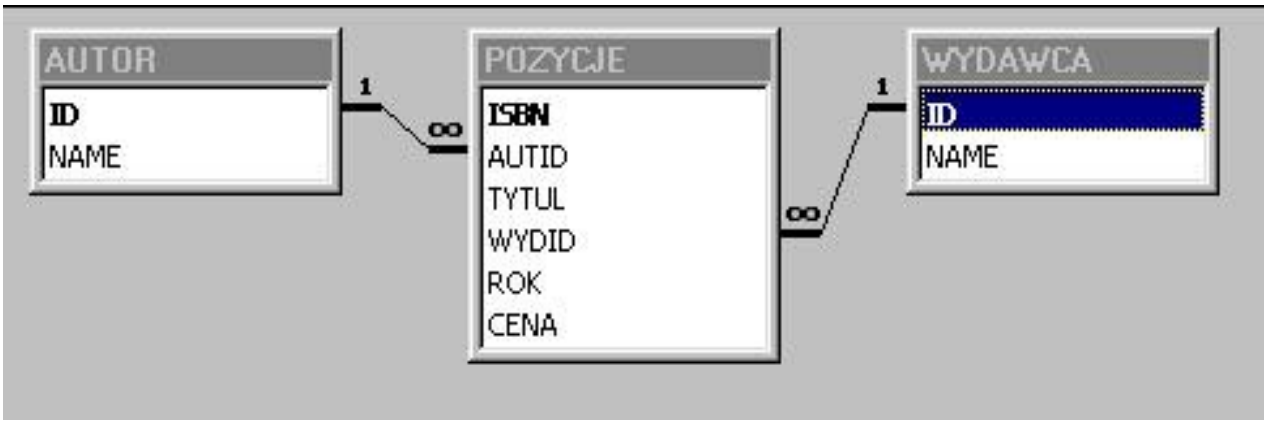
7. Java i bazy danych (JDBC)

Java jest doskonałym środowiskiem programowania dostępu do baz danych. Przyjrzymy się więc mechanizmom umożliwiającym pisanie takich programów.

7.1. Przykładowa baza danych

Schemat przykładowej bazy danych książek (może część BD księgarni internetowej) przedstawia poniższy rysunek.



Baza składa się z trzech powiązanych tabel (AUTOR, POZYCJE, WYDAWCA). Pola ID (identyfikatory) są kluczami głównymi w tabelach AUTOR i WYDAWCA, w tabeli POZYCJE odnoszą sie do nich klucze zewnętrzne (obce) AUTID i WYDID. Pole ISBN jest kluczem głownym tabeli POZYCJE. Podobnej bazy będziemy używac w przykładowych programach tego rozdziału.

Poniżej przedstawiono plik wsadowy z instrukcjami dla MySQL, które tworzą przykładową bazę. Na tej podstawie można się zorientować jak ta baza wygląda.

```
create database if not exists ksidb;
use ksidb;
drop table if exists AUTOR;
drop table if exists WYDAWCA;
drop table if exists POZYCJE;

create table AUTOR (
        AUTID integer not null AUTO_INCREMENT,
        NAME varchar(255) not null,
        PRIMARY KEY(AUTID)
        ) ENGINE=INNODB;
```

```
create table WYDAWCA (
        WYDID integer not null AUTO_INCREMENT,
        NAME varchar(255) not null,
        PRIMARY KEY(WYDID)
        ) ENGINE=INNODB;



load data infile '../BazySql/ksidb/AUTOR.TXT' replace into table AUTOR;
load data infile '../BazySql/ksidb/WYDAWCA.TXT' replace into table WYDAWCA;

create table POZYCJE (
        ISBN char(13) not null,
        AUTID integer not null,
        TYTUL varchar(255) not null,
        WYDID integer not null,
        ROK integer not null,
        CENA real,
        PRIMARY KEY(ISBN),

        INDEX(AUTID),
        FOREIGN KEY(AUTID) REFERENCES AUTOR(AUTID),

        INDEX(WYDID),
        FOREIGN KEY(WYDID) REFERENCES WYDAWCA(WYDID)

        ) ENGINE=INNODB;

load data infile '../BazySql/ksidb/POZYCJE.TXT' replace into table POZYCJE;
```

Podobny skrypt dla Derby w trybie Embedded :

```
connect 'jdbc:derby:ksidb;create=true';

drop table POZYCJE;
drop table AUTOR;
drop table WYDAWCA;


create table AUTOR (
        AUTID integer not null generated by default as identity,
        NAME varchar(255) not null,
        PRIMARY KEY(AUTID)
        );

create table WYDAWCA (
        WYDID integer not null generated by default as identity,
        NAME varchar(255) not null,
        PRIMARY KEY(WYDID)
        );

CALL SYSCS_UTIL.SYSCS_IMPORT_TABLE
(null,'AUTOR','AUTOR.TXT',null,null,null,0);
CALL SYSCS_UTIL.SYSCS_IMPORT_TABLE
(null,'WYDAWCA','WYDAWCA.TXT',null,null,null,0);

create table POZYCJE (
        ISBN char(13) not null,
```

```
        AUTID integer not null,
        TYTUL varchar(255) not null,
        WYDID integer not null,
        ROK integer not null,
        CENA real,
        PRIMARY KEY(ISBN),
        FOREIGN KEY(AUTID) REFERENCES AUTOR(AUTID),
        FOREIGN KEY(WYDID) REFERENCES WYDAWCA(WYDID)
        );

CALL SYSCS_UTIL.SYSCS_IMPORT_TABLE
(null,'POZYCJE','POZYCJE.TXT',null,null,null,0);
```

7.2. Dlaczego Java?

Zazwyczaj "poważne" RDBMS nie dostarczają gotowych (zadowalających) rozwiązań w zakresie graficznych interfejsów dostępu do baz danych lub nieco bardziej zaawansowanych środków przetwarzania danych na styku klient – serwer bazodanowy.

Zamiast tego udostępniane są programistyczne interfejsy (API), dzięki którym można takie problemy rozwiązywać.
Każdy RDBMS ma zdefiniowane dla różnych języków programowania odpowiednie interfejsy programistyczne dostępu do BD (C, C++, Cobol, PL/I etc; nie wspomnę już o Visual Basicu czy językach specyficznych dla danego RDBMS).

Są to jednak biblioteki dynamiczne, skompilowane (i zlinkowane) dla konkretnych platform sprzętowych i systemowych. Każde takie API różni się w też w zależności od RDBMS.

Programistyczny interfejs dostępu do baz danych z poziomu Javy
JDBC (Java Database Connectivity API):

- jest niezależny od maszyny bazodanowej (RDBMS)
- jest niezależny od platformy sprzętowej
- jest niezależny od systemu operacyjnego

Jest zatem jednolity i uniwersalny, a do tego łatwy w użyciu i aktualny (np. umożliwia działania, wykorzystujące nowe konstrukcje SQL – przewijalne tabele wynikowe czy typy danych SQL3 – oraz programowanie z uwzględnieniem wymagań środowisk rozproszonych).

Wszystko co chcielibyśmy robić z dowolnymi relacyjnymi bazami danych z poziomu programów użytkowych – możemy zrobić w Javie, w jej duchu i konwencji, mając jednocześnie do dyspozycji przebogate środowisko Javy.

Znając Javę możemy szybko i łatwo tworzyć aplikacje bazodanowe, które wykraczają poza samą interakcję z RDBMS i mogą włączać wszystko co Java ma do zaoferowania (programowanie sieciowe, rozproszone, multimedialne itp.)

7.3. JDBC

JDBC jest zestawem klas i interfejsów, umożliwiających:

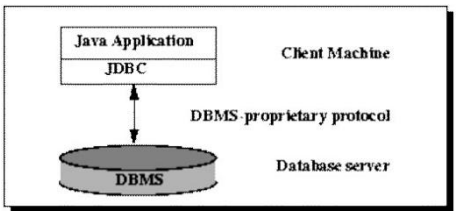
1. Połączenie z bazą danych
2. Wykonywanie instrukcji SQL na bazie danych
3. Otrzymywanie i przetwarzanie wyników instrukcji SQL (np. tabel wynikowych)

Wersja JDBC 1.0 dostarcza podstawowych środków działania na BD.
Wersje JDBC 2.0 i - aktualna JDBC 4.0 dają dodatkowe możliwości np.

- przewijalne i modyfikowalne tabele wynikowe,
- bezpośrednie modyfikowanie tabel wynikowych za pomoc metod klasy Statement
- wsadowe przetwarzanie instrukcji SQL
- obsługę typów danych SQL3
- wspomaganie JNDI (Java Naming and Directory Services) – czyli możliwość katalogowania i prowadzenia nazw źródeł danych na poziomie logicznym (podobnie jak to jest w hierarchicznym systemie plikowym)
- pooling połączeń (przechowywanie puli połączeń w pamięci w celu ew. ponownego użycia i przyspieszenia transakcji)
- transakcje rozproszone (przesyłanie danych w sieci do takich klientów jak np. przeglądarki lub laptopy)
- dostęp do praktycznie każdej formy tabularyzowanych danych (w tym arkuszy kalkulacyjnych i zwykłych plików),
- obsługę typu XML.

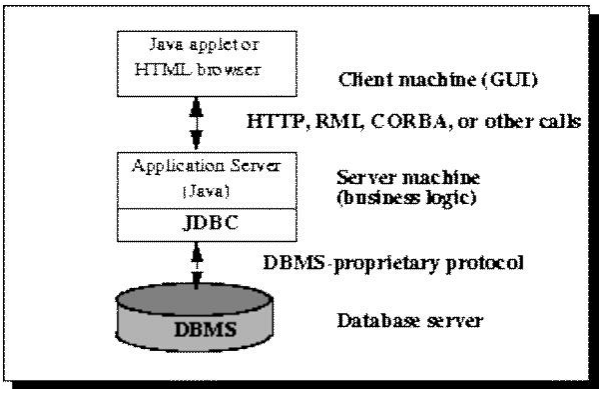
JDBC pozwala na działanie w architekturze dwu- i trzy-warstwowej.

Architektura dwuwarstwowa



Źródło: JDBC User's Guide. Javasoft

Architektura trzywarstwowa



Źródło: JDBC User's Guide. Javasoft

Zalety warstwy pośredniej: efektywność, kontrola, bezpieczeństwo, ułatwienie utrzymywania i rozwoju systemu, możliwości integracji z innymi podsystemami (middleware).

7.4. Sterowniki JDBC

Aby połączyć się z bazą danych i móc wykonywać na niej operacje należy skorzystać ze specjalnego sterownika, który tłumaczy odwołania z poziomu Javy na odwołania właściwe dla danego RDBMS.

Istnieją 4 typy sterowników.

