Bazy Danych

Ćwiczenie 10 – JDBC

Wywoływanie poleceń SQL z języków programowania.

Ćwiczenie 10 – JDBC

Na dotychczasowych ćwiczeniach wykorzystywaliście państwo program pozwalający na bezpośrednie wydawanie poleceń SQL systemowi zarządzania bazą danych. Oczywiście, nie jest to jedyny sposób na korzystanie z SZBD. Celem tego ćwiczenia jest zademonstrowanie państwu API JDBC (ang. *Java Database Connectivity*), które pozwala na komunikowanie się z SZBD bezpośrednio z programów napisanych w języku Java. Dzięki temu możliwe jest wykonywanie zapytań oraz aktualizacji danych za pomocą własnych programów.

Wymagania:

Umiejętność pisania prostych programów w języku Java. Znajomość tematyki omawianej na poprzednich ćwiczeniach, w szczególności umiejętność konstrukcji zapytań i poleceń służących do wstawiania, modyfikacji i usuwania danych w języku SQL.

Plan ćwiczenia

- Wprowadzenie do laboratorium.
- Nawiązywanie i zamykanie połączeń.
- Wykonywanie zapytań .
- Wykonywanie aktualizacji.
- Zarządzanie transakcjami.
- Prekompilowane polecenia.

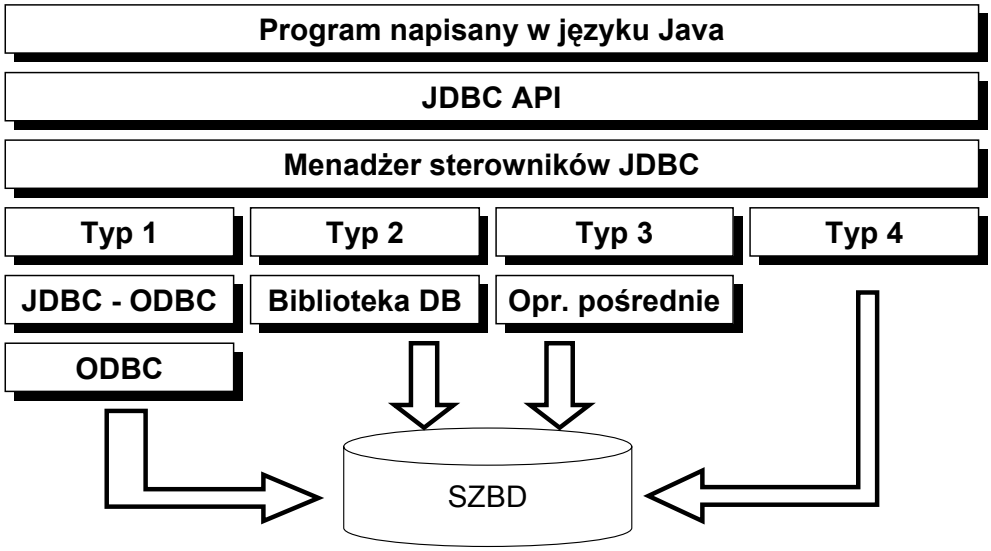
Ćwiczenie rozpoczniemy od wprowadzenia do laboratorium, na którym przedstawimy architekturę JDBC, oraz opiszemy jakie wymagania należy spełnić, aby móc pisać programy korzystające z tego API. Następnie, przedstawimy państwu interfejsy i metody pozwalające na nawiązywanie i zamykanie połączeń z SZBD oraz wykonywanie zapytań i aktualizacji. Omówimy również metody pozwalające na optymalizację realizacji zapytań oraz poleceń aktualizacji danych, takie jak tworzenie i wykorzystywanie prekompilowanych poleceń,...

Plan ćwiczenia – cd.

- Zwiększanie liczby przesyłanych krotek.
- Aktualizowanie wsadowe.
- Modyfikowalne zbiory wyników.
- SQLJ.
- Podsumowanie.

... zwiększanie liczby przesyłanych jednorazowo krotek i (wprowadzowe w JDBC 2.0) wsadowe wykonywanie poleceń aktualizacji. Prócz wsadowego wykonywania operacji aktualizacji, wersja 2.0 API JDBC wprowadza m. in. modyfikowalne zbiory wyników, które również zostaną omówione. Ostatecznie przedstawimy państwu w ogólny sposób standard SQLJ pozwalający na zagnieżdżanie poleceń SQL bezpośrednio w kodzie języka Java. Ćwiczenie zakończymy podsumowaniem. Omawiane na zajęciach tematy będą ilustrowane zadaniami do samodzielnego wykonania.

Wprowadzenie do laboratorium



JDBC (ang. *Java Database Connectivity*) jest API zdefiniowanym dla języka Java, dzięki któremu aplikacja kliencka może nawiązać połączenie z SZBD w celu wykonywania zapytań oraz aktualizacji danych w relacyjnych bazach danych. Rysunek na slajdzie przedstawia architekturę JDBC. Najwyższą warstwę architektury JDBC stanowi program napisany w języku Java, który wykorzystuje JDBC API do komunikacji z SZBD poprzez wywoływanie standardowych, zdefiniowanych w API metod. Wywołania standardowych metod muszą być następnie przekształcone w wywołania charakterystyczne dla SZBD. Służą do tego sterowniki, które muszą zostać zaimplementowane dla konkretnego SZBD. W momencie nawiązywania połączenia z SZBD, menadżer sterowników dobiera odpowiedni dla podanego adresu sterownik, który następnie nawiązuje połączenie z SZBD i przekazuje do niego polecenia użytkownika. Wyróżnia się 4 typy sterowników JDBC:

- Typ 1 sterowników to mostek tłumaczący wywołania JDBC na wywołania ODBC. Dzięki niemu możliwe jest połączenie ze wszystkimi SZBD wspierającymi interfejs ODBC, pod warunkiem, że odpowiednie biblioteki, sterowniki i pliki pomocnicze ODBC są zainstalowane na komputerze klienckim. Sam mostek jest dostarczany razem z Java 2 SDK, ale sterowniki ODBC specyficzne dla SZBD już nie.

- Typ 2 sterowników to sterowniki wykorzystujące interfejs JNI (ang. *Java Native Interface*) do komunikacji z API konkretnego SZBD. Sterowniki te są szybsze od sterowników typu 1, niestety również wymagają one instalacji oprogramowania klienckiego na komputerze.

- Typ 3 sterowników to sterowniki napisane w języku Java wykorzystujące protokół sieciowy do komunikacji z oprogramowaniem pośrednim (ang. m*iddleware*). Wszystkie wywołania JDBC są przesyłane do oprogramowania pośredniego, które następnie tłumaczy te wywołania na wywołania charakterystyczne dla SZBD. Dzięki takiemu rozwiązaniu, sterowniki tego typu nie wymagają żadnego dodatkowego oprogramowania zainstalowanego na komputerze.

- Typ 4 sterowników to sterowniki napisane w języku Java, które komunikują się bezpośrednio z SZBD za pomocą jego własnego protokołu. Nie wymagają one również żadnego dodatkowego oprogramowania.

JDBC doczekało się 3 wersji (1.0, 2.0 i 3.0), a w czasie, kiedy tworzone jest to ćwiczenie, została zgłoszona ostatnia propozycja wersji 4.0. Każda nowa wersja jest kompatybilna z poprzednią i jedynie wprowadza nową funkcjonalność. Na tym ćwiczeniu zapoznacie się państwo z częścią funkcjonalności wersji 1.0 i 2.0 JDBC API.

Aby możliwe było pisanie i uruchamianie programów wykorzystujących JDBC API, konieczne jest, aby w zmiennej CLASSPATH umieszczone były archiwa jar z klasami JDBC (znajdują się w standardowych bibliotekach instalowanych razem z Java 2 SDK), oraz archiwa jar albo ścieżki z klasami sterownika do komunikacji z SZBD. Pliki ze sterownikami są dostarczane razem z SZBD, oraz, bardzo często, można je pobrać ze strony producenta. Poniżej przedstawiono często spotykane nazwy plików ze sterownikami JDBC do różnych SZBD (w ogólności pliki te mogą się nazywać inaczej).

