Zachodniopomorski Uniwersytet Technologiczny w Szczecinie

Wydział Informatyki

KATEDRA INŻYNIERII OPROGRAMOWANIA http://kio.wi.zut.edu.pl/

INŻYNIERIA OPROGRAMOWANIA

Zaawansowane Techniki Programowania Java

#02 : JDBC (*java.sql*)

Prowadzący:

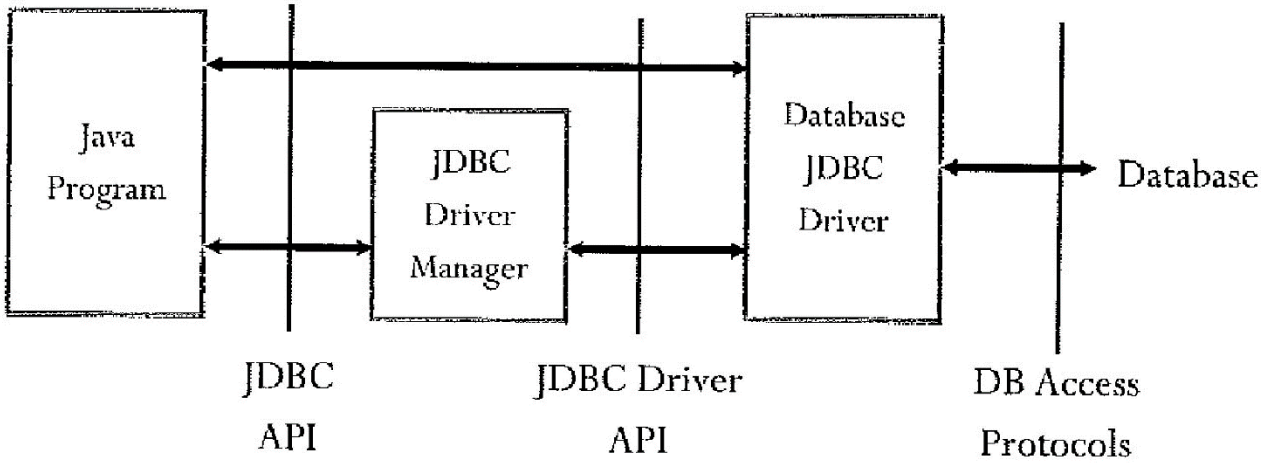
Krzysztof Kraska

email: kkraska@wi.zut.edu.pl

Szczecin, 17 marca 2018 r.

JDBC Overview

- JDBC is based on the X/Open SQL Call Level Interface (CLI) and Microsoft's ODBC.
- ODBC API is a C interface (procedural). JDBC is intended for direct use from Java (Object-oriented)
- JDBC is the Java API for executing SQL statements in database independent ways

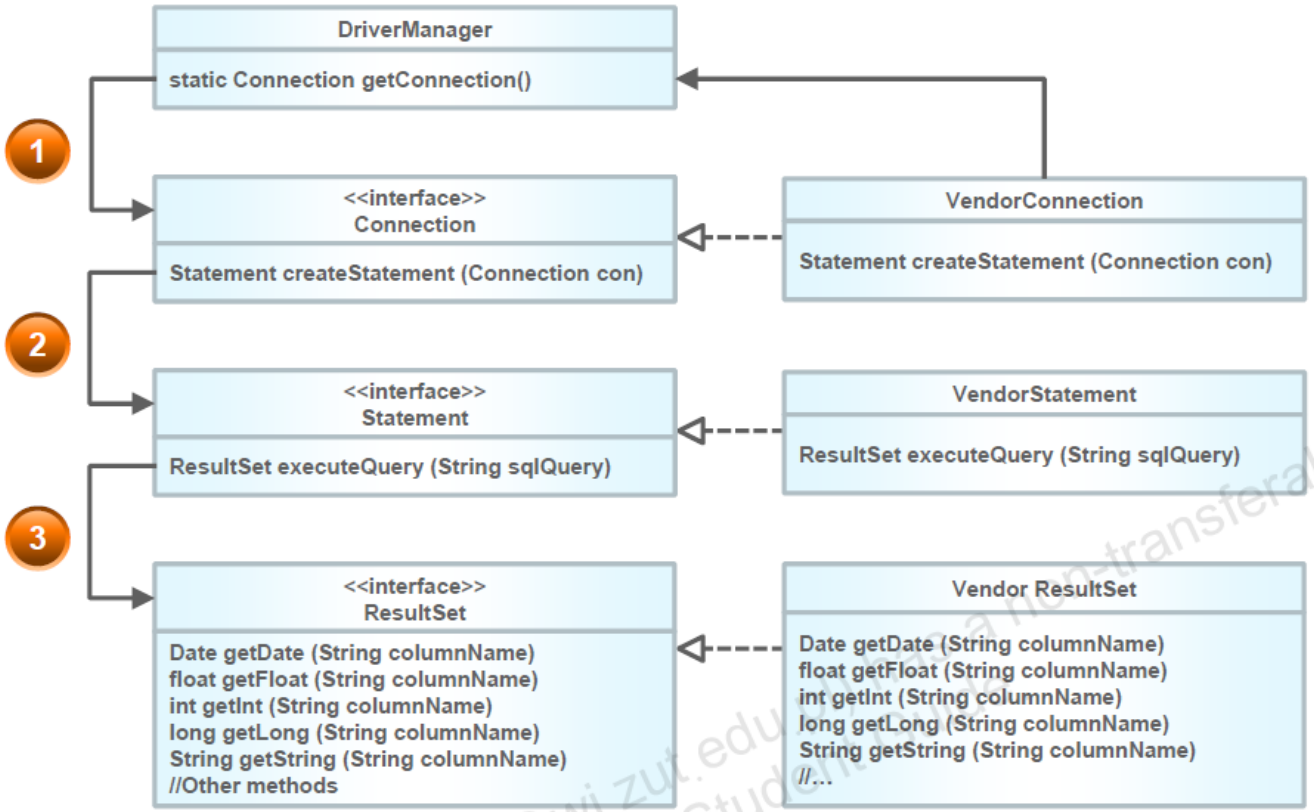


JDBC Drivers

- JDBC drivers are software that adapt the JDBC API to the specific database access protocols.
- Each vendor will supply a JDBC driver - with different class names.
- The drivers all look the same because JDBC is based on interfaces.
- A driver must supply concrete implementation for the JDBC API interfaces. The most common ones are shown here:
  - java.sql.Connection
  - java.sql.Statement
  - java.sql.PreparedStatement
  - java.sql.CallableStatement
  - java.sql.ResultSet
- Also a driver must supply a class which implements the java.sql.Driver interface.