Typ sterownika	Wyjaśnienia	Zastosowanie
1 - ***JDBC-ODBC bridge*** ***+ sterownik ODBC***	Dostęp do BD przez ODBC. JDBC-ODBC bridge komunikuje się ze sterownikiem ODBC a ten z bazą danych. Natywny kod ODBC musi być załadowany po stronie klienta.	Wszelkie BD spełniające protokół ODBC. Kiedy nie ma problemów z ładowaniem natywnego kodu po stronie klienta
2 - *Native-API* ***partly-Java driver***	Sterownik JDBC tłumaczy odwołania na natywny kod konkretnego API klienta danego RDBMS.	Sterowniki są specyficzne dla RDBMS dostarczane przez firmy np. Oracle, Sybase, IBM DB2 (UDB) etc.
3 - *JDBC-Net* ***pure Java driver***	Tylko kod javowy. Odwołania tłumaczone są na uniwersalny, niezależny od	Najbardziej elastyczne rozwiązanie, ale w przypadku użycia w

	RDBMS, protokół sieciowy, a następnie przez serwer na kody specyficzne dla RDBMS.	Internecie wymaga, by sterownik/serwer zapewniały odpowiedni poziom bezpieczeństwa
4 - *Native-protocol* ***pure Java driver***	Tylko kod javowy. Sterownik tłumaczy odwołania na specyficzny dla danego RDBMS protokół sieciowy	Pozwala na b. efektywną, bo bezpośrednią komunikację klient-serwer bazodanowy. Doskonałe w intranecie. Głównym źródłem są producenci RDBMS np. Oracle, Sybase, Informix, IBM DB2, Inprise InterBase, Microsoft SQL Server

7.5. Łączenie z bazą danych

Połączenie z bazą danych wymaga dwóch kroków:

- załadowania sterownika JDBC,
- zażądania od sterownika połączenia i ew. uzyskania go w postaci obiektu typu Connection.

Załadowanie sterownika odbywa się za pomocą wywołania statycznej metody klasy Class o nazwie forName i z argumentem – nazwa klasy (sterownika). Ogólnie metoda ta zwraca obiekt-klasę o podanej nazwie. Jeśli klasa ta nie jest załadowana do JVM, następuje jej załadowanie. Klasy-sterowniki są tak napisane, że przy ich ładowaniu rejestrują się jako obiekty typu Driver.
Zwykle obiekt ten (klasa) nie interesuje nas (dlatego w wywołaniu pomijamy zwracany rezultat).

Przykłady:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Class.forName("postgresql.Driver");
Class.forName("oracle.jdbc.driver.OracleDriver");
Class.forName("com.mysql.jdbc.Driver");
```

Nad załadowanymi sterownikami kontrolę sprawuje DriverManager (nazwa klasy). Prowadzi on listę zarejestrowanych sterowników.
Statyczna metoda **getConnection** z klasy DriverManager pozwala na uzyskanie połączenia z bazą, której URL podajemy jako argument metody.
DriverManager przegląda listę zarejestrowanych sterowników i wybiera ten, który może połączyć się z podaną bazą.

Po połączeniu z bazą zwracany jest obiekt typu Connection, który reprezentuje połączenie.

Connection con = DriverManager.getConnection(dbUrl,
userID,
password);

lub (jeśli dopuszczalne jest "domyślne" połączenie – bez podania nazwy użytkownika i hasła)

Connection con = DriverManager.getConnection(dbUrl);

Wszystkie argumenty metody getConnection są typu String.

Forma lokatorów (urli) zależna jest od sterownika i konkretnej bazy danych np.

// źródło danych ODBC o nazwie ksidb
String dbUrl = "jdbc:odbc:ksidb"

// łączenie z Oraclem z dodatkowymi specyfikacjami
String dbUrl = "jdbc:oracle:thin:user/password@(description=(address_list=(address=(protocol=tcp) (host=dbmachine)(port=1521)))(source_route=yes) (connect_data=(sid=ksidb)))";

// MySQL:
String dbUrl = "jdbc:mysql://localhost/ksidb";

Uwaga: klasa sterownika powinna być dostępna dla odwołań z naszego programu. Odpowiedni JAR można np. umieścić w katalogu jre/lib/ext.

W trakcie ładowania klasy sterownika i przy próbie połączenia mogą powstać wyjątki, które musimy obsłużyć.

```
....
 String driverName = "com.mysql.jdbc.Driver";
 String url = "jdbc:mysql://localhost/ksidb";
 String uid = "jakis";
 String pwd = "haslo";
 Connection con;

 try {
     Class.forName(driverName);
     con = DriverManager.getConnection(url, uid, pwd);
    } catch (ClassNotFoundException  exc)  {  // brak klasy sterownika
        System.out.println("Brak klasy sterownika");
        System.out.println(exc);
        System.exit(1);
    } catch(SQLException exc) {  // nieudane połączenie
```

```
        System.out.println("Nieudane połączenie z " + url);
        System.out.println(exc);
        System.exit(1);
}

.....
```

Możemy też przechwycić oba wyjątki w jednej klauzuli catch(Exception exc) ...

Innym sposobem uzyskania połaczenie jest wykorzystaie serwisów JNDI oraz tzw. źródeł danych - zapoznamy sie z nim w rozdziale "Aplikacje WEB".

Sterowniki spełniające specyfikację JDBC 4.0 (jeśli odpowiednie JARy spełniają protokół Service Provider) mogą być odnajdywane bez jawnego załadowania klasy. Np. jeśli nasza aplikacja ma dostęp do pliku derby.jar (jest na ścieżce dostępu klas), to uzyskać połaczenie możemy prościej:

```
Connection con = DriverManager.getConnection("jdbc:derby:ksidb");
```

Dzieje się tak dlatego, że w derby.jar w katalogu META-INF/services znajduje się plik java.sql.Driver, zawierający nazwę klasy sterowanika.

Przy tej okazji - parę słów o Derby.
Derby jest niewielkim i wygodnym w użyciu SZBD, całkowicie napisanym w Javie, dostarczanym w dystrybucji Javy 6.
Może działać w dwóch trybach:

- embedded - SZBD działa w tej samej maszynie wirtualnej co nasza aplikacja i nie wymaga działania serwera,
- klient-serwer (wymaga startu serwera Derby)

Bardzo ważną kwestią jest ustalenie systemowej właściwości Javy ***derby.system.home***, wskazującej na katalog, w którym zajdują się bazy danych. Jesli tej właściwości nie ustalimy, to zostanie przyjęty bieżący katalog lub katalog podany bezpośrednio przy specyfikacji URLa bazy danych.
Właściwość derby.system.home możemy określić podając opcję - **Dderby.system.home=*nazwa_katalogu*** przy starcie JVM (czy to nazej aplikacji, czy serwera Derby czy też CLI, który w Derby nazywa się **ij**.

Załóżmy, że:
JAVA_HOME wskazuje na katalog instalacyjny Javy i katalog %JAVA_HOME%/bin jest naścieżce PATH
DERBY_HOME - katalog instalacyjny Derby,
DERBY_JARS - zawiera nazwy niezbędnych bibliotek JAR z

katalogu %DERBY_HOME%/lib, w szczególności: (rozdzielone srednikami):
%DERBY_HOME%/lib/derby.jar
%DERBY_HOME%/lib/derbynet.jar
;%DERBY_HOME%/lib/derbyclient.jar
%DERBY_HOME%/lib/derbytools.jar

Start CLI w trybie embedded ze skryptem tworzącym bazę danych ksidb w katalogu D:\DerbyDbs

```
java -Dderby.system.home=D:/DerbyDbs -cp "%DERBY_JARS" -
Dij.protocol=jdbc:derby:
org.apache.derby.tools.ij nazwa_skryptu
```

Start aplikacji App w trybie embedded Derby (dostęp do bd ksidb umieszczonej w katalogu D:\DerbyDbs):

```
java -Dderby.system.home=D:/DerbyDbs -cp %DERBY_HOME%/derby.jar App

// dostęp do ksidb w programie:
Connection con = DriverManager.getConnection("jdbc:derby:ksidb");
```

Start serwera Derby (z ustaleniem derby.system.home):

```
java -Dderby.system.home=D:/DerbyDbs -cp "%DERBY_JARS"
org.apache.derby.drda.NetworkServerControl start
```

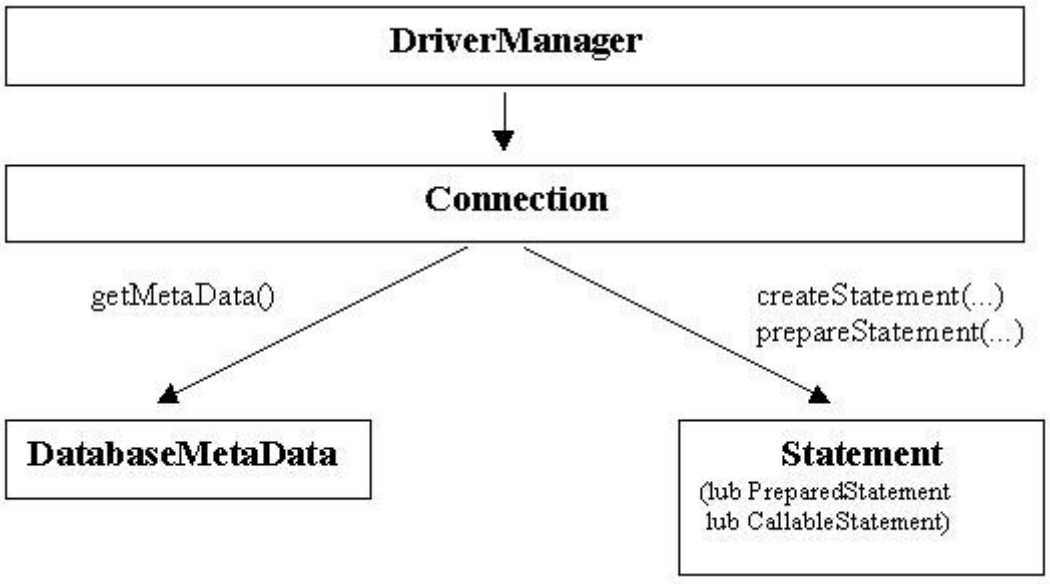
Dostęp do bazy danych za pomocą protokołu sieciowego (po starcie serwera):

```
  String driverName = "org.apache.derby.jdbc.ClientDriver";
  String url = "jdbc:derby://localhost/ksidb";
  try {
      Class.forName(driverName).newInstance();
      Connection con = DriverManager.getConnection(url);
      // ...
  }

lub jeśli dostępnym JARem jest tylko derbyclient.jar:

try {
     Connection con =
DriverManager.getConnection(jdbc:derby://localhost/ksidb);
// ...
}
```

Dostęp w trybie embedded do bazy danych umieszczonej w katalogu D:\DerbyDbs (niezależnie od tego czy właściwość derby.system.home została ustalona czy nie):