IBM DB2:

db2java.zip (typ 2) albo db2jcc.jar, db2jcc_license_cu.jar (typ 4)

MS/SQL Server/Sybase:

jtds.jar (Typ 4, darmowy - LGPL) albo sqljdbc.jar (Typ 4, stworzony przez Microsoft, są osobne wersje dla Microsoft Windows i Linux)

mySQL :

mysql-connector-java-X.X.XX-bin.jar (X.X.XX – wersja)

Oracle:

Java 1.2 - 1.3: classes12.zip, nlscharset12.zip, ocrs12.zip; Java 1.4 ojdbc1.4.zip, nlscharset12.zip, ocrs12.zip

PostgreSQL:

pgXXjdbc2.jar, postgresql-X.X.-XXX.jdbc2.jar (X.X.XX – wersja)

Nawiązywanie i zamykanie połączeń

① `import java.sql.*;`

②
```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

③
```
Connection con = DriverManager.getConnection(
 "jdbc:oracle:thin:@dblab.cs.put.poznan.pl:1521:dblab10g",
 "elearning_user", "elearning_pass");
```

④ `con.close();`

Większość klas JDBC API znajduje się w pakiecie „java.sql”. Należy zatem pakiet ten zaimportować, aby możliwa była praca z JDBC (1). Pierwszą czynnością po rozpoczęciu programu, którą należy wykonać, jest rejestracja sterownika (2). Rejestrację można wykonać za pomocą instrukcji:

Class.forName(łańcuch_zawierający_nazwę_klasy_sterownika);

W ten sposób można rejestrować dowolną liczbę sterowników. Jeżeli podana nazwa klasy sterownika jest niepoprawna, zgłaszany jest wyjątek „ClassNotFoundException”. Poniżej przedstawiono kilka nazw klas sterowników dla różnych SZBD (nazwy te mogą się zmienić wraz z rozwojem sterownika):

IBM DB2:

„COM.ibm.db2.jdbc.app.DB2Driver” (Typ 2), „com.ibm.db2.jcc.DB2Driver” (Typ 4)

MS/SQL Server/Sybase:

„net.sourceforge.jtds.jdbc.Driver” (dla sterownika jDTS)

„com.microsoft.sqlserver.jdbc.SQLServerDriver„ dla wersji 2005 MS/SQL Server (dla sterownika Microsoftu)

„com.microsoft.jdbc.sqlserver.SQLServerDriver„ dla wersji 2000 MS/SQL Server (dla sterownika Microsoftu)

mySQL:

„com.mysql.jdbc.Driver”

Oracle:

„oracle.jdbc.driver.OracleDriver”

PostgreSQL:

„org.postgresql.Driver”

Po zarejestrowaniu sterownika możliwe jest nawiązanie połączenia z SZBD. Połączenie należy utworzyć za pomocą statycznej metody „getConnection” klasy „DriverManager”. Pierwszym parametrem tej metody jest URL połączenia z SZBD, a drugim i trzecim są odpowiednio: nazwa użytkownika i hasło. Adres URL określa z jakim SZBD należy nawiązać połączenie, oraz jego adres sieciowy. Metoda zwraca obiekt klasy implementującej interfejs „Connection”, który reprezentuje nawiązane połączenie. Jeżeli nawiązanie połączenia z jakiegoś powodu się nie powiedzie, zostanie zgłoszony wyjątek „SQLException”. Format adresu URL potrafi być dość skomplikowany i zawierać wiele różnych opcji połączenia. Poniżej przedstawiono kilka przykładowych adresów URL, w ich najprostszej wersji, dla różnych SZBD:

IBM DB2:

jdbc:db2//server:port/database (Typ 4 sterownika)
jdbc:db2//server/database (Typ 4 sterownika)

jdbc:db2:database (Typ 2 sterownika)

MS/SQL Server/Sybase:

jdbc:jtds:server_type://server:port[/database] (dla sterownika jTDS)

jdbc:sqlserver://server:port;databasename=database (MS/SQL 2005, sterownik Microsoftu)

jdbc:microsoft:sqlserver://server:port;databasename=database (MS/SQL 2000, sterownik Microsoftu)

mySQL:

jdbc:mysql://server:port/database

Oracle:

jdbc:oracle:thin:@//server:port/service (Typ 4 sterownika)

jdbc:oracle:thin:@server:port:sid (Typ 4 sterownika)

jdbc:oracle:oci:@//server:port/service (Typ 2 sterownika)

jdbc:oracle:oci:@server:port:sid (Typ 2 sterownika)

PostgreSQL:

jdbc:postgresql://server:port/database

W powyższych adresach URL, „server”, „port” i „database” oznaczają kolejno: adres serwera, port na którym nasłuchuje serwer oraz nazwa bazy danych. Parametr „server_type” jest specyficzny dla sterownika jTDS i oznacza „sqlserver” lub „sybase”, w zależności od SZBD z którym chcemy nawiązać połączenie. W SZBD Oracle bazę danych można identyfikować na dwa sposoby: albo poprzez SID, albo poprzez nazwę usługi („service”). Stąd, podano wersje URL wykorzystujące jeden i drugi sposób identyfikacji.

Rozważmy przykład (3) na slajdzie. Instrukcja na tym przykładzie rozpoczyna się od deklaracji zmiennej „con” typu „Connection”, której przypisywany jest następnie wynik działania metody „getConnection”. Metodzie „getConnection” przekazano trzy parametry aktualne:

- URL: jdbc:oracle:thin:@dblab.cs.put.poznan.pl:1521:dblab10g oznaczający że połączenie ma zostać nawiązane za pomocą sterownika typu 4, z serwerem bazy danych znajdującym się pod adresem dblab.cs.put.poznan.pl, nasłuchuącym na porcie 1521, z bazą danych o SID dblab10g,
- nazwę użytkownika: elearning_user,
- hasło: elearning_pass.

Powyższe parametry są przykładowe. Podczas wykonywania zadań, należy podać metodzie getConnection parametry odpowiednie dla konfiguracji środowiska, na którym wykonywane są ćwiczenia. Jeżeli parametry te są nieznane, należy się o nie zapytać prowadzącego zajęcia. W wyniku instrukcji przedstawionej na przykładzie (3) do zmiennej „con” zostanie zapisany obiekt reprezentujący nawiązane połączenie.

Połączenie zamykane jest za pomocą metody „close”, którą należy aktywować na rzecz obiektu reprezentującego nawiązane połączenie (4). Jeżeli zamknięcie połączenia nie odbędzie się w sposób prawidłowy, zgłaszany jest wyjątek „SQLException”.

Pełny kod programu, którego fragmenty przedstawiono na slajdzie, załączono do kursu w pliku JDBCELearning1.java.

Wykonywanie zapytań

① `Statement stmt = con.createStatement() ;`

②
```
ResultSet rs=stmt.executeQuery(
  "SELECT nazwisko,placa_pod FROM pracownicy");
```

③
```
while (rs.next()) {
  String nazwisko=rs.getString("NAZWISKO");
  float placa=rs.getFloat(2);
  System.out.println(nazwisko+" "+placa);
}
```

④
```
rs.close();
stmt.close();
```


Po nawiązaniu połączenia, kolejnym krokiem jaki należy wykonać, jest utworzenie obiektu implementującego interfejs „Statement”. Jest to obiekt związany z konkretnym połączeniem, za pomocą którego JDBC wykonuje polecenia SQL w SZBD. Obiekt ten można utworzyć za pomocą bezparametrowej metody „createStatement” interfejsu „Connection”. Rozważmy przykład (1). Instrukcja przedstawiona na tym przykładzie rozpoczyna się od deklaracji zmiennej „stmt” typu „Statement”. Do tej zmiennej przypisywany jest następnie wynik działania metody „createStatement” aktywowanej na rzecz obiektu reprezentującego nawiązane połączenie (patrz poprzedni slajd). W wyniku instrukcji przedstawionej na przykładzie (1) w zmiennej „stmt” znajdzie się obiekt pozwalający na wykonywanie -poleceń SQL w SZBD.

Omawianie sposobów wykonywania poleceń SQL rozpoczniemy od omówienia metod pozwalających na wykonywanie zapytań do baz danych i odczytywanie ich wyników. Aby wykonać zapytanie należy użyć metody „executeQuery” interfejsu „Statement”. Jest to metoda, której jedynym parametrem jest łańcuch zawierający treść polecenia SELECT. Metoda „executeQuery” zwraca obiekt implementujący interfejs „ResultSet”, który reprezentuje otrzymany w wyniku zapytania zbiór krotek. Jeżeli przy wykonywaniu zadanego zapytania wystąpi problem, zgłaszany jest wyjątek „SQLException”. Sposób użycia metody executeQuery przedstawiono na przykładzie (2). Instrukcja na tym przykładzie rozpoczyna się od deklaracji zmiennej „rs” typu „ResultSet”, której przypisywany jest wynik działania metody „executeQuery”. Metoda „executeQuery” jest aktywowana na rzecz utworzonego wcześniej obiektu zapisanego w zmiennej „stmt”, który pozwala na wykonywanie poleceń SQL w SZBD. Metoda „executeQuery” przesyła zapytanie, przekazane jako parametr aktualny, do SZBD i zwraca obiekt reprezentujący zbiór krotek otrzymanych w wyniku zapytania. Zwróćmy uwagę na zapytanie przekazane jako parametr. Jest to proste zapytanie, którego zadaniem jest odszukanie nazwisk i płac wszystkich pracowników.

Ciekawą rzeczą jest to, iż polecenie SQL nie jest zakończone średnikiem. Użycie tutaj średnika zakończyłoby się błędem i dlatego w JDBC nie należy używać średników do zakończenia polecenia SQL.

Obiekt otrzymany w wyniku działania metody „executeQuery” zawiera metody pozwalające na odczytanie wyniku zapytania. Obiekt ten jest rodzajem iteratora, którego wskaźnik, zaraz po wykonaniu zapytania, znajduje się przed pierwszą krotką wyniku. Aby przesunąć wskaźnik należy użyć metody „next”, która przesuwa wskaźnik na kolejną krotkę wyniku (lub pierwszą przy pierwszym wykonaniu). Metoda „next” zwraca wartość logiczną „true”, jeśli wskaźnik został przesunięty na prawidłową krotkę, bądź „false”, jeśli został przesunięty poza ostatnią krotkę. Dzięki zwracanym przez metodę „next” wartościom można użyć jej wywołania, jako warunku zakończenia pętli „while”, której celem jest przejrzenie całego wyniku zapytania (patrz przykład (3)). Kiedy wskaźnik wskazuje na poprawną krotkę, można odczytać wartości jej atrybutów za pomocą metod interfejsu „ResultSet”, o nazwach „getXXX”, gdzie XXX jest nazwą typu zwracanej wartości. Należy stosować odpowiednie metody „getXXX”, dla odpowiednich atrybutów, aby otrzymać wartości odpowiednich typów. Poniżej podano sugerowane metody „getXXX”, dla różnych typów SQL:

- „getString” (typ zwracanej wartości: „String”) :

Można użyć praktycznie dla każdego typu SQL, żeby otrzymać wartość atrybutu w postaci łańcucha. W ogólności jednak najlepiej stosować tą metodę dla typów: CHARACTER(n), CHAR(n), CHARACTER VARYING(n), CHAR VARYING(n), NATIONAL CHARACTER(n), NATIONAL CHAR(n), NATIONAL CHARACTER VARYING (n), NATIONAL CHAR VARYING(n), NCHAR VARYING(n). Jeżeli atrybut zawiera wartość NULL, to metoda zwraca wartość „null” (pusty wskaźnik w języku Java).

- „getShort” (typ zwracanej wartości: „short”):

Należy używać dla atrybutów całkowitoliczbowych, których wartości mieszczą się w przedziale <-32768,32767>. Sugerowane jest używanie tej metody dla atrybutów typu SMALLINT. Jeżeli atrybut zawiera wartość NULL zwracane jest 0.

- „getInt” (typ zwracanej wartości: „int”):

Należy używać dla atrybutów całkowitoliczbowych, których wartości mieszczą się w przedziale <-2147483648,2147484647>. Sugerowane jest używanie tej metody dla atrybutów typów: INTEGER i INT. Jeżeli atrybut zawiera wartość NULL zwracane jest 0.

- „getFloat” (typ zwracanej wartości: „float”):

Należy używać dla atrybutów przechowujących wartości liczbowe zmiennoprzecinkowe o pojedynczej precyzji. Sugerowane jest używanie tej metody dla atrybutów typu REAL. Jeżeli atrybut zawiera wartość NULL zwracane jest 0.

- „getDouble” (typ zwracanej wartości: „double”):

Należy używać dla atrybutów przechowujących wartości liczbowe zmiennoprzecinkowe o zwiększonej precyzji. Sugerowane jest używanie tej metody dla atrybutów typów FLOAT(b) i DOUBLE PRECISION. Jeżeli atrybut zawiera wartość NULL zwracane jest 0.

- „getBigDecimal” (typ zwracanej wartości: „java.math.BigDecimal”):

Należy używać dla atrybutów liczbowych o definiowanej przez użytkownika precyzji. Sugerowane jest używanie tej metody dla atrybutów typów NUMERIC(p,s) i DECIMAL (p,s). Jeżeli atrybut zawiera wartość NULL, to metoda zwraca wartość „null” (pusty wskaźnik w języku Java).

-„getTimestamp” (typ zwracanej wartości: „java.sql.Timestamp”):

Należy używać dla atrybutów typu zezwalającego na przechowywanie zarówno dat jak i czasu. Sugerowane jest używanie tej metody dla atrybutów typu TIMESTAMP(n) i DATE (w SZBD Oracle). Jeżeli atrybut zawiera wartość NULL, to metoda zwraca wartość „null” (pusty wskaźnik w języku Java).

- „getDate” (typ zwracanej wartości: „java.sql.Date”):

Należy używać dla atrybutów przechowujących daty. Sugerowane jest używanie tej metody dla atrybutów typu DATE (za wyjątkiem SZBD Oracle). Jeżeli atrybut zawiera wartość NULL, to metoda zwraca wartość „null” (pusty wskaźnik w języku Java).

- „getTime” (typ zwracanej wartości: „java.sql.Time”):

Należy używać dla atrybutów przechowujących czas. Sugerowane jest używanie tej metody dla atrybutów typu TIME(n). Jeżeli atrybut zawiera wartość NULL, to metoda zwraca wartość „null” (pusty wskaźnik w języku Java).

- „getBlob” (typ zwracanej wartości „java.sql.Blob”):

Należy używać dla atrybutów typu BLOB. Jeżeli atrybut zawiera wartość NULL, to metoda zwraca wartość „null” (pusty wskaźnik w języku Java).

- „getClob” (typ zwracanej wartości „java.sql.Clob”):

Należy używać dla atrybutów typu CLOB. Jeżeli atrybut zawiera wartość NULL, to metoda zwraca wartość „null” (pusty wskaźnik w języku Java).

Dla typu INTERVAL nie przewidziano w JDBC odpowiednich metod ani typów. Istnieją rozwiązania specyficzne dla producentów konkretnych SZBD, jednak nie będziemy ich opisywać. W ogólności możliwe jest używanie innych niż sugerowane metod dla różnych atrybutów (np. „getFloat” dla atrybutów typu NUMERIC), należy jednak w takiej sytuacji zwrócić szczególną uwagę na to, by w wyniku zapytania nie zostały zwrócone wartości, które nie mieszczą się w zmiennych typu zwracanego przez metodę „getXXX”.

Każda z wyżej wymienionych metod przyjmuje jeden parametr pozwalający na określenie, którego atrybutu wartość należy pobrać. Możliwe jest albo podanie nazwy atrybutu w postaci łańcucha, albo numeru atrybutu w klauzuli SELECT liczonego od 1. Jeżeli podczas pobierania wartości atrybutu wystąpi błąd, zgłaszany jest wyjątek „SQLException”.

Należy tutaj zwrócić uwagę na jeszcze jedną rzecz. Niektóre z metod „getXXX” zwracają wartość 0 zamiast „null”, gdyż typ zwracanych przez nie wartości, nie przewiduje wartości „null”. Aby sprawdzić, czy ostatnia odczytana za pomocą metody „getXXX” wartość zawierała NULL, należy użyć bezparametrowej metody „wasNULL”, która zwraca wartość logiczną „true”, jeśli w atrybucie faktycznie znajdował się NULL, albo „false”, jeśli w atrybucie znajdowała się wartość 0.

Rozważmy przykład (3). Przykład ten zawiera pętlę „while”, w której warunkiem zakończenia jest przyjęcie przez omawianą wcześniej metodę „next” wartości „false”. Zgodnie z tym, co mówiono o metodzie „next” wcześniej, w kolejnych iteracjach pętli wskaźnik zapisany wewnątrz obiektu zapisanego w zmiennej „rs”, będzie wskazywał na kolejne krotki wyniku. Pętla zakończy się w momencie, kiedy zostaną odczytane wszystkie krotki. Rozważmy pierwszą instrukcję wewnątrz pętli:

