Relacyjny model danych

Relacyjny model danych posiada trzy podstawowe składowe:

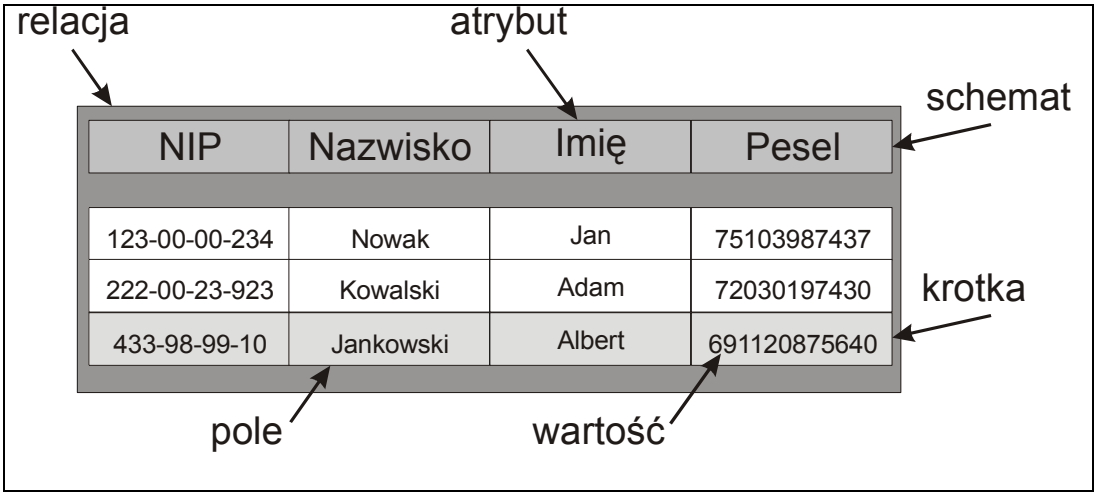
- relacyjne struktury danych
- operatory algebry relacyjnej, które umożliwiają tworzenie, przeszukiwanie i modyfikowanie danych
- ograniczenia (więzy) integralnościowe jawnie lub niejawnie określającymi możliwe/dopuszczalne wartości danych.

Podstawową strukturą danych modelu relacyjnego jest relacja, będąca podzbiorem iloczynu kartezjańskiego wybranych domen i przedstawiana w postaci dwuwymiarowej tabeli. W tym ujęciu pojęcia relacji (ang. *relation*) i tabeli (ang. *table*) są synonimami. Pojęcie relacji używane jest na poziomie teorii baz relacyjnych i modelu konceptualnego, a pojęcie tabeli na poziomie realizacji fizycznej.

Relację, nieformalnie rzecz biorąc, definiuje się za pomocą predykatu jako pewien zbiór krotek (ang. *tuples*) (rekordów) posiadających taką samą strukturę (schemat) i różne wartości. Zbiór ten jest przedstawiany w postaci wierszy tablicy, tzn. na poziomie fizycznym krotka jest przedstawiana jako wiersz tabeli, zawierający wartość co najmniej jednego atrybutu o określonej dziedzinie (kolumny tablicy). Inaczej mówiąc: każda krotka danej relacji ma dokładnie taką samą strukturę, zgodą ze schematem relacji. Każda krotka posiada przynajmniej jeden atrybut (ang. *attribute*), który jest przedstawiany w postaci kolumny.

Relacja posiada następujące właściwości:

- wszystkie jej krotki są różne
- wszystkie jej atrybuty są różne
- kolejność krotek nie ma znaczenia i w ogólności nie jest ona znana
- kolejność atrybutów nie ma znaczenia
- wartości atrybutów są niepodzielne (atomowe), tj. nie mogą być zbiorem wartości.



SQL

Język SQL używany jest do pracy z relacyjną bazą danych. Jest to język nieproceduralny, należący do grupy języków deklaratywnych. Semantyka SQL wyraża, co ma być zrobione, a nie jak. Problem „jak” przeniesiony został na poziom systemu zarządzania bazą danych.

Wyrażenia SQL nazywane są kwerendami. Jak wynika z powyższego opisu, kwerendy mogą służyć do: uzyskiwania informacji z bazy danych (wtedy nazywamy je zapytaniami); modyfikowania bazy danych (wstawiania lub usuwania danych, modyfikacji schematu tabeli, itp.); sterowania danymi.

Zapytania (podzbiór kwerend) służą do uzyskiwania informacji z bazy danych. Jeżeli odpowiedzią na zapytanie jest relacja, to pytanie w minimalnej formie musi zawierać definicję schematu relacji odpowiedzi i definicję źródła danych, z którego dane są pobierane. Relacji odpowiedzi może służyć jako źródło danych do kolejnego zapytanie (z punktu widzenia pierwszego zapytania będzie to podzapytanie).

Historia języka SQL

Historia SQL'a rozpoczęła się w zeszłym stuleciu. W ramach prac w firmie IBM nad językiem do obsługi baz danych powstał język SEQUEL (Structured English Query Language). Język następnie został rozwinięty i nazwany jako SQL (Structured Query Language - Strukturalny Język Zapytań).

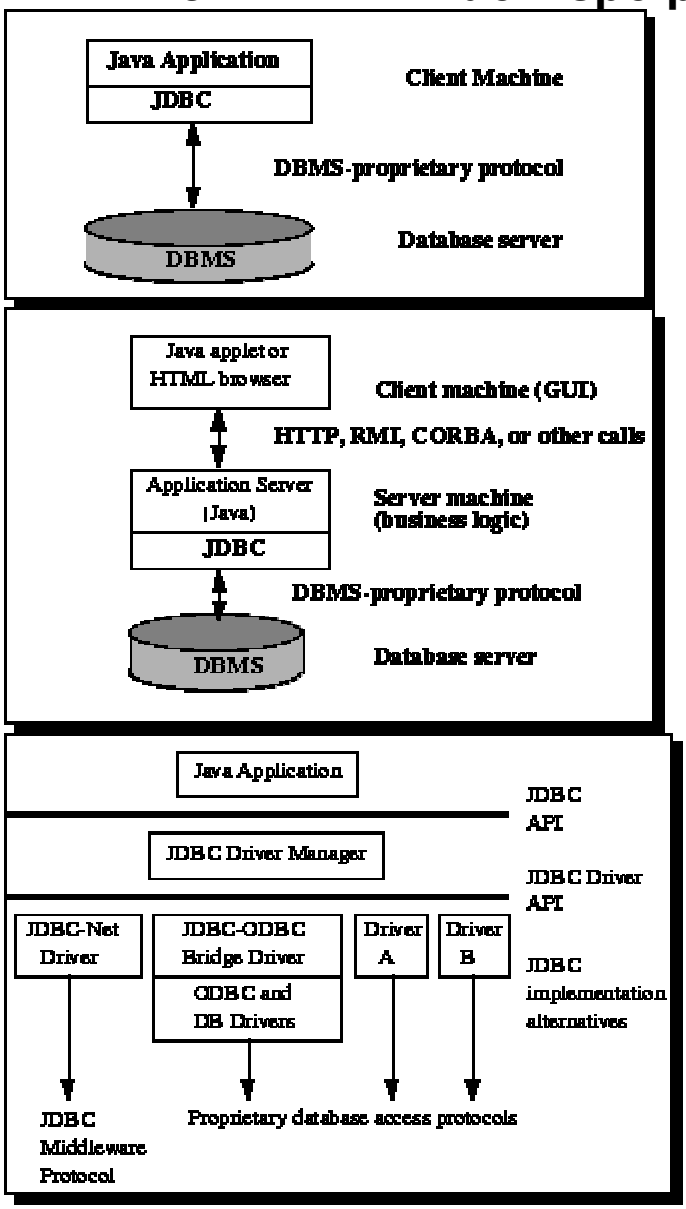
1986 – ANSI zdefiniowało standard SQL-86
1989 – ANSI modernizuje poprzedni standard, powstaje SQL-89 (brak w nim rozkazów do zmiany schematu bazy danych i brak dynamicznego SQL'a)
1992 – Kolejna modernizacja standardu przez ANSI, powstaje SQL-92 (SQL 2) (jest to już szerszy standard, zawierającym wszystkie elementy niezbędne do zmian schematu bazy danych (DB), sterowaniq transakcjami i sesjami).

Równocześnie z zakończeniem prac nad standardem SQL 2 rozpoczęto prace nad standardem SQL 3 mającym elementy obiektowości.

