROM EMPLOYEES;
```

### Operatory porównania

Operatory te są podobne do tych, które poznaliśmy już w samej Javie. Możemy stosować te operatory przy określaniu jakie warunki mają spełniać dane, które chcemy `SELECTować`. Możemy ich używać przykładowo w warunku `WHERE`. Pamiętajmy, że wynikiem operatorów poniżej jest wartość `true`/`false`.

| Operator | Opis | Przykład |
|---|---|---|
| = | Operator porównania, sprawdza, czy operandy po obu stronach wyrażenia są sobie równe | SELECT * FROM EMPLOYEES WHERE NAME = 'Roman'; |
| != | Operator nierówności, sprawdza czy operandy po obu stronach wyrażenia są sobie nierówne | SELECT * FROM EMPLOYEES WHERE NAME != 'Roman'; |
| <> | Operator różności, sprawdza czy operandy po obu stronach wyrażenia są różne, czyli w sumie to samo co poprzedni operator | SELECT * FROM EMPLOYEES WHERE NAME <> 'Roman'; |
| < | Operator mniejszości, sprawdza czy lewa część wyrażenia jest mniejsza niż prawa | SELECT * FROM EMPLOYEES WHERE SALARY < 5000; |
| > | Operator większości, sprawdza czy lewa część wyrażenia jest większa niż prawa | SELECT * FROM EMPLOYEES WHERE SALARY > 5000; |
| ⇐ | Operator mniejsze-równe, sprawdza czy lewa część wyrażenia jest mniejsza lub równa prawej | SELECT * FROM EMPLOYEES WHERE SALARY ⇐ 5000; |
| >= | Operator większe-równe, sprawdza czy lewa część wyrażenia jest większa lub równa prawej | SELECT * FROM EMPLOYEES WHERE SALARY >= 5000; |

### Operatory logiczne

Poznaliśmy już wcześniej operatory logiczne jako `&&` lub `||`. Tutaj mamy trochę więcej możliwości niż w samej Javie. Wcześniej w notatce wspomniane zostały już 2 operatory `OR` oraz `AND`. Pamiętajmy, że wynikiem operatorów poniżej jest wartość `true`/`false`.

| Operator | Opis | Przykład |
|---|---|---|
| OR | Operator LUB (alternatywa), używany przykładowo w klauzuli WHERE | SELECT * FROM EMPLOYEES WHERE NAME = 'Roman' OR NAME = 'Agnieszka'; |
| AND | Operator I (koniunkcja), używany przykładowo w klauzuli WHERE | SELECT * FROM EMPLOYEES WHERE NAME = 'Roman' AND SURNAME = 'Romański'; |
| IN | Operator sprawdzający, czy wartość w kolumnie jest równa jednej z podanych wartości. Działanie w przykładzie jest analogiczne do przykładu w operatorze OR | SELECT * FROM EMPLOYEES WHERE NAME IN ('Roman', 'Agnieszka'); |
| LIKE | Operator działający podobnie do String.contains() | SELECT * FROM EMPLOYEES WHERE NAME LIKE '%Ro'; |
| BETWEEN | Operator sprawdzający, czy wartość w kolumnie jest zawarta w przedziale podanym przy operatorze | SELECT * FROM EMPLOYEES WHERE AGE BETWEEN 20 AND 30; |
| IS NULL | Operator sprawdzający, czy wartość w kolumnie jest NULL | SELECT * FROM EMPLOYEES WHERE AGE IS NULL; |

| Operator | Opis | Przykład |
|---|---|---|
| NOT | Operator odwracający znaczenie innych operatorów | SELECT * FROM EMPLOYEES WHERE NAME NOT IN ('Roman', 'Agnieszka'); |

### LIKE - Lubię to

Operator `LIKE` specjalnie wyciągam pod oddzielny fragment ze względu na to, że jest często używany.

`LIKE` działa podobnie do `String.contains()`, ale należy przy tym pamiętać o znaku charakterystycznym `%`. Oznacza on brak znaku albo jeden lub więcej dowolnych znaków. Przykładowo:

Znajdź rekordy, gdzie imię zaczyna się od dowolnych znaków ale kończy się znakami `RO`:

```
SELECT *
FROM EMPLOYEES
WHERE NAME LIKE '%Ro';
```

Znajdź rekordy, gdzie imię zaczyna się od `Ro`, ale kończy się dowolnymi znakami:

```
SELECT *
FROM EMPLOYEES
WHERE NAME LIKE 'Ro%';
```

Znajdź rekordy, gdzie imię ma w środku `Ro`, może zaczynać i kończyć się dowolnymi znakami. Inaczej mówiąc, dopiero ten zapis odzwierciedla metodę `String.contains()`:

```
SELECT *
FROM EMPLOYEES
WHERE NAME LIKE '%Ro%';
```

## Sortowanie zwracanego wyniku

Wynik zwracany możemy posortować po konkretnej kolumnie, albo nawet po kilku.

```
SELECT *
FROM EMPLOYEES
ORDER BY AGE DESC;
```

Powyższe zapytanie zwróci nam rekordy z tabeli `EMPLOYEES` posortowane po wieku malejąco. Jeżeli chcielibyśmy posortować te wiersze rosnąco, to albo zamiast `DESC` możemy napisać `ASC`, albo napisać to tak:

```
SELECT *
FROM EMPLOYEES
ORDER BY AGE;
```

Domyślnie sortowanie odbywa się rosnąco, dlatego nie ma potrzeby pisać `ASC`.

Możemy również posortować wynik po kilku kolumnach w kolejności:

```
SELECT *
FROM EMPLOYEES
ORDER BY SALARY DESC, AGE ASC;
```

# Ograniczenie ilości zwracanych wierszy

W PostgreSQL do tego służy słówko kluczowe `LIMIT`. Wspominam tutaj o PostgreSQL, bo inne bazy mogą mieć to zrealizowane w inny sposób. Poniższe zapytanie zwróci nam tylko 2 wiersze posortowane domyślnie.

```
SELECT *
FROM EMPLOYEES
LIMIT 2;
```

Natomiast jeżeli interesowałoby nas zwrócenie 5 najmłodszych pracowników, moglibyśmy napisać to tak:

```
SELECT *
FROM EMPLOYEES
ORDER BY AGE ASC
LIMIT 5;
```

# Zwrócenie tylko unikalnych wartości

Wyobraźmy sobie, że potrzebujemy zwrócić tylko unikalne wartości jakie występują w danej kolumnie. Przykładowo chcemy się dowiedzieć jakie imiona ludzi występują wśród pracowników naszej firmy. Do tego służy słówko `DISTINCT`:

```
SELECT DISTINCT NAME
FROM EMPLOYEES;
```

# Grupowanie

Zanim poruszymy grupowanie to musimy wspomnieć o funkcjach agregujących. Jest to nic innego jak funkcja która z kliku elementów w jakiś sposób zwróci jakąś jedną wartość. Przykładowo może być to wartość maksymalna, minimalna, suma wartości, średnia itp.

Poruszymy takie funkcje agregujące:

| Funkcja | Działanie |
|---|---|
| COUNT | Zlicza ilość elementów w zbiorze |

| Funkcja | Działanie |
|---|---|
| SUM | Sumuje wartości elementów w zbiorze |
| AVG | Wylicza średnią wartość elementów w zbiorze |
| MIN | Określa wartość minimalną dla elementów w zbiorze |
| MAX | Określa wartość maksymalną dla elementów w zbiorze |

Funkcje powyżej mogą być wykonywane bez klauzuli `GROUP BY`, która jest poruszana poniżej, przykładowo:

```
SELECT
    COUNT(AGE),
    SUM(AGE),
    AVG(AGE),
    MIN(AGE),
    MAX(AGE)
FROM EMPLOYEES;
```

Znając już funkcje agregujące możemy przejść do klauzuli `GROUP BY`. Pamiętasz ze Streamów w programowaniu funkcyjnym, że mieliśmy możliwość pogrupowania obiektów po jakiejś wartości i otrzymywaliśmy wtedy mapę `klucz:lista_wartości`? Tutaj jest podobnie. Wyobraźmy sobie, że chcemy pogrupować rekordy po wieku. Otrzymalibyśmy wtedy mapę `wiek:lista_ludzi_w_tym_wieku`. Natomiast z racji, że przedstawiamy dane w tabelce, to musimy taki zapis wepchnąć do jednego wiersza. Przykładowo zapytanie poniżej nie zostanie wykonane poprawnie. Musimy określić funkcję agregującą te listy ludzi dla danego wieku.

```
SELECT *
FROM EMPLOYEES
GROUP BY AGE;
```

Może pojawić się teraz pytanie, czy możliwe jest ominięcie tej agregacji i przedstawienie w tabeli mapy, która została wspomniana w taki sposób, żeby było widać całą listę dla klucza, tak jak poniżej:

| AGE | NAME |
|---|---|
| 33 | Aleksander, Roman, Stefan |
| 28 | Agnieszka, Karol, Michał |
| 34 | Anna, Urszula, Jolanta |

Od razu odpowiadam, jest to możliwe, ale o wiele trudniejsze niż poziom, którego uczymy się teraz. Dlatego skupiamy się na funkcjach agregujących.

Zapytanie poniżej zliczy nam ile jest osób w każdym wieku. Najpierw grupujemy osoby w danym wieku `GROUP BY`, dostajemy wtedy mapę `wiek:lista_ludzi_w_tym_wieku`. Następnie wykorzystujemy funkcję `COUNT`, aby zliczyć rozmiary tych list i przedstawić mapę `wiek:ilość_ludzi_w_tym_wieku` w formie tabelki. Możemy w tym celu wykorzystywać również inne funkcje agregujące.