```
String nazwisko=rs.getString("NAZWISKO");
```

Instrukcja rozpoczyna się od deklaracji zmiennej nazwisko typu „String”, do której przypisywany jest wynik działania metody „getString” aktywowanej na rzecz obiektu zapisanego zmiennej „rs”. Parametrem tej metody jest nazwa atrybutu relacji wynikowej zapytania. W tym przypadku jest to „NAZWISKO”. W wyniku działania metody „getString”, nazwisko z aktualnie wskazywanej krotki w wyniku zapytania jest zapisywane do zmiennej nazwisko. Druga instrukcja wewnątrz pętli ma analogiczną konstrukcję:

```
float placa=rs.getFloat(2);
```

Instrukcja rozpoczyna się od deklaracji zmiennej „placa” typu „float”, do której następnie przypisywany jest wynik działania metody „getFloat”. Parametrem tej metody jest numer atrybutu relacji wynikowej zapytania (podano tutaj 2, czyli atrybut PLACA_POD). W wyniku działania metody „getFloat”, płaca podstawowa z aktualnie wskazywanej krotki w wyniku zapytania jest zapisywana do zmiennej „placa”. W ostatnim wierszu pętli, na konsoli wypisywane są odczytane wyniki.

Po przeczytaniu wszystkich wyników, trzeba w sposób jawny zamknąć zbiór wyników za pomocą metody „close”. Jeżeli nie mamy zamiaru wykonywać kolejnych poleceń SQL, należy zamknąć za pomocą metody „close” również obiekt implementujący interfejs „Statement” (4). Aby wytłumaczyć konieczność jawnego zamykania zbioru wyników i obiektu pozwalającego na wysyłanie poleceń do SZBD, należy wprowadzić termin „kursor”. Kursorem nazywamy obszar pamięci operacyjnej na serwerze SZBD 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 obiekt „Statement” przypada jeden kursor, a zatem może na niego przypadać tylko jeden obiekt ResultSet.

Pełny kod programu, którego fragmenty przedstawiono na slajdzie, załączono do kursu w pliku JDBCELearning2.java.

Zadanie (1)

- Napisz program, który wyświetla następujące informacje:

```
Zatrudniono X pracowników, w tym:
 X1 w zespole Y1,
 X2 w zespole Y2,
 ....
Ranking etatów według średniej płacy:
 1. Etat1
 2. Etat2
 ....
```

- W miejsce X, X_i, Y_i i $Etat_i$ wpisz dane odczytane z bazy danych.

Rozwiązanie (1)

```
ResultSet rs = stmt.executeQuery(
 "SELECT COUNT(*) FROM pracownicy");

rs.next();
System.out.println("Zatrudniono " + rs.getInt(1) +
  " pracowników, w tym:");
rs.close();
```

```
rs = stmt.executeQuery(
  "SELECT COUNT(nazwisko),nazwa " +
  "FROM pracownicy NATURAL RIGHT JOIN zespoly " +
  "GROUP BY nazwa");
while (rs.next())
  System.out.println(" " + rs.getInt(1) + " w zespole " +
    rs.getString(2));
rs.close();
```


Na tym i kolejnym slajdzie przedstawiono najważniejsze fragmenty programu stanowiącego rozwiązanie zadania (1), którego treść przedstawiono poniżej. Pełny program umieszczono w pliku JDBCELearningZad1.java, który został załączony do kursu.

Napisz program, który wyświetla następujące informacje:

Zatrudniono X pracowników, w tym:

X1 w zespole Y1,

X2 w zespole Y2,

....

Ranking etatów według średniej płacy:

1. Etat1
2. Etat2

....

W miejsce X, X_i, Y_i i $Etat_i$ wpisz dane odczytane z bazy danych.

Rozwiązanie (1) – cd.

```
rs = stmt.executeQuery(
  "SELECT etat FROM pracownicy " +
  "GROUP BY etat ORDER BY AVG(placa_pod) DESC");

System.out.println("Ranking etatów według średniej
  płacy:");
int i=1;
while (rs.next()) {
  System.out.println(" "+i+". " + rs.getString(1));
  i++;
}
rs.close();
```

Wykonywanie aktualizacji danych

```
Statement stmt = con.createStatement() ;
int changes;
①
changes=stmt.executeUpdate(
  "INSERT INTO pracownicy(id_prac,nazwisko)"+
  "VALUES(300,'Zieliński')");
System.out.println("Wstawiono "+changes+" krotek.");
②
changes=stmt.executeUpdate("UPDATE pracownicy " +
  "SET placa_pod=placa_pod*1.5");
System.out.println("Zmodyfikowano "+changes+" krotek.");
③
changes=stmt.executeUpdate(
  "DELETE FROM pracownicy WHERE id_prac=300");
System.out.println("Usunieto "+changes+" krotek.");

stmt.close();
```


Wykonywanie aktualizacji danych jest analogiczne do wykonywania zapytań. Polecenia aktualizacji danych należy wykonywać za pomocą metody „executeUpdate” interfejsu „Statement”. Metoda ta przyjmuje jako parametr łańcuch zawierający polecenie DML, a zwraca liczbę utworzonych, zmodyfikowanych bądź usuniętych krotek. Przeanalizujmy przykłady na slajdzie. W przykładowym fragmencie programu tworzony jest obiekt typu „Statement”, który zostanie wykorzystany następnie do przesłania do SZBD poleceń DML oraz deklarowana jest zmienna „changes” typu „int”, której będą przypisywane liczby zmian w bazie danych. Rozważmy przykład (1). W przykładzie tym, jako parametr metody „executeUpdate” podano polecenie wstawiające nowego pracownika do relacji PRACOWNICY. Jak łatwo zauważyć, polecenie to nie różni się składnią od poleceń INSERT konstruowanych przez państwa poprzednio, za wyjątkiem tego, że na końcu polecenia nie napisano średnika. W wyniku wykonania metody „executeUpdate”, do relacji PRACOWNICY zostanie wstawiony pracownik, a liczba wstawionych krotek zapisana do zmiennej „changes”. Przykłady (2) i (3) są skonstruowane analogicznie i wykorzystuję ten sam obiekt typu „Statement”, który został wykorzystany do pierwszego polecenia aktualizacji. Na końcu przykładowego fragmentu programu obiekt ten jest zamykany w celu zwolnienia kursora.

Polecenia DDL można również wykonywać za pomocą metody „executeUpdate”. Jedyną 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.

Pełny kod programu, którego fragmenty przedstawiono na slajdzie załączono do kursu w pliku JDBCELearning3.java.

Zadanie (2)

- 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).

(!) Sekwencja utworzona w tym zadaniu przyda się również przy wykonywaniu kolejnych zadań.

Rozwiązanie (2)

CREATE SEQUENCE seqprac **START WITH** 300 **INCREMENT BY** 1;

```
for (int i=0;i<zwolnienia.length;i++){
  stmt.executeUpdate("DELETE FROM pracownicy "+
    "WHERE id_prac="+zwolnienia[i]);
}

for (int i=0;i<zatrudnienia.length;i++) {
  stmt.executeUpdate(
   "INSERT INTO pracownicy(id_prac,nazwisko) " +
   "VALUES (seqprac.nextval,'"+zatrudnienia[i]+"')");
}
```


Na slajdzie przedstawiono najważniejsze fragmenty programu stanowiącego rozwiązanie zadania (2), którego treść przedstawiono poniżej. Pełny program umieszczono w pliku JDBCELearningZad2.java, który został załączony do kursu.

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).

Zarządzanie transakcjami

```
private static int ilePracownikow(Connection con)
  throws SQLException {
  int result;
  Statement stmt=con.createStatement();
  ResultSet rs=stmt.executeQuery(
    "SELECT COUNT(*) FROM pracownicy");
  rs.next();
  result=rs.getInt(1);
  rs.close();
  stmt.close();
  return result;
}
```


W celu zademonstrowania działania metod pozwalających na zarządzanie transakcjami wykorzystana zostanie metoda „ilePracownikow”, której implementację przedstawiono na slajdzie. Metoda ta, jako parametr dostaje obiekt reprezentujący połączenie z SZBD, tworzy obiekt typu „Statement”, który następnie wykorzystuje do wykonania zapytania obliczającego liczbę krotek w relacji pracownicy. Po wykonaniu zapytania, metoda odczytuje wynik do zmiennej „result”, zamyka zbiór wynikowy i obiekt typu „Statement”, i na końcu zwraca wynik zapytania.

Zarządzanie transakcjami – cd.

① ② ③ ④

```
con.setAutoCommit(false);
Statement stmt=con.createStatement();
System.out.println(ilePracownikow(con)); // 14

changes=stmt.executeUpdate(
  "DELETE FROM pracownicy WHERE id_prac=150");
System.out.println("Usunieto "+changes+" krotek.");
con.rollback();
System.out.println(ilePracownikow(con)); //14

changes=stmt.executeUpdate(
  "DELETE FROM pracownicy WHERE id_prac=150");
System.out.println("Usunieto "+changes+" krotek.");
con.commit();
System.out.println(ilePracownikow(con)); //13
```


W ramach jednego połączenia z SZBD równocześnie może być wykonywana tylko jedna transakcja. Dotychczas, wszystkie wykonywane polecenia SQL były automatycznie zatwierdzane, a zatem każda transakcja składała się tylko z jednego polecenia SQL. Aby wyłączyć automatyczne zatwierdzanie, należy użyć metody „setAutoCommit” interfejsu „Connection”. Metoda ta posiada jeden parametr typu „boolean”, który określa, czy automatyczne zatwierdzanie ma być włączone („true”) czy nie („false”). Sposób użycia tej metody przedstawiono na przykładzie (1). Przykład (2) wypisuje na konsoli liczbę pracowników zapisanych w relacji PRACOWNICY. Liczba ta jest konieczna w celu zademonstrowania działania kolejnych metod. Jeżeli w relacji PRACOWNICY nie wprowadzono żadnych zmian, powinna zostać wypisana liczba 14. Kolejnymi, po „setAutoCommit”, metodami interfejsu „Connection”, służącymi do zarządzania transakcjami, są metody „rollback” i „commit”, służące odpowiednio do wycofywania i zatwierdzania zmian wykonanych w ramach transakcji. Przykłady (3) i (4) demonstrują ich użycie. Rozpocznijmy od przykładu (3). W przykładzie tym wykonywane jest polecenie DELETE usuwające jednego pracownika z bazy danych, a następnie wypisywana jest liczba usuniętych krotek. Jeżeli w relacji PRACOWNICY nie wprowadzono żadnych zmian, powinien zostać usunięty jeden pracownik. W kolejnym kroku aktywowana jest metoda „rollback” na rzecz obiektu reprezentującego połączenie z bazą danych. W wyniku działania tej metody, zmiany wykonane w relacji PRACOWNICY są wycofywane, i kolejne polecenie wypisuje liczbę pracowników równą 14, pomimo tego, że jedna krotka została w miedzy czasie usunięta. Przykład (4) jest taki sam jak przykład (3), z jedną różnicą: zamiast metody rollback, użyto metody commit. Tym razem zmiany w relacji PRACOWNICY zostają zatwierdzone, czego dowodem jest liczba pracowników wypisana przez ostatnie polecenie tego przykładu (13 pracowników).

Pełny kod programu, którego fragmenty przedstawiono na tym i poprzednim slajdzie, załączono do kursu w pliku JDBCELearning4.java.

Zadanie (3)

- 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.

Rozwiązanie (3)

```
private static void listaEtatow(Connection con)
  throws SQLException {
  Statement stmt=con.createStatement();
  ResultSet rs=stmt.executeQuery(
    "SELECT nazwa FROM etaty");

  while (rs.next()) {
    System.out.println(rs.getString(1));
  }

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


Na tym, i kolejnych dwóch slajdach przedstawiono najważniejsze fragmenty programu stanowiącego rozwiązanie zadania (3), którego treść przedstawiono poniżej. Pełny program umieszczono w pliku JDBCELearningZad3.java, który został załączony do kursu.