```
Connection con = DriverManager.getConnection(jdbc:derby:D:/DerbyDbs/ksidb);
```

Po uzyskaniu połączenia otrzymany obiekt Connection wykorzystujemy do operacji na bazie danych za pośrednictwem innych obiektów, który uzyskamy od obiektu Connection.
Pokazuje to poniższy rysunek.



Po zakończeniu operacji na bazie danych warto zwolnić uzyskane zasoby (takie jak Statement) oraz połączenie, wywołując odpowiednie metody close() na rzecz obiektów reprezentujących zasoby/ połączenie.
Nie zawsze jest to obowiązkowe, bo zwykle zasoby są zwalniane automatycznie przy zakończeniu programu, ale należy do dobrej praktyki programistycznej, mogą się bowiem zdarzyć takie sytuacje, kiedy zasoby nie zostaną automatycznie zwolnione.

7.6. Uzyskiwanie metainformacji o bazie danych (przykład)

```
 Connection con;
 DatabaseMetaData md;  // metadane

 // ... uzyskane połączenie
// reprezentuje obiekt con

// uzyskanie metadanych
 md = con.getMetaData();
```

```
// odpytywanie metadanych o różne
// informacje
md.getDatabaseProductName();
md.getDatabaseProductVersion();
md.getDriverName();
md.getURL();
md.getUserName();
md.supportsAlterTableWithAddColumn();
md.supportsAlterTableWithDropColumn();
md.supportsANSI92FullSQL();
md.supportsBatchUpdates();
md.supportsMixedCaseIdentifiers();
md.supportsMultipleTransactions();
md.supportsPositionedDelete();
md.supportsPositionedUpdate();
md.supportsSchemasInDataManipulation();
md.supportsTransactions();
md.supportsResultSetType(ResultSet.TYPE_SCROLL_INSENSITIVE);
md.supportsResultSetType(ResultSet.TYPE_SCROLL_SENSITIVE);
md.insertsAreDetected(ResultSet.TYPE_SCROLL_INSENSITIVE);
md.updatesAreDetected(ResultSet.TYPE_SCROLL_INSENSITIVE);
```

Przykładowe wyniki:

```
DatabaseProductName: ACCESS
DatabaseProductVersion: 3.5 Jet
DriverName: JDBC-ODBC Bridge (ODBCJT32.DLL)
URL: jdbc:odbc:ksidb
UserName: admin
supportsAlterTableWithAddColumn: true
supportsAlterTableWithDropColumn: true
supportsANSI92FullSQL: false
supportsBatchUpdates: true
supportsMixedCaseIdentifiers: true
supportsMultipleTransactions: true
supportsPositionedDelete: false
supportsPositionedUpdate: false
supportsSchemasInDataManipulation: false
supportsTransactions: true
ResultSet  TYPE_SCROLL_INSENSITIVE :true
ResultSet  TYPE_SCROLL_SENSITIVE :false
insertsAreDetected :false
updatesAreDetected :false

DatabaseProductName: MySQL
DatabaseProductVersion: 3.23.33-debug
DriverName: Mark Matthews' MySQL Driver
URL: jdbc:mysql:///test
UserName: Admin
supportsAlterTableWithAddColumn: true
supportsAlterTableWithDropColumn: true
supportsANSI92FullSQL: false
supportsBatchUpdates: false
supportsMixedCaseIdentifiers: false
supportsMultipleTransactions: true
supportsPositionedDelete: false
supportsPositionedUpdate: false
supportsSchemasInDataManipulation: false
supportsTransactions: false
ResultSet  TYPE_SCROLL_INSENSITIVE :true
ResultSet  TYPE_SCROLL_SENSITIVE :false
insertsAreDetected :false
updatesAreDetected :false
```

Interfejs DatabaseMetaData zawiera również metody umożliwiające uzyskanie informacji o:

- podtrzymywanych przez RDBMS typach danych
- zestawie tabel w bazie danych.

7.7. Wykonywanie instrukcji SQL

Do wykonywanie instrukcji SQL służy obiekt typu:

Statement (oznacza instrukcje SQL)

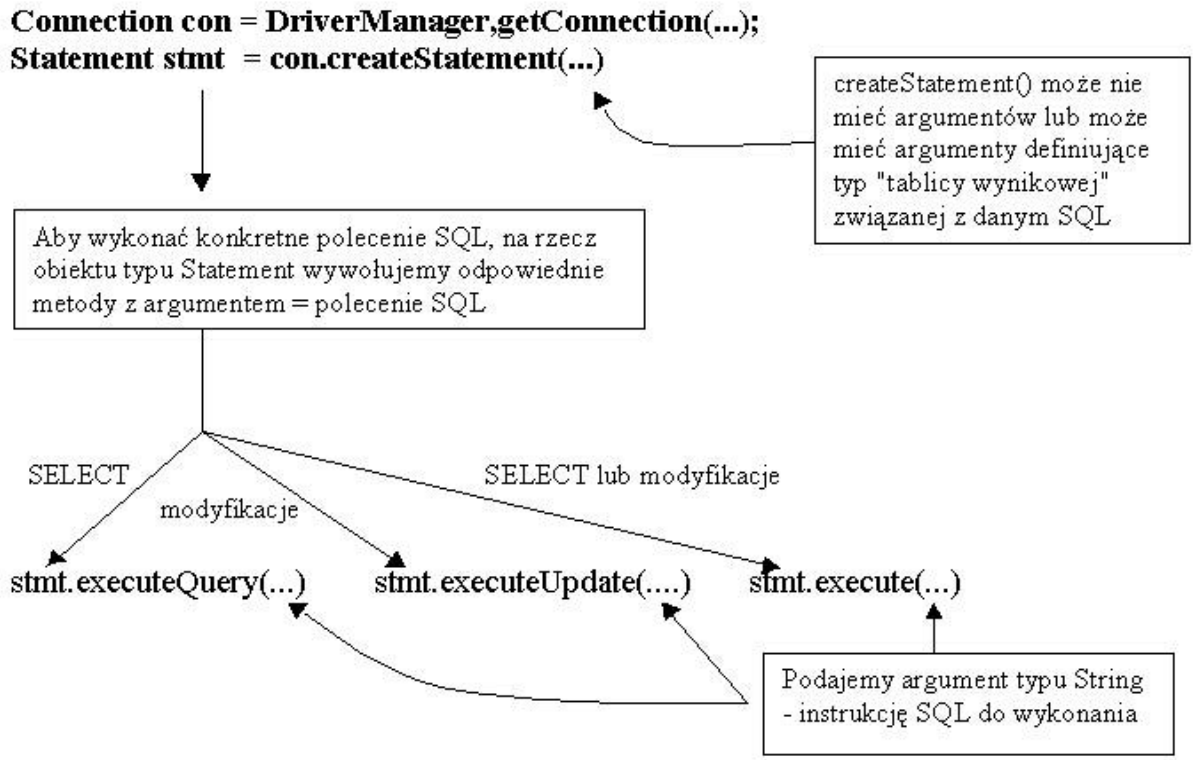
a także obiekty typu interfejsów pochodnych:

PreparedStatement (prekompilowane instrukcje SQL)

CallableStatement (przechowywane procedury)

Uzyskujemy je od obiektu typu Connection za pomocą odwołań (odpowiednio): createStatement(...), prepareStatement(...) i prepareCall(...)

Poniższy schemat obrazuje sposób posługiwania się tymi interfejsami.



Różnice pomiędzy w/w metodami są następujące.

Argumenty metod **Metody**	**SELECT...**	**CREATE TABLE...** **DROP TABLE...** **INSERT...** **UPDATE...** **DELETE...**
executeQuery(...)	**zwraca tabelę wynikową**	**-**
executeUpdate(...)	**-**	**zwraca liczbę zmodyfikowanych rekordów lub –1 (np. dla CREATE...)**
execute(...)	**wykonuje dowolną instrukcję SQL i zwraca wartość boolean (true – jeśli powstała tabela wynikowa, false – jeśli nie; prawdziwy wynik – tabelę wynikową lub liczbę zmodyfikowanych rekordów uzyskujemy za pomocą dodatkowego odwołania do obiektu Statement)**	

Ten sam obiekt typu Statement może być wielokrotnie używany do wykonania różnych instrukcji SQL np.

```
Statement stmt;
...
String[] creTab = { "CREATE TABLE  A (ID INTEGER, NAME CHAR(30))",
                    "CREATE TABLE  B (ID INTEGER, ADR CHAR(30))",
};
...
for (int i = 0; i < creTab.length; i++) {
      stmt.executeUpdate(creTab[i]);
}
stmt.executeUpdate("INSERT INTO A VALUES(1, 'Pies')");
stmt.executeUpdate("INSERT INTO B VALUES(1, 'Buda')");
...
```

7.8. Obsługa wyjątków SQLException

Zarówno createStatement() jak i metody executeUpdate(...), executeQuery(...) i execute(...) mogą generować wyjątki typu SQLException.
Wyjątki te sygnalizują błędy, wykrywane albo przez sam sterownik (np. brak jakiegoś trybu działania) , albo przez RDBMS (np. błędy składniowe w SQL lub próba naruszenia ograniczeń – jednoznaczności, spójności referencyjnej itp.).

Wyjątki te musimy obsługiwać.
A w trakcie obsługi możemy uzyskać wiele cennych informacji o przyczynie błędu.

Na przykład:

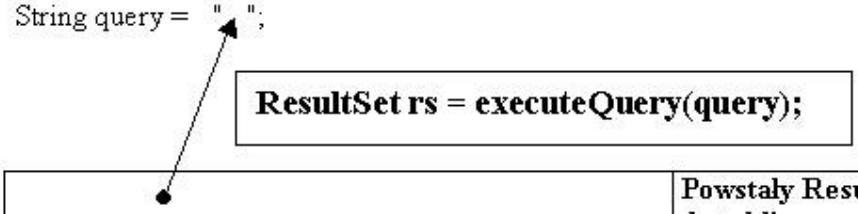
```
Connection con;
Statement stmt;
try {
     Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
     con = DriverManager.getConnection("jdbc:odbc:ksidb");
     stmt = con.createStatement();
   } catch (Exception exc)  {
     System.out.println(exc);
     System.exit(1);
   }