Jedną z ostatnich wersji standardu jest SQL:2003 (ISO/IEC 9075-X:2003). Standard ten jest tak obszerny, że nie wszystkie komercyjne systemy baz danych w pełni się z nim zgadzają. Lista komend i słów kluczowych tego języka przedstawia poniższa tabela:

ALTER DOMAIN	DECLARE CURSOR	FREE LOCATOR
ALTER TABLE	DECLARE TABLE	GET DIAGNOSTICS
CALL	DELETE	GRANT
CLOSE	DISCONNECT	HOLD LOCATOR
COMMIT	DROP ASSERTION	INSERT
CONNECT	DROP CHARACTER SET	OPEN
CREATE ASSERTION	DROP COLLATION	RELEASE SAVEPOINT
CREATE CHARACTER SET	DROP DOMAIN	RETURN
CREATE COLLATION	DROP ORDERING	REVOKE
CREATE DOMAIN	DROP ROLE	ROLLBACK
CREATE FUNCTION	DROP SCHEMA	SAVEPOINT
CREATE METHOD	DROP SPECIFIC FUNCTION	SELECT
CREATE ORDERING	DROP SPECIFIC PROCEDURE	SET CONNECTION
CREATE PROCEDURE	DROP SPECIFIC ROUTINE	SET CONSTRAINTS
CREATE ROLE	DROP TABLE	SET ROLE
CREATE SCHEMA	DROP TRANSFORM	SET SESSION AUTHORIZATION
CREATE TABLE	DROP TRANSLATION	SET SESSION CHARACTERISTICS
CREATE TRANSFORM	DROP TRIGGER	SET TIME ZONE
CREATE TRANSLATION	DROP TYPE	SET TRANSACTION
CREATE TRIGGER	DROP VIEW	START TRANSACTION
CREATE TYPE	FETCH	UPDATE
CREATE VIEW		

JDBC – API do współpracy z bazami danych



1. *JDBC-ODBC bridge plus ODBC driver:* tłumaczy JDBC na ODBC i wykorzystuje wobec tego sterownik ODBC do kokmunikacji z bazą danych. Pakiet JDK zawiera jeden sterownik tego typu – tj. mostek JDBC/ODBC. Wymaga on właściwego zainstalowania i skonfigurowania sterownika ODBC. Jego używanie jest zalecane głównie do przeprowadzania testów. Można do stosowac również w sieci korporacyjnych, gdzie instalacja binarnych plików ODBC na klienckich stanowiskach nie stanowi problemu, lub też w aplikacjach z serwerem napisanym w Java w architekturze trzy poziomowej.
2. *Native-API partly-Java driver:* Napisany jest częściowo w Javie, częściowo w kodzie macierzystym danej platformy. Komunikuje się z bazą danych, używając interfejsu programowego klienta danego systemu baz danych. Wykorzystując taki sterownik, oprócz bibliotek Javy, należy instalować oprogramowanie specyficzne do danej platformy (API for Oracle, Sybase, Informix, IBM DB2, i inne.
3. *JDBC-Net pure Java driver:* napisany jest wyłącznie w Javie i korzysta z protokołu transmisji niezależnego od systemu bazy danych. Komunikacja ta odbywa się z komponentem serwera, który tłumaczy żądania sterownika na specyficzny protokół określonego systemu bazy danych.

This driver translates JDBC calls into a DBMS-independent net protocol, which is then translated to a DBMS protocol by a server. This net server middleware is able to connect its pure Java clients to many different databases. The specific protocol used depends on the vendor. In general, this is the most flexible JDBC alternative. It is likely that all vendors of this solution will provide products suitable for intranet

use. In order for these products to support Internet access as well, they must handle the additional requirements for security, access through firewalls, and so forth, that the Web imposes.

4. *Native-protocol pure Java driver:* napisany jest wyłącznie w języku Java i tłumaczy żądania JDBC na specyficzny protokół danego systemu bazy danych.

This kind of driver converts JDBC calls directly into the network protocol used by DBMSs. This allows a direct call from the client machine to the DBMS server and is an excellent solution for intranet access. Since many of these protocols are proprietary, the database vendors themselves are the primary source. Several that are now available include Oracle, Sybase, Informix, IBM DB2, Inprise InterBase, and Microsoft SQL Server.

Nawiązanie połączenia

1. załadowania drivera
2. utworzenia połączenia

Do zarządzania driverami (tj. klasami odpowiedzialnymi za komunikację z bazami danych) odpowiedzialna jest klasa DriverManager. Klasa ta posiada metodę DriverManager.registerDriver, służącą do rejestracji driverów, jednak użytkownik normalnie jej nie wywołuje. Metoda ta powinna być wywołana automatycznie przez klasę drivera, kiedy jest ona ładowana. Klasa drivera jest ładowana (i automatycznie rejestrowana w DriverManager na dwa sposoby:

1. przez wywołanie Class.forName:

```
Class.forName("acme.db.Driver");
```

   Jeśli acme.db.Driver został zaimplementowany w taki sposób, że jego ładowanie tworzy instancję klasy drivera i rejestruje ją wywołując DriverManager.registerDriver (z instancją jako parametrem), wtedy załadowany driver pojawia się w liście driverów DriverManager i jest od tej chwili dostępny do tworzenia połączeń.

2. przez dodanie nazwy klasy drivera do własności systemowej jdbc.drivers (. Jest to lista nazw klas driverów, oddzielona średnikami, które DriverManager ładuje domyślnie (przeglądanie własności jdbc.drivers odbywa się w chwili, kiedy klasa DriverManager jest inicjalizowana. Kiedy użytkownik wprowadził do niej nazwy jednej lub więcej klas driverów, DriverManager przystąpi do ich ładowania przy pierwszym wywołaniu jego metody.

```
jdbc.drivers=foo.bah.Driver:wombat.sql.Driver:bad.test.ourDriver
```

Ładowanie driverów

Można to zrobić w jednej linii programu (rozważany jest przypadek drivera JDBC-ODBC Bridge):

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Dokumentacja drivera opisuje, jakiej nazwy klasy należy użyć. Na przykład, jeśli nazwą klasy jest jdbc.DriverXYZ , wtedy załadowanie drivera ma postać:

```
Class.forName("jdbc.DriverXYZ");
```

Wywołanie Class.forName automatycznie tworzy instancję drivera i rejestruje go z DriverManager. Tak jest dla implementacji zgodnej ze standardem JDBC. Tworzenie własnej instancji nie jest konieczne (byłoby niepotrzebnym tworzeniem duplikatu). Niestety, często drivery nie są zgodne ze standardem i wymagają użycia następującego kodu:

```
Class.forName(DriverClassName).newInstance();
```

Utworzenie połączenia

Łączy się tu driver z odpowiednim serverem (DBMS). W ogólności wygląda to tak:

```
Connection con = DriverManager.getConnection(url, "myLogin","myPassword");
```

Dla drivera JDBC-ODBC Bridge driver, url (tzn. JDBC URL) zaczyna się od jdbc:odbc:. Resztą url jest w ogólności nazwą źródła danych lub też systemu baz danych.

```
String url = "jdbc:odbc:Fred";
Connection con = DriverManager.getConnection(url, "Fernanda", "J8");
```

Drivery JDBC dostarczane przez różnych producentów oprogramowania mogą wymagać podania podprotokołu, tzn. w url po jdbc: powinno coś się jeszcze pojawić. Jeśli podprotokół zarejestrowany został z nazwą acme, url będzie wyglądać tak: jdbc:acme:. Ostatnia część url identyfikuje źródło danych.

Jeśli któryś z załadowanych driverów rozpozna ciąg url podany w metodzie getConnection klasy DriverManager, ten właśnie driver nawiąże połączenie z DBMS. Klasa DriverManager jest odpowiedzialna za wszystkie szczegóły tej operacji, ukrywając je przed użytkownikiem. Tak więc, jeśli nie pisze się własnego drivera, prawdopodobnie nie będzie się korzystać z metod interfejsu Driver, a jedyną metodą wykorzystywaną klasy DriverManager będzie getConnection.

Połączenie zwracane przez DriverManager.getConnection jest otwartym połączeniem, poprzez które można przesyłać SQL'owe żądania do SZBD (*systemu zarządzania bazą danych, ang. DBMS*), zawarte w tworzonych wyrażeniach JDBC.

Zamykanie połączenia:

```
con.close()
```

Metoda Connection.isClosed() zwraca true, jeśli wywołana jest po Connection.close().

JDBC URL - Standardowa składnia

jdbc:<subprotocol>:<subname>	
<subprotocol>-nazwa drivera lub mechanizmu połączenia z bazą danych	jdbc:dcenaming:accounts-payable //serwis nazw DCE jdbc:odbc:fred // JDBC-ODBC bridge
<subname>- nazwa źródła danych	//hostname:port/subsubname
jdbc:odbc:<data-source-name>[;<attribute-name>=<attribute-value>]	jdbc:odbc:wombat;CacheSize=20;ExtensionCase=LOWER jdbc:odbc:qeora;UID=kgh;PWD=fooey

Operacje na tabelach

Tworzenie tabeli w SQL

Tabela COFFEES

COF_NAME	SUP_ID	PRICE	SALES	TOTAL
Colombian	101	7.99	0	0
French_Roast	49	8.99	0	0
Espresso	150	9.99	0	0
Colombian_Decaf	101	8.99	0	0
French_Roast_Decaf	49	9.99	0	0

VARCHAR (32 znaki)	INTEGER	FLOAT	FLOAT	INTEGER

Tabela SUPPLIERS

SUP_ID	SUP_NAME	STREET	CITY	STATE	ZIP
101	Acme, Inc.	99 Market Street	Groundsville	CA	95199
49	Superior Coffee	1 Party Place	Mendocino	CA	95460
150	The High Ground	100 Coffee Lane	Meadows	CA	93966

Polecenie SQL tworzące pustą tabelę COFFEES:

```
CREATE TABLE COFFEES
(COF_NAME VARCHAR(32),
 SUP_ID INTEGER,
 PRICE FLOAT,
 SALES INTEGER,
 TOTAL INTEGER)
```

(końcówka polecenia, tj. znak ';' lub słowo 'go' jak w Sybase, dostawiona będzie przez driver, gdy polecenie to wykorzystane będzie przez JDBC). Typy danych w powyższym poleceniu są podstawowymi typami SQL (*generic SQL types* – nazywane również *JDBC types*). Typy te zdefiniowane są w klasie java.sql.Types. Jeśli DBMS używa własnych nazw typów, trzeba działać inaczej. Rozróżnialność wielkości liter zależy od używanej bazy danych.