-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.

Rozwiązanie (3) – cd.

```
con.setAutoCommit(false);
System.out.println("Zawartosc tabeli ETATY na poczatku "+
  "zadania:");
listaEtatow(con);
stmt.executeUpdate("INSERT INTO etaty(nazwa) VALUES "+
  "('NADREKTOR')");
System.out.println("Zawartosc tabeli ETATY po "+
  "wstawieniu etatu:");
listaEtatow(con);
con.rollback();
...
```

Rozwiązanie (3) – cd.

```
...
System.out.println("Zawartosc tabeli ETATY "+
  "po wycofaniu transakcji:");
listaEtatow(con);
stmt.executeUpdate("INSERT INTO etaty(nazwa) "+
  "VALUES ('NADREKTOR')");
con.commit();
System.out.println("Zawartosc tabeli ETATY po "+
  "wstawieniu nowego etatu i zatwierdzeniu transakcji:");
listaEtatow(con);
```

Prekompilowane polecenia

```
①  PreparedStatement stmt = con.prepareStatement(
     "SELECT nazwisko FROM pracownicy WHERE id_prac=?");

   ResultSet rs;
②  stmt.setInt(1,140); rs=stmt.executeQuery();
   while (rs.next()) {
     System.out.println(rs.getString("NAZWISKO"));
   }
   rs.close();

③  stmt.setInt(1,170); rs=stmt.executeQuery();
   while (rs.next()) {
     System.out.println(rs.getString("NAZWISKO"));
   }
   rs.close();
   stmt.close();
```


Na dotychczasowych slajdach omówiono podstawy pracy z JDBC. Obecnie przejdziemy do omówienia kilku mechanizmów pozwalających na zwiększenie wydajności pracy z SZBD. Pierwszym z tych mechanizmów jest tworzenie prekompilowanych poleceń. Większość aplikacji współpracujących z 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. Te operacje zajmują bardzo często nawet 90% czasu realizacji zapytania. Problem ten można rozwiązać tworząc prekompilowane polecenia, w którym można zmieniać jedynie pewne parametry. W JDBC istnieje możliwość tworzenia tego typu poleceń. Każde prekompilowane polecenie jest reprezentowane przez obiekt typu „PreparedStatement” (odpowiednik obiektów typu „Statement” z poprzednich slajdów). Aby utworzyć taki obiekt, należy użyć metody „prepareStatement” interfejsu „Connection”. Jedynym parametrem tej metody jest treść polecenia SQL, w której wszystkie literały, które zmieniają się pomiędzy wywołaniami tego polecenia zastąpiono znakami zapytania. Rozważmy przykład (1). W przykładzie tym deklarowana jest zmienna „stmt” typu „PreparedStatement”, której następnie przypisywany jest wynik działania metody „prepareStatement”. Jako parametr aktualny tej metody przekazano następujące zapytanie: „SELECT nazwisko FROM pracownicy WHERE id_prac=?”. Jak łatwo zauważyć, jest to zapytanie odczytujące nazwisko pracownika, którego identyfikator zostanie podany w klauzuli WHERE. Tutaj, wartość identyfikatora zastąpiono znakiem zapytania, co pozwoli na późniejsze wstawianie w to miejsce różnych wartości. W wyniku działania procedury „prepareStatement” zapytanie zostanie wysłane do SZBD i skompilowane, oraz zostanie zarezerwowany kursor na potrzeby późniejszego jego wykonania.

Aby przypisać w miejsce znaku zapytania konkretną wartość należy użyć jednej z metod „setXXX” interfejsu „PreparedStatement”, gdzie XXX oznacza typ wartości zapisywanej do zapytania. Metody te mają nazwy analogiczne do metod „getXXX” stosowanych przy odczytywaniu wyników zapytań. Znajdują tutaj również zastosowanie sugestie dotyczące stosowania odpowiednich metod do odpowiednich typów ANSI SQL.. Pierwszym parametrem metody „setXXX” jest zawsze numer znaku zapytania w prekompilowanym poleceniu SQL (liczony od 1). Drugim parametrem jest wartość, która ma zostać zapisana w miejsce odpowiedniego znaku zapytania. Typ wartości zależy od użytej metody „setXXX”. Kiedy wszystkim znakom zapytania zostanie przypisana odpowiednia wartość można wykonać polecenie za pomocą bezparametrowej metody „executeQuery” (dla zapytań), bądź „executeUpdate” (dla poleceń aktualizacji danych). W wyniku działania tych metod otrzymujemy odpowiednio: obiekt typu „ResultSet”, bądź liczbę typu „int” oznaczającą liczbę wprowadzonych modyfikacji. Rozważmy przykład (2). W przykładzie tym, za pomocą metody „setInt” pierwszemu (i jedynemu) znakowi zapytania przypisujemy wartość 140, a następnie wykonujemy prekompilowane zapytanie. W wyniku wykonania zapytania otrzymujemy zbiór wyników, który znaną już państwu pętlą wypisywany jest na konsoli. Ostatecznie zbiór wyników jest zamykany. Przykład (3) jest identyczny. Jedyną różnicą jest tutaj to, iż za pomocą metody „setInt” podano tutaj inną wartość identyfikatora (170). Ponowne wykonanie zapytania zwróci inny wynik. Jak łatwo zauważyć, na tym przykładzie, jedno polecenie jest wielokrotnie wykonywane, bez konieczności wielokrotnej kompilacji.

Dodatkową zaletą prekompilowanych poleceń jest to, iż programy napisane z ich użyciem są odporne na ataki typu *SQL injection*.

Pełny kod programu, którego fragmenty przedstawiono na slajdzie, załączono do kursu w pliku JDBCELearning5.java.

Prekompilowane polecenia – cd.

```
①
PreparedStatement stmt = con.prepareStatement(
  "UPDATE pracownicy SET placa_pod= ?, etat = ?
   WHERE id_prac= ?");

②
stmt.setFloat(1,2000);
stmt.setString(2,"PROFESOR");
stmt.setInt(3,140);
int changes =stmt.executeUpdate();

System.out.println("Zmodyfikowano "+changes+
  " krotek");

③
stmt.close();
```


Ten slajd pokazuje przykładowe użycie prekompilowanych poleceń do aktualizacji danych. Na przykładzie (1) tworzone jest prekompilowane polecenie UPDATE, które przypisuje nową pensję i etat pracownikowi o podanym identyfikatorze. Na przykładzie (2) kolejnym znakom zapytania przypisywane są odpowiednie wartości. Pierwszemu znakowi zapytania, który reprezentuje nową płacę przypisywana jest wartość 2000, drugiemu nowy etat (PROFESOR), a trzeciemu przypisywany jest identyfikator pracownika, który ma zostać zmodyfikowany. Następnie, za pomocą bezparametrowej metody „executeUpdate”, wykonywane jest prekompilowane polecenie aktualizacji. W wyniku działania metody otrzymujemy liczbę zmodyfikowanych krotek. Ostatecznie, zamykany jest obiekt typu „PreparedStatement” i zwalniany jest zajmowany przez niego kursor, za pomocą metody „close” (3).

Pełny kod programu, którego fragmenty przedstawiono na slajdzie, załączono do kursu w pliku JDBCELearning6.java.