```
SELECT AGE, COUNT(AGE)
```

```
FROM EMPLOYEES
GROUP BY AGE;
```

# UPDATE rekordu w bazie

Rekordy w bazie danych mogły być tworzone od zera, ale bardzo często zdarzy się, że taki rekord będziemy musieli zaktualizować. Przykładowo możemy napisać, żeby od dzisiaj wszystkie `Anny` w naszej firmie zarabiały `10000` pieniędzy.

```
UPDATE EMPLOYEES
SET SALARY = 10000
WHERE NAME = 'Anna';
```

Jak widzisz używamy słowa kluczowego `UPDATE`, a następnie określamy jakie pola chcemy zaktualizować. Ważne też jest aby pamiętać o klauzuli `WHERE` inaczej zaktualizujemy wypłatę dla wszystkich pracowników.

A co jeżeli chcielibyśmy zaktualizować jednocześnie dane w kilku kolumnach? Niech każdy `Roman` ma na nazwisko `Zajavkowy` i ma `20` lat.

```
UPDATE EMPLOYEES
SET SURNAME = 'Zajavkowy', AGE = 20
WHERE NAME = 'Roman';
```

# DELETE rekordu w bazie

Dane możemy również z bazy usuwać. Należy jednak pamiętać ponownie, aby nie skasować danych z całej tabeli jednocześnie. Jeżeli pominiemy klauzulę `WHERE`, usuniemy wszystkie dane z tabeli.

```
DELETE
FROM EMPLOYEES
WHERE ID = 5;
```

# Notatki - SQL - Relacje

# Spis treści



# Relacje między tabelami

Wspomniałem wcześniej, że mówimy o relacyjnych bazach danych. No dobrze, to gdzie te relacje?

Tabele mogą zawierać wzajemne odnośniki do siebie. Przedstawię to obrazowo w taki sposób. Wyobraźmy sobie, że każdy pracownik ma zapisany w bazie danych adres, ale z jakiegoś powodu jest to tylko miasto i ulica. Nie mamy zapisanego numeru bloku ani mieszkania. Moglibyśmy zapisać to w naszej tabelce w taki sposób:

| ID | NAME | SURNAME | AGE | SALARY | DATE_OF_EMPLOYMENT | CITY | STREET |
|---|---|---|---|---|---|---|---|
| 1 | Aleksander | Wypłata | 33 | 8791.12 | 2018-03-12 | Warszawa | Marszałkowska |
| 2 | Roman | Pomidorowy | 43 | 7612.12 | 2012-01-01 | Gdańsk | Oliwska |
| 3 | Anna | Rosół | 38 | 5728.90 | 2015-07-18 | Warszawa | Marszałkowska |
| 4 | Urszula | Nowak | 39 | 3817.21 | 2014-12-15 | Gdańsk | Oliwska |
| 5 | Stefan | Romański | 38 | 9201.23 | 2020-07-14 | Warszawa | Marszałkowska |
| 6 | Jolanta | Kowalska | 27 | 6521.22 | 2012-06-04 | Szczecin | Biała |

W przypadku gdy jakiś rodzaj danych zaczyna się bardzo często powtarzać, nie ma sensu zapisywać go "na płasko" - czyli wszystko do jednej tabeli. Druga kwestia to 'podział obowiązków', tabela powinna być odpowiedzialna za jedną i tylko jedną rzecz, a nie za przetrzymywanie wszystkich możliwych informacji o naszych pracownikach. W tym celu wykorzystuje się relacje. Sytuację wyżej można przedstawić w ten sposób:

Tabela EMPLOYEES

| ID | NAME | SURNAME | AGE | SALARY | DATE_OF_EMPLOYMENT | ADDRESS_ID |
|---|---|---|---|---|---|---|
| 1 | Aleksander | Wypłata | 33 | 8791.12 | 2018-03-12 | 1 |
| 2 | Roman | Pomidorowy | 43 | 7612.12 | 2012-01-01 | 2 |
| 3 | Anna | Rosół | 38 | 5728.90 | 2015-07-18 | 1 |
| 4 | Urszula | Nowak | 39 | 3817.21 | 2014-12-15 | 2 |
| 5 | Stefan | Romański | 38 | 9201.23 | 2020-07-14 | 1 |
| 6 | Jolanta | Kowalska | 27 | 6521.22 | 2012-06-04 | 3 |

Tabela ADDRESSES

| ID | CITY | STREET |
|---|---|---|
| 1 | Warszawa | Marszałkowska |
| 2 | Gdańsk | Oliwska |
| 3 | Szczecin | Biała |

W ten sposób możemy stworzyć relację między tabelami i między rekordami, wiedząc, że jeżeli `ADDRESS_ID` ma wartość `3`, to musimy szukać rekordu w tabeli `ADDRESS` pod `ID = 3`.

# Stwórzmy teraz takie tabele w praktyce

```
CREATE TABLE ADDRESSES(
    ID     INT         NOT NULL,
    CITY   VARCHAR(32) NOT NULL,
    STREET VARCHAR(64) NOT NULL,
    PRIMARY KEY (ID)
);

CREATE TABLE EMPLOYEES(
    ID                 INT           NOT NULL,
    NAME               VARCHAR(20)   NOT NULL,
    SURNAME            VARCHAR(20)   NOT NULL,
    AGE                INT,
    SALARY             NUMERIC(7, 2) NOT NULL,
    DATE_OF_EMPLOYMENT DATE,
    ADDRESS_ID         INT           NOT NULL,
    PRIMARY KEY (ID),
    CONSTRAINT fk_address
        FOREIGN KEY (ADDRESS_ID)
            REFERENCES ADDRESSES (ID)
);
```

Dodając dopisek:

```
CONSTRAINT fk_address
    FOREIGN KEY (ADDRESS_ID)
        REFERENCES ADDRESSES (ID)
```

Dodajemy FOREIGN KEY (**klucz obcy**) w tabeli EMPLOYEES. Klucz ten mówi, że kolumna ADDRESS_ID ma wskazywać na PRIMARY KEY (**klucz główny**) z tabeli ADDRESSES. Klucz obcy jest oznaczeniem połączenia między tabelami, który oznacza, że dane z kolumny ADDRESS_ID z tabeli EMPLOYEES wskazują na dane z kolumny ID w tabeli ADDRESSES. Nazwa `fk_address` oznacza nazwę klucza obcego (**fk - foreign key**), który jest jednoznacznym identyfikatorem połączenia między tabelami EMPLOYEES i ADDRESSES. Oznacza to, że nie możemy w swojej bazie danych mieć innego klucza obcego o tej samej nazwie (fk_address).

Po stworzeniu tabel w sposób pokazany wyżej należy pamiętać o kilku kwestiach.

# Uwaga 1

ADDRESS_ID w tabeli EMPLOYEES jest NOT NULL. Oznacza to, że nie dodamy rekordu do tabeli EMPLOYEES nie mając odnośnika do danych w tabeli ADDRESSES. Czyli najpierw musimy wstawić rekord do tabeli ADDRESSES, a dopiero mając ID rekordu w tabeli ADDRESSES możemy wstawiać dane do tabeli EMPLOYEES. Mówiąc kodem:

```
INSERT INTO ADDRESSES (ID, CITY, STREET) VALUES (1, 'Warszawa', 'Marszałkowska');
INSERT INTO ADDRESSES (ID, CITY, STREET) VALUES (2, 'Gdańsk', 'Oliwska');
INSERT INTO ADDRESSES (ID, CITY, STREET) VALUES (3, 'Szczecin', 'Biała');