JDBC

Using the JDBC API



Szczecin, 17 marca 2018 r.

JDBC Driver Types (1 of 2)

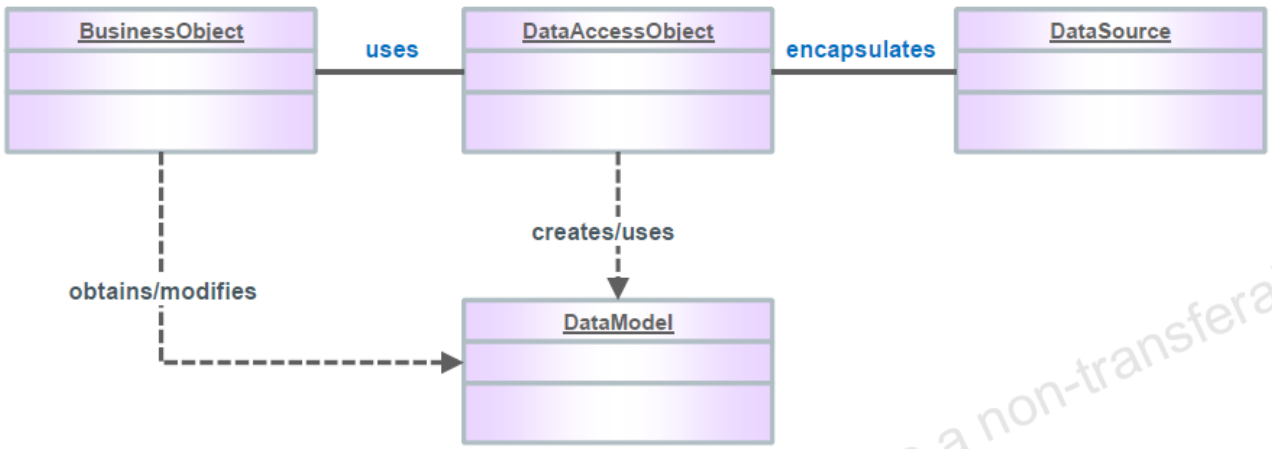
- Type1: JDBC-ODBC bridge plus ODBC driver
  - Implemented in both Java and native code
  - Must be preinstalled on client
- Type 2: Native API partly Java
  - Implemented in both Java and native code
  - Communicates to database in vendor-specific protocol (DB2, Oracle, and so forth)
  - Must be preinstalled on client

JDBC Driver Types (2 of 2)

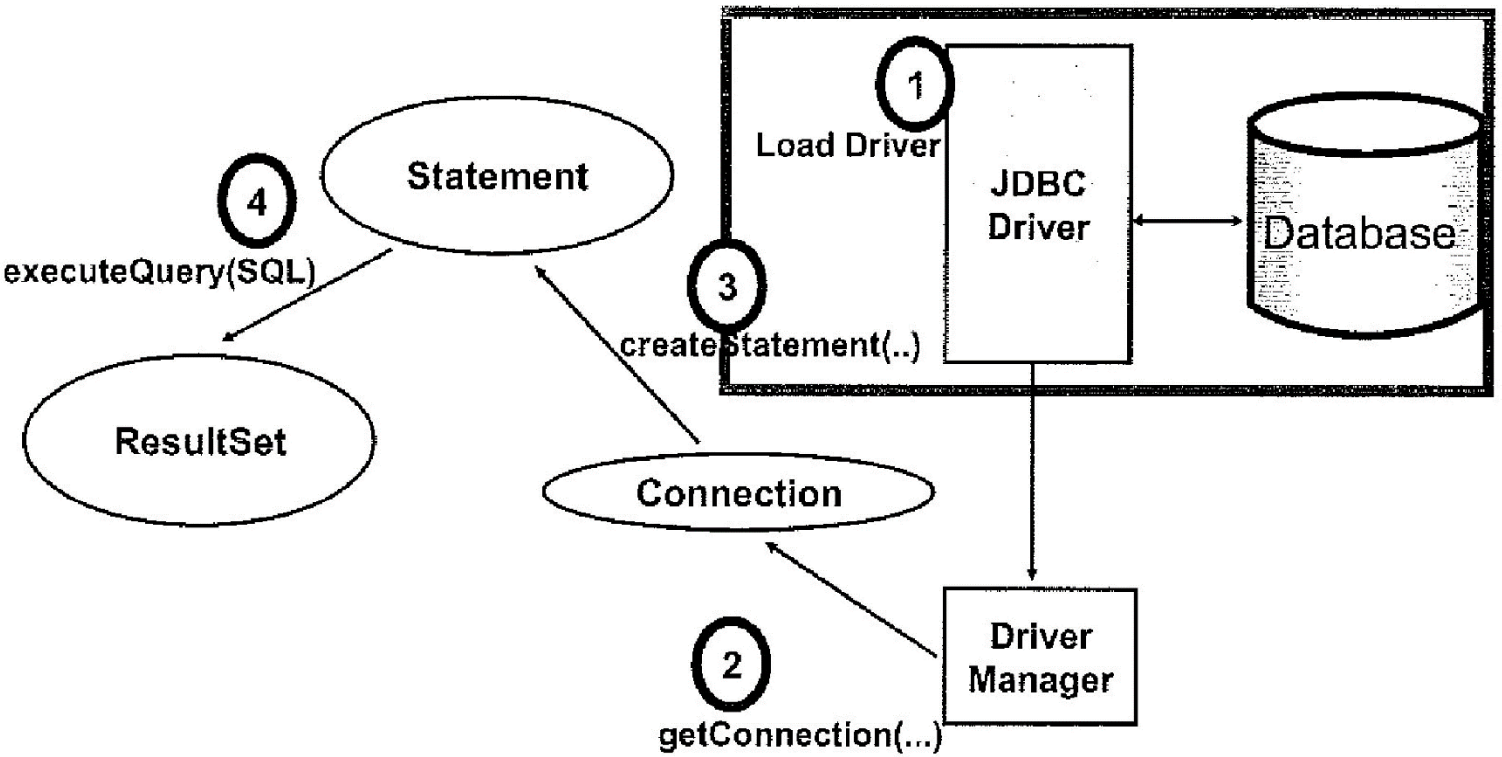
- Type 3: JDBC-Net pure Java
  - Implemented in pure Java
  - Communicates in standard protocol (HTTP, HTTPS, and so forth)
  - Can be downloaded and probably will pass through proxies (firewalls)
- Type 4: Native protocol pure Java
  - Implemented in pure Java
  - Communicates in vendor specific protocol
  - Can be downloaded - may have problems passing through proxies