Zadanie (4)

- (!) Spróbuj wstawić po 2000 pracowników korzystając z normalnego sposobu wykonywania poleceń („Statement”), oraz wykorzystując mechanizm prekompilowanych poleceń („PreparedStatement”). Całe zadanie wykonaj w ramach jednej transakcji.
  - Wykonaj pomiary czasu potrzebnego do wykonania jednego i drugiego wstawiania.
  - Hint: 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;
```


(!). Jeżeli to zadanie będzie powtarzane wielokrotnie, może łatwo dojść do przekroczenia przez sekwencję maksymalnej dopuszczalnej wartości klucza podstawowego relacji PRACOWNICY. Dlatego też warto, przed wykonaniem tego ćwiczenia, zwiększyć zakres wartości klucza podstawowego, przykładowo, za pomocą polecenia:

ALTER TABLE pracownicy MODIFY id_prac numeric(10);

Uwaga! Podano tutaj liczbę 2000 pracowników. W zależności od konfiguracji serwera SZBD na którym zadanie jest wykonywane, liczba ta może być za duża (całe program będzie działał zbyt długo), bądź za mała (nie będzie widać wyraźnej przewagi poleceń prekompilowanych nad zwykłymi). Jeżeli zatem za pierwszym wykonaniem okaże się, że liczba ta jest niepoprawna, należy dobrać inną samemu.

Rozwiązanie (4)

```
con.setAutoCommit(false);
long start=System.nanoTime();
Statement stmt = con.createStatement();
for (int i=0;i<2000;i++) {
  stmt.executeUpdate(
    "INSERT INTO pracownicy(id_prac, nazwisko) " +
    "VALUES (seqprac.nextval,'Pracownik"+i+"')");
}
long time=System.nanoTime()-start;
System.out.println("Wstawianie zajelo "+
  (double)time/1000000000 + " sekund.");
stmt.close();
```


Na tym, i kolejnym slajdzie przedstawiono najważniejsze fragmenty programu stanowiącego rozwiązanie zadania (4), którego treść przedstawiono poniżej. Pełny program umieszczono w pliku JDBCELearningZad4.java, który został załączony do kursu.

Spróbuj wstawić po 2000 pracowników korzystając z normalnego sposobu wykonywania poleceń („Statement”), oraz wykorzystując mechanizm prekompilowanych poleceń („PreparedStatement”). Całe zadanie wykonaj w ramach jednej transakcji. Wykonaj pomiary czasu potrzebnego do wykonania jednego i drugiego wstawiania.

Rozwiązanie (4) – cd.

```
start=System.nanoTime();
PreparedStatement pstmt=con.prepareStatement(
  "INSERT INTO pracownicy(id_prac, nazwisko) " +
  "VALUES (seqprac.nextval,?)");
for (int i=2000;i<4000;i++) {
  pstmt.setString(1,"Pracownik"+i);
  pstmt.execute();
}
time=System.nanoTime()-start;
System.out.println("Wstawianie zajelo" +
  (double)time/1000000000 + " sekund.");
pstmt.close();
```

Aktualizowanie wsadowe

①
```
PreparedStatement stmt = con.prepareStatement(
  "UPDATE pracownicy SET placa_pod=placa_pod * ?
   WHERE id_prac = ? ");
```

②
```
stmt.setFloat(1,new Float(0.9));
stmt.setInt(2,130);
stmt.addBatch();

stmt.setFloat(1,new Float(1.5));
stmt.setInt(2,150);
stmt.addBatch();
```

③
```
int []changes =stmt.executeBatch();
System.out.println("Zmodyfikowano"+changes[0]+",„
  +changes[1]+" krotek");
```


W wersji 2.0 JDBC wprowadzono nową funkcjonalność, która pozwala na wsadowe wykonywanie poleceń aktualizacji. Rozważmy przykładowy fragment programu na slajdzie. Przykład (1) pokazuje instrukcję przygotowującą prekompilowane polecenie UPDATE które pozwala na danie podwyżki zadanemu pracownikowi. Przykład (2) demonstruje utworzenie zbioru dwóch instrukcji aktualizacji. Za pomocą metod „setXXX” przypisywane są odpowiednie wartości dla kolejnych znaków zapytania w poleceniu. Po uzupełnieniu wszystkich wartości zapytania aktywowana jest metoda „addBatch” interfejsu „PreparedStatement”, która powoduje zapamiętanie danego zestawu wartości. Następnie, za pomocą tych samych metod przygotowywany jest kolejny zestaw wartości. Kiedy utworzone zostaną wszystkie zestawy wartości, aktualizacje są uruchamiane za pomocą metody „executeBatch” (3). W wyniku działania „executeBatch” zwracana jest tablica wartości „int” reprezentująca liczby modyfikacji wykonanych przez kolejne polecenia aktualizacji. Prócz dodatnich wartości liczbowych, mogą się w tablicy pojawić wartości ujemne reprezentowane przez stałe „Statement.EXECUTE_FAILED” (wykonanie polecenie nie powiodło się) i „Statement.SUCCESS_NO_INFO” (nie zwrócono żadnych wyników, ale polecenie powiodło się). Aby usunąć już utworzony zbiór poleceń należy użyć metody „clearBatch” interfejsu „PreparedStatement”.

Pełny kod programu, którego fragmenty przedstawiono na slajdzie, załączono do kursu w pliku JDBCELearning7.java.

Zadanie (5)

- Dane są nastepujące tablice opisujące nowych pracowników:

```
String [] nazwiska={„Woźniak”, „Dąbrowski”,
  „Kozłowski”};
int [] place={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 wsadowej aktualizacji.

Rozwiązanie (5)

```
PreparedStatement pstmt=con.prepareStatement(
  "INSERT INTO pracownicy(id_prac,nazwisko,placa_pod,etat)"+
  "VALUES (seqprac.nextval,?,?,?)");

for (int i=0;i<nazwiska.length;i++) {
  pstmt.setString(1,nazwiska[i]);
  pstmt.setFloat(2,place[i]);
  pstmt.setString(3,etaty[i]);
  pstmt.addBatch();
}

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


Na slajdzie przedstawiono najważniejsze fragmenty programu stanowiącego rozwiązanie zadania (5), którego treść przedstawiono poniżej. Pełny program umieszczono w pliku JDBCELearningZad5.java, który został załączony do kursu.

Dane są nastepujące tablice opisujące nowych pracowników:

String [] nazwiska={„Woźniak”, „Dąbrowski”, „Kozłowski”};

int [] place={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 wsadowej aktualizacji.

Zwiększanie przesyłanej liczby krotek

①
```
Statement stmt = con.createStatement();
((OracleStatement)stmt).setRowPrefetch(10);
```

②
```
ResultSet rs=stmt.executeQuery(
  "SELECT nazwisko FROM pracownicy");

while (rs.next()) System.out.println(rs.getString(1));

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


Kiedy wykonywane jest zapytanie, z reguły wszystkie krotki stanowiące jego wynik nie są od razu przesyłane z SZBD do klienta (choć zależy to również od implementacji sterownika). W najprostszym przypadku, każde wywołanie metody „next” powoduje przesłanie jednej krotki z SZBD do klienta. Jak łatwo zauważyć, jest to rozwiązanie wysoce niewydajne, gdyż dla każdej krotki wyniku zapytania należy poświęcić czas konieczny na wymianę komunikatów pomiędzy klientem a SZBD. W sterowniku JDBC dla SZBD Oracle zaimplementowano możliwość zmiany liczby przesyłanych jednorazowo krotek. Można to wykonać za pomocą metody „setRowPrefetch” dostępnej poprzez typ „OracleStatement” stanowiący implementację interfejsu „Statement” z JDBC API. Przykład (1) pokazuje sposób użycia tej metody. W pierwszym kroku tworzony jest nowy obiekt typu „Statement”. Następnie, za pomocą rzutowania, typ tego obiektu jest zmieniany na typ „OracleStatement” (taka operacja uda się jedynie w sytuacji, kiedy korzystamy ze sterownika Oracle’a). Po zmianie typu obiektu mamy dostęp do metody „setRowPrefetch”, której jako parametr przekazujemy liczbę 10. Oznacza to, że kiedy zajdzie potrzeba pobrania krotek wyniku zapytania z SZBD, to pobrane zostanie od razu 10 krotek. Typ „OracleStatement” zawiera również metodę, która pozwala na ustawienie domyślnej liczby przesyłanych krotek – „setDefaultRowPrefetch”. W sytuacji, kiedy chcemy zwiększyć liczbę przesyłanych krotek dla zapytań przekompilowanych, należy użyć rzutowania na typ „OraclePreparedStatement”. Przykład (2) stanowi proste wykonanie zapytania o listę nazwisk pracowników. W tej pętli, komunikacja pomiędzy SZBD a aplikacją kliencką nastąpi jedynie dwukrotnie (najpierw pierwsze 10 krotek, potem kolejne 4).

Rozwiązanie przedstawione na tym slajdzie jest specyficzne dla SZBD Oracle. Inne SZBD mogą implementować analogiczne rozwiązania. Niestety nie są one objęte specyfikacją JDBC. Postanowiono zatem nie przedstawiać zadań demonstrujących to rozwiązanie.

Pełny kod programu, którego fragmenty przedstawiono na slajdzie, załączono do kursu w pliku JDBCELearning8.java.