   String crestmt = "CREATE TABLE WYDAWCA ( " +
                    "    ID  INTEGER,      " +
                    "    NAME VARCHAR(120), " +
                    "    CONSTRAINT WYDPK PRIMARY KEY(ID) )";
   try  {
     stmt.executeUpdate(crestmt);
     System.out.println("Table created.");
   } catch (SQLException exc)  {
     // różne informacje, które można uzyskać o wyjątku SQLException
     System.out.println("SQL except.: " + exc.getMessage());  // komunikat
     System.out.println("SQL state  : " + exc.getSQLState()); // kod std
     System.out.println("Vendor errc: " + exc.getErrorCode()); // kod RDBMS
     System.exit(1);
   } finally {   // klauzula finally wykona się zawsze
      try {      // wykorzystujemy to do prawidłowego zwolnienia zasobów
        stmt.close();
        con.close();
      } catch(SQLException exc) {
        System.out.println(exc);
        System.exit(1);
      }
   }
```

7.9. Instrukcja SQL SELECT, tabele wynikowe, ResultSet i kursory

W wyniku wykonania instrukcji SELECT powstaje tabela wynikowa.
Jest ona w Javie dostępna poprzez obiekt typu ResultSet.

w kontekście:
Connection con = ... // uzyskane połączenia
Statement stmt = con.createStatement();

String query = " ";



	Powstały ResultSet rs daje dostęp do tablicy wynikowej zawierającej:
SELECT * FROM AUTOR	Wszystkie rekordy i wszystkie kolumny z tablicy AUTOR
SELECT TYTUL, CENA FROM POZYCJE WHERE CENA < 20	Kolumny TYTUL i CENA oraz rekordy spełniające warunek CENA < 20 z tablicy POZYCJE
select autor.autor, pozycje.tytul from autor, pozycje where autor.id = pozycje.autid and pozycje.tytul like '%Java%'	Połączenie tablic AUTOR i POZYCJE, dające nazwisko autora oraz tytuł książki, przy czym pokazywane są tylko te rekordy, dla których w kolumnie TYTUL występuje ciąg znaków Java

Przy czym:

- ResultSet możemy przeglądać za pomocą kursora,
- kursor inicjalnie jest ustawiony przed pierwszym rekordem tabeli wynikowej,
- w zależności od typu ResultSet możemy przemieszczać kursor tylko w kierunku od początku tabeli wynikowej do końca (typ: ResultSet.TYPE_FORWAD_ONLY) lub w obu kierunkach (typy ResultSet.TYPE_SCROLL_INSENSITIVE lub ResultSet.TYPE_SCROLL_SENSITIVE).
- interfejs ResultSet zawiera metody przemieszczające kursor, z których korzystamy przy przeglądaniu tabeli wynikowej.
- metody przemieszczające kursor zwracają wartość logiczną false, gdy żądane przemieszczenie kursora nie jest możliwe np. polecenie przejścia do następnego rekordu wyprowadza nas poza tabelę,
- jeśli kursor ustawiony jest na jakimś rekordzie tabeli wynikowej, to możemy pobrać wartości jego pól za pomocą odpowiednich metod interfejsu ResultSet; metody te zapewniają automatyczne przekształcenie typów SQL do odpowiadających im typów Javy

7.10. Przemieszczanie kursora

W kontekście:

ResultSet rs = stmt.executeQuery(query);

Odwołanie	Ustawia kursor	Typ ResultSet	
		nieprzewijalny	przewijalny
rs.beforeFirst();	Przed pierwszym rekordem	NIE	TAK
rs.first();	Na pierwszym rekordzie	NIE	TAK
rs.next();	Na następnym rekordzie	TAK	TAK
rs.previous();	Na poprzednim rekordzie	NIE	TAK
rs.last();	Na ostatnim rekordzie	NIE	TAK
rs.afterLast();	Za ostatnim rekordem	NIE	TAK
rs.absolute(n);	Na n-tym rekordzie	NIE	TAK
rs.relative(n);	Na rekordzie oddalonym o n miejsc od bieżącego (jeśli n < 0 – to do początku)	NIE	TAK

Przykład:
ile rekordów zawiera tabela wynikowa?

```
int count = 0;
while (rs.next()) count++;
```

lub:

```
rs.last();
int count = rs.getRow() // numer bieżącego rekordu
```

Uwaga: działanie na ResultSet nie oznacza, że wszystkie rekordy tabeli wynikowej są "ściągane" z RDBMS. Jest zwykle ściągana jakaś rozsądna porcja, gdy kursor zbliża się do pozycji od której te rekordy mogą być potrzebne.
Dlatego drugi sposób (dostępny tylko dla przewijalnych tabel wynikowych) jest bardziej efektywny od pierwszego

Oczywiście, ResultSet przeglądamy zwykle po to by pobierać wartości pól poszczególnych rekordów i wykonywać na nich jakieś operacje (choćby raportowania).

7.11. Odpowiedniość typów danych SQL i Javy. Pobieranie wartości pól

Typy danych zapisane w BD różnią się od typów danych Javy.
Aby sprawnie działać na wartościach pól poszczególnych rekordów trzeba wiedzieć w jaki sposób typy SQL są odzwierciedlane w typy Javy.

Standardowy typ SQL	Podstawowy typ Javy	Obiektowy typ Javy
CHAR	String	String
VARCHAR	String	String
LONGVARCHAR	String	String
NUMERIC	java.math.BigDecimal	j ava.math.BigDecimal
DECIMAL	java.math.BigDecimal	java.math.BigDecimal
BIT	boolean	Boolean
TINYINT	byte	Integer
SMALLINT	short	Integer
INTEGER	int	Integer
BIGINT	long	Long
REAL	float	Float
FLOAT	doub le	Double
DOUBLE	double	Double
BINARY	byte[]	byte[]
VARBINARY	byte[]	byte[]
LONGVARBINARY	byte[]	byte[]
DATE	java.sql.Date	java.sql.Date
TIME	java.sql.Time	java.sql.Time
TIMESTAMP	java.sql.Timestamp	java.sql.Timestamp
CLOB	java.sql.Clob	java.sql.Clob
BLOB	java.sql.Blob	java.sql.Blob
ARRAY	java.sql.Array	java.sql.Array
STRUCT	java.sql.Struct	java.sql.Struct
REF	java.sql.Ref	java.sql.Ref

Ta informacja jest ważna, jeśli chcemy tworzyć nieco bardziej elastyczne aplikacje (np. uniwersalne edytory tabel bazodanowych).

Do pobierania wartości kolumn tabeli wynikowej służą metody getTTT(...) interfejsu ResultSet, które dokonują automatycznej konwersji pomiędzy SQL-owym typem pola, a typem Javy TTT (TTT – oznacza tu jakiś typ np. int lub String).

Najprostszy szablon:

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

while (rs.next()) {
   TTT pole =  rs.getTTT(•);
   ...
}
```

Nazwa lub numer kolumny tablicy wynikowej

Możliwości użycie metod getTTT(...) wobec określonych typów SQL wyjaśnia następująca rysunek.

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP	CLOB	BLOB	ARRAY	REF	STRUCT	JAVA OBJECT
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									
getClob																				X					
getBlob																					X				
getArray																						X			
getRef																							X		
getCharacterStream											x	x	X	x	x	x									
getObject	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	X	X

getString dokonuje konwersji większości typów SQL do String

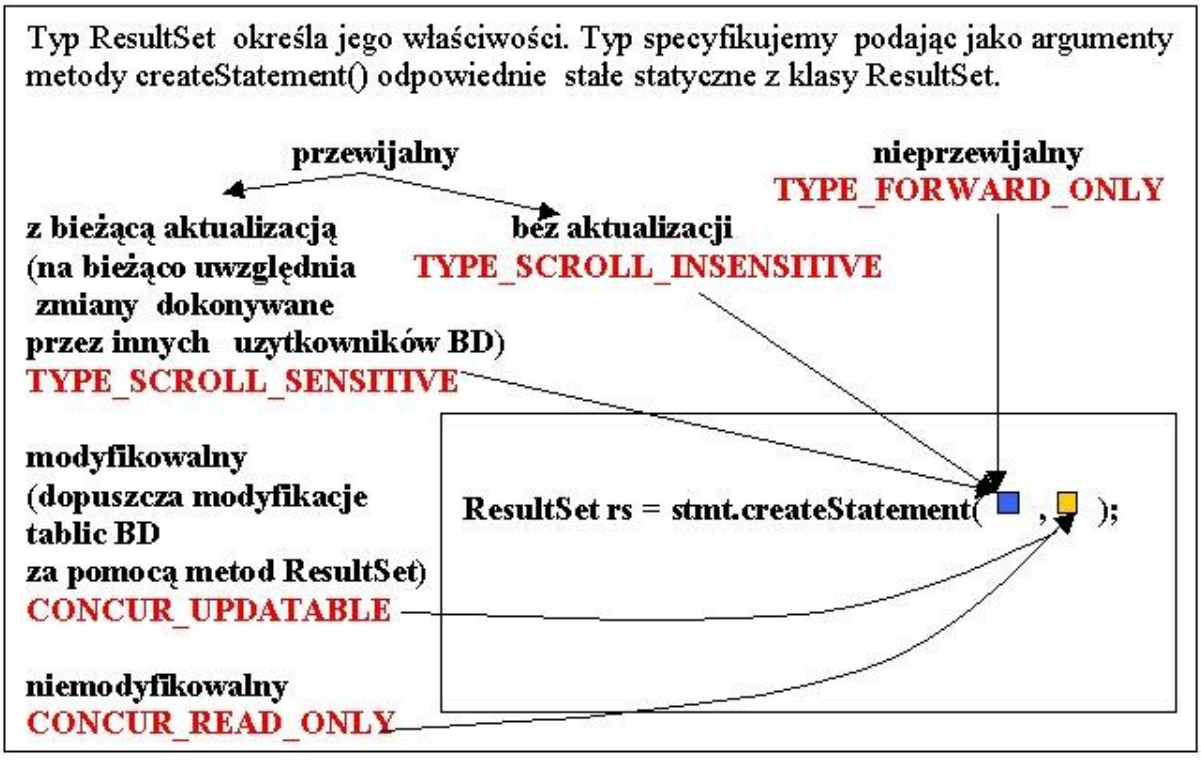
getObject może być użyte wobec każdego typu SQL, ale potem sami musimy zapisać właściwą konwersję

x – oznacza możliwość użycia metody, X – preferowaną metodę
Źródło: Getting Started with JDBC API

Przykład:

```
String sel = "select tytul, cena from pozycje where cena > 40";
   try  {
      Statement stmt = con.createStatement();
      ResultSet rs = stmt.executeQuery(sel);
      while (rs.next())  {                    //  można też użyć:
         String tytul = rs.getString(3);      //   rs.getString("tytul");
         float cena  = rs.getFloat(6);        //   rs.getFloat("cena");
         float usd = cena/4;
         System.out.println("Tytul: " + tytul);
         System.out.println("Cena : " + cena + " PLN");
         System.out.println("USD  : " + usd + " USD");
         System.out.println("-----------------");
      }
      rs.close();
      stmt.close();
      con.close();
   } catch (SQLException exc)  {
     System.out.println(exc.getMessage());
   }
```

Uwaga: należy zamykać ResultSet po wykorzystaniu (rs.close()), aby na pewno zwolnić zasoby.
ResultSet jest zamykany automatycznie, gdy zamykamy Statement (stmt.close()) lub gdy ten sam obiekt typu Statment wykorzystywany jest ponownie do wykonania innej instrukcji SQL (ew. powstaje wtedy nowy ResultSet).