Wstawianie danych do tabeli w SQL

Polecenie SQL wstawiające jeden wiersz do tabeli COFFEES:

```
INSERT INTO COFFEES VALUES ('Colombian', 101, 7.99, 0, 0)
```

Wynikiem tego wyrażenia jest nowy wiersza w tabeli

Colombian	101	7.99	0	0

Odczytywanie danych z tabeli w SQL

Dla pełnej tabeli wynikiem zapytania: SELECT * FROM COFFEES jest zbiór wynikowy:

```
COF_NAME          SUP_ID  PRICE  SALES  TOTAL
---------------   ------  -----  -----  -----
Colombian         101     7.99   0      0
French_Roast      49      8.99   0      0
Espresso          150     9.99   0      0
Colombian_Decaf   101     8.99   0      0
French_Roast_Decaf 49     9.99   0      0
```

Dla pełnej tabeli wynikiem zapytania SELECT COF_NAME, PRICE FROM COFFEES jest zbiór wynikowy:

```
COF_NAME                PRICE
--------                -----
Colombian               7.99
French_Roast            8.99
Espresso                9.99
Colombian_Decaf         8.99
French_Roast_Decaf      9.99
```

Wynikiem zapytania SELECT COF_NAME, PRICE FROM COFFEES WHERE PRICE < 9.00 jest:

COF_NAME	PRICE
Colombian	7.99
French_Roast	8.99
Colombian Decaf	8.99

Modyfikacja danych w tabeli

Wynikiem UPDATE COFFEES SET SALES = 75 WHERE COF_NAME LIKE 'Colombian' jest:

COF_NAME	SUP_ID	PRICE	SALES	TOTAL
Colombian	101	7.99	75	0
French_Roast	49	8.99	0	0
Espresso	150	9.99	0	0
Colombian_Decaf	101	8.99	0	0
French_Roast_Decaf	49	9.99	0	0

(kolumna TOTAL pozostała jeszcze nie zmieniona).

Wyrażenia JDBC

Metody do tworzenia wyrażeń:

createStatement (SQL bez parametrów)	prepareStatement (SQL parametryzowane)	prepareCall (procedury zapisane)
createStatement() createStatement(int resultSetType, int resultSetConcurrency) createStatement(int resultSetType, int resultSetConcurrency, int resultlSetHoldability)	prepareStatement(String sql) prepareStatement(String sql, int resultSetType, int resultSetConcurrency) prepareStatement(String sql, int resultSetType, int resultSetConcurrency, int resultlSetHoldability)	prepareCall(String sql) prepareCall(String sql, int resultSetType, int resultSetConcurrency) prepareCall(String sql, int resultSetType, int resultSetConcurrency, int resultlSetHoldability)

Obiekt Statement jest właściwie tym, co przesyła polecenie SQL do SZBD. Obiekt Statement tworzy się oraz wykonuje się go, określając odpowiednią do wyrażenia SQL metodę. Dla wyrażenia SELECT metodą tą jest executeQuery. Dla wyrażeń, które tworzą lub zmieniają zawartość tabel, metodą jest executeUpdate.

Aby stworzyć obiekt Statement należy skorzystać z otwartego połączenia:

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

Linijka ta tworzy stmt, ale jeszcze bez polecenia SQL. Polecenie to należy dostarczyć w metodzie wykonującej stmt.

```
stmt.executeUpdate(createTableCoffees);
```

Argumentem metody jest obiekt typu String. W poleceniu użyto obiektu, który był utworzony wcześniej:

```
String createTableCoffees = "CREATE TABLE COFFEES " +
   "(COF_NAME VARCHAR(32), SUP_ID INTEGER, PRICE FLOAT, " +
   "SALES INTEGER, TOTAL INTEGER)";
```

Wykonywanie wyrażeń

Wykorzystano executeUpdate, gdyż wyrażenie SQL ukryte w createTableCoffees jest typu DDL. Aby stworzyć, poprawić, zmniejszyć tabelę, używa się właśnie wyrażeń DDL, uruchamianych metodą executeUpdate. Jednak najczęściej wykonywaną metodą jest executeQuery. Używa się jej do wykonania wyrażeń SELECT.

Wstawianie danych do tabeli

Wstawienie jednego wiersza danych do tabeli

```
Statement stmt = con.createStatement();
stmt.executeUpdate( "INSERT INTO COFFEES VALUES ('Colombian', 101, 7.99, 0, 0)");
```

Wynikiem tego wyrażenia jest nowy wiersz w tabeli COFFEES

Colombian	101	7.99	0	0

Wstawienie kolejnych wierszy:

```
stmt.executeUpdate("INSERT INTO COFFEES " + "VALUES ('French_Roast', 49, 8.99, 0, 0)");
stmt.executeUpdate("INSERT INTO COFFEES " + "VALUES ('Espresso', 150, 9.99, 0, 0)");
stmt.executeUpdate("INSERT INTO COFFEES " + "VALUES ('Colombian_Decaf', 101, 8.99, 0,
0)");
stmt.executeUpdate("INSERT INTO COFFEES " + "VALUES ('French_Roast_Decaf', 49, 9.99,
0, 0)");
```

Zbiór wynikowy

JDBC zwraca odpowiedź na zapytanie (SELECT) w obiekcie ResultSet. Aby przechowywać tą odpowiedź, należy zadeklarować referencję do klasy ResultSet:

```
ResultSet rs = stmt.executeQuery("SELECT COF_NAME, PRICE FROM COFFEES");
```

Użycie metody Next

Zmienna rs, będąca instancją klasy ResultSet, zawiera wiersze z nazwami kaw i ich ceną. Korzysta ona z kursora identyfikującego bieżący wiersz. Pozycję tego kursora przesuwa metoda next. W JDBC 2.0 API można przesuwać kursor do przodu, wstecz i relatywnie względem bieżącej pozycji.

Używanie metod getXXX

getXXX służy do odczytywania wartości z kolumn tabeli dla bieżącego wiersza. W przykładzie tabeli COFFEES, pierwsza kolumna w wierszu to COF_NAME , zawierająca wartości SQL'owego typu VARCHAR . Taką wartość można odczytać metodą getString. Wartości typu SQL'owego FLOAT odczytuje metoda getFloat.

Przykład:

zapytanie	wynik

String query = "SELECT COF_NAME, PRICE FROM COFFEES";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
  String s = rs.getString("COF_NAME");
  float n = rs.getFloat("PRICE");
  System.out.println(s + " " + n);
}
	Colombian 7.99
	French_Roast 8.99
	Espresso 9.99
	Colombian_Decaf 8.99
	French_Roast_Decaf 9.99

JDBC oferuje dwa sposoby do identyfikowania kolumn w metodach getXXX. W jednym podaje się nazwę kolumny, w drugim jej indeks (ale w zbiorze wynikowym, nie w bazie danych).

```
String s = rs.getString(1);
float n = rs.getFloat(2);
```

getInt można używać do odczytywania liczb jak i znaków. Jednak odczytywane wartości zawsze będą konwertowane do int. Jeśli więc mamy w tabeli wartość VARCHAR, JDBC spróbuje odczytać z tej wartości liczbę całkowitą. getInt jest zalecany do odczytywania typu SQL'owego INTEGER, lecz nie można jej używać do BINARY , VARBINARY , LONGVARBINARY , DATE , TIME , lub TIMESTAMP .

Używanie metody getString

Chociaż metoda getString zalecane jest do odczytu wartości typów CHAR i VARCHAR, możliwe jest jej użycie również do innych podstawowych typów SQL (nie dotyczy to nowych typów wg SQL3). Jednak wiąże się to z podwójną konwersją (najpierw do obiektu String, a potem do, np. wartości numerycznej).

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
getByte	X	x	x	x	x	x	x	x	x	x	x	x	x						
getShort	x	X	x	x	x	x	x	x	x	x	x	x	x						
getInt	x	x	X	x	x	x	x	x	x	x	x	x	x						
getLong	x	x	x	X	x	x	x	x	x	x	x	x	x						
getFloat	x	x	x	x	X	x	x	x	x	x	x	x	x						
getDouble	x	x	x	x	x	X	X	x	x	x	x	x	x						
getBigDecimal	x	x	x	x	x	x	x	X	X	x	x	x	x						
getBoolean	x	x	x	x	x	x	x	x	x	X	x	x	x						
getString	x	x	x	x	x	x	x	x	x	x	X	X	x	x	x	x	x	x	x
getBytes														X	X	x			
getDate											x	x	x				X		x
getTime											x	x	x					X	x
getTimestamp											x	x	x				x	x	X
getAsciiStream											x	x	X	x	x	x			
getUnicodeStream											x	x	X	x	x	x			
getBinaryStream														x	x	X			
getObject	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

"x" - getXXX może być użyta do odczytu danego typu JDBC.

"X " - getXXX method jest zalecane do odczytu danego typu JDBC.

Uaktualnianie tablic

Modyfikacja kolumny SALES w tablicy COFFEES (wynik działania tego polecenia SQL zaprezentowano już wcześniej):

```
String updateString = "UPDATE COFFEES " +
        "SET SALES = 75 " + "WHERE COF_NAME LIKE 'Colombian'";
stmt.executeUpdate(updateString);
```

A oto przykład, jak można podglądnąć wykonane zmiany (wypisać COF_NAME i SALES):

program	wynik

String query = "SELECT COF_NAME, SALES FROM COFFEES " +
"WHERE COF_NAME LIKE 'Colombian'";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
String s = rs.getString("COF_NAME");
int n = rs.getInt("SALES");
System.out.println(n + " pounds of " + s +
" sold this week.");
}
 	
	75 pounds of Colombian sold this week.