INSERT INTO EMPLOYEES (ID, NAME, SURNAME, AGE, SALARY, DATE_OF_EMPLOYMENT, ADDRESS_ID)
    VALUES (1, 'Aleksander', 'Wypłata', 33, 8791.12, '2018-03-12', 1);
INSERT INTO EMPLOYEES (ID, NAME, SURNAME, AGE, SALARY, DATE_OF_EMPLOYMENT, ADDRESS_ID)
    VALUES (2, 'Roman', 'Pomidorowy', 43, 7612.12, '2012-01-01', 2);
INSERT INTO EMPLOYEES (ID, NAME, SURNAME, AGE, SALARY, DATE_OF_EMPLOYMENT, ADDRESS_ID)
    VALUES (3, 'Anna', 'Rosół', 38, 5728.90, '2015-07-18', 1);
INSERT INTO EMPLOYEES (ID, NAME, SURNAME, AGE, SALARY, DATE_OF_EMPLOYMENT, ADDRESS_ID)
    VALUES (4, 'Urszula', 'Nowak', 39, 3817.21, '2014-12-15', 2);
INSERT INTO EMPLOYEES (ID, NAME, SURNAME, AGE, SALARY, DATE_OF_EMPLOYMENT, ADDRESS_ID)
    VALUES (5, 'Stefan', 'Romański', 38, 9201.23, '2020-07-14', 1);
INSERT INTO EMPLOYEES (ID, NAME, SURNAME, AGE, SALARY, DATE_OF_EMPLOYMENT, ADDRESS_ID)
    VALUES (6, 'Jolanta', 'Kowalska', 27, 6521.22, '2012-06-04', 3);
```

Nie bylibyśmy w stanie dodać danych w odwrotnej kolejności ze względu na constraint NOT NULL na kolumnie ADDRESS_ID w tabeli EMPLOYEES.

# Uwaga 2

Można w tym momencie spokojnie odpytywać o dane z tych tabel niezależnie, połączenie jest potrzebne wtedy gdy chcemy zobaczyć dane w 'sklejonym' widoku, czyli w takim. Poruszymy tę tematykę za moment

| ID | NAME | SURNAME | AGE | SALARY | DATE_OF_EMPLOYMENT | CITY | STREET |
|---|---|---|---|---|---|---|---|
| 1 | Aleksander | Wypłata | 33 | 8791.12 | 2018-03-12 | Warszawa | Marszałkowska |
| 2 | Roman | Pomidorowy | 43 | 7612.12 | 2012-01-01 | Gdańsk | Oliwska |
| 3 | Anna | Rosół | 38 | 5728.90 | 2015-07-18 | Warszawa | Marszałkowska |
| 4 | Urszula | Nowak | 39 | 3817.21 | 2014-12-15 | Gdańsk | Oliwska |
| 5 | Stefan | Romański | 38 | 9201.23 | 2020-07-14 | Warszawa | Marszałkowska |
| 6 | Jolanta | Kowalska | 27 | 6521.22 | 2012-06-04 | Szczecin | Biała |

## Uwaga 3

`Klucz obcy` powoduje, że należy uważać z kolejnością usuwania danych z bazy danych. Jeżeli tabela `EMPLOYEES` ma dowiązanie do danych z tabeli `ADDRESSES`, to nie możemy najpierw usunąć danych z tabeli `ADDRESSES`, bo na te konkretnie informacje, które byśmy chcieli usunąć wskazuje klucz obcy z tabeli `EMPLOYEES`.

# JOINy

Jak teraz wyświetlić połączone ze sobą tabele? Trzeba wykorzystać klauzulę `JOIN`. JOINy są używane do złączania ze sobą tabel przy wykorzystaniu kluczy obcych.

Da się wykonać połączenie między tabelami po kolumnach, które nie są kluczami obcymi, ale w praktyce zawsze robi się klucze obce. A przynajmniej tak być powinno ☺.

Teraz trzeba pamiętać o pewnej kwestii, gdyż poniższe zapytanie zwróci błąd.

```
SELECT ID, NAME, SURNAME, AGE, SALARY, DATE_OF_EMPLOYMENT, CITY, STREET
FROM EMPLOYEES AS EMP
    INNER JOIN ADDRESSES ADR ON EMP.ADDRESS_ID = ADR.ID;
```

Mamy 2 tabele, które mają teraz kolumnę `ID`, dlatego trzeba konkretnie określić o którą kolumnę nam chodzi. Przepiszmy zatem te zapytanie w ten sposób (różnica to `EMP.ID`):

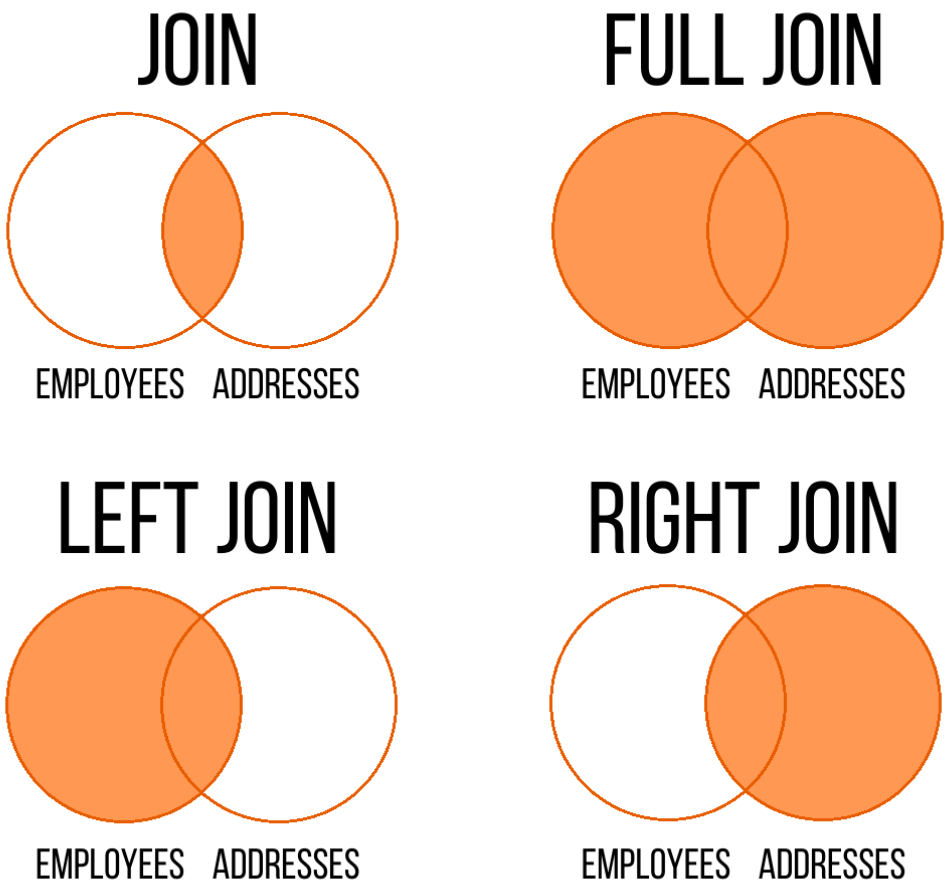
```
SELECT EMP.ID, NAME, SURNAME, AGE, SALARY, DATE_OF_EMPLOYMENT, CITY, STREET
FROM EMPLOYEES AS EMP
    INNER JOIN ADDRESSES ADR ON EMP.ADDRESS_ID = ADR.ID;
```

Wykorzystane tutaj zostały 2 aliasy `EMP` i `ADR` - jest to nazwa własna, która na potrzeby tego zapytania będzie używana jak zmienne. Używamy fragmentu `INNER JOIN`, który pozwala nam złączyć ze sobą 2 tabele na podstawie podanych kolumn `EMP.ADDRESS_ID = ADR.ID`.

W praktyce można napisać ten sam fragment w ten sposób (**zamiast `INNER JOIN` piszemy samo `JOIN`**). Oznacza to to samo, co `INNER JOIN`, ale napisanie `INNER JOIN` jest czytelniejsze, bo podajemy jawnie rodzaj JOINa.

```
SELECT EMP.ID, NAME, SURNAME, AGE, SALARY, DATE_OF_EMPLOYMENT, CITY, STREET
FROM EMPLOYEES AS EMP
    JOIN ADDRESSES ADR ON EMP.ADDRESS_ID = ADR.ID;
```

Czekaj, rodzaj JOINa? Tak, wyróżnia się 4 podstawowe rodzaje JOINów. Rozróżnienie to wynika z tego, że możemy mieć taką sytuację, że dane w 2 tabelach nie pasują do siebie 1 do 1 i możemy mieć rekordy w tabeli EMPLOYEES, dla których nie ma istniejących rekordów powiązanych w tabeli ADDRESSES i odwrotnie. Stąd poniższa grafika, gdzie zostało wyjaśnione działanie każdego JOINa wykorzystując analogię do zbiorów. Grafika powstała na przykładzie tabel EMPLOYEES i ADDRESSES.




*Obraz 1. Rodzaje JOINów*

Na grafice mamy wyróżnione 4 rodzaje joinów:

- **JOIN** (domyślnie JOIN oznacza INNER JOIN) - szukamy przecięcia 2 zbiorów, czyli wyświetlimy tylko te rekordy z tabeli EMPLOYEES, dla których znajdziemy dopasowanie w tabeli ADDRESSES i jednocześnie wyświetlimy tylko takie rekordy z tabeli ADDRESSES, dla których znajdziemy dopasowanie w tabeli EMPLOYEES. Stąd analogia do przecięcia zbiorów,
- **FULL JOIN** - zwrócimy wszystkie rekordy z tabeli EMPLOYEES, nawet te, dla których nie znajdziemy dopasowania w tabeli ADDRESSES i jednocześnie zwrócimy wszystkie rekordy z tabeli ADDRESSES, nawet te, dla których nie znajdziemy dopasowania w tabeli EMPLOYEES,
- **LEFT JOIN** - zwrócimy wszystkie rekordy z tabeli EMPLOYEES, nawet te, dla których nie znaleźliśmy

dopasowania w tabeli ADDRESSES i jednocześnie zwrócimy tylko te rekordy z tabeli ADDRESSES, dla których znaleźliśmy dopasowanie w tabeli EMPLOYEES,

- **RIGHT JOIN** - zwrócimy wszystkie rekordy z tabeli ADDRESSES, nawet te, dla których nie znaleźliśmy dopasowania w tabeli EMPLOYEES i jednocześnie zwrócimy tylko te rekordy z tabeli EMPLOYEES, dla których znaleźliśmy dopasowanie w tabeli ADDRESSES.

Przykład w praktyce? Musimy z definicji tabelki EMPLOYEES pozbyć się constrainta NOT NULL na kolumnie ADDRESS_ID.

```
ALTER TABLE EMPLOYEES
ALTER COLUMN ADDRESS_ID DROP NOT NULL;
```

Teraz możemy mieć taką sytuację, że dodamy rekord do tabeli EMPLOYEES, który nie będzie miał żadnego dowiązania do tabeli ADDRESSES. Możemy również dodać rekordy do tabeli ADDRESSES, do których nie będziemy w żaden sposób się łączyć z tabeli EMPLOYEES.