JDBC

The Data Access Object (DAO) Pattern



Simple JDBC Client (1 of 4)



Loading a Driver

- Each driver must be loaded by your application code
  - `Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");`
- More than one driver can be loaded at once
  - Only needs to happen once per driver
  - Loading registers the driver with the `DriverManager`
    - Usually in a static initializer in the Driver class

JDBC

DriverManager

Any JDBC 4.0 drivers that are found in the class path are automatically loaded. The `DriverManager.getConnection` method will attempt to load the driver class by looking at the file `META_INF/services/java.sql.Driver`. This file contains the name of the JDBC driver's implementation of `java.sql.Driver`. For example, the contents of the `META-INF/services/java.sql.driver` file in `derbyclient.jar` contain `org.apache.derby.jdbc.ClientDriver`.

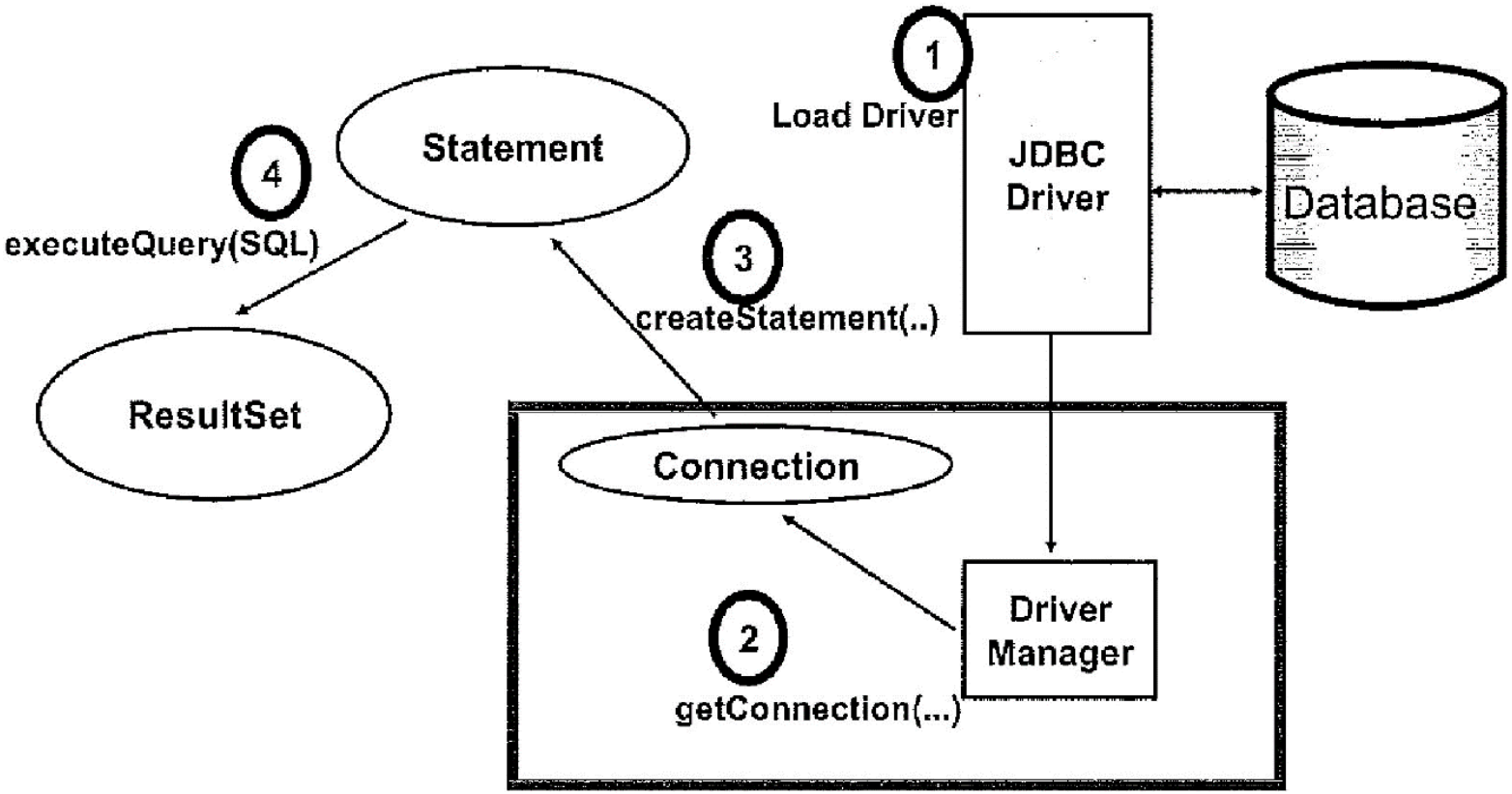
Drivers before JDBC 4.0 must be loaded manually by using:

```
try {
    java.lang.Class.forName("<fully qualified path of the driver>");
} catch (ClassNotfoundException c) {
}
```