Ponieważ klauzula WHERE ogranicza selekcję tylko do jednego wiersza, zbiór wynikowy rs zawierał tylko jedną linię. Dlatego też można byłoby pominąć pętlę while:

```
rs.next();
String s = rs.getString(1);
int n = rs.getInt(2);
System.out.println(n + " pounds of " + s + " sold this week.");
```

Teraz kolumna TOTAL zostanie zmodyfikowana poprzez dodanie wartości tygodniowej sprzedaży (75) do całkowitej sumy. Suma ta zostanie wypisana na ekran:

```
String updateString = "UPDATE COFFEES " + "SET TOTAL = TOTAL + 75 " +
                      "WHERE COF_NAME LIKE 'Colombian'";
stmt.executeUpdate(updateString);
String query = "SELECT COF_NAME, TOTAL FROM COFFEES " +
                                  "WHERE COF_NAME LIKE 'Colombian'";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
                String s = rs.getString(1);
                int n = rs.getInt(2);
                System.out.println(n + " pounds of " + s + " sold to date.");
}
```

W powyższym kolumna 1 to COF_NAME, kolumna 2 to TOTAL.

Użycie przygotowanych wyrażeń

Klasa PreparedStatement dziedziczy z klasy Statement. Używa się jej do deklaracji obiektów definiujących specjalny typ wyrażeń. Używa się jej wtedy, gdy obiekt Statement miałby być wykonany wiele razy.

Główną cechą obiektu PreparedStatement jest to, że przypisane zostaje mu wyrażenie SQL już w czasie tworzenia. W rezultacie wyrażenie to przesyłane jest do DBMS, gdzie zostaje skompilowane w trybie natychmiastowym. Dlatego obiekt PreparedStatement nie będzie wykonywał wyrażenia SQL'owego, tylko wyrażenie prekompilowane (tzn. wyrażenia, których wykonanie nie wiąże się już z koniecznością jego kompilacji przez DBMS).

Najczęściej obiekt PreparedStatement jest parametryzowany. Pozwala to na jego wielokrotne użycie z różną wartością parametrów.

Tworzenie obiektu PreparedStatement

Odbywa się to podobnie jak w przypadku obiektu Statement:

```
PreparedStatement updateSales = con.prepareStatement(
   "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");
```

Dwuargumentowa zmienna updateSales w powyższym przykładzie zawiera wyrażenie SQL'owe: "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?", które zostanie wstępnie skompilowana przez DBMS.

Parametry PreparedStatement

Podaje się ich wartości, wywołując którąś z metod setXXX (setInt dla int; setString dla String, itd.). Pierwszy parametr w tych metodach to pozycja znaku zapytania w wyrażeniu SQL'owym, drugi to wartość podstawiana:

```
updateSales.setInt(1, 75);
updateSales.setString(2, "Colombian");
```

A oto większy fragment kodu:

a) bez PreparedStatement:

```
String updateString = "UPDATE COFFEES SET SALES = 75 " +
                      "WHERE COF_NAME LIKE 'Colombian'";
stmt.executeUpdate(updateString);
```

b) z PreparedStatement:

```
PreparedStatement updateSales = con.prepareStatement(
        "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ? ");
updateSales.setInt(1, 75);
updateSales.setString(2, "Colombian");
updateSales.executeUpdate():
```

executeUpdate w drugim przypadku nie ma argumentu (bo przecież zawiera już wyrażenie SQL'owe). Nadane parametrom wartości pozostaną niezmienione w przygotowanym wyrażeniu do chwili, kiedy: albo wykonana zostanie metoda setXXX, albo metoda clearParameters:

```
updateSales.setInt(1, 100);
updateSales.setString(2, "French_Roast");
updateSales.executeUpdate();  // w kolumnie SALES dla French Roast uaktualnia wartość do 100
updateSales.setString(2, "Espresso");
updateSales.executeUpdate();  // w kolumnie SALES dla Espresso uaktualnia wartość do 100
```

Pętle w uaktualnianiu wartości

```
PreparedStatement updateSales;
String updateString = "update COFFEES " +
                      "set SALES = ? where COF_NAME like ?";
updateSales = con.prepareStatement(updateString);
int [] salesForWeek = {175, 150, 60, 155, 90};
String [] coffees = {"Colombian", "French_Roast", "Espresso",
                     "Colombian_Decaf", "French_Roast_Decaf"};
int len = coffees.length;
for(int i = 0; i < len; i++) {
        updateSales.setInt(1, salesForWeek[i]);
        updateSales.setString(2, coffees[i]);
        updateSales.executeUpdate();
    }
```

Wartości zwracane przez metodę executeUpdate

executeQuery zwraca obiekt ResultSet zawierający odpowiedź SZBD na zgłoszone zapytanie. executeUpdate zwraca zaś wartość typu int mówiącą, ile wierszy w tablicach zostało zmodyfikowane:

```
updateSales.setInt(1, 50);
updateSales.setString(2, "Espresso");
int n = updateSales.executeUpdate(); // n = 1 bo tylko jeden wiersz został zmieniony
```

Kiedy metoda executeUpdate użyta jest do wykonania wyrażenia DDL, jak np. do wyrażenia tworzącego tablicę, wartością zwracaną jest int 0.

```
int n = executeUpdate(createTableCoffees); // n = 0
```

Użycie powiązań (Joins)

Powiązanie jest operacją na bazie danych, w której wiąże się dwie lub więcej tabel na podstawie wartości, które są dla nich wspólne (COFFEES oraz SUPPLIERS mają wspólną kolumnę SUP_ID, która może posłużyć do ich powiązania).

Stworzenie i wypełnienie tablicy SUPPLIERS:

```
String createSUPPLIERS = "create table SUPPLIERS " +
 "(SUP_ID INTEGER, SUP_NAME VARCHAR(40), " +
 "STREET VARCHAR(40), CITY VARCHAR(20), " +
 "STATE CHAR(2), ZIP CHAR(5))";
stmt.executeUpdate(createSUPPLIERS);

stmt.executeUpdate("insert into SUPPLIERS values (101, " +
    "'Acme, Inc.', '99 Market Street', 'Groundsville', " + "'CA', '95199'");
stmt.executeUpdate("Insert into SUPPLIERS values (49," +
    "'Superior Coffee', '1 Party Place', 'Mendocino', 'CA', " + "'95460'");
stmt.executeUpdate("Insert into SUPPLIERS values (150, " +
    "'The High Ground', '100 Coffee Lane', 'Meadows', 'CA', " + "'93966'");
```

Pobranie zawartości tablicy SUPPLIERS:

```
ResultSet rs = stmt.executeQuery("select * from SUPPLIERS");
```

SUP_ID	SUP_NAME	STREET	CITY	STATE	ZIP
------	----------	----------	----------	-----	-----
101	Acme, Inc.	99 Market Street	Groundsville	CA	95199
49	Superior Coffee	1 Party Place	Mendocino	CA	95460
150	The High Ground	100 Coffee Lane	Meadows	CA	93966

W celu odróżnienia kolumn o tej samej nazwie, poprzedza się je nazwą tabeli (np. " COFFEES.SUP_ID "). Zaznaczenie kawy kupionej Acme, Inc (stmt jest obiektem Statement zawierającym odpowiednie wyrażenie):

```
String query = "
SELECT COFFEES.COF_NAME " +
  "FROM COFFEES, SUPPLIERS " +
  "WHERE SUPPLIERS.SUP_NAME LIKE 'Acme, Inc.' " +
  "and SUPPLIERS.SUP_ID = COFFEES.SUP_ID";

ResultSet rs = stmt.executeQuery(query);
```

```
System.out.println("Coffees bought from Acme, Inc.: ");
while (rs.next()) {
   String coffeeName = rs.getString("COF_NAME");
   System.out.println("     " + coffeeName);
}
```

A oto efekt:

```
Coffees bought from Acme, Inc.:
    Colombian
    Colombian_Decaf
```

Użycie transakcji (Transactions)

Operacje na bazach danych mogą być czasem zależne od siebie. Na przykład modyfikacja pola tabeli mówiącego sprzedaży tygodniowej powinno odbywać się przed modyfikacją pola mówiącego o sprzedaży całkowitej (a nie na odwrót). Aby zapewnić poprawność danych wprowadzono transakcje. Transakcje są zbiorem składającym się z jednego lub więcej wyrażeń, które wykonywane są jako pewna całość, tak, że albo wszystkie się wykonają, albo żadne z nich.

Włączanie i wyłączanie trybu Auto-commit

Gdy tworzone jest połączenie, jest ono tworzone w trybie automatycznego zatwierdzania. Znaczy to, że każde pojedyncze wyrażenie SQL wykonane na tym połączeniu jest traktowane jako transakcja. Wynik tego polecenia pojawi się w bazie danych natychmiast po jego wykonaniu. Aby umożliwić wykonywanie transakcji składającej się z grupy wyrażeń, należy wyłączyć tryb auto-commit:

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

Zatwierdzanie transakcji

Po wyłączeniu trybu auto-commit, wyniki wyrażeń SQL będą zatwierdzane jako całość dopiero po jawnym wykonaniu metody commit. Wywołanie commit powoduje, że wszystkie zmiany spowodowane wyrażeniami w transakcji stają się w bazie danych permanentne.

```
con.setAutoCommit(false);
PreparedStatement updateSales = con.prepareStatement(
   "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");
updateSales.setInt(1, 50);
updateSales.setString(2, "Colombian");
updateSales.executeUpdate();
PreparedStatement updateTotal = con.prepareStatement(
   "UPDATE COFFEES SET TOTAL = TOTAL + ? WHERE COF_NAME LIKE ?");
updateTotal.setInt(1, 50);
updateTotal.setString(2, "Colombian");
updateTotal.executeUpdate();
con.commit();
con.setAutoCommit(true);
```

Uwaga: wyłącznie trybu auto-commit sensowne jest tylko w rzeczywiście ważnych przypadkach (bo może to powodować problemy podczas współdzielenia bazy danych przez wielu użytkowników).