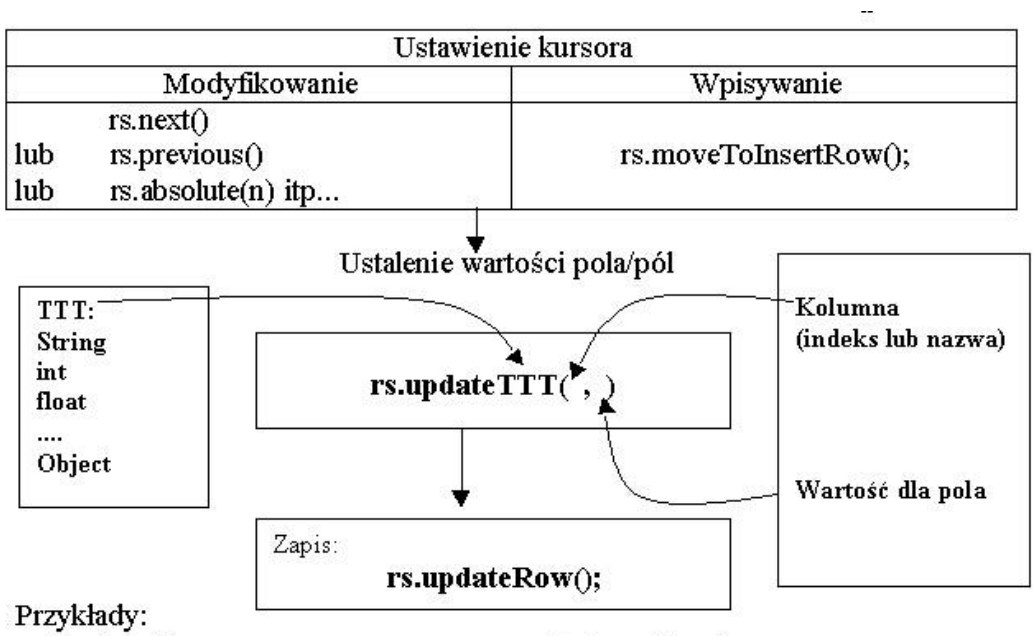
7.12. Modyfikowalny ResultSet

Jeżeli sterownik JDBC dopuszcza modyfikowalny ResultSet (typ: ResultSet.TYPE_CONCUR_UPDATABLE), to możemy użyć wobec obiektu typu ResultSet metod updateTTT(...), updateRow(), insertRow() i deleteRow(...). Pozwalają one na: dodawanie, modyfikowanie i usuwanie rekordów bez bezpośredniego użycia instrukcji SQL, operując na obiekcie typu ResultSet.

Przed wywołaniem tych metod należy ustawić kursor, tak by wskazywał odpowiedni rekord.

```
// Np. usuwanie rekordu 5
ResultSet rs;
...
rs.absolute(5);
rs.deleteRow();
```

Metoda updateRow() służy zarówno do wpisywania jak i modyfikowania rekordów. Przy wpisywaniu ustawiamy kursor na specjalnym "rekordzie" – nowym wierszu, za pomocą metody moveToInsertRow().
Ustalenie wartości pól (w nowym lub modyfikowanym) rekordzie odbywa się za pomocą metod updateTTT(...) (gdzie TTT – javowy typ pola) z dwoma argumentami: oznaczenie kolumny (indeks lub nazwa) i wpisywana wartość.



Przykłady:

```
rs.absolute(5);
rs.updateString(2, "Alabama");
rs.updateRow();
```

```
rs.moveToInsertRow();
rs.updateInt(1, 111);
rs.updateString("STAN", "Nebraska");
rs.updateRow();
```

7.13. Metainformacje o tabeli wynikowej

Specjalny obiekt typu **ResultSetMetaData** dostarcza informacji o kolumnach tabeli wynikowej. Obiekt ten uzyskujemy od obiektu ResultSet za pomocą metody getMetaData():

```
ResultSet rs ...
...
ResultSetMetaData rsmd =  rs.getMetaData();
```

a następnie używamy metod interfejsu ResultSetMetaData by otrzymać konkretne informacje.

Przykład
(mamy otwarte połaczenie Connection con i używamy dodatkowej metody
 void say(String s) { System.out.print(s); })

```
String sel ="SELECT AUTOR.ID, AUTOR.AUTOR, POZYCJE.TYTUL,"
                  "WYDAWCA.NAME AS WYDAWCA " +
                  "FROM POZYCJE,AUTOR, WYDAWCA " +
                  "WHERE WYDAWCA.ID = POZYCJE.WYDID " +
                  "AND AUTOR.ID = POZYCJE.AUTID " +
                  "ORDER BY AUTOR ASC;";
  try  {
     Statement stmt = con.createStatement();
     ResultSet rs = stmt.executeQuery(sel);
     ResultSetMetaData rsmd = rs.getMetaData();
     int cc = rsmd.getColumnCount();               // liczba kolumn
    for (int i = 1; i <= cc; i++)  {               // i-ta kolumna:
       say('\n'+ rsmd.getColumnName(i));           // - nazwa
       say(" " + rsmd.getColumnDisplaySize(i));    // - szerokość
       say(" " + rsmd.getColumnClassName(i));      // - klasa Javy
       say(" " + rsmd.getColumnType(i));           // - typ SQL
       say(" " + rsmd.getColumnTypeName(i));       // - typ RDBMS
     }
     stmt.close();
     con.close();

  } catch (SQLException exc)  {
    System.out.println(exc.getMessage());
  }
```

Uwaga:
typ SQL – stała int z java.sql.Types
klasa Javy – jakiej klasy obiekt zwróci getObject() użyty wobec tej kolumny ResultSet

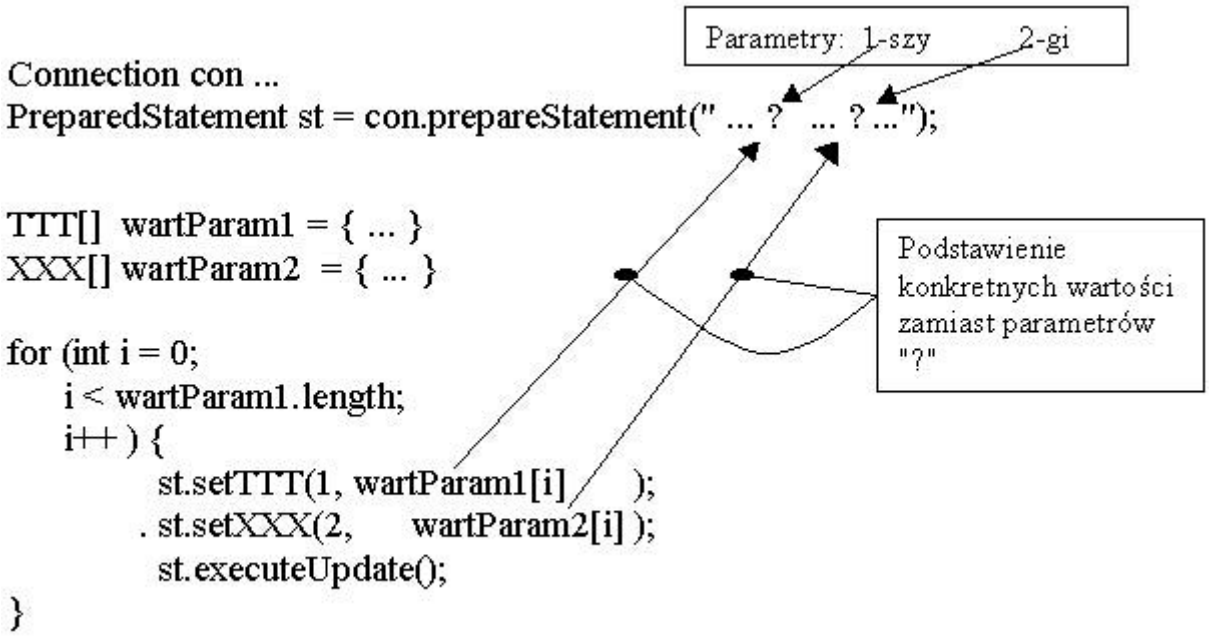
Możliwy wynik:

```
ID 11 java.lang.Integer 4 LONG
AUTOR 255 java.lang.String 12 TEXT
TYTUL 255 java.lang.String 12 TEXT
WYDAWCA 120 java.lang.String 12 TEXT
```

7.14. Instrukcje prekompilowane

Prekompilowane instrukcje SQL są przed wykonaniem wysyłane do RDBMS i podlegają tam prekompilacji, swoistemu przygotowaniu, które następnie przyspiesza ich wielokrotne użycie (wykonanie).
Oczywiście nie ma sensu wykonywać tej samej instrukcji wielokrotnie. Dlatego w instrukcjach prekompilowanych używane są znaki ? jako symbole parametrów. Przy każdym wykonaniu w miejsce znaków ? podstawia się odpowiednie wartości.
Instrukcje prekompilowane w Javie reprezentowane są jako obiekty typu PreparedStatement.
Tworzymy instrukcję prekompilowaną za pomocą metody prepareStatement (zamiast createStatement), podając jako argument odpowiednią instrukcję SQL (z parametrami ?). Zwykle instrukcje takie wykonujemy w pętli ustalając wartości parametrów za pomocą metod set... interfejsu PrepareStatement .