```
INSERT INTO ADDRESSES (ID, CITY, STREET) VALUES (1, 'Warszawa', 'Marszałkowska');
INSERT INTO ADDRESSES (ID, CITY, STREET) VALUES (2, 'Gdańsk', 'Oliwska');
INSERT INTO ADDRESSES (ID, CITY, STREET) VALUES (3, 'Szczecin', 'Biała');
INSERT INTO ADDRESSES (ID, CITY, STREET) VALUES (4, 'Szczecin', 'Niebieska');
INSERT INTO ADDRESSES (ID, CITY, STREET) VALUES (5, 'Zakopane', 'Wodna');
INSERT INTO ADDRESSES (ID, CITY, STREET) VALUES (6, 'Zakopane', 'Piaskowa');
INSERT INTO ADDRESSES (ID, CITY, STREET) VALUES (7, 'Kraków', 'Wawelska');

INSERT INTO EMPLOYEES (ID, NAME, SURNAME, AGE, SALARY, DATE_OF_EMPLOYMENT)
    VALUES (1, 'Aleksander', 'Wypłata', 33, 8791.12, '2018-03-12');
INSERT INTO EMPLOYEES (ID, NAME, SURNAME, AGE, SALARY, DATE_OF_EMPLOYMENT, ADDRESS_ID)
    VALUES (2, 'Roman' 'Pomidorowy' 43, 7612.12, '2012-01-01', 2);
INSERT INTO EMPLOYEES (ID, NAME, SURNAME, AGE, SALARY, DATE_OF_EMPLOYMENT)
    VALUES (3, 'Anna', 'Rosół', 38, 5728.90, '2015-07-18');
INSERT INTO EMPLOYEES (ID, NAME, SURNAME, AGE, SALARY, DATE_OF_EMPLOYMENT, ADDRESS_ID)
    VALUES (4, 'Urszula', 'Nowak', 39, 3817.21, '2014-12-15', 2);
INSERT INTO EMPLOYEES (ID, NAME, SURNAME, AGE, SALARY, DATE_OF_EMPLOYMENT, ADDRESS_ID)
    VALUES (5, 'Stefan', 'Romański', 38, 9201.23, '2020-07-14', 1);
INSERT INTO EMPLOYEES (ID, NAME, SURNAME, AGE, SALARY, DATE_OF_EMPLOYMENT)
    VALUES (6, 'Jolanta', 'Kowalska', 27, 6521.22, '2012-06-04');
```

Teraz mając te przykłady możemy pobawić się kolejnymi rodzajami joinów:

# INNER JOIN

```
SELECT ID, NAME, SURNAME, AGE, SALARY, DATE_OF_EMPLOYMENT, CITY, STREET
FROM EMPLOYEES AS EMP
    INNER JOIN ADDRESSES ADR ON EMP.ADDRESS_ID = ADR.ID;
```

# FULL JOIN

```
SELECT EMP.ID, NAME, SURNAME, AGE, SALARY, DATE_OF_EMPLOYMENT, CITY, STREET
FROM EMPLOYEES AS EMP
    FULL JOIN ADDRESSES ADR ON EMP.ADDRESS_ID = ADR.ID;
```

# LEFT JOIN

```
SELECT EMP.ID, NAME, SURNAME, AGE, SALARY, DATE_OF_EMPLOYMENT, CITY, STREET
FROM EMPLOYEES AS EMP
    LEFT JOIN ADDRESSES ADR ON EMP.ADDRESS_ID = ADR.ID;
```

# RIGHT JOIN

```
SELECT EMP.ID, NAME, SURNAME, AGE, SALARY, DATE_OF_EMPLOYMENT, CITY, STREET
FROM EMPLOYEES AS EMP
    RIGHT JOIN ADDRESSES ADR ON EMP.ADDRESS_ID = ADR.ID;
```

Przy czym zwróć uwagę, że w przypadku `LEFT JOIN` oraz `RIGHT JOIN`, lewy oraz prawy zbiór (odnoszę się do obrazka) są kolejno rozumiane jako tabela określona w sekcji `FROM` (lewy zbiór) oraz tabela dodawana w sekcji `JOIN` (prawy zbiór).

# Notatki - JDBC - cz.1

# Spis treści



# JDBC

To skoro już potrafimy poruszać się w świecie baz danych, to możemy zacząć rozmawiać o tym jak z taką bazą danych może się dogadać Java.

Zanim natomiast przejdziemy do konkretów, chciałbym opisać dosyć ciekawą kwestię, której jeszcze nie widzieliśmy w całym procesie nauki. Tak jak wspominałem, różnych rodzajów baz danych jest dużo. Java nie wie konkretnie jak do każdej z tych baz danych ma się podłączyć. To znaczy, nie ma w JRE zapisanego potężnego ifa, który wyglądałby jakoś tak:

```
if (naszaBazaDanych = "PostgreSQL") {
    polacz_sie_w_ten_sposob();
}
if (naszaBazaDanych = "MySQL") {
    polacz_sie_w_inny_sposob();
}
```

Określone są natomiast pewne interfejsy (jak już wiemy, będące kontraktem), mówiące, jakie zachowania "coś" ma implementować, aby móc podłączyć się do konkretnej bazy danych. Wiemy też już, że aby interfejs był w ogóle użyteczny to musimy dostarczyć konkretną implementację metod z interfejsu.

Aby dostarczyć takie konkretne implementacje interfejsów, na których będziemy operować należy dostarczyć sterownik (driver) do konkretnej implementacji bazy danych. Sterownik taki będzie dostarczony w postaci pliku `.jar` (skrót od **Java Archive**) i będzie on zawierał konkretne implementacje wymaganych interfejsów, aby móc podłączyć się do konkretnej bazy danych.

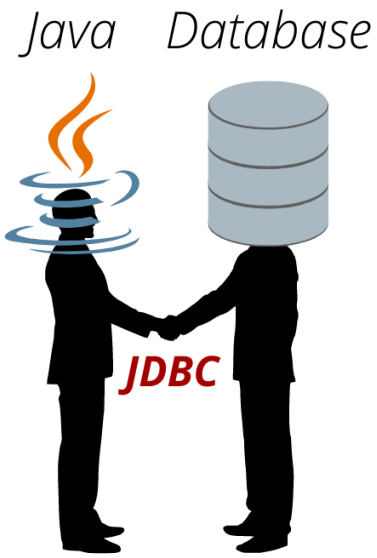
Możemy taki sterownik znaleźć w internecie, wpisując przykładowo **postgresql jdbc driver**. Przy pobieraniu Chrome nas zapyta, czy jesteśmy pewni, że chcemy pobrać ten plik bo może on być potencjalnie niebezpieczny. Prawda, może być on potencjalnie niebezpieczny, jeżeli nie znamy źródła pliku, jeżeli natomiast pobierzemy go ze strony Link to powinien on być bezpieczny ☺. Plik taki może się

nazywać przykładowo `postgresql-42.2.23.jar`. Najważniejsze jest to, że klasy, które są zawarte w pliku `.jar` wiedzą jak podłączyć się konkretnie do tej bazy danych, która nas interesuje.

Następnie mając taki plik `.jar` należy dodać go do naszego projektu i zaznaczyć, że jest on 'biblioteką'. Możemy to ustawić w ustawieniach naszego projektu **CTRL** + **ALT** + **SHIFT** + **S** w zakładce **libraries**.

Dzięki temu, możemy używać interfejsów w naszym projekcie podczas pisania kodu, natomiast aby faktycznie ten kod zadziałał, to musimy dostarczyć implementację naszych interfejsów. Dostarczamy ją dołączając sterownik dedykowany do konkretnej bazy danych do naszego projektu.

Przy następnych przykładach będziemy posługiwać się interfejsami `Driver`, `Connection`, `Statement` i `ResultSet`. Aby program mógł poprawnie działać należy dostarczyć sterownik, który zawiera implementację tych interfejsów. Nie interesuje nas natomiast jak konkretnie nazywają się klasy implementujące te interfejsy.




*Obraz 1. JDBC*

# Podłączenie do bazy danych

Spróbujemy przejść przez przykłady w logicznej kolejności, to znaczy, że najpierw spróbujemy podłączyć się do bazy danych, aby później otrzymać wynik zapytania, które będziemy na takiej bazie wykonywać.

Zanim natomiast nastąpi połączenie do takiej bazy, musimy podać w jakiś sposób gdzie taka baza się znajduje, czyli podać jej adres.

## JDBC URL

Jeżeli funkcjonujesz już trochę w Internecie to wydaje się naturalnym taki zapis https://www.youtube.com/. Jest to adres strony internetowej, na której możemy oglądać filmy o śmiesznych kotkach.

Bazy danych również mają swoje adresy i musimy podać taki adres, aby móc do takiej bazy danych się podłączyć. W przypadku baz danych, adres taki wygląda w ten sposób:

```
<protokol>:<rodzaj_bazy_danych>://<adres_bazy_danych>/<nazwa_bazy_danych>
```

Jeżeli teraz zmienne w nawiasach trójkątnych zastąpimy wartościami, to taki adres mógłby wyglądać w taki sposób:

```
jdbc:postgresql://localhost:5432/zajavka
```

Znaczenie kolejnych fragmentów:

- **jdbc** - nazwa protokołu jaki wykorzystujemy do połączenia
- **postgres** - nazwa bazy danych do jakiej chcemy się podłączyć
- **localhost** - nazwa hosta (inaczej strony internetowej), podając ten adres możemy znaleźć w sieci gdzie fizycznie znajduje się nasza baza danych. **Localhost** oznacza, że jest ona na naszej maszynie. Możemy ewentualnie spotkać zapis **127.0.0.1**, również oznacza on naszą maszynę
- **5432** - port, na którym baza danych jest dostępna. Możemy to sobie wyobrazić w taki sposób, że localhost (albo inny adres) jest miastem i aby wpłynąć do danego miasta statkiem potrzebujemy portu. Ale w jednym mieście może być wiele portów. Tak samo uruchamiając jakiś serwer na komputerze, jeżeli chcemy dać możliwość, aby taki serwer przyjmował jakikolwiek ruch to musi wystawiać otwarty port. Portów mamy wiele, bo gdyby serwer miał tylko jeden port, to na jednym komputerze moglibyśmy uruchomić tylko jedną aplikację. Jeżeli każda aplikacja/baza danych/inne rzeczy, których jeszcze nie poznaliśmy mogą być uruchamiane na różnych portach, to na jednym komputerze możemy uruchomić wiele aplikacji, które pozwalają się komunikować ze sobą przez sieć
- **zajavka** - nazwa bazy danych

# Spróbujmy połączyć się do bazy danych

Teraz gdy już wiemy jak skonstruować adres, możemy spróbować faktycznie połączyć się do tej bazy danych.