Użycie transakcji do zachowania integralności danych

Metoda rollback z klasy Connection pozwala przerwać transakcję, przywracając wartości zmienione do stanu przed próbą ich modyfikacji. Przy konfliktach może to zaburzyć integralność danych. Aby zapobiec konfliktom w czasie transakcji, SZBD używa blokad (locks), tzn. blokuje dostęp do danych dla innych transakcji, poza bieżąco wykonywaną transakcją. Blokada trwa do chwili, gdy transakcja jest zatwierdzona lub odwrócona (rolled back).

Poziom blokad określany jest przez coś, co nazywa się poziomem izolacji transakcji. Blokady mogą mieć zakres od braku zabezpieczeń po bardzo twarde definicje reguł dostępu.

Przykładem poziomu izolacji transakcji jest TRANSACTION_READ_COMMITTED, który nie pozwala na dostęp do wartości dopóki nie zostanie ona zatwierdzona. Interfejs Connection zawiera pięć wartości reprezentujące różne poziomy izolacji dla JDBC.

Normalnie DBMS ustawia domyślny poziom izolacji. JDBC pozwala go odczytać metoda getTransactionIsolation z Connection. Ustawienie nowego poziomu izolacji zapewnia metoda setTransactionIsolation. Dzieje się tak, jeśli driver implementuje takie operacje.

Kiedy wywołać rollback

Rollback stosuje się wtedy, kiedy podczas wykonywania kilku wyrażeń w transakcji zgłoszony zostanie wyjątek: SQLException

```
import java.sql.*;
public class TransactionPairs {
        public static void main(String args[]) {
                String url = "jdbc:mySubprotocol:myDataSource";
                Connection con = null;
                Statement stmt;
                PreparedStatement updateSales;
                PreparedStatement updateTotal;
                String updateString = "update COFFEES " +
                                        "set SALES = ? where COF_NAME like ?";
                String updateStatement = "update COFFEES " +
                                "set TOTAL = TOTAL + ? where COF_NAME like ?";
                String query = "select COF_NAME, SALES, TOTAL from COFFEES";

                try {
                        Class.forName("myDriver.ClassName");
                } catch(java.lang.ClassNotFoundException e) {
                        System.err.print("ClassNotFoundException: ");
                        System.err.println(e.getMessage());
                }

                try {
                        con = DriverManager.getConnection(url,
                         "myLogin", "myPassword");
                        updateSales = con.prepareStatement(updateString);
                        updateTotal = con.prepareStatement(updateStatement);
                        int [] salesForWeek = {175, 150, 60, 155, 90};
                        String [] coffees = {"Colombian", "French_Roast",
                                                "Espresso", "Colombian_Decaf",
                                                "French_Roast_Decaf"};
                        int len = coffees.length;
                        con.setAutoCommit(false);
                        for (int i = 0; i < len; i++) {
                                updateSales.setInt(1, salesForWeek[i]);
                                updateSales.setString(2, coffees[i]);
                                updateSales.executeUpdate();
```

```
                        updateTotal.setInt(1, salesForWeek[i]);
                        updateTotal.setString(2, coffees[i]);
                        updateTotal.executeUpdate();
                        con.commit();
                    }

                    con.setAutoCommit(true);
                    updateSales.close();
                    updateTotal.close();

                    stmt = con.createStatement();
                    ResultSet rs = stmt.executeQuery(query);

                    while (rs.next()) {
                        String c = rs.getString("COF_NAME");
                        int s = rs.getInt("SALES");
                        int t = rs.getInt("TOTAL");
                        System.out.println(c + "     " +  s + "    " + t);
                    }

                    stmt.close();
                    con.close();

                } catch(SQLException ex) {
                    System.err.println("SQLException: " + ex.getMessage());
                    if (con != null) {
                        try {
                            System.err.print("Transaction is being ");
                            System.err.println("rolled back");
                            con.rollback();
                        } catch(SQLException excep) {
                            System.err.print("SQLException: ");
                            System.err.println(excep.getMessage());
                        }
                    }
                }
        }
}
```

Procedury zapisane

Grupa wyrażeń SQL'owych tworzących logiczna całość, pozwalająca wykonać konkretne zadanie, nazywa może zostać zadeklarowana jako procedura zapisana. Procedur zapisanych używa się do kapsułkowania zbioru operacji lub żądań, które mają być wykonane na serwerze bazy danych.

Na przykład operacje: zatrudnij, zwolnij, awansuj, wykonywane na bazie danych pracowników, mogłyby być zaimplementowane jako zapisane procedury wykonywane przez kod aplikacji.
Zapisane procedury mogą być skompilowane (po stronie bazy danych) i wykonywane z różnymi parametrami, mogą mieć też dowolną kombinację parametrów i, o, i/o. Procedury takie udostępniane są w DBMS, ale trudno tu mówić o jakimś standardzie.

Wyrażenia SQL'owe do tworzenia procedur zapisanych

Przykładowy “kod” SQL:

```
create procedure SHOW_SUPPLIERS
as
select SUPPLIERS.SUP_NAME, COFFEES.COF_NAME
from SUPPLIERS, COFFEES
where SUPPLIERS.SUP_ID = COFFEES.SUP_ID
order by SUP_NAME
```

A tak wygląda ten kod w JAVIE:

```
String createProcedure = "create procedure SHOW_SUPPLIERS " +
                         "as " +
                         "select SUPPLIERS.SUP_NAME, COFFEES.COF_NAME " +
                         "from SUPPLIERS, COFFEES " +
                         "where SUPPLIERS.SUP_ID = COFFEES.SUP_ID " +
                         "order by SUP_NAME";
Statement stmt = con.createStatement();
stmt.executeUpdate(createProcedure);
```

Procedura SHOW_SUPPLIERS będzie skompilowana i zapamiętana w bazie danych jako obiekt, który może być wywołany, podobnie do innych metod, w sposób omówiony poniżej.

Wywołanie procedur zapisanych w JDBC

W pierwszym kroku należy stworzyć obiekt CallableStatement. Obiekt ten zawiera wywołanie (a nie procedurę). Przykład, w którym użyto executeQuery (a nie executeUpdate)!

```
CallableStatement cs = con.prepareCall("{call SHOW_SUPPLIERS}");
ResultSet rs = cs.executeQuery();
```

W wyniku rs będzie przechowywał zawartość jak niżej:

```
SUP_NAME              COF_NAME
---------------       ----------------------
Acme, Inc.            Colombian
Acme, Inc.            Colombian_Decaf
Superior Coffee       French_Roast
Superior Coffee       French_Roast_Decaf
The High Ground       Espresso
```

Gdy rezultatów (zbiorów wynikowych) jest więcej, do wykonania CallableStatement powinna być użyta metoda execute.