Driver classes can also be passed to the interpreter on the command line:

```
java -djdbc.drivers=<fully qualified path to the driver> <class to run>
```

Simple JDBC Client (2 of 4)



Creating a Connection

- Connection request made on **DriverManager**

```
Connection conn = DriverManager.
                  getConnection(url);
Connection conn = DriverManager.
                  getConnection(url, properties);
Connection conn = DriverManager.
                  getConnection(url,login,password);
```

- You should ALWAYS close the connection when finished

```
conn.close();
```

URL for the Connection

- JDBC specifies individual databases with a URL format
  - `jdbc:<subprotocol>:<subname>`
    - `subprotocol:` vendor connection name (db2, oracle, odbc, etc.)
    - `subname:` driver specific. In DB2 type 2 driver it is the database name, for example, SAMPLE, WOMBANK, and so forth
  - Local example jdbc:db2:sample
  - Remote example jdbc:db2://mydb.com:8888/sample

Using a Vendor's Driver Class

The `DriverManager` class is used to get an instance of a Connection object by using the JDBC driver named in the JDBC Uniform Resource Identifier (URI).

```
String uri = "jdbc:derby://localhost:1527/EmployeeDB";
Connection con = DriverManager.getConnection (url);
```

- The URI syntax for a JDBC driver is:

```
jdbc:<driver>:[subsubprotocol:][databaseName][;attribute=value]
```

- Each vendor can implement its own subprotocol.
- The URI syntax for an Oracle Thin driver is:

```
jdbc:oracle:thin:@//[HOST][:PORT]/SERVICE
```

Example:

```
jdbc:oracle:thin:@//myhost:1521/orcl
```

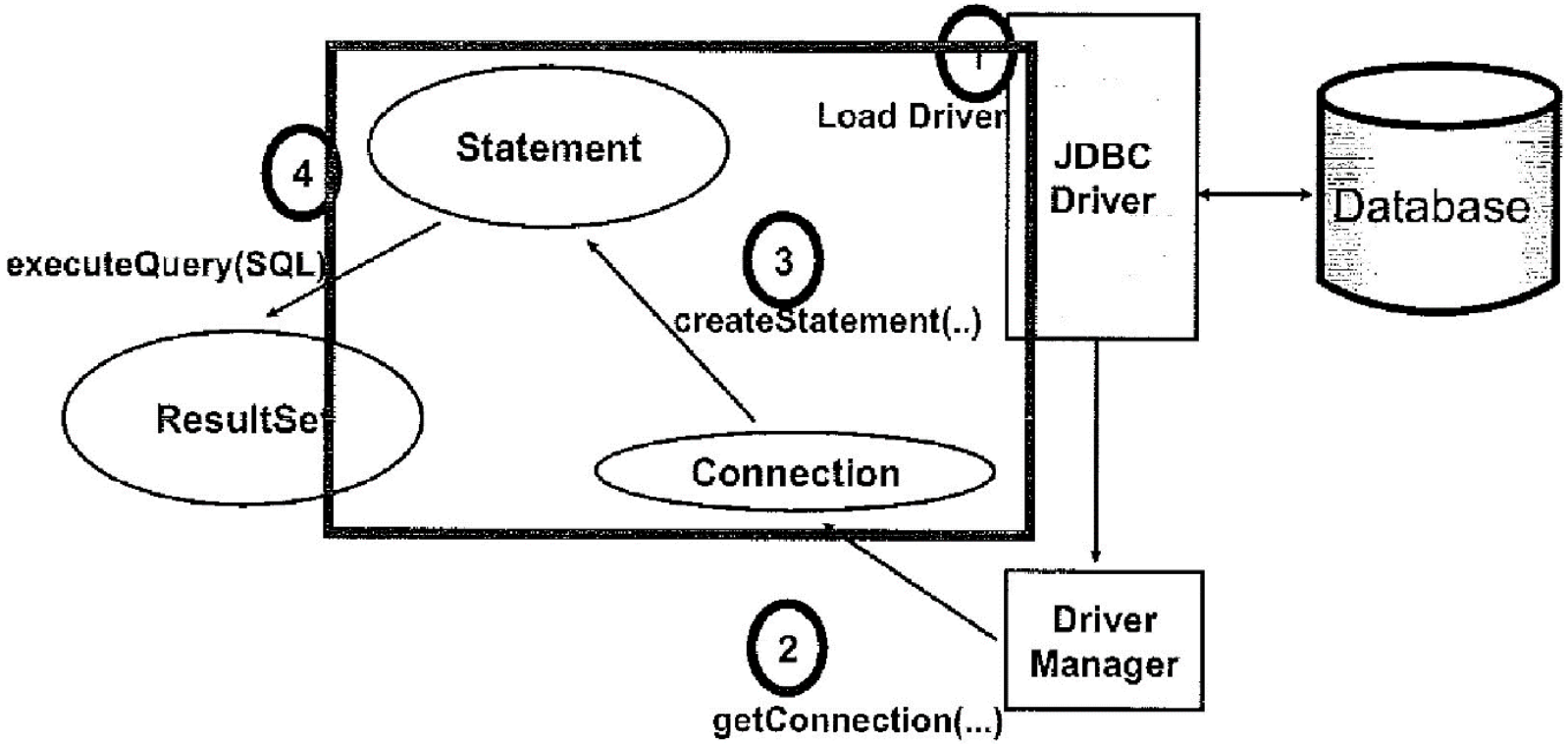
Connection Example

```
// Load a driver
Class.forName(
   "COM.ibm.db2.jdbc.app.DB2Driver");

// create a Connection for a local DB2 db
Connection current =
   DriverManager.getConnection(
       "jdbc:db2:SAMPLE", "user1", "wombat");

// Connections auto-commit by default
...
current.close();
```