Przewijalne zbiory wyników

```
①
Statement stmt = con.createStatement(
  ResultSet.TYPE_SCROLL_SENSITIVE,
  ResultSet.CONCUR_READ_ONLY);

②
ResultSet rs = stmt.executeQuery(
 "SELECT nazwisko FROM pracownicy");
rs.absolute(1); System.out.println(rs.getString(1));
rs.absolute(-2); System.out.println(rs.getString(1));
rs.relative(-1); System.out.println(rs.getString(1));
rs.relative(2); System.out.println(rs.getString(1));

③
rs.afterLast();
while(rs.previous())System.out.println(rs.getString(1));
```


W JDBC 2.0 rozbudowano funkcjonalność obiektu typu „ResultSet” reprezentującego wynik zapytania. Dzięki nowej funkcjonalności możliwe jest przeglądanie zbioru wyników w dowolnej kolejności. Aby w wyniku zapytania tworzone były odpowiednie obiekty typu „ResultSet”, należy utworzyć obiekt typu „Statement”, za pomocą innej wersji metody „createStatement”. Inna wersja metody „createStatement”, przyjmuje dwa parametry, które są typu liczbowego, a konkretne wartości są reprezentowane przez odpowiednie stałe. Pierwszy parametr określa dopuszczalny sposób przeglądania wyników zapytania, a drugi określa czy zbiór wyników można modyfikować. Pierwszy parametr może przyjmować następujące wartości:

ResultSet.TYPE_SCROLL_FORWARD – stary rodzaj wyników zapytania, wyniki mogą być przeglądane jedynie sekwencyjnie, „do przodu”,

ResultSet.TYPE_SCROLL_INSENSITIVE – wyniki mogą być przeglądane w dowolny sposób, ale nie odzwierciedlają zmian wykonanych na relacjach, do których odnosi się zapytanie, wykonanych przez innych użytkowników,

ResultSet.TYPE_SCROLL_SENSITIVE – wyniki mogą być przeglądane w dowolny sposób i odzwierciedlają zmiany wykonane przez innych użytkowników.

Drugi parametr może przyjąć jedną z dwóch wartości:

ResultSet.CONCUR_READ_ONLY – zbioru wyników nie można modyfikować,

ResultSet.CONCUR_UPDATABLE – zbiór wyników można modyfikować.

Tematyka związana z modyfikowalnymi zbiorami wyników zostanie opisana później. Jeżeli chcemy korzystać z prekompilowanych poleceń i przewijalnych zbiorów wyników, to istnieje również rozbudowana wersja metody „prepareStatement”, której pierwszym parametrem jest łańcuch z poleceniem SQL (tak samo jak poprzednio), a kolejnymi dwoma parametrami są parametry identyczne z parametrami rozbudowanej wersji metody „createStatement”.

Kiedy zostanie utworzony obiekt typu „Statement”, można go użyć do wykonania w znany już sposób dowolnego zapytania. W wyniku zapytania otrzymujemy obiekt typu „ResultSet”, który posiada następujące metody pozwalające na poruszanie się po zbiorze wyników:

- „absolute(n)” – 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),
- „relative(n)” – przesunięcie wskaźnika o n krotek względem aktualnej pozycji. Wartość n może być zarówno dodatnia (przesunięcie do przodu), jak i ujemna (cofnięcie się),
- „beforeFirst” – przesunięcie wskaźnika na pozycję przed pierwszą krotką,
- „afterLast” – przesunięcie wskaźnika na pozycję za ostatnią krotką,
- „first” – przesunięcie wskaźnika na pierwszą krotkę,
- „last” – przesunięcie wskaźnika na ostatnią krotkę,
- „next” – znane z poprzednich slajdów, przesuwa wskaźnik na następną krotkę i zwraca prawdę, jeśli nie znalazł się on za ostatnią krotką,
- „previous” – odwrotność „next”, przesuwa wskaźnik na poprzednią krotkę i zwraca prawdę, jeśli nie znalazł się on przed pierwszą krotką,
- „isAfterLast” – zwraca „true”, jeśli wskaźnik znajduje się za ostatnią krotką i „false” w przeciwnym wypadku,
- „isBeforeFirst” – zwraca „true”, jeśli wskaźnik znajduje się przed pierwszą krotką i „false” w przeciwnym wypadku,
- „isFirst” – zwraca „true” jeśli wskaźnik znajduje się na pierwszej krotce i „false” w przeciwnym wypadku,
- „isLast” – zwraca „true” jeśli wskaźnik znajduje się na ostatniej krotce i „false” w przeciwnym wypadku.

Rozważmy przykład na slajdzie. Na przykładzie (1) pokazano instrukcję tworzącą obiekt typu „Statement”, który pozwoli na otrzymywanie przewijalnych, ale nie modyfikowalnych wyników. Na przykładzie (2) pokazano wykonanie zapytania, a następnie wykorzystanie metod „absolute” i „relative” do odczytania krotek z relacji wynikowej. Pierwsze polecenie przesuwa wskaźnik na pierwszą krotkę relacji wynikowej. Następnie odczytywana jest wartość nazwiska z tej krotki i wypisywane na konsoli. Następnie, wskaźnik jest przesuwany na drugą od końca krotkę, potem na krotkę znajdującą się przed aktualnie wskazywaną i ostatecznie, na krotkę znajdującą się dwie pozycje za aktualnie wskazywaną. Przykład (3) demonstruje użycie metod „afterLast” i „previous”. Za pomocą „afterLast” wskaźnik przesuwany jest na pozycję za ostatnią krotką. Następnie, w pętli „while”, analogicznej do używanych na poprzednich slajdach, odczytywane są wszystkie nazwiska z relacji. Jedyną różnicą pomiędzy tą, a poprzednimi pętlami jest to, że użyto metody „previous”, która jest odpowiednikiem metody „next”, ale poruszającym się „do tyłu”.

Pełny kod programu, którego fragmenty przedstawiono na slajdzie, załączono do kursu w pliku JDBCELearning9.java.

Zadanie (6)

- 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.

Rozwiązanie (6)

```
Statement stmt=con.createStatement(
  ResultSet.TYPE_SCROLL_SENSITIVE,
  ResultSet.CONCUR_READ_ONLY);
ResultSet rs=stmt.executeQuery(
  "SELECT nazwisko FROM pracownicy " +
  "WHERE etat='ASYSTENT' ORDER BY placa_pod DESC");
rs.last();
System.out.println("Asystent zarabiający najmniej: "+
  "+rs.getString(1));
rs.relative(-2);
System.out.println("Trzeci najmniej zarabiający asystent:"+
  "+rs.getString(1));
rs.absolute(2);
System.out.println("Przedostatni najmniej zarabiający "+
  "asystent: "+rs.getString(1));
rs.close(); stmt.close();
```


Na slajdzie przedstawiono najważniejsze fragmenty programu stanowiącego rozwiązanie zadania (6), którego treść przedstawiono poniżej. Pełny program umieszczono w pliku JDBCELearningZad6.java, który został załączony do kursu.

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.

Modyfikowalne zbiory wyników

```
①  Statement stmt = con.createStatement(
     ResultSet.TYPE_SCROLL_SENSITIVE,
     ResultSet.CONCUR_UPDATABLE);

②  ResultSet rs = stmt.executeQuery(
    "SELECT id_prac,nazwisko,placa_pod FROM pracownicy");

③  rs.last();
   rs.updateString(2,rs.getString(2)+"123");
   rs.updateFloat("PLACA_POD", new Float(rs.getFloat(3)*0.9));
   rs.updateRow();// albo rs.cancelRowUpdates();

④  rs.moveToInsertRow();
   rs.updateInt(1,310);
   rs.updateString(2,"Zieliński");
   rs.updateFloat(3,new Float(1234.5));
   rs.insertRow();
```


Na poprzednim slajdzie wspomniano, że wynik zapytania można modyfikować. Przypomnijcie sobie państwo dyskusję dotyczącą perspektyw. Ponieważ wynik zapytania jest relacją, to można go przetwarzać, w celu modyfikowania relacji bazowych. W JDBC 2.0 wprowadzono funkcjonalność pozwalającą na modyfikowanie wyniku zapytania dostępnego poprzez obiekt typu „ResultSet". Wszystkie wymagania, co do warunków, które zapytanie perspektywy musi spełnić, aby perspektywa była modyfikowalna, odnoszą się również do zapytań, dla których zbiory wyników mają być modyfikowalne. Aby utworzony w wyniku wykonania zapytania obiekt typu „ResultSet" pozwalał na modyfikację danych, obiekt typu „Statement" musi być utworzony z parametrem „ResultSet.CONCUR_UPDATABLE" (patrz poprzedni slajd).