CallableStatement jest klasą dziedziczącą z PreparedStatement, a więc może mieć parametry wejściowe. Dodatkowo, obiekt CallableStatement może mieć parametry we, wy, oraz we/wy. Parametry we/wy używane są w metodzie execute rzadko.

```
import java.sql.*;

public class DbTest {

   Connection con;
   Statement stmt;

   public void connect() {
      //  Register the driver
      try {
         Class.forName("org.gjt.mm.mysql.Driver").newInstance();
```

```
        }
        catch (Exception E) {
            System.err.println("Unable to load driver.");
        }

        //  Make the connection.
        try   {
            String url = "jdbc:mysql://localhost/test";

//  Example string to connect to a computer on a network..."jdbc:mysql:ip address
// or url, user id, password"

            con = DriverManager.getConnection(url);
            stmt = con.createStatement();
        }
        catch (SQLException e)   {
            System.out.println("SQLException:  " + e.getMessage());
        }
    }

    //  Close the connection....
    public void disconnect()  {
        try {
            con.close();
        }
        catch (SQLException e)  {
            System.out.println("SQLException:  " + e.getMessage());
        }
    }
    public void insert() {
        try {
            String insert = "INSERT INTO simpletable VALUES (1, 'record one')";
            stmt.executeUpdate(insert);

            insert = "INSERT INTO simpletable VALUES (2, 'record two')";
            stmt.executeUpdate(insert);

            insert = "INSERT INTO simpletable VALUES (3, 'record three')";
            stmt.executeUpdate(insert);
        }
        catch (SQLException e) {
            System.out.println("SQLException:  " + e.getMessage());
        }
    }

    // Create the table.
    public void create() {
        try {
            String create = "create table simpletable (id int, text char(20)) ";
            stmt.executeUpdate(create);
        }
        catch (SQLException e) {
            System.out.println("SQLException:  " + e.getMessage());
        }
    }
```

```
    public void addNewTable()  {
        try  {
          String createString;

          createString = "create table COFFEES " +
                         "(COF_NAME VARCHAR(32), " +
                         "SUP_ID INTEGER, " + "PRICE FLOAT, " +
                         "SALES INTEGER, " + "TOTAL INTEGER)";
          stmt.executeUpdate(createString);

        } catch(SQLException ex) {
            System.err.println("SQLException: " + ex.getMessage());
        }
    }
    // Drop the table if it already exists...
    public void drop()  {
        try {
            String drop = "drop table simpletable";
            stmt.executeUpdate(drop);
        }
        catch (SQLException e) {
            //  Table doesn't exist...
        }
    }

    public static void main(String[] args)  {
        DbTest table = new DbTest();
        table.connect();

        System.out.println("after connect");

        table.drop();   System.out.println("after drop");

        table.create(); System.out.println("after create");

        table.addNewTable();
        table.insert();  System.out.println("after insert");

        table.disconnect();   System.out.println("after disconnect");

    }
}
```

```
/**
 * This is a demonstration JDBC applet.
 * It displays some simple standard output from the Coffee database.
 */

import java.applet.Applet;
import java.awt.Graphics;
import java.util.Vector;
import java.sql.*;
```

```
public class OutputApplet extends Applet implements Runnable {
   private Thread worker;
   private Vector queryResults;
   private String message = "Initializing";

   public synchronized void start() {
         // Every time "start" is called we create a worker thread to
         // re-evaluate the database query.
         if (worker == null) {
            message = "Connecting to database";
          worker = new Thread(this);
            worker.start();
         }
   }

   /**
    * The "run" method is called from the worker thread.  Notice that
    * because this method is doing potentially slow databases accesses
    * we avoid making it a synchronized method.
    */

   public void run() {
         String url = "jdbc:mySubprotocol:myDataSource";
         String query = "select COF_NAME, PRICE from COFFEES";

         try {
            Class.forName("myDriver.ClassName");
         } catch(Exception ex) {
            setError("Can't find Database driver class: " + ex);
            return;
         }

         try {
            Vector results = new Vector();
            Connection con = DriverManager.getConnection(url,
                                                "myLogin", "myPassword");
            Statement stmt = con.createStatement();
            ResultSet rs = stmt.executeQuery(query);
            while (rs.next()) {
                  String s = rs.getString("COF_NAME");
                  float f = rs.getFloat("PRICE");
                  String text = s + "     " + f;
                  results.addElement(text);
            }

            stmt.close();
            con.close();

            setResults(results);

         } catch(SQLException ex) {
            setError("SQLException: " + ex);
         }
   }
```

```
    /**
     * The "paint" method is called by AWT when it wants us to
     * display our current state on the screen.
     */

    public synchronized void paint(Graphics g) {
        // If there are no results available, display the current message.
        if (queryResults == null) {
            g.drawString(message, 5, 50);
            return;
        }

        // Display the results.
        g.drawString("Prices of coffee per pound:  ", 5, 10);
        int y = 30;
        java.util.Enumeration enum = queryResults.elements();
        while (enum.hasMoreElements()) {
            String text = (String)enum.nextElement();
            g.drawString(text, 5, y);
            y = y + 15;
        }
    }

    /**
     * This private method is used to record an error message for
     * later display.
     */

    private synchronized void setError(String mess) {
        queryResults = null;
        message = mess;
        worker = null;
        // And ask AWT to repaint this applet.
        repaint();
    }

    /**
     * This private method is used to record the results of a query, for
     * later display.
     */

    private synchronized void setResults(Vector results) {
        queryResults = results;
        worker = null;
        // And ask AWT to repaint this applet.
        repaint();
    }
}
```

Nowe możliwości JDBC 2.0 API (JDK1.2)

- Przesuwanie kursora wprzód i wstecz w zbiorze wyników

- Modyfikacja tabel za pomocą metod JAVY bez konieczności używania komend SQL
- Wysyłanie wielu wyrażeń SQL'owych jako jedno polecenie (tryb wsadowy, batch)
- Użycie typu danych SQL3 jako wartości kolumn

Przesuwanie kursora w przewijanym zbiorze wynikowym.

Tworzenie przewijanego zbioru wyników (obiektu ResultSet):

```
Statement stmt =
con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery("SELECT COF_NAME, PRICE FROM COFFEES");
```

Argumenty createStatement:

1) typ obiektu:
   TYPE_FORWARD_ONLY ,
   TYPE_SCROLL_INSENSITIVE
   TYPE_SCROLL_SENSITIVE ;
2) właściwości obiektu (poziomy izolacji):
   CONCUR_READ_ONLY,
   CONCUR_UPDATABLE.

Domyślnie (bez podania argumentu) mamy: TYPE_FORWARD_ONLY oraz CONCUR_READ_ONLY (tak jak jest to w JDBC 1.0 API).

TYPE_SCROLL_INSENSITIVE : nie odzwierciedla dokonanych zmian;
TYPE_SCROLL_SENSITIVE : odzwierciedla dokonane zmiany.
Zmiany w zbiorze wyników będą widoczne, jeśli zostaną one zamknięte i ponownie otwarte (niezależnie od zadeklarowanego typu). Niemniej i tak wszystko zależy od drivera.

Metody zbioru wynikowego:

next – przesuwa kursor wprzód (zwraca false, jeśli przekroczony jest zakres zbioru)
previous – przesuwa kursor wstecz (zwraca false, jeśli przekroczony jest zakres zbioru)
afterLast – przesuwa kursor bezpośrednio za ostatni wiersz
first - przesuwa kursor do pierwszego wiersza
last – przesuwa kursor do pierwszego wiersza
beforeFirst – przesuwa kursor przed pierwszy wiersza
absolute – przesuwa kursor do wiersza o numerze podanym w argumencie, przy czym argumenty dodatnie określają numer wiersza względem początku, argumenty ujemne zaś względem końca

```
srs.absolute(4); // ustaw kursor na czwartym wierszu
srs.absolute(-4); // ustaw kursor na 7 wierszu, jeśli wierszy było 10
```

relative – przesuwa kursor względem bieżącej pozycji (argumenty dodatnie to przesunięcia wprzód, ujemne zaś to przesunięcia wstecz)

```
srs.absolute(4); // ustaw kursor na 4 wierszu
srs.relative(-3);// ustaw kursor na 1 wierszu
srs.relative(2); // ustaw kursor na 3 wierszu
```

getRow – pozwala określić bieżącą pozycję kursora

```
srs.absolute(4);
int rowNum = srs.getRow(); // będzie rowNum=4
```

isFirst – sprawdza, czy kursor jest na pierwszym wierszu (wynikiem jest wartość logiczna)
isLast – sprawdza, czy kursor jest na ostatnim wierszu (wynikiem jest wartość logiczna)
isBeforeFirst – sprawdza, czy kursor jest na przedostatnim wierszu (wynikiem jest wartość logiczna)

isAfterLast – sprawdza, czy kursor jest za ostatnim wierszem (wynikiem jest wartość logiczna)

Przykład:	Wynik:

Statement stmt =
con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery(
"SELECT COF_NAME, PRICE FROM COFFEES");
while (srs.next()) {
String name = srs.getString("COF_NAME");
float price = srs.getFloat("PRICE");
System.out.println(name + " " + price);
}
	Colombian 7.99
	French_Roast 8.99
	Espresso 9.99
	Colombian_Decaf 8.99
	French_Roast_Decaf 9.99

Przykład:	Wynik:

Statement stmt =
con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery(
"SELECT COF_NAME, PRICE FROM COFFEES");
srs.afterLast();
while (srs.previous()) {
String name = srs.getString("COF_NAME");
float price = srs.getFloat("PRICE");
System.out.println(name + " " + price);
}
	French_Roast_Decaf 9.99
	Colombian_Decaf 8.99
	Espresso 9.99
	French_Roast 8.99
	Colombian 7.99

```
if (srs.isAfterLast() == false) {
        srs.afterLast();
}
while (srs.previous()) {
        String name = srs.getString("COF_NAME");
        float price = srs.getFloat("PRICE");
        System.out.println(name + "     " + price);
}
```

moveToInsertRow, moveToCurrentRow – wyjaśnienie poniżej.

Modyfikacja bazy danych z poziomu JAVY

Do modyfikacji tabel w JDBC 1.0 API można używać tylko komend SQL (wykorzystując obiekt przesuwalny i modyfikowalny):

```
Statement stmt = con.createStatement(
                    ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
stmt.executeUpdate("UPDATE COFFEES SET PRICE = 10.99" + "WHERE COF_NAME =
FRENCH_ROAST_DECAF");
```

JDBC 2.0 API pozwala zrobić to samo z poziomu Javy (zmiany dotyczą kolumny w bieżącym wierszu, przy czym stmt utworzone powinno być jak wyżej):

```
ResultSet uprs = stmt.executeQuery("SELECT COF_NAME, PRICE FROM COFFEES");
uprs.last();
uprs.updateFloat("PRICE", 10.99);
uprs.updateRow();
```

Wynikiem będzie zmodyfikowana tabela w bazie danych:

```
COF_NAME          PRICE
----------------- -----
Colombian         7.99
French_Roast      8.99
Espresso          9.99
Colombian_Decaf   8.99
French_Roast_Decaf 10.99
```

updateXXX – metody modyfikujące zbiór wynikowy, wymagające podania dwóch argumentów (identyfikatora kolumny (nazwy lub numeru) i wartości wstawianej). Zbiór modyfikowany jest lokalnie.

updateRow – uaktualnia bazę danych (stosuje się po modyfikacji wiersza w zbiorze wynikowym przy niezmienionej pozycji kursora).

cancelRowUpdates – przywraca oryginalne wartości dla wszystkich wierszy w zbiorze wynikowym (metoda musi być wywołana przed updateRow, gdyż po updateRow jej działanie nie przyniesie efektu).

```
uprs.last();
uprs.updateFloat("PRICE", 10.99);
uprs.cancelRowUpdates();
uprs.updateFloat("PRICE", 10.79);
uprs.updateRow();
uprs.previous();
uprs.updateFloat("PRICE", 9.79);
uprs.updateRow();
```

Programowe wstawianie i usuwanie wierszy

W JDBC 1.0 API wstawianie wiersza wygląda następująco:

```
Connection con = DriverManager.getConnection("jdbc:mySubprotocol:mySubName");
Statement stmt = con.createStatement(
                    ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
stmt.executeUpdate("INSERT INTO COFFEES " + "VALUES ('Kona', 150, 10.99, 0, 0)");
```