Simple JDBC Client (3 of 4)



Methods in the Connection Class

- Create a `Statement`

```
-Statement createStatement()
      throws SQLException
```

- Create a `PreparedStatement/CallableStatement`

```
-PreparedStatement prepareStatement(
      String sql) throws SQLException
-CallableStatement prepareCall(String sql)
     throws SQLException
```

- Commit or roll back a transaction

```
-void commit() throws SQLException
-void rollback() throws SQLException
```

- Change auto-commit mode

```
-void setAutoCommit(boolean autoCommit)
      throws SQLException
```

- Set transaction level

```
-void setTransactionIsolation(Connection.
                  TRANSACTION_READ_COMMITTED)
```

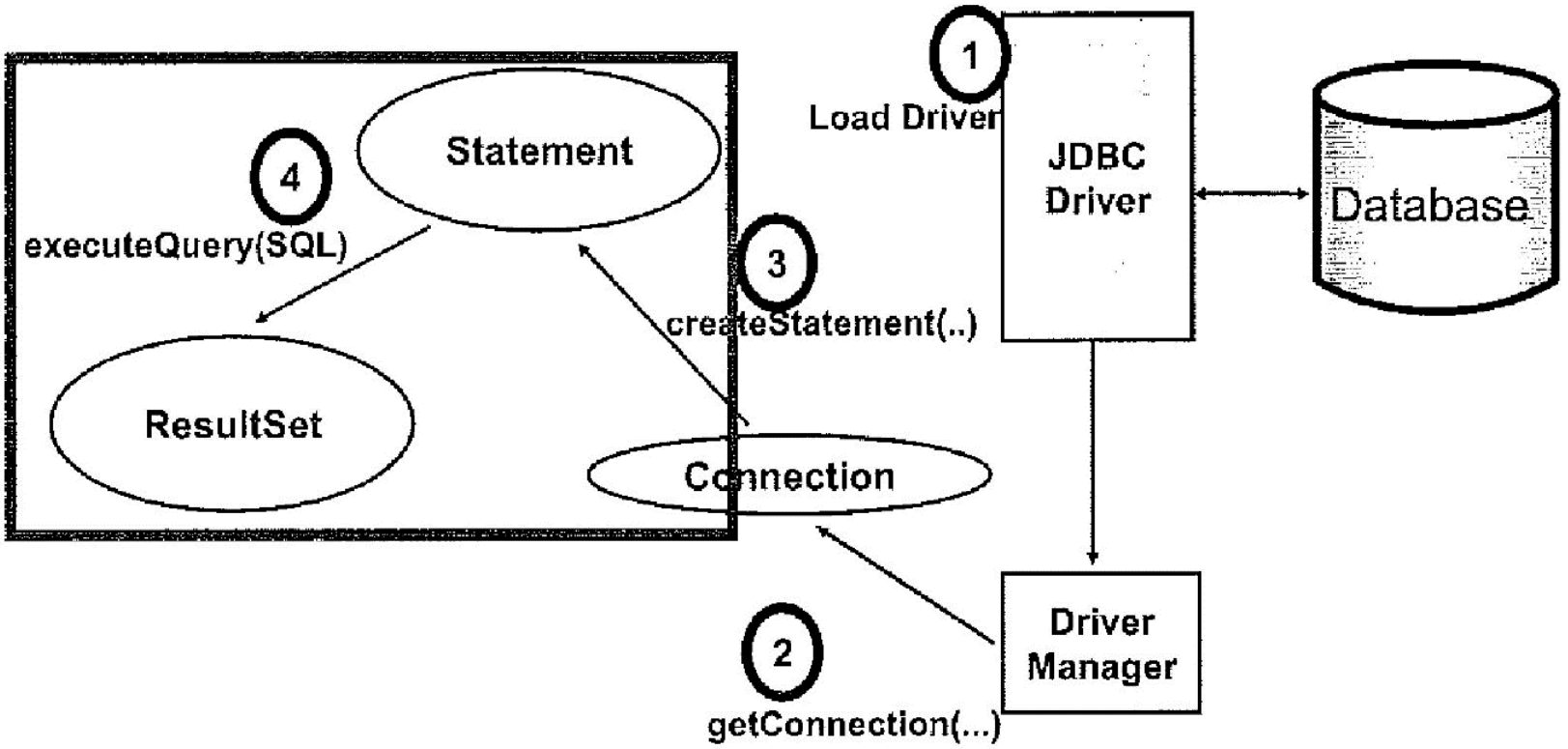
Statement Example

```
// Load a driver and create a connection
Class.forName ("COM.ibm.db2.jdbc.app.DB2Driver");
Connection current = DriverManager.getConnection
          ("jdbc:db2:SAMPLE", "user1", "wombat");
current.setAutoCommit(false);

// Create a statement
Statement query = current.createStatement();

// Close resources
current.commit();
current.close();
```

Simple JDBC Client (4 of 4)



Methods in the Statement Class

- A `Statement` executes an SQL statement and obtains the results produced by it
- Execute an SQL `SELECT` statement.

```
-ResultSet executeQuery(
   String sql) throws SQLException
```

- Execute an SQL `DELETE`, `UPDATE`, `INSERT` or `DDL`.

```
-int executeUpdate(String sql)
   throws SQLException
```

- Execute an SQL statement of unknown type

```
-boolean execute(String sql)
   throws SQLException
```

Query Example

```
// Load a driver and create a connection
Class.forName ("COM.ibm.db2.jdbc.app.DB2Driver");
Connection current = DriverManager.getConnection
          ("jdbc:db2:SAMPLE", "user1", "wombat");
current.setAutoCommit(false);

// Create a statement and execute a query
Statement query = current.createStatement();
ResultSet result =
    query.executeQuery("SELECT * FROM Employee");

// Close resources
current.commit();
query.close();
current.close();
```