```
public class JdbcConnectionExample {

    public static void main(String[] args) {
        try {
            Connection conn = DriverManager.getConnection(
                    "jdbc:postgresql://localhost:5432/zajavka",
                    "postgres",
                    "password"
            );
            System.out.println(conn);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Jesteś już w stanie rozpoznać w powyższym kodzie JDBC URL, który musimy podać aby podłączyć się do

bazy danych. `postgres` to nazwa użytkownika bazy danych, natomiast `password` to przykładowe hasło dla tego użytkownika. Jeżeli nie udaje Ci się uzyskać połączenia, może to być kwestia niezgodności podanego adresu/użytkownika/hasła. Pamiętaj, że hasło było ustalane na etapie instalacji. Jeżeli coś nie działa na tym etapie, polecam googlować, musimy się uczyć rozwiązywać takie problemy ☺.

Gdy uruchomimy ten kod, na ekranie zostanie wydrukowane coś w tym stylu (no chyba, że wcześniej dodaliśmy sterownik do naszego projektu):

```
java.sql.SQLException: No suitable driver found for jdbc:postgresql://localhost:5432/zajavka
    at java.sql/java.sql.DriverManager.getConnection(DriverManager.java:702)
    at java.sql/java.sql.DriverManager.getConnection(DriverManager.java:251)
    at pl.zajavka.JdbcConnectionExample.main(JdbcConnectionExample.java:10)
```

Dodajemy teraz sterownik do naszego projektu w oknie z ustawieniami naszego projektu **CTRL** + **ALT** + **SHIFT** + **S** w zakładce "libraries". Spróbujmy uruchomić teraz ten kod ponownie, na ekranie wydrukuje się coś podobnego do:

```
org.postgresql.jdbc.PgConnection@b59d31
```

Czyli konkretna klasa implementująca nasz interfejs `Connection`. W przypadku podłączenia do innej bazy danych, klasa ta nazywałaby się inaczej.

Trochę rzeczy stało się tutaj jakby magicznie. Wszystko dzięki metodzie `DriverManager.getConnection()`, która jest odpowiedzialna za znalezienie odpowiedniej klasy implementującej nasz interfejs (nie musimy tego podawać sami).

Od razu też uprzedzę, że to co zrobiliśmy tutaj zostało przedstawione w formie edukacyjnej. W praktyce prawdopodobnie nawet nie będziesz znać hasła do bazy danych na serwerze produkcyjnym, zostaną do tego wykorzystane inne mechanizmy.

Druga kwestia do wyjaśnienia - prawdopodobnie nie będziesz też ręcznie pisać/używać klas `Connection`, `Statement` itp. w praktyce. W praktyce stosowane są biblioteki, które robią to za Ciebie. Musimy natomiast wiedzieć jak to się zachowuje pod spodem, żeby nie była to dla nas jeszcze większa magia.

# Statement

Następnym interfejsem, który powinniśmy omówić, jest `Statement`. Reprezentuje on `SQL Statement`, o których to SQLach uczyliśmy się wcześniej.

Najprostszym sposobem na uzyskanie `Statement` jest `connection.createStatement()`:

```
public class JdbcConnectionExample {

    public static void main(String[] args) {
        Optional<Connection> connection = getConnection();
        Optional<Statement> statement = connection.flatMap(JdbcConnectionExample::createStatement);
    }

    private static Optional<Connection> getConnection() {
        try {
```

```
            Optional<Connection> connection = Optional.of(DriverManager.getConnection(
                "jdbc:postgresql://localhost:5432/zajavka",
                "postgres",
                "password"
            ));
            System.out.println(connection);
            return connection;
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return Optional.empty();
    }

    private static Optional<Statement> createStatement(final Connection connection) {
        try {
            return Optional.of(connection.createStatement());
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return Optional.empty();
    }
}
```

Są jednak dostępne przeładowane wersje metody `createStatement()`, które mogą przyjmować parametry. Nie chcę natomiast mocno się rozpisywać na ich temat, gdyż w większości przypadków będziesz stosować podstawowy wariant metody `createStatement()`.

Oba te parametry dotyczą `ResultSet`, czyli interfejsu, który będzie reprezentował wynik otrzymany po poprawnym wykonaniu zapytania na bazie danych. Parametry te określają, co możemy z takim `ResultSet` po jego otrzymaniu zrobić innego niż zachowanie domyślne.

### ResultSet Type

Pierwszym parametrem, który możemy określić jest `ResultSet Type`. Określa on w jakim kierunku możemy czytać dane ze zbioru, który otrzymaliśmy. W większości przypadków będziemy czytać je dokładnie w takim, w jakim je otrzymaliśmy, bez żadnego kombinowania. Zapewni nam to parametr `ResultSet.TYPE_FORWARD_ONLY`, który jest wartością domyślną tego parametru w metodzie `createStatement()`. Jeżeli ktoś będzie potrzebował tutaj coś kombinować to odsyłam do dokumentacji ☺.

### ResultSet Concurrency Mode

Drugi parametr jaki możemy przekazać do metody `createStatement()` dotyczy tego, czy z otrzymanego `ResultSet` możemy tylko czytać, czy może też próbować przez niego zapisywać dane do bazy danych. W większości przypadków z `ResultSet` będziemy tylko czytać i zapewnia nam to wartość domyślna tego parametru `ResultSet.CONCUR_READ_ONLY`, która jest używana w metodzie `createStatement()`. Jeżeli chcemy dokonać aktualizacji danych w bazie danych, raczej robi się to przy wykorzystaniu `UPDATE QUERY`, niż przez `ResultSet`. Jeżeli ktoś będzie potrzebował tutaj coś kombinować to odsyłam do dokumentacji ☺.

# Zamykanie otwartych zasobów

Przejdźmy jeszcze do tematu zamykania otwartych połączeń/zasobów. Rzecz polega na tym, że na bazach danych ustawia się limity otwartych połączeń, więc jeżeli nie będziemy ich zamykać możemy

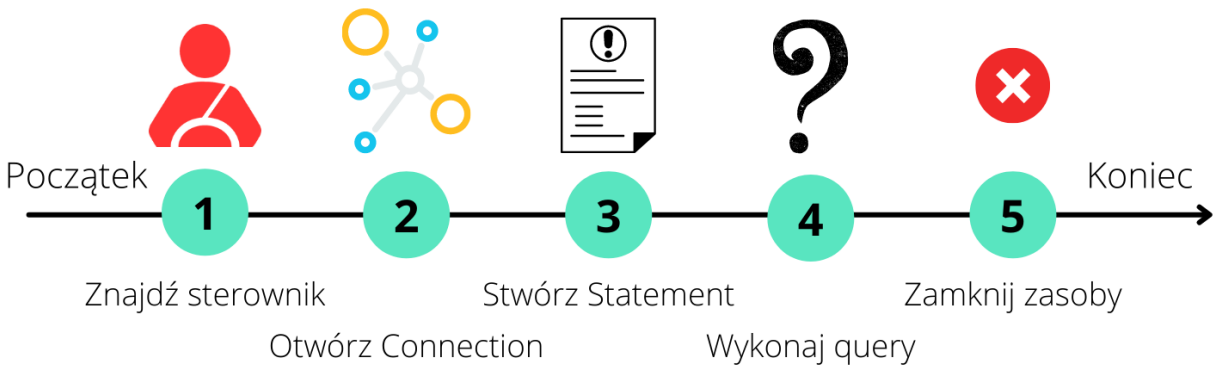
doprowadzić do takiej sytuacji, że nie będziemy w stanie podłączyć się do bazy danych, bo próbujemy ustanowić nowe połączenie, ale nie możemy tego zrobić bo nie zamknęliśmy starych i limit się wyczerpał. W praktyce stosowane są takie praktyki jak pule połączeń, aby móc reużywać otwarte już połączenia, ale nie jest to temat na teraz.

Interface `ResultSet` będzie omawiany w następnej kolejności. Tutaj jest on przywołany aby pokazać mechanizm otwierania i zamykania wszystkich zasobów, na których będziemy pracować.