JDBC 2.0 API pozwala zrobić to samo, wykorzystując InsertRow (pojęcie InsertRow należy interpretować jako bufor reprezentujący nowy, wstawiany wiersz):

```
ResultSet uprs = stmt.executeQuery("SELECT * FROM COFFEES");
uprs.moveToInsertRow();
uprs.updateString("COF_NAME", "Kona");
uprs.updateInt("SUP_ID", 150);
uprs.updateFloat("PRICE", 10.99);
uprs.updateInt("SALES", 0);
uprs.updateInt("TOTAL", 0);
uprs.insertRow();
```

moveToInsertRow - przesuwa kursor do wiersza wstawianego

insertRow – wstawia wiersz (tj. zmodyfikowany bufor) do zbioru wynikowego oraz modyfikuje bazę danych. Jeśli we wstawianym wierszu brak jest jakiejś wartości, to podstawiana jest wartość NULL (jeśli jest to dopuszczalne) albo zgłaszany jest wyjątek: SQLException.

moveToCurrentRow – przesuwa kursor do bieżącego wiersza, jeśli kursor ustawiony jest na wierszu wstawianym (tj. przesuwa kursor do wiersza, który był bieżącym wierszem przed przesunięciem kursora na wiersz wstawiany)

Przykład wstawiania wiersza

Uwaga: nawet przy deklaracji TYPE_SCROLL_SENSITIVE może zdarzyć się, że po wstawieniu wiersza getXXX nie zadziała dla niego poprawnie. Zobacz interfejs DatabaseMetaData, który dostarcza metod na sprawdzenie, co jest widoczne i co jest rozpoznawalne w zbiorach wynikowych dla konkretnego drivera. Najpewniejszym rozwiązaniem w takiej sytuacji jest jednak zamknięcie zbioru wynikowego i powtórne jego wczytanie.

```
import java.sql.*;

public class InsertRows {
   public static void main(String args[]) {
      String url = "jdbc:mySubprotocol:myDataSource";
      Connection con;
      Statement stmt;
   try {
      Class.forName("myDriver.ClassName");
   } catch(java.lang.ClassNotFoundException e) {
      System.err.print("ClassNotFoundException: ");
      System.err.println(e.getMessage());
   }
   try {
      con = DriverManager.getConnection(url, "myLogin", "myPassword");
      stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                  ResultSet.CONCUR_UPDATABLE);
      ResultSet uprs = stmt.executeQuery("SELECT * FROM COFFEES");
      uprs.moveToInsertRow();
      uprs.updateString("COF_NAME", "Kona");
      uprs.updateInt("SUP_ID", 150);
      uprs.updateFloat("PRICE", 10.99f);
      uprs.updateInt("SALES", 0);
      uprs.updateInt("TOTAL", 0);
      uprs.insertRow();
      uprs.updateString("COF_NAME", "Kona_Decaf");
      uprs.updateInt("SUP_ID", 150);
      uprs.updateFloat("PRICE", 11.99f);
      uprs.updateInt("SALES", 0);
      uprs.updateInt("TOTAL", 0);
      uprs.insertRow();
      uprs.beforeFirst();
      System.out.println("Table COFFEES after insertion:");
      while (uprs.next()) {
         String name = uprs.getString("COF_NAME");
         int id = uprs.getInt("SUP_ID");
         float price = uprs.getFloat("PRICE");
         int sales = uprs.getInt("SALES");
         int total = uprs.getInt("TOTAL");
         System.out.print(name + "   " + id + "   " + price);
         System.out.println("   " + sales + "   " + total);
      }
      uprs.close();
      stmt.close();
      con.close();
   } catch(SQLException ex) {
      System.err.println("SQLException: " + ex.getMessage());
```

```
    }
}
```

Usuwanie wiersza

Strategia usuwania wierszy polega na przesunięciu kursor do wiersza, który należy usunąć, i wywołaniu metody deleteRow:

```
        uprs.absolute(4);
        uprs.deleteRow();
```

deleteRow – usuwa wiersz ze zbioru wynikowego i z bazy danych

Uwaga: usuwanie wiersza może zostawić wiersz pusty w zbiorze wynikowym (zależy to od drivera). Aby nie popełniać błędów przy kolejnych przesuwaniach kursora, należy używać metody absolute. Usuwanie bowiem wierszy nie zmienia ich numeracji w zbiorze wynikowym.

Obserwacja zmian w zbiorze wynikowych

- Należy korzystać z interfejsu DatabaseMetaData
- Należy sprawować kontrolę nad poziom izolacji transakcji, ale pojawia się tu problem z efektywnością.
  con.setTransactionIsolation(TRANSACTION_READ_COMMITTED);
  con.setTransactionIsolation(TRANSACTION_REPEATABLE_READ);

refreshRow – odświeżenie zawartości wiersza bezpośrednio z bazy danych, przy czym obiekt ResultSet powinien być TYPE_SCROLL_INSENSITIVE (ale może to być bardzo kosztowne wywołanie)

```
Statement stmt = con.createStatement(
                                ResultSet.TYPE_SCROLL_SENSITIVE,
                                ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery(
                "SELECT COF_NAME, PRICE FROM COFFEES");
uprs.absolute(4);
Float price1 = uprs.getFloat("PRICE");
// do something. . .
uprs.absolute(4);
uprs.refreshRow();
Float price2 = uprs.getFloat("PRICE");
if (price2 > price1) {
        // do something. . .
}
```

Wsadowa aktualizacja bazy danych (*Batch Updates*)

Wsad jest to zbiór wielu wyrażeń modyfikujących bazę danych (dostępne w JDBC 2.0 API).

Użycie obiektów Statement do wsadowej aktualizacji bazy danych

W JDBC 1.0 API, używa się obiektu Statement z metodą executeUpdate. Kilka wywołań executeUpdate można wysłać w jednej transakcji. Jednak mimo, że zatwierdzane czy też odwracane jako jednostka, wywołania te są przetwarzane indywidualnie. Jest to prawda zarówno dla Statement, jak i PreparedStatement oraz CallableStatement, które mają własne wersje metody executeUpdate.

W JDBC 2.0 API obiekty Statement, PreparedStatement oraz CallableStatement potrafią zapamiętać listę komend, które mają być przesłane razem do wykonania (jako wsad). Są one tworzone ze stowarzyszoną listą (wstępnie pustą).

addBatch – dodaje komendy SQL do listy

clearBatch – czyści listę

executeBatch – przesyła do DBMS listę komend do wykonania (jako wsad); po wykonaniu lista komend będzie pusta

```
con.setAutoCommit(false);
Statement stmt = con.createStatement();
stmt.addBatch("INSERT INTO COFFEES VALUES('Amaretto', 49, 9.99, 0, 0)");
stmt.addBatch("INSERT INTO COFFEES VALUES('Hazelnut', 49, 9.99, 0, 0)");
stmt.addBatch("INSERT INTO COFFEES VALUES('Amaretto_decaf', 49, 10.99, 0, 0)");
stmt.addBatch("INSERT INTO COFFEES VALUES('Hazelnut_decaf', 49, 10.99, 0, 0)");
int [] updateCounts = stmt.executeBatch();
```

Polecenie:

```
int [] updateCounts = stmt.executeBatch();
```

wysyła listę komend jako wsad. W tablicy updateCounts pojawią się wartości mówiące o tym, ile wierszy zostało zmodyfikowanych przez każdą z komend.

Wyjątki w *Batch Update*

Mogą pojawić się dwa rodzaje wyjątków: SQLException i BatchUpdateException.

SQLException – jeśli pojawią się problemy z dostępem do bazy danych, albo jeśli na liście komend umieszczono komendę zwracającą zbiór wyników (do listy wstawiać tylko komendy zwracające liczbę zmodyfikowanych wierszy: INSERT INTO , UPDATE , DELETE , CREATE TABLE , DROP TABLE, ALTER TABLE , itp.).

BatchUpdateException – jeśli nie pojawił się SQLException, to znaczy że są jakieś dodatkowe problemy z wykonaniem executeBatch. Wyjątek ten, oprócz zwykłych informacji, zawiera tablicę z liczbami określającymi ilość zmodyfikowanych wierszy przez komendy, które wykonały się poprawnie przed zgłoszeniem wyjątku. BatchUpdateException dziedziczy SQLException.

```
try {
        // make some updates
} catch(BatchUpdateException b) {
        System.err.println("SQLException: " + b.getMessage());
        System.err.println("SQLState:  " + b.getSQLState());
        System.err.println("Message:  " + b.getMessage());
        System.err.println("Vendor:  " + b.getErrorCode());
        System.err.print("Update counts:  ");
        int [] updateCounts = b.getUpdateCounts();
        for (int i = 0; i < updateCounts.length; i++) {
```

```
                System.err.print(updateCounts[i] + "   ");
        }
}
```

Typy danych SQL3

Typy danych SQL mapowane są do odpowiednich typów w Javie.