ResultSet Processing

- A `ResultSet` provides access to a table of data generated by executing a `Statement`
  - Maintains a cursor pointing to its current row of data
  - Initially the cursor is positioned before the first row
- The `next()` method moves the cursor to the next row
  - Returns **false** if there are no more rows
  - The table rows are retrieved in sequence

Processing a ResultSet

- `ResultSet` has methods to retrieve column values for the current row
  - Within a row, its column values can be accessed in any order
- The methods all follow the following templates:
  - `<JavaType> get<JavaType>(int)` - by column number
  - `<JavaType> get<JavaType>(String)` - by column name
- The access methods perform the necessary type conversion between SQL types and Java types.
- **getString()** can be performed on any column.

```
String fname = resultSet.getString("firstnme");
int id = resultSet.getInt(1);
```

Mapping SQL Types

SQL Type	ResultSet method	Java Type
CHAR	getString	String
VARCHAR	getString	String
LONGVARCHAR	getString,	String
	getAsciiStream,	InputStream
	getCharacterStream	Reader
NUMERIC	getBigDecimal	java.math.BigDecimal
DECIMAL	getBigDecimal	java.math.BigDecimal
BIT	getBoolean	boolean
TINYINT	getByte	byte
SMALLINT	getShort	short
INTEGER	getInt	int
BIGINT	getLong	long
REAL	getFloat	float
FLOAT	getDouble	double
DOUBLE	getDouble	double
DATE	getDate	java.sql.Date
TIME	getTime	java.sql.Time
TIMESTAMP	getTimestamp	java.sql.Timestamp
BLOB	getBlob	java.sql.Blob
CLOB	getClob	java.sql.Clob

ResultSet Example

```
// Need the previous driver load & connection
Statement query = current.createStatement();
ResultSet result = query.executeQuery
 ("SELECT FIRSTNME,BIRTHDATE FROM Employee");

// iterate through result and print out
while (result.next()) {
 System.out.println(result.getString("firstnme"));
 System.out.println(result.getDate("birthdate"));
}
result.close();
```

Scrollable ResultSet

- Added JDBC 2.0
- Scroll forward or backwards through a ResultSet
- Developer can specify number of rows fetched when more rows are needed (suggestion to DBM only)
- Read-only and updatable results sets are allowed

Options for Create Connection Method

- Connection class createStatement and prepareStatement methods support scrollable ResultSets:

```
Statement createStatement
          (resultSetType, resultSetConcurrency);
PreparedStatement prepareStatement(sql,
           resultSetType, resultSetConcurrency);
```

- `resultSetType` controls scrollability

- `resultSetConcurrency` controls whether rows in the ResultSet can be updated, inserted, or deleted

Methods for Scrollable ResultSet

- Scroll

```
result.first();
result.last();
result.previous();
```

- Set the number of rows fetched

```
result.setFetchSize(int);
```

- Set the fetch direction

```
result.setFetchDirection(direction);
result.afterLast();
result.beforeFirst();
```

- Where `direction` is:

```
ResultSet.FETCH_FORWARD
ResultSet.FETCH_REVERSE
ResultSet.FETCH_UNKNOWN
```

ResultSet Type

ResultSet Type	Meaning
`ResultSet.TYPE_FORWARD_ONLY`	Same as old JDBC 1.0 ResultSet processing
`ResultSet.TYPE_SCROLL_INSENSITIVE`	Static view of the fetched data
`ResultSet.TYPE_SCROLL_SENSITIVE`	Dynamic view of the fetched data. Changes in underlying rows are visible to the application

ResultSet Concurrency

ResultSet Concurrency
ResultSet.CONCUR_READ_ONLY
ResultSet.CONCUR_UPDATABLE

ResultSet Example

```
Statement query = conn.createStatement
  (ResultSet.TYPE_SCROLL_INSENSITIVE,
   ResultSet.CONCUR_READ_ONLY);
ResultSet result = query.executeQuery
      ("SELECT ID, NAME, DEPT FROM STAFF");
result.afterLast();
while (result.previous()) {
   System.out.println(result.getInt("id"));
   System.out.println(result.getString("name"));
   System.out.println(result.getInt(3));
}
result.close();
conn.close();
```

ResultSet Objects

- `ResultSet` maintains a cursor to the returned rows. The cursor is initially pointing before the first row.
- The `ResultSet.next()` method is called to position the cursor in the next row.
- The default `ResultSet` is not updatable and has a cursor that points only forward.
- It is possible to produce `ResultSet` objects that are scrollable and/or updatable. The following code fragment, in which `con` is a valid `Connection` object, illustrates how to make a result set that is scrollable and insensitive to updates by others, and that is updatable:

```
Statement stmt
        = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                              ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT a, b FROM TABLE2");
```

Note: Not all databases support scrollable result sets.

`ResultSet` has accessor methods to read the contents of each column returned in a row. `ResultSet` has a `getter` method for each type.