Jeśli TTT i XXX oznaczają (różny) typ Javy (np. int, String, float, etc) to (przykładowe) wykonanie instrukcji prekompilowanej można przedstawić schematycznie w następujący sposób:



Przykład:

```
Connection con;
PreparedStatement stmt;
...
  String[] wyd =  { "PWN", "PWE", "Czytelnik", "Amber", "HELION",
                              "MIKOM" };
  int beginKey = 10,
  try  {
     stmt = con.prepareStatement("INSERT INTO WYDAWCA VALUES(?,?)");
     for (int i=0; i < wyd.length; i++)   {
       stmt.setInt(1, beginKey + i);
       stmt.setString(2, wyd[i]);
       stmt.executeUpdate();     //         Uwaga: inna forma
executeUpdate()
 }
     con.close();
   } catch(SQLException exc)  {
      System.out.println(exc);
   }
```

7.15. Obsługa transakcji

Transakcja to grupa instrukcji, traktowanych jako całość: jeżeli któraś z nich nie zostanie wykonana – nie mogą być wykonane inne; np. przelew z konta na konto)

Sterowniki JDBC zwykle używają domyślnie trybu autoCommit (wykonanie każdej instrukcji INSERT, DELETE, UPDATE powoduje zmiany w bazie danych; transakcją jest jedna instrukcja).

```
Connection con;
Statement stmt;
...
con.autoCommit(false);
try {
     stmt.executeUpdate(pierwszaInstrukcjaTransakcji);
     stmt.executeUpdate(drugaInstrukcjaTransakcji);
     ...
     con.commit();  //
} catch(SQLException exc) {
   con.rollback();  //
}
```

Zapis zmian do bazy danych

Wycofanie się z wykonanych instrukcji

7.16. Zastosowanie architektury "Model-View-Controller" przy tworzeniu graficznych interfejsów BD za pomocą komponentów Swingu

Java wyjątkowo dobrze nadaje się do tworzenia graficznych interfejsów użytkownika dostępu do baz danych. Szczególną rolę odgrywają tu komponenty Swingu ze względu na ich elastyczność, atrakcyjność graficzną, niezależny od platformy i konfigurowalny wygląd oraz realizację koncepcji MVC. Szczególnie atrakcyjnym (w kontekście interakcji z bazą danych) komponentem Swingu jest tabela (klasa JTable).

Zobaczymy teraz przykład realizacji modelu danych tabeli dla przedstawienia tabeli wynikowej instrukcji SELECT (i nie tylko – praktycznie każdego ResultSetu).
Komórki tabeli będą edytowalne, a ich edycja ma powodować zmiany w tabelach BD.

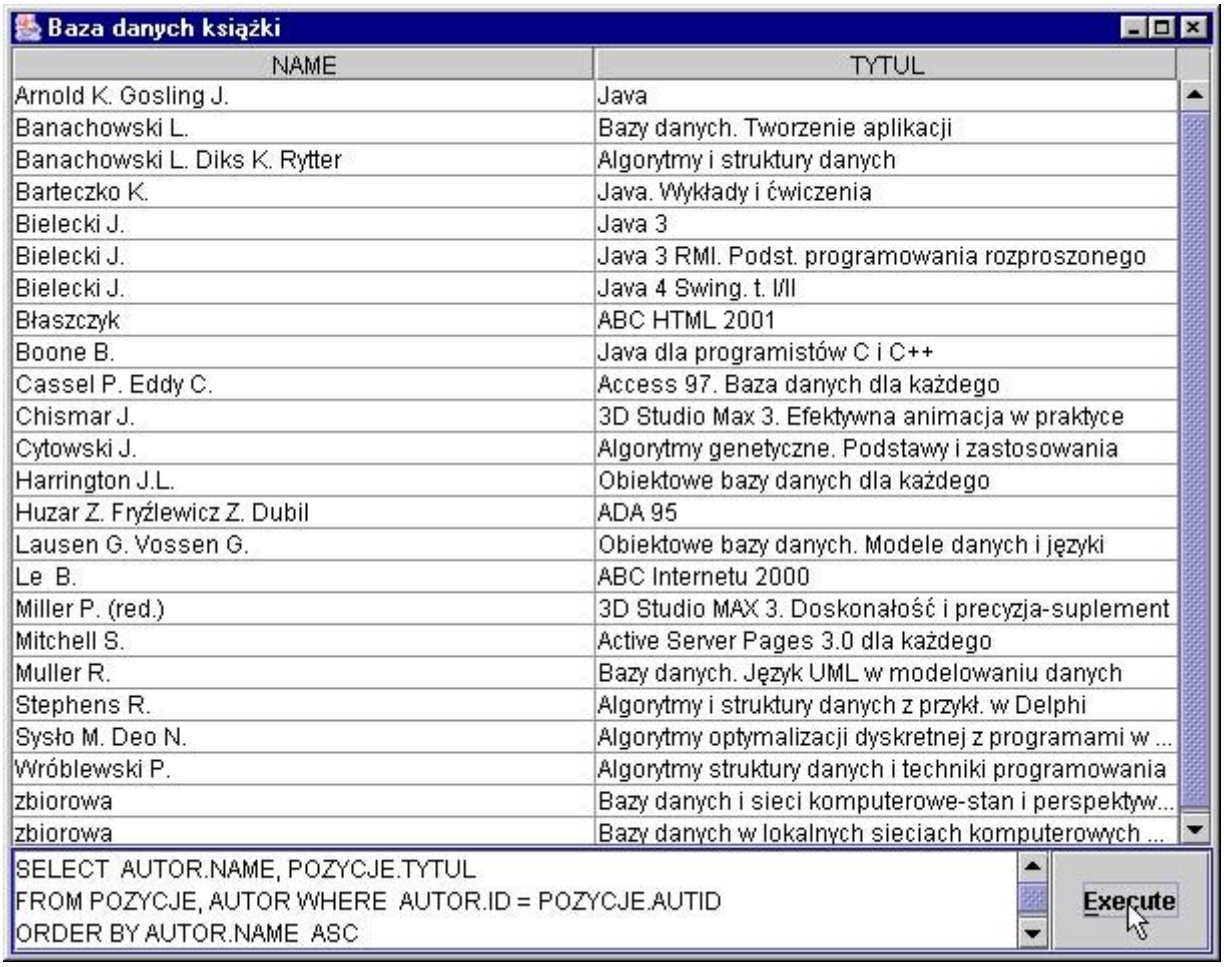
```
// Model danych dla tabeli pokazującej dowolny ResultSet

import java.util.*;
import java.sql.*;
import javax.swing.*;
import javax.swing.table.*;
import javax.swing.event.*;

public class DbTable extends AbstractTableModel  {
    private Connection con;
    private ResultSet rs;
    private String[]  columnNames;
    private int[]  columnTypes;
    private boolean[] readOnly;
    private String tableName = "";
    private List  rows;
    private ResultSetMetaData   md;
    private boolean editable = false;

public DbTable(Connection conn, String query, ResultSet resultSet, boolean
ed)  {
   rs = resultSet;
   editable = ed;
   con = conn;
   tableName = getTableName(query);
   try {
     md = rs.getMetaData();
     int cc =  md.getColumnCount();
     columnNames = new String[cc];
     columnTypes = new int[cc];
     readOnly = new boolean[cc];
     for(int col = 0; col < cc; col++) {
       columnNames[col] = md.getColumnName(col+1);
       columnTypes[col] = md.getColumnType(col+1);
```

```
        readOnly[col] = md.isReadOnly(col+1);
      }

      rows = new ArrayList();
      while (rs.next()) {
         List row = new ArrayList();
         for (int i = 1; i <= getColumnCount(); i++) {
           row.add(rs.getObject(i));
         }
         rows.add(row);
      }
      rs.close();
      fireTableChanged(null); // Nowa tabela
    } catch (SQLException ex) {
            System.out.println(ex.getMessage());
      }
}

// Niedoskonala wersja
private String getTableName(String q)  {
   if (q == null || q.equals("")) return "";
   StringTokenizer st = new StringTokenizer(q);
   while (st.hasMoreTokens())  {
     String w = st.nextToken();
     w = w.toUpperCase();
     if (w.equals("FROM")) {
         String t = st.nextToken();
         if (t.indexOf(',') == -1) return t;
         break;
     }
   }
   return "";
}

// Obowiązkowe metody interfejsu TableModel
public String getColumnName(int column) {
    if (columnNames[column] != null) return columnNames[column];
     else return "";
}

public Class getColumnClass(int column) {
   String type;
   Class c = null;
   try {
     type = md.getColumnClassName(column+1);
     c = Class.forName(type);
     }
     catch (Exception e) {
       return super.getColumnClass(column);
   }
   return c;
}

public boolean isCellEditable(int row, int column) {
    if (!editable) return false;
    if (tableName.equals("")) return false;
    return !readOnly[column];
}

public int getColumnCount() {
   return columnNames.length;
```

```
}

public int getRowCount() {
   return rows.size();
}

public Object getValueAt(int r, int c) {
    List row = (List)rows.get(r);
    return row.get(c);
}

 public String dbValue(int col, Object value) {
    int type;
    if (value == null) return "null";
    type = columnTypes[col];

    switch(type) {
        case Types.CHAR:
        case Types.VARCHAR:
        case Types.LONGVARCHAR:
             return "\'"+value.toString()+"\'";
        case Types.BIT:
            return ((Boolean)value).booleanValue() ? "1" : "0";
        default:
            return value.toString();
        }
}

 public void setValueAt(Object value, int r, int c) {
    List row = (List) rows.get(r);
    String oldval = row.get(c).toString();
    if (oldval.equals(value.toString())) return;
    String colName = getColumnName(c);
    String query = " update " + tableName +
                   " set " + colName + " = " + dbValue(c, value) +
                   " where ";
    for(int j = 0; j < getColumnCount(); j++) {
      colName = getColumnName(j);
      if (colName.equals("")) continue;
      if (j != 0)  query += " and ";
      query += colName +" = "+  dbValue(j, getValueAt(r, j));
    }
    query +=  ";";
    try {
       Statement s = con.createStatement();
       int updCount = s.executeUpdate(query);
       row.set(c, value);
       System.out.println("Zmieniono rekordów: " + updCount);
    } catch (SQLException e) {
        System.out.println(query);
        System.out.println(e.getMessage());
   }
 }
}
```

Stworzymy również prosty graficzny interfejs do wydawania zleceń SQL oraz oglądania wyników w postaci tabeli.