Jak zamknąć połączenie?

```
private static void oldWay() throws SQLException {
    Connection connection = null;
    Statement statement = null;
    ResultSet resultSet = null;
    try {
        connection = DriverManager.getConnection(
            "jdbc:postgresql://localhost:5432/zajavka",
            "postgres",
            "password"
        );
        statement = connection.createStatement();
        resultSet = statement.executeQuery("SELECT * FROM CUSTOMER;");
        while (resultSet.next()) {
            System.out.println(resultSet.getString("name"));
        }
    } finally {
        try {
            if (resultSet != null)
                resultSet.close();
        } catch (SQLException e) {
            System.err.println("Failed to close ResultSet, " + e.getMessage());
        }
        try {
            if (statement != null)
                statement.close();
        } catch (SQLException e) {
            System.err.println("Failed to close Statement, " + e.getMessage());
        }
        try {
            if (connection != null)
                connection.close();
        } catch (SQLException e) {
            System.err.println("Failed to close Connection, " + e.getMessage());
        }
    }
}
```

Zwróć uwagę, że zamykamy nie tylko połączenie, ale również `ResultSet` i `Statement`. Generalnie jest dobrym nawykiem zamykać wszystkie 3 zasoby ręcznie (`Connection`, `Statement` i `ResultSet`), ale nie jest to konieczne. Teoretycznie jeżeli zamkniemy `Connection`, powinno ono zamknąć `Statement` i `ResultSet`. Jeżeli zamkniemy sam `Statement`, to powinien on zamknąć również `ResultSet`. Ważne jest natomiast aby zamykać te zasoby w odwrotnej kolejności niż zostały utworzone.




*Obraz 2. Kroki połączenia JDBC*

# A da się prościej?

Pamiętasz `try-with-resources`? Kod pokazany powyżej da się mocno uprościć o ile zastosujemy konstrukcję `try-with-resources`.

```
private static void newWay() throws SQLException {
    try (Connection connection = DriverManager
            .getConnection("jdbc:postgresql://localhost:5432/zajavka", "postgres", "password");
         Statement statement = connection.createStatement();
         ResultSet resultSet = statement.executeQuery("select name from animal")
    ) {
        while (resultSet.next()) {
            System.out.println(resultSet.getString("name"));
        }
    }
}
```

Konstrukcja `try-with-resources` zamykała zasoby w odwrotnej kolejności niż zostały one podane w zapisie. Czyli najpierw zostanie zamknięty `ResultSet`, potem `Statement`, a potem `Connection`. W przykładzie, w którym nie używaliśmy tej konstrukcji, trzeba było dodać samemu logowanie ewentualnych błędów, które mogły wystąpić na etapie zamykania każdego z zasobów. Tutaj jeżeli takie błędy wystąpią, będą one dostępne jako `Suppressed Exception`.

# Notatki - JDBC - cz.2

## Spis treści



# Wywołajmy w końcu jakąś komendę ☺

Możemy nareszcie przejść do faktycznego wywołania jakiegoś zapytania SQL.

Zacznijmy od QUERY typu INSERT, gdyż jest to pierwszy rodzaj zapytania jakiego będziemy potrzebowali aby zasilić tabelę danymi. Mamy 3 podstawowe rodzaje metod stosowanych przy interface Statement aby wykonywać zapytania SQL. Jest to metoda execute(), executeQuery() i executeUpdate().

## Dane do przykładów

W kolejnych przykładach będę odnosił się cały czas do struktury tabel, która jest pokazana poniżej. Będziemy działać na tabelach: CUSTOMER, PRODUCER, PRODUCT, PURCHASE, OPINION.

*Tabela CUSTOMER*

```
CREATE TABLE CUSTOMER(
  ID              INT         NOT NULL,
  USER_NAME       VARCHAR(32) NOT NULL,
  EMAIL           VARCHAR(32) NOT NULL,
  NAME            VARCHAR(32) NOT NULL,
  SURNAME         VARCHAR(32) NOT NULL,
  DATE_OF_BIRTH   DATE,
  TELEPHONE_NUMBER VARCHAR(64),
  PRIMARY KEY (ID),
  UNIQUE (USER_NAME),
  UNIQUE (EMAIL)
);
```

*Tabela PRODUCER*

```
CREATE TABLE PRODUCER(
  ID            INT          NOT NULL,
  PRODUCER_NAME VARCHAR(32)  NOT NULL,
  ADDRESS       VARCHAR(128) NOT NULL,
  PRIMARY KEY (ID),
  UNIQUE (PRODUCER_NAME)
);
```

*Tabela PRODUCT*

```
CREATE TABLE PRODUCT(
  ID            INT           NOT NULL,
  PRODUCT_CODE  VARCHAR(32)   NOT NULL,
  PRODUCT_NAME  VARCHAR(64)   NOT NULL,
  PRODUCT_PRICE NUMERIC(7, 2) NOT NULL,
  ADULTS_ONLY   BOOLEAN       NOT NULL,
  DESCRIPTION   TEXT          NOT NULL,
  PRODUCER_ID   INT           NOT NULL,
  PRIMARY KEY (ID),
  UNIQUE (PRODUCT_CODE),
  CONSTRAINT fk_product_producer
    FOREIGN KEY (PRODUCER_ID)
      REFERENCES PRODUCER (ID)
);
```

*Tabela PURCHASE*

```
CREATE TABLE PURCHASE(
  ID          INT  NOT NULL,
  CUSTOMER_ID INT  NOT NULL,
  PRODUCT_ID  INT  NOT NULL,
  QUANTITY    INT  NOT NULL,
  DATE_TIME   TIMESTAMP WITH TIME ZONE NOT NULL,
  PRIMARY KEY (ID),
  CONSTRAINT fk_purchase_customer
    FOREIGN KEY (CUSTOMER_ID)
      REFERENCES CUSTOMER (ID),
  CONSTRAINT fk_purchase_product
    FOREIGN KEY (PRODUCT_ID)
      REFERENCES PRODUCT (ID)
);
```

*Tabela OPINION*

```
CREATE TABLE OPINION(
  ID          INT  NOT NULL,
  CUSTOMER_ID INT  NOT NULL,
  PRODUCT_ID  INT  NOT NULL,
  STARS       INT  CHECK (STARS IN (1, 2, 3, 4, 5)) NOT NULL,
  COMMENT     TEXT  NOT NULL,
  DATE_TIME   TIMESTAMP WITH TIME ZONE NOT NULL,
  PRIMARY KEY (ID),
  CONSTRAINT fk_purchase_customer
    FOREIGN KEY (CUSTOMER_ID)
      REFERENCES CUSTOMER (ID),
  CONSTRAINT fk_purchase_product
    FOREIGN KEY (PRODUCT_ID)
      REFERENCES PRODUCT (ID)
);
```

# executeUpdate()

Używana do zapytań INSERT, UPDATE, DELETE. Jest o tyle ciekawa, że jej nazwa zawiera słowo **update**, a

używa się jej również do operacji INSERT oraz DELETE. Wynikiem jej wywołania jest liczba zmodyfikowanych wierszy w bazie danych.

Przykład INSERT, UPDATE i DELETE:

*Klasa JdbcConnectionExecuteUpdateExample*

```
public class JdbcConnectionExecuteUpdateExample {

    public static void main(String[] args) {
        String datbaseURL = "jdbc:postgresql://localhost:5432/zajavka";
        String user = "postgres";
        String password = "password";
        try (
            Connection connection = DriverManager.getConnection(datbaseURL, user, password);
            Statement statement = connection.createStatement()
        ) {
            String query1 = "INSERT INTO PRODUCER (ID, PRODUCER_NAME, ADDRESS) " +
                "VALUES (21, 'Zajavka Group', 'Zajavkowa 15, Warszawa');";
            String query2 = "UPDATE PRODUCER SET ADDRESS = 'Nowy adres naszej siedziby' WHERE ID = 21;";
            String query3 = "DELETE FROM PRODUCER WHERE ID = 21;";

            Optional.of(statement.executeUpdate(query1))
                .ifPresent(result -> System.out.printf("Inserted %s row(s)%n", result));
            Optional.of(statement.executeUpdate(query2))
                .ifPresent(result -> System.out.printf("Updated %s row(s)%n", result));
            Optional.of(statement.executeUpdate(query3))
                .ifPresent(result -> System.out.printf("Deleted %s row(s)%n", result));
        } catch (Exception e) {
            System.err.printf("Error while working on database: %s%n", e.getMessage());
        }
    }
}
```

Każde wywołanie executeUpdate() zwraca ilość zmodyfikowanych wierszy.

Jeżeli natomiast chcielibyśmy wywołać komendę, która zwraca nam jakieś informacje, w tym celu możemy użyć executeQuery():

# executeQuery()

Używana do zapytań `SELECT`, zwraca otrzymany z zapytania rezultat w postaci interfejsu `ResultSet`.

```
public class JdbcConnectionExecuteQueryExample {

    public static void main(String[] args) {
        String datbaseURL = "jdbc:postgresql://localhost:5432/zajavka";
        String user = "postgres";
        String password = "password";
        String query = "SELECT * FROM PRODUCER;";
        try (
            Connection connection = DriverManager.getConnection(datbaseURL, user, password);
            Statement statement = connection.createStatement();
            ResultSet resultSet = statement.executeQuery(query)
        ) {
            System.out.printf("Selected %s%n", resultSet);
        } catch (Exception e) {
            System.err.printf("Error while working on database: %s%n", e.getMessage());
        }

    }
}
```

Typem zwracanym z tego kodu jest `ResultSet` i sama próba wydrukowania go na ekranie kończy się czymś podobnym do wydruku poniżej - dołożone zostało słówko `Selected`.

```
Selected org.postgresql.jdbc.PgResultSet@4d49af10
```

Zaraz omówimy jak z takiego rezultatu `ResultSet` wyciągnąć faktyczne dane pobrane z bazy.