Exception Handling

- Most of the methods in `Statement` and `Connection` can throw `SQLException`
  - `int getErrorCode()` returns the vendor-specific SQL error code
  - `String getSQLState()` returns the SQLState
  - `SQLException getNextException()` returns a nested SQLException
  - SQLException is a checked exception
  - Left out of the examples for space and simplicity
- Can be handled in one of two ways:
  - Catch and recover locally (possibly by re-acquiring the connection)
  - Catch and convert the exception by throwing another exception

JDBC Efficiency

- Database access is the performance bottleneck in most Java applications that use a database
- Using JDBC efficiently depends on a number of factors:
  - Use the right driver
  - Use JDBC's `PreparedStatement` and `CallableStatement` when appropriate

PreparedStatements

- An SQL statement can be precompiled and stored in a `PreparedStatement` object
  - Variable parameter data specified as "?"

```
PreparedStatement insert =
 currentConnection.prepareStatement("INSERT INTO
 AddressBook (NAME, PHONENO, EMAIL)
VALUES (?, ?, ?)");
```

- This object can then be used to efficiently execute this statement many times
- You simply set the variable parameters and reexecute the statement

```
insert.setString(1, "Eric");
insert.setString(2, "555-1212");
insert.setString(3, "monkeyboy@redondo.com");
insert.executeUpdate();
```

Methods on the PreparedStatement Class

- Setting parameter values
  - One `set<Type>` method per Java datatype, for example, `setInt(int,int)`, `setDate(int,Date)`
- `ResultSet executeQuery()`
  - Used when substituting parameters in a `WHERE` clause
- `int executeUpdate()`
  - Used when substituting data parameters in an `INSERT`, `UPDATE`, or `DELETE`
- PreparedStatements are also very useful for passing long text or non-character data

PreparedStatement Example

```
// Driver load and connect as previous
// Create a preparedStatement
PreparedStatement insert =
 currentConnection.prepareStatement("INSERT INTO
 Employee (firstnme, midinit, lastname, edlevel)
 VALUES (?, ?, ?, ?)");

// This code would be executed many times
insert.setString(1, "John");
insert.setString(2, "Q");
insert.setString(3, "Public");
insert.setInt(4, 12);
insert.executeUpdate();
```

CallableStatement

- `CallableStatement` allows calls to SQL stored procedures
  - Can be faster than `PreparedStatements`
  - Stored Procedure resides with database
  - Not portable to other databases
- IN parameter values are set using the set methods inherited from `PreparedStatement`
- OUT parameter types must be registered prior to executing the stored procedure
  - Values are retrieved by `get<JavaType>(int)` methods
  - A result set can also be returned

CallableStatement Example (1 of 2)

```
CallableStatement cstmt = con.prepareCall(
                "{call getTestData(?, ?)}");
cstmt.setByte(1,25);
cstmt.setBigDecimal(2, 83.75);
// register parameters
cstmt.registerOutParameter(1,
                    java.sql.Types.TINYINT);
cstmt.registerOutParameter(2,
                    java.sql.Types.NUMERIC);
ResultSet rs = cstmt.executeQuery();
```

CallableStatement Example (2 of 2)

```
// retrieve and print values
while (rs.next()) {
    String name = rs.getString(1);
    int score = rs.getInt(2);
    int percentile = rs.getInt(3);
    System.out.print("name = " + name +
                                    ", score =" + score);
    System.out.println(",percentile = " + percentile);
}
// retrieve output parameters
byte x = cstmt.getByte(1);
java.math.BigDecimal n = cstmt.getBigDecimal(2);
```

Batch Processing

- JDBC 2.0 provides support for batch processing of SQL
  - multiple update statements executed in a single database call
- Available for Statement, PreparedStatement, CallableStatement
- Additional methods of:
  - `addBatch()` for each statement in the batch
  - `executeBatch()` to send the batch to the database
- Allowed for update activity (INSERT, UPDATE, DELETE) only

Batch Statement Example

```
connection.setAutoCommit(false);
Statement insert = connection.createStatement();
insert.addBatch
   ("INSERT INTO EMPL VALUES (10, 'M. Vogel')");
insert.addBatch
   ("INSERT INTO DEPT VALUES (42, 'OO A/D')");
insert.addBatch
   ("INSERT INTO EMPL_DEPT VALUES (10, 42)");
// submit a batch of update commands for execution
int[] counts = insert.executeBatch();
// …
connection.commit();
```

Batch PreparedStatement Example

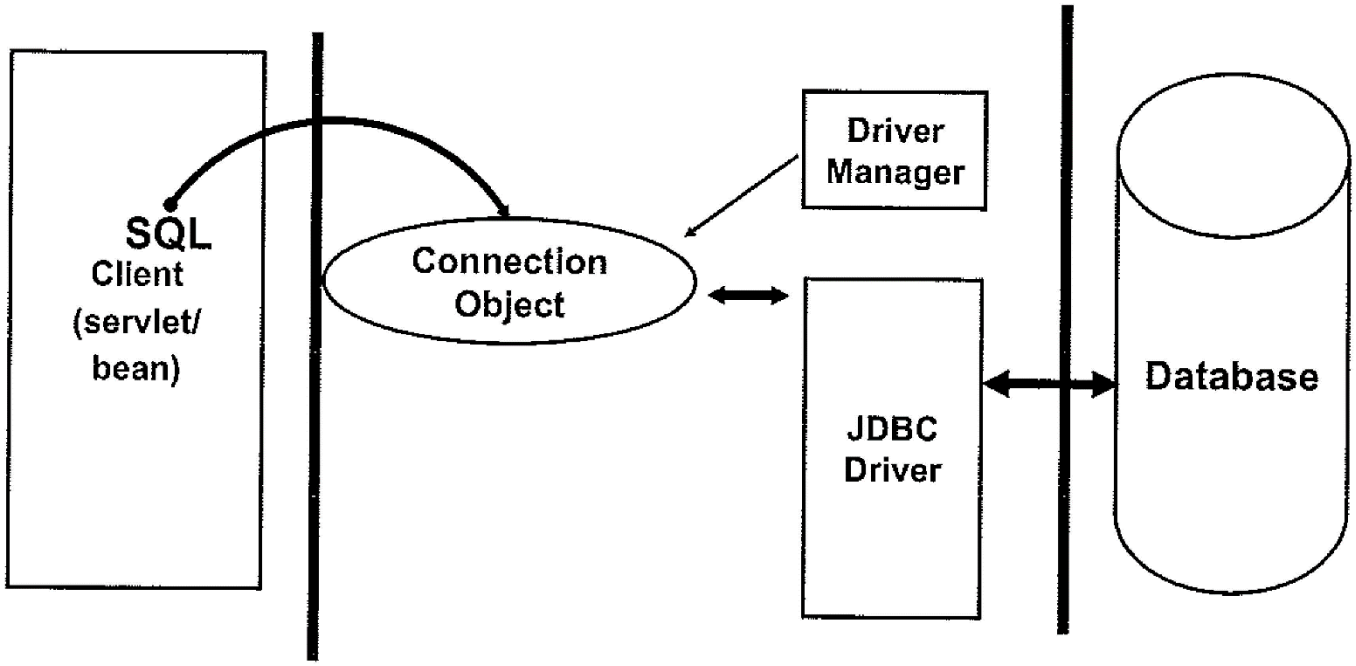
```
connection.setAutoCommit(false);
PreparedStatement insert=connection.prepareStatement
          ("INSERT INTO EMPL VALUES (?, ?)");
insert.setInt(1, 10);
insert.setString(2, "K. Brown");
insert.addBatch();
insert.setInt(1, 20);
insert.setString(2, "M. Bell");
insert.addBatch();
// submit a batch of update commands for execution
int[] counts = insert.executeBatch();
// …
connection.commit();
```