Przykładowe okno tego programiku wygląda tak:



a jego kod pokazano poniżej:

```
// Testowy interfejs SQL

import java.sql.*;
import javax.swing.*;
import javax.swing.text.*;
import java.awt.event.*;
import java.awt.*;
import java.util.*;

public class TestSQL extends JFrame implements ActionListener {

   private Connection con = null;
   private Statement stmt;
   private ResultSet rs = null;
   private String query;
   private JTable table = new JTable();
   private JTextArea ta = new JTextArea(3,40);
   private DefaultListModel history = new DefaultListModel();
   private JList hlis = new JList(history);
   private JWindow wh = new JWindow();
```

```
 public TestSQL(String URL, String driver, String user,
                       String passwd) {
      super("Baza danych książki");
      setDefaultCloseOperation(3);

      try {
         Class.forName(driver);
         con = DriverManager.getConnection(URL);
         stmt = con.createStatement();
      } catch (Exception exc)  {
         System.out.println(exc.getMessage());
         System.exit(1);
      }

      JScrollPane scrollpane = new JScrollPane(table);
      scrollpane.setPreferredSize(new Dimension(600, 400));
      JPanel p = new JPanel();
      p.setLayout(new BorderLayout());
      ta.setLineWrap(true);

      JScrollPane tsp = new JScrollPane(ta);
      p.add(tsp, "Center");
      JButton b = new JButton("Execute");
      b.setMnemonic('E');
      b.addActionListener(this);

      p.add(b, "East");
      p.setBorder(BorderFactory.createLineBorder(Color.blue));
      getContentPane().add(scrollpane, "Center");
      getContentPane().add(p, "South");

      createHistoryList();

      pack();
      setVisible(true);
  }

 public void actionPerformed(ActionEvent e)  {
    String new_query = ta.getText();
    if (new_query.equals(query)) return;
    query = new_query;
    if (!history.contains(query)) history.addElement(query);
    execute(query);
 }

 void execute(String query)  {
    try  {
      rs = stmt.executeQuery(query);
      DbTable dbt = new DbTable(con, query, rs, true);
      table.setModel(dbt);
    } catch(SQLException exc)  {
       System.out.println(exc.getMessage());
    }
 }

void createHistoryList()  {

  ta.addMouseListener(new MouseAdapter()  {
      public void mouseReleased(MouseEvent e)  {
```

```
        if (e.isPopupTrigger())  {
           wh.pack();
           wh.show();
        }
      }
    });

    hlis.addMouseListener(new MouseAdapter()  {
      public void mouseClicked(MouseEvent e)  {
        if (e.getClickCount() == 2)   {
           String s = (String) hlis.getSelectedValue();
           if (s != null) ta.setText(s);
           wh.setVisible(false);
        }
      }
    });

    JScrollPane hsp = new JScrollPane(hlis);
    hsp.setPreferredSize(new Dimension(200, 300));
    JPanel hp = new JPanel(new BorderLayout());
    hp.setBorder(BorderFactory.createLoweredBevelBorder());
    hp.add(hsp, "Center");
    JPanel bhp = new JPanel();

    ActionListener hlHandler = new ActionListener()  {
      public void actionPerformed(ActionEvent e)  {
        String cmd = e.getActionCommand();
        if (cmd.equals("Cancel")) wh.setVisible(false);
        else if (cmd.equals("Clear all")) history.clear();
        else  {
          int index = hlis.getSelectedIndex();
          if (index == -1) return;
          if (cmd.equals("Clear")) history.remove(index);
          else if (cmd.equals("Execute"))  {
            String new_query = (String) hlis.getSelectedValue();
            if (new_query.equals(query)) return;
            query = new_query;
            wh.setVisible(false);
            execute(query);
            ta.setText(query);
          }
        }
      }
    };

    JButton  b = new JButton("Cancel");
    b.addActionListener(hlHandler);
    bhp.add(b);
    b = new JButton("Clear");
    b.addActionListener(hlHandler);
    bhp.add(b);
    b = new JButton("Clear all");
    b.addActionListener(hlHandler);
    bhp.add(b);
    b = new JButton("Execute");
    b.addActionListener(hlHandler);
    bhp.add(b);
    hp.add(bhp, "South");

    wh.getContentPane().add(hp);
    ta.addMouseListener(new MouseAdapter()  {
```

```
        public void mouseReleased(MouseEvent e)  {
           if (e.isPopupTrigger())  {
             wh.setLocation( getX()+10, getY()+50);
             wh.pack();
             wh.show();
           }
        }
      });
  }

  public static void main(String[] args) {
    String driverName = "com.mysql.jdbc.Driver";
    String url = "jdbc:mysql:///ksidb";
    String uid = "pies";
    String pwd = "kuba";
    new TestSQL(url, driverName, uid, pwd);
  }
}
```

7.17 Zadania i ćwiczenia

Ze względu na wagę problematyki programowania dostępu do baz danych ćwiczenia będą dość obszerne, ale za to proste i stopniowo wprowadzające w temat.
Używana w ćwiczeniach przykładowa baza danych książek zrealizowana jest w MySQL.
Po instalacji MySQL i sterownika Connector/J (jego plik jar można umieścić w katalogu jre/lib/ext) należy stworzyć bazę danych uruchamiając plik wsadowy o następującej postaci:

```
create database if not exists ksidb;
use ksidb;
drop table if exists AUTOR, WYDAWCA, POZYCJE;

create table if not exists AUTOR (
        ID integer not null AUTO_INCREMENT,
        NAME varchar(255) not null,
        PRIMARY KEY(ID)
        );
load data infile 'AUTOR.TXT' replace into table AUTOR;


create table if not exists WYDAWCA (
        ID integer not null AUTO_INCREMENT,
        NAME varchar(255) not null,
        PRIMARY KEY(ID)
        );
load data infile 'WYDAWCA.TXT' replace into table WYDAWCA;


create table if not exists POZYCJE (
        ISBN char(13) not null,
        AUTID integer not null,
```

```
        TYTUL varchar(255) not null,
        WYDID integer not null,
        ROK int not null,
        CENA real,
        PRIMARY KEY(ISBN),
        FOREIGN KEY(AUTID) REFERENCES AUTOR(ID),
        FOREIGN KEY(WYDID) REFERENCES WYDAWCA(ID),
        );
load data infile 'POZYCJE.TXT' replace into table POZYCJE;
```

Przykładowe pliki z danymi dołaczone są na CD.

CZĘŚĆ 1. DBLETY

Proponowane "deblety" są krótkimi programikami ćwiczeniowymi pokazującymi podstawowe działania z bazami danych z poziomu Javy. Tutaj pokazane sa częściowymi programy, które należy uzpełnić, tak by właściwie działały.

Zad. 1 (Łączenie z bazą danych i uzyskiwanie metainformacji o bazie danych)

Program pokazuje, że do połączenia z BD potrzebne są dwa kroki:
- załadowanie odpowiedniej klasy sterownika
- uzyskanie połączenie poprzez uzyskanie obiektu typu Connection

Od obiektu Connection możemy otrzymać metainformacje związane ze sterownikiem, systemem zarządzania BD i samą BD poprzez uzyskanie obiektu typu DatabaseMetaData, który możemy odpytywać za pomocą wielu metod interfejsu DatabaseMetaData.

Należy napisać program, łączący się z bazą danych książek i uzyskujący niektóre informacje o bazie danych.

Częściowy gotowy program (bez części odpowiedzialnej za połączenie z bazą i uzyskanie metainformacji jest pokazany poniżej. Nalezy go uzupełnić o brakujące fragmenty kodu.

```
import java.sql.*;
import java.lang.reflect.*;


public class Con1  {


 // .... tu czegoś brakuje

 public Con1()  {
   // ... i tu rownież
 }

// Metoda raportująca informacje zebrane w DatabaseMetaData
// w wywołaniach metody info podano jako argumenty nazwy metod tego
interfejsu
// a w metodzie info korszystamy z metod refleksji;
// ten sposób oprogramowania jest zaawansowany, ale wygodny, bo dużo mniej
pisania
// i kod jest bardziej klarowny
```

```
// klauzula throws SQLException mówi o tym, że w trakcie działania
reportInfo może powstać wyjątek
// SQLException, ale nie będziemy go tu obsługiwać, obsługę przekażemy do
miejsca wywołania
// czyli bloku try w konstruktorze

void reportInfo() throws SQLException {

   info("getDatabaseProductName");
   info("getDatabaseProductVersion");
   info("getDriverName");
   info("getURL");
   info("getUserName");

   info("supportsAlterTableWithAddColumn");
   info("supportsAlterTableWithDropColumn");
   info("supportsANSI92FullSQL");
   info("supportsBatchUpdates");
   info("supportsMixedCaseIdentifiers");
   info("supportsMultipleTransactions");
   info("supportsPositionedDelete");
   info("supportsPositionedUpdate");
   info("supportsSchemasInDataManipulation");
   info("supportsTransactions");

   System.out.println("ResultSet  TYPE_SCROLL_INSENSITIVE :" +
       md.supportsResultSetType(ResultSet.TYPE_SCROLL_INSENSITIVE));
   System.out.println("ResultSet  TYPE_SCROLL_SENSITIVE :" +
       md.supportsResultSetType(ResultSet.TYPE_SCROLL_SENSITIVE));
   System.out.println("insertsAreDetected :" +
       md.insertsAreDetected(ResultSet.TYPE_SCROLL_INSENSITIVE));
   System.out.println("updatesAreDetected :" +
       md.updatesAreDetected(ResultSet.TYPE_SCROLL_INSENSITIVE));
 }

// Metoda info korzysta z metod refleksji do wywołania metod podanych
"przez" nazwy.
 void info(String metName) {
   Class mdc  = DatabaseMetaData.class;
   Class[] paramTypes =  { };
   Object[] params =  { };
   String infoTyp;
   if (metName.startsWith("get"))
     infoTyp = metName.substring(3,metName.length());
   else infoTyp = metName;
   try  {
     Method m = mdc.getDeclaredMethod(metName, paramTypes);
     System.out.println(infoTyp + ": " + m.invoke(md, params));  //
dynamiczne wywołanie metody
   } catch(Exception exc)  {   // Możliwe powody wyjątków: nie ma takiej
metody, niewłaściwe wywołanie
     System.out.println(exc);
   }
 }

  public static void main(String[] args)  {
   new Con1();
 }
```

Zadanie 2 (tworzenie tabeli)

Uwaga: Przed wykonaniem tego zadania należy zrobić kopię bazy.

Przykład pokazuje następujące ważne kwestie:

- polecenia DDL lub SQL są wykonywane za pośrednictwem obiektu typu Statement
- obiekt Statement uzyskujemy od obiektu Connection za pomocą zlecenia createStatement()
- wszelkie zmiany w bazie danych (w tym usuwanie i tworzenie tabel) wykonujemy za pomocą metody executeUpdate aktywowanej na rzecz obiektu Statement
- "na tym samym" obiekcie Statement możemy wykonać dowolnie wiele poleceń SQL/DDL
- od obiektu typu SQLException (wyjątku SQL) możemy się dowiedzieć wielu rzeczy np. o standardowy "SQL State" lub zależny od dostawcy RDBMS kod błędu.

Zadanie: utworzyć tabelę WYDAWCA z kolumnami:
ID (całkowitoliczbowy klucz pierwotny)
NAME (łańcuch znakowy zmiennej długości o maks. 255 znakach) – nazwa wydawcy.