SQL3	Typy JAVA	metody getXXX	metody setXXX	metody updateXXX
BLOB	Blob	getBlob	setBlob	updateBlob
CLOB	Clob	getClob	setClob	updateClob
ARRAY	Array	getArray	setArray	updateArray
Structured type	Struct	getObject	setObject	updateObject
REF (structured type)	Ref	getRef	setRef	updateRef

BLOB (Binary Large Object), CLOB (Character Large Object)

Przykład (kolumna SCORES w tablicy STUDENTS zawiera wartości typu ARRAY):

```
ResultSet rs = stmt.executeQuery(
   "SELECT SCORES FROM STUDENTS   WHERE ID = 2238");
rs.next();
Array scores = rs.getArray("SCORES");
```

Zmienna scores jest logicznym wskaźnikiem do obiektu SQL ARRAY zapisanego w tablicy STUDENTS dla wiersza (studenta) 2238.
Przykład: zapisywanie obiektu Clob:

```
Clob notes = rs.getClob("NOTES");
PreparedStatement pstmt = con.prepareStatement(
   "UPDATE MARKETS SET COMMENTS = ? WHERE SALES < 1000000",
   ResultSet.TYPE_SCROLL_INSENSITIVE,
   ResultSet.CONCUR_UPDATABLE);
pstmt.setClob(1, notes);
```

Obiekty Blob, Clob, and Array

Ważną cechą obiektów Blob , Clob i Array jest to, że można operować na nich bez ich całościowego odczytania z bazy danych. Instancje bowiem tych typów są logicznymi wskaźnikami to odpowiednich obiektów w bazie danych.
Komendy SQL oraz JDBC 1.0 lub 2.0 API można używać z obiektami Blob, Clob, Array tak jakby operowało się na aktualnych obiektach w bazie danych. Jeśli jednak obiekty te mają być obiektami z sensie JAVY (obiekty tego języka), to muszą one być ściągnięte z bazy danych w całości. Ściąganie danych w całości nazywane jest materializacją obiektu.
Aby użyć obiekt SQL'owy ARRAY w aplikacji JAVY jako tablicę, obiekt ARRAY musi zostać zmaterializowany po stronie klienta, a następnie skonwertowany do tablicy takiej, jakiej używa się w j. JAVA. Interfejsy Blob , Clob oraz Array posiadają metody do materializacji reprezentowanych przez siebie obiektów.

Typy strukturalne i wyróżnione

Typy strukturalne SQL oraz wyróżnione mogą być definiowane przez użytkownika (UDT - user-defined types). Można je stworzyć SQL'owym wyrażeniem CREATE TYPE:

```
CREATE TYPE PLANE_POINT
(
        X FLOAT,
        Y FLOAT
)
```

Aby odczytać dane takiego typy używa się klasy Struct. Obiekty tej klasy zawierają wartości dla każdego z atrybutów struktury i dlatego nie mogą być logicznymi wskaźnikami do obiektów w bazie danych. Dla instancji Struct muszą być używane getObject i setObject.

Przykład: obiekt PLANE_POINT w kolumnie POINTS tabeli PRICES.

```
ResultSet rs = stmt.executeQuery(
   "SELECT POINTS FROM PRICES  WHERE PRICE > 3000.00");
while (rs.next()) {
        Struct point = (Struct)rs.getObject("POINTS");
        // do something with point
}
```

Typ wyróżniony SQL podobny jest do typedef w C lub C++ w tym sensie, że jest to typ bazujący na istniejącym typie.

Przykład (definicja typu MONEY jako liczby typu NUMERIC który będzie reprezentowany jako wartość dziesiętna z dwoma cyframi po przecinku)

```
CREATE TYPE MONEY AS NUMERIC(10, 2)
```

Typy wyróżnione SQL (*distinct type*) mapowane są zgodnie z mapowaniem typów, jakie użyto przy ich definicji. Ponieważ typ NUMERIC mapowany jest do java.math.BigDecimal, typ MONEY mapowany będzie do java.math.BigDecimal. Aby więc odczytać obiekt typu MONEY powinno się użyć metody ResultSet.getBigDecimal lub CallableStatement.getBigDecimal. Do zapisu obiektu MONEY powinno się użyć PreparedStatement.setBigDecimal.

Możliwości standardowego rozszerzenia Javy

Pakiet javax.sql dostarcza:

Rowsets
encapsulates zbiór wierszy ze zbioru wynikowego i może pozostawać w połączeniu z bazą danych lub może być odłączony od źródła danych. rowset jest ziarnem Javy (komponentem JavaBeans) i może być stworzony na etapie projektowania i związany z innymi widocznymi komponentami (podczas tworzenia aplikacji w visual JavaBeans builder).

JNDI *tm* for Naming Databases
Java *tm* Naming and Directory Interface *tm* (JNDI) pozwala na łączenie z bazą danych używając logicznej nazwy zamiast twardo zaszytego w kodzie drivera i bazy danych.

Connection Pooling
Jest cachem otwartych połączeń, pozwalający na powtórne ich użycie.

Distributed Transaction Support
Umożliwia wykonywanie rozproszonych transakcji przez umożliwienie driverom korzystania ze standardowego "two-phase commit protocol" używanego przez Java Transaction API (JTA). (zobacz Enterprise JavaBeans components).

Savepoints (punkty odniesienia)

W JDBC 3.0 API mamy interfejs Savepoint.

Tworzenie punktu odniesienia:

```
Savepoint save1 = con.setSavepoint("SAVEPOINT_1");
```

Przykład:

```
Statement stmt = con.createStatement();
int rows = stmt.executeUpdate("INSERT INTO AUTHORS VALUES " +
                              "(LAST, FIRST, HOME) 'TOLSTOY', 'LEO', 'RUSSIA'");
Savepoint save1 = con.setSavepoint("SAVEPOINT_1");

int rows = stmt.executeUpdate("INSERT INTO AUTHORS VALUES " +
                            "(LAST, FIRST, HOME) 'MELVOY', 'HAROLD', 'FOOLAND'");
...
con.rollback(save1);
...
con.commit();
```

Zwalnianie punktu odniesienia:

```
con.releaseSavepoint(save1);
```

Użycie mapowania typów

Mapowanie typów UDT (strukturalnych lub wyróżnionych - typy SQL99) można deklarować poprzez obiekt java.util.Map. Obiekt Map może być związany z połączeniem lub może być przekazany do metody.
Deklaracja obiektu Map odbywa się przez deklarację dwóch parametrów:

1. nazwa mapowanego typu UDT
2. obiekt Class, tj. klasa Javy, do której odbywa się mapowanie (klasa ta powinna implementować interfejs SQLData)

Jeśli Connection stworzono dla drivera pozwalającego na mapowanie, instancja odpowiedzialna za mapowanie typów (java.util.Map, które w Java 2 wyparło java.util.Dictionary) początkowo zawiera puste mapowania (Struct dla STRUCT, typ podrzędny dla typów DISTINCT).

Przykład:

```
java.util.Map map = con.getTypeMap();
map.put("SchemaName.ADDRESSES", Class.forName("Addresses"));
con.setTypeMap();
```

Nazwa typu UDT powinna być w pełni wyspecyfikowana, np. dla niektórych SZBD nazwą może być: catalogName.schemaName.UDTName. Pomocą w określaniu pełnej nazwy mogą być metody DatabaseMetaData takie jak:

getCatalogs,
getCatalogTerm,
getCatalogSeparator,
getSchemas,
getSchemaTerm.

Zamiast modyfikować, można też mapowania podmieniać (wykorzystując metodę Connection.setTypeMap):

```
java.util.Map newConnectionMap = new java.util.HashTable();
newConnectionMap.put("SchemaName.UDTName1", Class.forName("className1"));
newConnectionMap.put("SchemaName.UDTName2", Class.forName("className2"));
con.setTypeMap(newConnectionMap);
```

Niektórym metodom można podać mapowanie jako argument (jak w poniższej metodzie getArray):

```
java.util.Map arrayMap = new java.util.HashTable();
arrayMap.put("SchemaName.DIMENSIONS", Class.forName("Dimensions"));
Dimensions [] d = (Dimensions [])array.getArray(arrayMap);
```

Bezparametrowe getArray odbyłoby się z mapowaniem związanym z połączeniem, a jeśli i tam nie udałoby się znaleźć odpowiedniego mapowania, to mapowanie byłoby mapowaniem standardowym.

execute

public boolean **execute**(String sql)
throws SQLException

Executes the given SQL statement, which may return multiple results. In some (uncommon) situations, a single SQL statement may return multiple result sets and/or update counts. Normally you can ignore this unless you are (1) executing a stored procedure that you know may return multiple results or (2) you are dynamically executing an unknown SQL string.
The execute method executes an SQL statement and indicates the form of the first result. You must then use the methods getResultSet or getUpdateCount to retrieve the result, and getMoreResults to move to any subsequent result(s).

Parameters:
sql - any SQL statement

Returns:
true if the first result is a ResultSet object; false if it is an update count or there are no results

Throws:
`SQLException` - if a database access error occurs

See Also:
getResultSet(), getUpdateCount(), getMoreResults()

---

getResultSet

public ResultSet **getResultSet**()
throws SQLException

Retrieves the current result as a ResultSet object. This method should be called only once per result.

Returns:
the current result as a ResultSet object or null if the result is an update count or there are no more results

Throws:
`SQLException` - if a database access error occurs

See Also:
execute(java.lang.String)

---

getUpdateCount

public int **getUpdateCount**()
throws SQLException

Retrieves the current result as an update count; if the result is a ResultSet object or there are no more results, -1 is returned. This method should be called only once per result.

Returns:
the current result as an update count; -1 if the current result is a ResultSet object or there are no more results

Throws:
`SQLException` - if a database access error occurs

See Also:
execute(java.lang.String)

---

getMoreResults

public boolean **getMoreResults**()

throws SQLException

Moves to this Statement object's next result, returns true if it is a ResultSet object, and implicitly closes any current ResultSet object(s) obtained with the method getResultSet.

There are no more results when the following is true:

(!getMoreResults() && (getUpdateCount() == -1)

Returns:

true if the next result is a ResultSet object; false if it is an update count or there are no more results

Throws:

`SQLException` - if a database access error occurs

See Also:

execute(java.lang.String)