# execute()

Dodajmy jeszcze natomiast, że oprócz `executeUpdate()` i `executeQuery()` istnieje sama metoda `execute()`. Można jej używać w następujący sposób:

Metoda `execute()` może być uruchomiona do zapytań `INSERT`, `UPDATE`, `DELETE` i `SELECT`. Jej wynikiem jest `boolean` mówiący, czy jako wynik został zwrócony `ResultSet`. Jest to o tyle ważne, że tylko zapytanie `SELECT` może zwrócić `ResultSet`. Reszta nie zwraca wyniku bo służy do modyfikacji stanu bazy danych.

```
public class JdbcConnectionExecuteExample {

    public static void main(String[] args) {
        String datbaseURL = "jdbc:postgresql://localhost:5432/zajavka";
        String user = "postgres";
        String password = "password";
        String query = "SELECT * FROM PRODUCER;";
        try (
            Connection connection = DriverManager.getConnection(datbaseURL, user, password);
            Statement statement = connection.createStatement()
        ) {
            boolean resultSetExists = statement.execute(query);
```

```
            if (resultSetExists) {
                ResultSet resultSet = statement.getResultSet();
                System.out.println("ResultSet: " + resultSet);
            } else {
                int count = statement.getUpdateCount();
                System.out.println("Count: " + count);
            }
        } catch (Exception e) {
            System.err.printf("Error while working on database: %s%n", e.getMessage());
        }
    }

}
```

Nie jest natomiast dobrym pomysłem zwracanie z metody raz `ResultSet` a raz `Integer`, więc potraktuj ten przykład z przymrużeniem oka ☺. Dalej będziemy się raczej skupiać na metodach `executeUpdate()` i `executeQuery()`. Oczywiście możemy próbować kombinować i wywołać `executeUpdate()` z SELECTem w środku, ale dostaniemy wtedy `SQLException`.

Podsumowując, do jakich rodzajów zapytań można stosować określone metody:

| Metoda | INSERT | SELECT | UPDATE | DELETE |
|---|---|---|---|---|
| execute() | TAK | TAK | TAK | TAK |
| executeUpdate() | TAK | NIE | TAK | TAK |
| executeQuery() | NIE | TAK | NIE | NIE |

# Notatki - JDBC - cz.3

## Spis treści



## PreparedStatement

Trochę teraz o security. Zwróć uwagę, że jak na razie cały czas wpisywaliśmy `query`, które nie miały przekazywanych żadnych parametrów. W życiu raczej się takie rzeczy nie zdarzają, raczej będziemy tworzyć zapytania, które będą przyjmowały parametry z kodu Javowego.

Naturalnym wydaje się zatem napisanie czegoś takiego (potrzebujemy skasować użytkownika z naszego sklepu):

```
private static Optional<Integer> deleteUser(final Statement statement, String userName) {
    try {
        String query = "DELETE FROM CUSTOMER WHERE USER_NAME = '" + userName + "';";
        return Optional.of(statement.executeUpdate(query));
    } catch (SQLException e) {
        System.err.printf("Failed to executeUpdate: %s%n", e.getMessage());
    }
    return Optional.empty();
}
```

Wyobraźmy teraz sobie, że mamy napisaną aplikację w taki sposób, że nazwę użytkownika do usunięcia podaje sam użytkownik (w formie potwierdzenia, że na pewno to ten `user_name` ma być usunięty).

No i wszystko w porządku ☺, użytkownik na formularzu strony internetowej wpisuje swój `user_name` do usunięcia, query wygląda wtedy w ten sposób:

```
DELETE FROM CUSTOMER WHERE USER_NAME = 'asterix';
```

W tym momencie bardzo możliwe, że dostaniemy komunikat mówiący o tym, że naruszamy klucz obcy, czyli, że ten użytkownik jest wciąż używany w inych miejscach w bazie danych. Zatem dopiszmy zapytania, które usuwają również tego użytkownika z innych tabel:

```
DELETE FROM OPINION
    WHERE CUSTOMER_ID IN (SELECT ID FROM CUSTOMER WHERE USER_NAME = 'asterix');
DELETE FROM PURCHASE
    WHERE CUSTOMER_ID IN (SELECT ID FROM CUSTOMER WHERE USER_NAME = 'asterix');
DELETE FROM CUSTOMER
    WHERE USER_NAME = 'asterix';
```

I usunięcie przebiegło pomyślnie, usunęliśmy tylko użytkownika o `user_name = asterix`.

Potem przychodzi kolejny użytkownik i wpisuje w parametrze w formularzu coś takiego:

```
whatever' or 1=1 or USER_NAME = 'whateverAgain
```

Wtedy po uzupełnieniu naszego SQL parametrem user_name dostajemy takie SQL:

```
DELETE FROM OPINION
    WHERE CUSTOMER_ID
        IN (SELECT ID FROM CUSTOMER WHERE USER_NAME = 'whatever'
            or 1=1 or USER_NAME = 'whateverAgain');
DELETE FROM PURCHASE
    WHERE CUSTOMER_ID
        IN (SELECT ID FROM CUSTOMER WHERE USER_NAME = 'whatever'
            or 1=1 or USER_NAME = 'whateverAgain');
DELETE FROM CUSTOMER
    WHERE USER_NAME = 'whatever'
        or 1=1 or USER_NAME = 'whateverAgain';
```

Jeżeli spróbujemy wykonać takie zapytanie na bazie danych, to dostaniemy błąd, że nie możemy usunąć rekordów z tabeli CUSTOMER, gdyż istnieją klucze obce w innych tabelach wskazujące na rekordy z tej tabeli, ale skonstruowane zapytanie stara się usunąć nam wszystkich użytkowników z bazy danych.

Czyli gdybyśmy wykonali tego rodzaju akcję na tabeli, która nie ma wskazujących na nią kluczy obcych, np PURCHASE albo OPINION, to takie zapytanie skasowałoby nam wszystkie dane z tabeli. To, co teraz pokazałem nazywa się **SQL Injection** i jest szeroko znanym rodzajem ataku na aplikacje webowe.

Wiesz już, czemu nie należy za pomocą konkatenacji konstruować SQLi, które potem mają zostać wykonane na bazie danych?

Dlatego teraz na białym koniu wjeżdża PreparedStatement i rozwiązuje ten problem i kilka innych:

- **bezpieczeństwo** - jest bezpieczniejsze, ze względu na inną konstrukcję podawania parametrów do zapytania i niestraszne mu **SQL Injection**
- **czytelność** - w przypadku wielu parametrów, PreparedStatement jest czytelniejsze dla osoby czytającej kod
- **wydajność** - PreparedStatement jest bardziej wydajne niż Statement

# Wykorzystanie w praktyce

*Klasa JdbcPreparedStatementExample*

```
public class JdbcPreparedStatementExample {

    public static void main(String[] args) {
        String userName = "asterix";
        String query1 = "DELETE FROM OPINION " +
            "WHERE CUSTOMER_ID IN (SELECT ID FROM CUSTOMER WHERE USER_NAME = ?);";
        String query2 = "DELETE FROM PURCHASE " +
            "WHERE CUSTOMER_ID IN (SELECT ID FROM CUSTOMER WHERE USER_NAME = ?);";
        String query3 = "DELETE FROM CUSTOMER WHERE USER_NAME = ?;";
```

```
        String databaseURL = "jdbc:postgresql://localhost:5432/zajavka";
        String user = "postgres";
        String password = "password";
        try (
            Connection connection = DriverManager.getConnection(databaseURL, user, password);
            PreparedStatement statement1 = connection.prepareStatement(query1);
            PreparedStatement statement2 = connection.prepareStatement(query2);
            PreparedStatement statement3 = connection.prepareStatement(query3)
        ) {
            statement1.setString(1, userName);
            statement2.setString(1, userName);
            statement3.setString(1, userName);

            System.out.println("Changed: " + statement1.executeUpdate());
            System.out.println("Changed: " + statement2.executeUpdate());
            System.out.println("Changed: " + statement3.executeUpdate());

        } catch (Exception e) {
            System.err.printf("Error while working on database: %s%n", e.getMessage());
        }
    }
}
```

Czyli korzystamy z gotowego mechanizmu podstawiania zmiennych `statement.setString(1, userName);` Nie musimy się wtedy martwić o cudzysłowia, apostrofy itp. Jednocześnie zabezpiecza nas to przed atakiem pokazanym poprzednio.

# Notatki - JDBC i ResultSet

## Spis treści



# JDBC i ResultSet

## ResultSet

Otrzymaliśmy już wcześniej `ResultSet`, ale ani razu jeszcze nie wykorzystaliśmy go do faktycznego odczytania wartości.

W kodzie poniżej staramy się pobrać wszystkich klientów naszego sklepu, którzy mają imię zawierające litery `me`. W tym celu konstruujemy zapytanie:

```
SELECT * FROM CUSTOMER WHERE NAME LIKE '%me%';
```

Pamiętając, że w przypadku wykorzystania klauzuli `LIKE` należy zwrócić uwagę na procenty i miejsca, w których je umieszczamy. Procent oznacza brak znaku, jeden lub więcej dowolnych znaków.

Wykonujemy takie zapytanie, pobieramy `ResultSet` i wykorzystując pętle `while`, staramy się pobrać wartości z kolejnych kolumn i na tej podstawie zwrócić obiekt `Customer`. Następnie obiekty te dodajemy do listy.