JDBC: DataSources

Szczecin, 17 marca 2018 r.

JDBC

Database Connection via JDBC Calls



- Load the right JDBC Driver
- Get a Connection object
- Execute SQL (Queries / Updates) on the Connection object
- Close the Connection

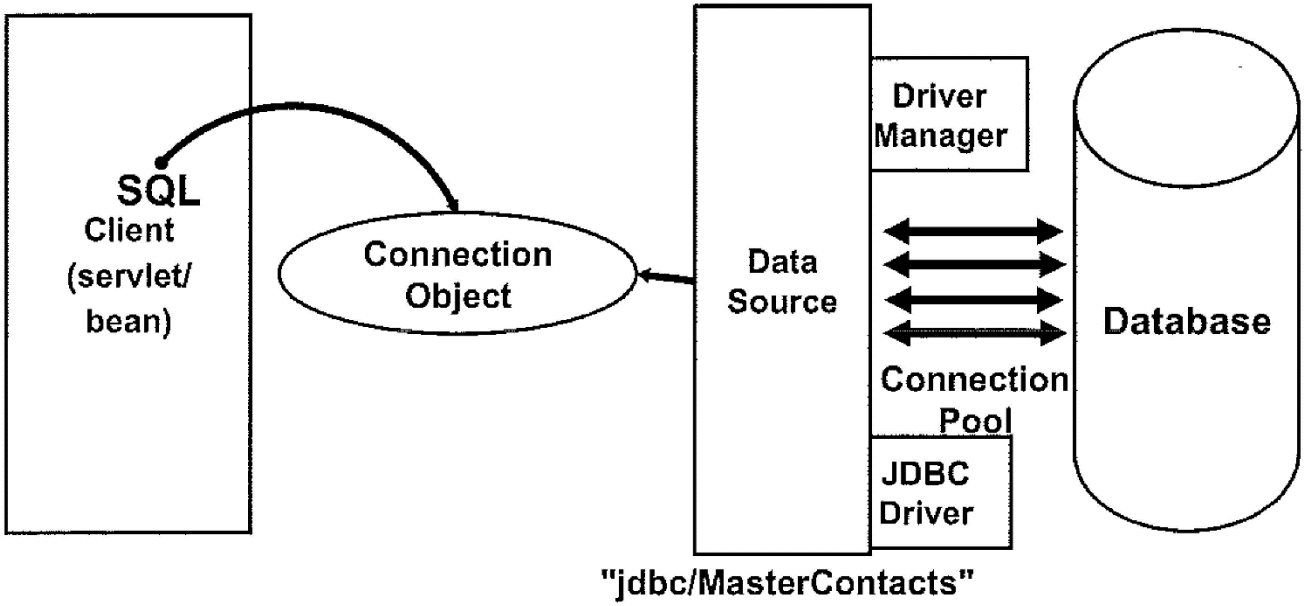
Connection Pooling

- Web-driven or client/server-driven database interactions are typically of a very short duration
- For performance and scalability, you want to:
  - Minimize DB connection setup and tear-down
  - Limit the maximum number of "active" sessions
- This is the goal of *connection pooling*
  - A Connection Pool is a cache of pre-built database connections maintained in memory, which allows the connections to be reused

Connection Pooling in JDBC

- JDBC supports Connection Pooling by using DataSources
- A DataSource:
  - Maintains a number of persistent connections to the database
  - Minimizes connection overhead
  - Spreads connection cost out over repeated uses
  - Is an Interface
  - Enhances code portability and maintenance

Database Connection Using a DataSource



- Look up the DataSource using JNDI
- Get a Connection object from the DataSource
- Execute SQL (Queries / Updates) on the Connection object
- Close the Connection (return to the pool)

Coding for DataSources

- Create the initial naming context
  - Hashtable parms = new Hashtable();
    parms.put(Context.INITIAL_CONTEXT_FACTORY,
    "COM.ibm.db2.jndi.DB2InitialContextFactory");
    Context ctx = new InitialContext(parms);
- Get a DataSource object
  - DataSource ds = (DataSource)
    ctx.lookup("jdbc/MasterContacts");
- Get a connection
  - Connection conn = ds.getConnection();
- Perform normal SQL processing
  - Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(
    "SELECT * FROM Employee");
- Close the connection
  - if (conn != null) conn.close();

Setting Up a DataSource

- In most cases, a Web application server and JDBC driver will create and manage the connection pool. Several administrative properties are set:
  - -DataSource name
  - -Actual driver and database name
  - -Minimum and maximum size of the pool
  - -Various timeouts
- Several classes/interfaces are used to support pooling; these are not visible to the client:
  - -ConnectionPoolDataSource
  - -PooledConnection
  - -ConnectionEventListener
  - -ConnectionEvent

KONIEC

DZIĘKUJĘ ZA UWAGĘ!!!

Szczecin, 17 marca 2018 r.