Przykład (1) pokazuje instrukcję tworzącą odpowiedni obiekt typu „Statement". Przykład (2) pokazuje wykonanie zapytania, którego wyniki będziemy modyfikować. Kiedy wskaźnik jest ustawiony na krotce, którą chcemy zmodyfikować, należy użyć metod „updateXXX" (XXX oznacza typ wartości), analogicznych do metod „getXXX", za pomocą których można modyfikować wartości w wyniku. Znajdują tutaj również zastosowanie sugestie dotyczące stosowania odpowiednich metod do odpowiednich typów ANSI SQL przedstawione przy okazji omawiania metod „getXXX". Metody „updateXXX" posiadają dwa parametry: pierwszy określa który atrybut ma zostać zmodyfikowany (albo jego nazwa, albo numer), drugi określa nową wartość atrybutu. Kiedy pożądane zmiany zostaną wprowadzone, należy albo je zatwierdzić za pomocą metody „updateRow" interfejsu „ResultSet", żeby zostały zapisane do bazy danych, albo wycofać za pomocą metody „cancelRowUpdates" interfejsu „ResultSet". Przesunięcie wskaźnika na inną krotkę w zbiorze wynikowym również powoduje wycofanie zmian w krotce. Przeanalizujmy przykład (3). W przykładzie tym, wskaźnik przesuwany jest na ostatnią krotkę wyniku. Jest to krotka, którą będziemy modyfikować.

Następnie za pomocą metody „updateString”, drugiemu atrybutowi relacji wynikowej (NAZWISKO) przypisywana jest stara wartość z dodanym na końcu łańcuchem „123”. Za pomocą metody „updateFloat” modyfikowany jest atrybut o nazwie PLACA_POD. Jego nową wartość obliczana jest jako 90% starej. Ostatecznie, zmiany są zatwierdzane za pomocą metody „updateRow”.

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 „SQLException”. 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, a której się znajdował przed aktywowaniem metody „moveToInsertRow”, należy użyć metody „moveToCurrentRow”. Rozważmy przykład (4). Przykład ten rozpoczyna się od przesunięcia wskaźnika do wirtualnej krotki. Następnie, za pomocą metod „updateXXX”, wstawiane są do kolejnych atrybutów odpowiednie wartości. Ostatecznie, krotka jest wstawiana do bazy danych za pomocą metody „insertRow”.

Usuwanie krotek jest bardzo proste. Wystarczy przesunąć wskaźnik na krotkę, którą chcemy usunąć, a następnie aktywować bezparametrową metodę „deleteRow” interfejsu „ResultSet”.

Pełny kod programu, którego fragmenty przedstawiono na slajdzie, załączono do kursu w pliku JDBCELearning10.java.

Zadanie (7)

- Napisz program, który zmieni na duże litery wszystkie litery w imionach i nazwiskach wszystkich pracowników.
- Do wykonania zadania wykorzystaj mechanizm modyfikacji wyniku zapytania.
- Hint: do zmiany wielkości liter wykorzystaj metodę toUpperCase typu String.

Rozwiązanie (7)

```
Statement stmt=con.createStatement(
  ResultSet.TYPE_SCROLL_SENSITIVE,
  ResultSet.CONCUR_UPDATABLE);
ResultSet rs=stmt.executeQuery(
  "SELECT imie,nazwisko FROM pracownicy");

while (rs.next()) {
  rs.updateString(1,rs.getString(1).toUpperCase());
  rs.updateString(2,rs.getString(2).toUpperCase());
  rs.updateRow();
}
rs.close();
stmt.close();
```


Na slajdzie przedstawiono najważniejsze fragmenty programu stanowiącego rozwiązanie zadania (7), którego treść przedstawiono poniżej. Pełny program umieszczono w pliku JDBCELearningZad7.java, który został załączony do kursu.

Napisz program, który zmieni na duże litery wszystkie litery w imionach i nazwiskach wszystkich pracowników. Do wykonania zadania wykorzystaj mechanizm modyfikacji wyniku zapytania.

SQLJ

①
```
import oracle.sqlj.runtime.Oracle;
import java.sql.*;
```

②
```
#sql iterator MyIter (String nazwisko);
```

③
```
Oracle.connect("jdbc:oracle:thin:@dblab.cs.put.poznan.pl:
  1521:dblab10g", „elearning_user", „elearning_pass");
```

④
```
MyIter zap;
#sql zap={select nazwisko from pracownicy};

while (zap.next()) System.out.println(zap.nazwisko());
```

⑤
```
Oracle.close();
```


Na zakończenie ćwiczenia omówimy państwu w ogólny sposób oparty na JDBC standard SQLJ i pokażemy przykład jego zastosowania. Ponieważ SQLJ jest rozwiązaniem wspieranym przez niewielką liczbę producentów, nie przedstawimy żadnych zadań demonstrujących użycie tego standardu. SQLJ jest standardem złożonym z trzech części. Część pierwsza opisuje sposób zagnieżdżania poleceń SQL w kodzie programu napisanego w języku Java. Część druga opisuje sposób składowania w bazie danych procedur i funkcji napisanych w języku Java. Część trzecia opisuje sposób składowania typów języka Java jako obiektów w bazie danych. Z naszego punktu widzenia najciekawsza jest pierwsza część tego standardu.

W standardzie zdefiniowano składnię, której można użyć aby zagnieździć polecenia SQL wewnątrz kodu w języku Java. Tak napisany program nie nadaje się oczywiście do kompilacji przez standardowy kompilator języka Java. Dlatego też stosuje się specjalne narzędzie, które całą składnię charakterystyczną dla SQLJ transformuje do standardowych wywołań JDBC. Tak przetransformowany plik można następnie poddać kompilacji za pomocą zwykłego kompilatora języka Java. Zaletą tego rozwiązania jest krótszy kod obsługujący komunikację z bazą danych, oraz kontrola typów zwracanych wartości wykonywana przez program dokonujący transformacji kodu zgodnego z SQLJ do czystego kodu w języku Java.

Na slajdzie pokazano przykładowy program, który wykorzystuje składnię SQLJ do wykonania prostego zapytania. Programy wykorzystujące SQLJ muszą importować zawartość pakietu „java.sql" i klasę „oracle.sqlj.runtime.Oracle" (1). Na przykładzie (2) pokazano deklarację typu iteratora, który zostanie użyty, w celu odczytania wyniku zapytania. Zapytanie, które będziemy wykonywać zwraca tylko jeden atrybut typu łańcuchowego – nazwisko. Na przykładzie (2) pokazano sposób nawiązania połączenia z SZBD.

Połączenie nawiązuje się za pomocą statycznej metody „connect” klasy „Oracle”. Parametry tej metody są analogiczne do parametrów metody „getConnection” klasy „DriverManager”. Przykład (4) pokazuje faktyczne wykonanie zapytania. Najpierw deklarowana jest zmienna zadeklarowanego wcześniej typu, a następnie wykonywane jest faktyczne zapytanie, którego wynik jest przypisywany do nowo zadeklarowanej zmiennej. Ostatecznie, za pomocą metody „next” otrzymanego w wyniku zapytania iteratora (metoda ta działa analogicznie do metody „next” w JDBC) odczytujemy i wypisujemy na konsoli kolejne odczytywane z bazy danych nazwiska. Jak łatwo zauważyć, zadeklarowana nazwa atrybutu w iteratorze, przełożyła się na jego metodę, która zwraca odpowiednią wartość. Bazę danych zamykamy za pomocą statycznej metody „close” klasy „Oracle” (5).

Pełny kod programu, którego fragmenty przedstawiono na slajdzie, załączono do kursu w pliku JDBCELearning11.sqlj.

W celu wykonania przykładowego programu, prócz standardowego pliku JAR ze sterownikiem JDBC, należy umieścić również w CLASSPATH następujące pliki JAR, które można pobrać ze strony Oracle’a:

- Runtime12.jar,
- Translator.jar.

Podsumowanie

- JDBC (*Java Database Connectivity*) to API pozwalające programom napisanym w języku Java na korzystanie z SZBD do zapisywania i odczytywania danych w bazie danych.
- JDBC pozwala na wykonywanie poleceń SQL tworzonych *ad hoc*, oraz poleceń prekompilowanych.
- W JDBC 2.0 poprzez wyniki zapytań można modyfikować relacje bazowe zapytania, oraz w dowolny sposób je przeglądać. Możliwe jest również również wsadowe wykonywanie aktualizacji danych.
- SQLJ pozwala na zagnieżdżanie zapytań bezpośrednio w kodzie programów napisanych w języku Java.

Na tym ćwiczeniu poznaliście państwo podstawy pracy z JDBC. Potraficie obecnie pisać programy, które komunikują się z SZBD w celu realizacji zapytań, oraz aktualizacji danych w bazie danych. Dowiedzieliście się jak można wykonywać polecenia SQL tworzone *ad hoc* jak również przygotowywać polecenia prekompilowane, które wykonywane są znacznie szybciej. Poznaliście również część funkcjonalności wprowadzonej przez drugą wersję JDBC. Dzięki tej funkcjonalności możliwe jest modyfikowanie relacji bazowych zapytania, poprzez modyfikację wyników zapytania, swobodne przeglądanie tych wyników oraz wsadowe wykonywanie poleceń aktualizacji. Poznaliście również wprowadzenie do standardu SQLJ, który pozwala na zagnieżdżanie kodu SQL w programach napisanych w języku Java. Dowiedzieliście się, że dzięki SQLJ możliwe jest skrócenie kodu programów komunikujących się SZBD, oraz kontrola typów zwracanych wartości na etapie kompilacji.