Wykorzystujemy metodę `next()` gdyż `ResultSet` posiada kursor, który określa obecną pozycję do odczytu danych w zbiorze danych będącym rezultatem zapytania. W momencie, gdy wywołujemy metodę `next()`, przesuwamy ten kursor o jedną pozycję do przodu. Gdy metoda `next()` zwróci `true`, oznacza to, że możemy odczytać dane. Gdy zwróci `false`, mówi to nam, że nie mamy w tym `ResultSet` już dalej danych. Dlatego można w prosty sposób wykorzystać tę metodę w pętli `while()`.

*Klasa JdbcResultSetExample*

```
public class JdbcResultSetExample {

    public static void main(String[] args) {
        String query = "SELECT * FROM CUSTOMER WHERE NAME LIKE ?;";
        String parameter = "%me%";

        String datbaseURL = "jdbc:postgresql://localhost:5432/zajavka";
        String user = "postgres";
        String password = "password";
        try (
            Connection connection = DriverManager.getConnection(datbaseURL, user, password);
            PreparedStatement statement = connection.prepareStatement(query)
        ) {
            statement.setString(1, parameter);
            try (
                ResultSet resultSet = statement.executeQuery();
            ) {
                List<Customer> customers = CustomerMapper.mapToCustomers(resultSet);
                customers.forEach(c -> System.out.println("Customer: " + c));
            }
        } catch (Exception e) {
            System.err.printf("Error while working on database: %s%n", e.getMessage());
        }
    }
}
```

*Klasa CustomerMapper*

```
public class CustomerMapper {

    static List<Customer> mapToCustomers(final ResultSet rs) {
        List<Customer> customers = new ArrayList<>();
        try {
            while (rs.next()) {
                long id = rs.getLong("id");
                String userName = rs.getString("user_name");
                String email = rs.getString("email");
                String name = rs.getString("name");
                String surname = rs.getString("surname");
                String dateOfBirth = rs.getString("date_of_birth");
                String telephoneNumber = rs.getString("telephone_number");
                customers.add(
                    new Customer(
                        id,
                        userName,
                        email,
                        name,
                        surname,
                        LocalDate.parse(dateOfBirth),
                        telephoneNumber));
            }
        } catch (SQLException e) {
            System.err.println("Failed to read ResultSet: " + e.getMessage());
        }
        return customers;
    }
```

```
}
```

Zwróć uwagę na metody w stylu `getLong()`, `getString()` i nazwę kolumny z bazy danych. Metod tego typu jest o wiele więcej: `getInt()`, `getDouble()`, `getBoolean()` itp. Ważne jest tutaj, aby `String` podany jako argument tych metod był dokładnie taki jak nazwa kolumny. Wielkość liter nie ma natomiast znaczenia.

Oprócz metod odwołujących się po nazwie kolumny możemy również odwołać się na podstawie indeksów kolejnych kolumn. W takim przypadku metoda `mapToCustomers()` wyglądałaby w ten sposób:

```
public class CustomerMapper {

    static List<Customer> mapToCustomersByIndex(final ResultSet rs) {
        List<Customer> customers = new ArrayList<>();
        try {
            while (rs.next()) {
                long id = rs.getLong(1);
                String userName = rs.getString(2);
                String email = rs.getString(3);
                String name = rs.getString(4);
                String surname = rs.getString(5);
                String dateOfBirth = rs.getString(6);
                String telephoneNumber = rs.getString(7);
                customers.add(
                    new Customer(
                        id,
                        userName,
                        email,
                        name,
                        surname,
                        LocalDate.parse(dateOfBirth),
                        telephoneNumber));
            }
        } catch (SQLException e) {
            System.err.println("Failed to read ResultSet: " + e.getMessage());
        }
        return customers;
    }
}
```

Pierwsza kwestia to zapamiętania przy stosowaniu metod odwołujących się po indeksach kolumn to to, że numeracja rozpoczyna się o `1`, a nie od `0`. Odwoływanie się po nazwach kolumn jest też czytelniejsze, bo określa w jawny sposób z perspektywy czytającego kod, która konkretnie kolumna nas interesuje.

Skoro poruszyliśmy już jak działa metoda `next()`, to jednocześnie widzimy, że nawet jeżeli `ResultSet` będzie zawierał przykładowo `1000` rezultatów, to możemy w taki sposób napisać kod, aby odczytać tylko jeden, albo tylko `10`. Kwestia tego jak napiszemy pętlę. W przypadku próby odczytu tylko pierwszego rekordu z `ResultSet`, możemy wykorzystać `if` zamiast pętli `while`.

Ważne też jest aby pamiętać, że należy sprawdzić, poprzez metodę `next()` czy kursor w `ResultSet` wskazuje obecnie na wiersz, który posiada dane. Jeżeli tego nie zrobimy i postaramy się wywołać metodę np. `rs.getString("id")`, rezultatem będzie `SQLException`, gdyż w `ResultSet` może nie być danych.

## Co w przypadku dat i czasów?

Wspomniałem, że mamy dostępne metody `get···()`, które pozwalają nam na odczyt typów takich jak `Boolean`, `Byte`, `Short`, `Integer`, `Double`, `Float` oraz `String`. Co ciekawe, nie ma metody `getChar()`, taka ciekawostka. W tym przypadku będziemy używać metody `getString()`. Co natomiast w przypadku dat? Mamy dostępne następujące metody:

| Metoda | Typ zwracany w Javie | Typ bazodanowy | Odpowiedni typ w Java 8 |
|---|---|---|---|
| getDate() | java.sql.Date | DATE | java.time.LocalDate |
| getTime() | java.sql.Time | TIME | java.time.LocalTime |
| getTimestamp() | java.sql.Timestamp | TIMESTAMP | java.time.LocalDateTime |

Jeżeli chcielibyśmy odczytywać daty z `ResultSet` to moglibyśmy to zrobić w następujący sposób:

```
public class DateMapper {

    static LocalDate getDate(final ResultSet rs) {
        try {
            if (rs.next()) {
                Date date = rs.getDate(1);
                return date.toLocalDate();
            }
        } catch (SQLException e) {
            System.err.println("Failed to read ResultSet: " + e.getMessage());
        }
        return null;
    }
}
```

Zwróć uwagę, że mamy dostępną metodę `toLocalDate()` w klasie `java.sql.Date`. Analogicznie, będziemy mieli dostępne metody `toLocalTime()` oraz `toLocalDateTime()` w klasach `java.sql.Time` oraz `java.sql.Timestamp`.

## getObject()

Dodam również, że mamy dostępną taką metodę jak `getObject()`, którą możemy wywołać na `ResultSet`. Wymaga ona od nas później ręcznego sprawdzenia typu, który się kryje w danej kolumnie. Pamiętasz, że w Javie wszystko (oprócz prymitywów) dziedziczy z klasy `Object`? Zatem wszystko może się kryć pod klasą `Object`. Dlatego kod korzystający z metody `getObject()` mógłby wyglądać w ten sposób:

```
public class ObjectMapper {

    static void mapObject(final ResultSet rs) {
        try {
            while (rs.next()) {
                Object id = rs.getObject("id");
                Object name = rs.getObject("name");
                if (id instanceof Integer) {
                    int idAsInt = (Integer) id;
                    System.out.println(idAsInt);
                }
                if (name instanceof String) {
                    String nameAsString = (String) name;
                    System.out.println(nameAsString);
                }
            }
        } catch (SQLException e) {
            System.err.println("Failed to read ResultSet: " + e.getMessage());
        }
```

```
    }
}
```

Jak widzisz, stosowanie tej metody jest o tyle problematyczne, że musimy sami zdecydować na jaki typ dany obiekt ma zostać rzutowany. Aczkolwiek mogą być w praktyce przypadki, kiedy będzie to potrzebne.

# Co daje nam wyjątek SQLException

Tak jak tytuł sekcji wskazuje, co takiego dodaje nam SQLException? Posiada on pewne metody, które pomogą nam namierzyć powód wystąpienia błędu. Aby to zobrazować wywołajmy taki fragment kodu:

```
public class JdbcSqlExceptionExample {

    public static void main(String[] args) {
        getConnection1();
        getConnection2();
    }

    private static Optional<Connection> getConnection1() {
        try {
            Optional<Connection> connection = Optional.of(DriverManager.getConnection(
                "jdbc:postgresql://localhost:5432/nonExistingDatabase",
                "wrongUsername",
                "wrongPassword"
            ));
            System.out.println(connection);
            return connection;
        } catch (SQLException e) {
            System.err.println("Failed to create connection: " + e.getMessage());
        }
        return Optional.empty();
    }

    private static Optional<Connection> getConnection2() {
        try {
            Optional<Connection> connection = Optional.of(DriverManager.getConnection(
                "jdbc:postgresql://localhost:5432/nonExistingDatabase",
                "wrongUsername",
                "wrongPassword"
            ));
            System.out.println(connection);
            return connection;
        } catch (SQLException e) {
            System.err.printf(
                "Failed to create connection, message: [%s], sqlState: [%s], errorCode: [%s]%n",
                e.getMessage(), e.getSQLState(), e.getErrorCode());
        }
        return Optional.empty();
    }
}
```

W przypadku złapania błędu Exception, na ekranie zostanie wydrukowana taka wiadomość:

```
Failed to create connection: FATAL: password authentication failed for user "wrongUsername"
```

Jeżeli natomiast złapiemy wyjątek SQLException, oprócz metody getMessage() mamy jeszcze dostępne metody getSQLState() oraz getErrorCode().Dzięki czemu możemy pobrać kody dające nam informacje o szczegółach błędu. Na podstawie tych kodów możemy poszukać dodatkowych informacji i dowiedzieć się czegoś więcej niż z samej wiadomości błędu.