Napisać program w taki sposób, by zawsze (niezależnie od tego czy już w bazie istnieje tabela WYDAWCA) była tworzona nowa tabela.
Uwaga: tabela WYDAWCA jest tabelą macierzystą dla tabeli POZYCJE (klucz zewnętrzny tabeli POZYCJE odnosi się do klucza pierwotnego tabeli WYDAWCA; relacja ta wymusza spójność referencyjnej).

```
import java.sql.*;

public class Cre1 {

  static public void main(String[] args) {
    new Cre1();
  }

Statement stmt;

Cre1()  {
   Connection con = null;
   try {
      // łączenie z bazą i utworzenie obiektu typu Statement
   } catch (Exception exc)  {
     System.out.println(exc);
     System.exit(1);
   }

   // metoda dropTable jest naszą własną metodą napisaną dla skrócenia
programu
   // usuwa ona tabelę podaną jako argument
   // Aby w każdych okolicznościach stworzyć nową tabelę WYDAWCA
   // musimy usunąć ew.  już istniejącą tabelę WYDAWCA
   dropTable("POZYCJE"); // usunięcie tabeli pochodnej, będącej w relacji z
tabelą WYDAWCA
   dropTable("WYDAWCA"); // usinięcie tabeli WYDAWCA

   String crestmt = ...

   try  {
```

```
     ....                  // wykonanie polecenia zapisanego w crestmt

   } catch (SQLException exc)  {                      // przechwycenie
wyjątku:
      System.out.println("SQL except.: " + exc.getMessage());
      System.out.println("SQL state  : " + exc.getSQLState());
      System.out.println("Vendor errc: " + exc.getErrorCode());
      System.exit(1);
   } finally {
      try {
        stmt.close();
        con.close();
      } catch(SQLException exc) {
        System.out.println(exc);
        System.exit(1);
      }
   }
}

private void dropTable(String tname  )  {
   // ....

}
```

Ćwiczenie dodatkowe:

1. przywrócić bazę danych do postaci wyjściowej
2. skompilować i wykonać program bez odwołania dropTable("POZYCJE")
3. obejrzeć dokładnie komunikaty o wyjątkach

Zad. 3 (wpisywanie rekordów do tabeli)

Dodać do tabeli WYDAWCA trzy rekordy reprezentujące jakichś wydawców.

Przykład ilustruje następujące kwestie:

- instrukcja SQL do wpisywania ma postać INSERT... (w kilka różnych formach)
- przy wpisywaniu rekordów używamy executeUpdate(...)
- przy wpisywaniu i modyfikowaniu metoda ta zwraca liczbę wpisanych/zmodyfikowanych rekordów,
- dane typu znakowego (CHAR, VARCHAR, LONGVARCHAR) są podawane w SQL w apostrofach

```
import java.sql.*;

public class Ins1 {

  static public void main(String[] args) {
    new Ins1();
  }
```

```
Statement stmt;

Ins1()  {
   Connection con = null;
   try {
     //...
   } catch (Exception exc)  {
     System.out.println(exc);
     System.exit(1);
   }

   String[] ins =  { "INSERT INTO WYDAWCA VALUES (1, \'Wyd 1\')",
                             "INSERT INTO WYDAWCA VALUES (2, \'Wyd 2\')",
                             "INSERT INTO WYDAWCA VALUES (3, \'Wyd 3\')",
                   };
   int insCount = 0;   // ile rekordów wpisano
   try  {
     for (int i=0; i < ins.length; i++) // wpisywanie rekordów
     // ...
   }
//...
}
}
```

Dodatkowe ćwiczenie:
wykonać program ponownie i
zobaczyć jak naruszone jest ograniczenie jednoznaczności klucza pierwotnego

Modyfikacja: użyć prekompilowanych instrukcji.
Ta modyfikacja ilustruje użycie instrukcji prekompilowanych:

- ins. prekompilowana przygotowywana i wykonywana jest za pomocą obiektu typu PreparedStatement
- obiekt ten jest tworzony poprzez (inne!) odwołanie do obiektu Connection: prepareStatement(...)
- argumentem prepareStatement jest String, w którym występują znaki zapytania – miejsca na "parametry" podstawiane przy kolejnych wykonaniach polecenia prekompilowanego
- metody set... interfejsu PreparedStatement pozwalają podstawiać za parametry-znaki zapytania kolejne wartości
- trzeba wiedzieć jaki jest typ wartości (pola) i użyć odpowiedniej metody set...

```
   // ...
   String[] wyd ={ "PWN", "PWE", "Czytelnik", "Amber", "HELION", "MIKOM" };
   int beginKey = 10,
       insCount =  0;
   try  {
     // przygotowanie instrukcji prekompilowanej
     stmt = con.prepareStatement("INSERT INTO WYDAWCA VALUES(?,?)");
     for (int i=0; i < wyd.length; i++)   {
       // ... ?
     }
```

```
  con.close();
} catch(SQLException exc)  {
   System.out.println(exc);
}
// ...
```

Zadanie 4 (SELECT i ResultSet)

Uwaga: aby wykonać to zadanie należy przywrócić wyjściową wersję bazy

Wyprowadzić z tabeli POZYCJE wszystkie rekordy, spełniające warunek CENA > 30 zł i pokazać dla każdego z nich tytuł i cenę w PLN i (obliczoną) cenę w USD.

Program ma ilustrować następujące kwestie:

- instrukcja SELECT wykonywana jest za pomocą executeQuery(..)
- executeQuery zwraca obiekt typy ResultSet (tzw. tabela wynikowa)
- z ResultSet związany jest tzw. kursor, który wskazuje bieżący rekord w tabeli wynikowej
- inicjalnie kursor ustawiony jest przed pierwszym rekordem tabeli wynikowej
- kursor możemy przesuwać (tylko w stronę końca tabeli, o ile nie wymagaliśmy tego, by ResultSet mógł być "skrolowany") za pomocą metody next() interfejsu ResultSet
- wartości poszczególnych kolumn z bieżącego rekordu możemy pobrać za pomocą metod get...

```
String sel = "SELECT AUTOR, TYTUL, CENA FROM POZYCJE WHERE CENA > 40";
try  {
   Statement stmt = con.createStatement();
   ResultSet rs = stmt.executeQuery(sel);
   while (rs.next())  {
      String tytul = // ... ?
      float cena  = //  ... ?
      float usd = cena/4;
      System.out.println("Tytul: " + tytul);
      System.out.println("Cena : " + cena + " PLN");
      System.out.println("USD  : " + usd + " USD");
      System.out.println("-----------------");
   }
   stmt.close();
   con.close();
} catch (SQLException exc)  {
  System.out.println(exc.getMessage());
}
```

Dodatkowe zadanie: wyprowadzić wszystkie rekordy tabeli wynikowej powstałej na skutek wykonania instrukcji SELECT od końca tabeli; a następnie wyprowadzić rekordy 3, 7 i 9.

To zadanie winno zilustrować:

- przewijalny ResultSet (typ deklarujemy w createStatement)
- absolutne pozycjonowanie w ramach tabeli wynikowej
- użycie metainformacji o kolumnach tabeli wynikowej (obiekt typu ResultSetMetaData możemy uzyskać za pomocą zlecenia wobec ResultSet – getMetaData(), następnie możemy go "odpytać" o różne informacje za pomocą metod interfejsu ResultSetMetaData)
- uniwersalność metody getString: jeśli potrzebna nam tylko znakowa reprezentacja informacji zawartej w kolumnach tabeli, getString (użyte wobec bieżącego rekordu ResultSet) dokona właściwej konwersji dla każdego typu danych w BD (oprócz typów definiowanych i SQL3

```
  String sel = // ... ?
   try  {
      Statement stmt =
con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                          ResultSet.CONCUR_READ_ONLY);
      ResultSet rs = stmt.executeQuery(sel);
      ResultSetMetaData rsmd = rs.getMetaData();
      int cc = rsmd.getColumnCount();
      for (int i = 1; i <= cc; i++)
        System.out.print(rsmd.getColumnLabel(i) + "     ");

      System.out.println("\n------------------------ przewijanie do góry");

      // ... ?

      System.out.println("\n------------------------ pozycjonowanie abs.");
      int[] poz =  { 3, 7, 9  };
      for (int p = 0; p < poz.length; p++)  {
         System.out.print("[ " + poz[p] + " ] ");
         // ... ?
         for (int i = 1; i <= cc; i++)
             System.out.print(rs.getString(i) + ", ");
         System.out.println("");
      }
      stmt.close();
      con.close();
   } catch (SQLException exc)  {
     System.out.println(exc.getMessage());
   }
```

CZĘŚĆ 2. Java jako język tworzenia interfejsów bazodanowych

Druga część ćwiczeń polega na przedstawieniu Javy jako wygodnego języka do tworzenia graficznych interfejsów użytkownika dostępu do baz danych. Prezentowany wczesniej (w p. 16) program korzysta z uniwersalnego modelu danych tabeli Swing, odzwierciedlającego tabelę wynikową zapytania SQL lub jakikolwiek inny ResultSet.

Program składa się z dwóch plików źródłowych, definiujących dwie klasy o tych samych nazwach co pliki:

- DbTable.java – odzwierciedla dowolny ResultSet w modelu danych tabeli Swingowej (JTable),
- TestSQL.java – jest graficznym interfejsem, umożliwiającym uzyskiwanie wyników zapytań SELECT w postaci tabeli JTable, dla której modelem jest klasa DbTable.

Skompilować obie klasy i uruchomić program.

Po uruchomieniu TestSQL jako głównej klasy uzyskujemy możliwość wpisywania poleceń SQL w wielowierszowym polu edycyjnym u dołu okna. Kliknięcie w przycisk Execute lub naciśnięcie alt-e (mnemonika) powoduje wykonanie instrukcji SELECT (nie tylko!) i przedstawienie jej wyników w tabeli w centrum okna. Tabela pozwala na bezpośrednie edytowanie pól w bazie danych (dbl-click na polu tabeli) – jeśli jest to możliwe na podstawie danego ResultSet.
Wydane polecenia SQL gromadzone są w postaci "listy historii"/
Możemy do niej sięgać poprzez prawy klik na polu edycyjnym.
Podwójne kliknięcie na elemencie historii (zapamiętanym poleceniu) powoduje jego przepisanie do pola edycyjnego.
U dołu okna listy historii znajdują się przyciski o następującym znaczeniu:
"Cancel" - zamknij listę
"Clear" – usuń zaznaczony element
"Clear all" – usuń wszystki elementy historii
"Execute" – wykonaj zaznaczoną na liście historii instrukcję SQL

Komentarze:

1. Model danych tabeli (plik DbTable.java) jest dość uniwersalny – pozwala przedstawić dowolny ResultSet w postaci tabeli Swingowej
2. Konkretne GUI (TestSQL) może być dowolnie zmieniane bez ingerencji w związki DB – Swing table model
3. Realizacja tego GUI zajęła mało czasu: okazuje się, że program w Javie o zaawansowanych możliwościach może liczyć mniej niż 200 wierszy (tzn. niezwykle krótki). Nb. większość kodu tego programu zajmuje się obsługą listy historii.

Stworzyć bardziej interesujące GUI. Zastanowić się w jaki sposób można by było uzyskiwac w klasie DBTable dostęp do ResultSet i prezentację go w modlu bez przepisywania rekordów do wewnętrznych struktur danych,