JDBC

Jeszenszky, Péter
University of Debrecen
jeszenszky.peter@inf.unideb.hu

Kocsis, Gergely
University of Debrecen
kocsis.gergely@inf.unideb.hu

Last modified: 20.04.2016.

JDBC

- A Java API to reach relational data
  - It allows the execution of SQL queries, the process of the resulted data and the modification of the data source
  - It provides mapping between Java language and SQL
- A part of Java SE and also Java EE
  - Contained by `java.sql` and `javax.sql` packages

Planning goals

- Fit into Java SE and Java EE platforms
- Allow the use of possibilities used by wide spread implementations in a provider independent way
- Serve as a basis of higher level tools and API-s
- Simplicity

Specification

- *JSR 221: JDBC 4.2 API Specification* (March 2014) https://jcp.org/en/jsr/detail?id=221
  - Supported SQL standard: SQL:2003

History

- Introduced in JDK 1.1 in 1997
- Current version is 4.2 that is contained by Java SE 8
  - New features:
    - New interfaces and classes: `java.sql.SQLType`, `java.sql.DriverAction`, `java.sql.JDBCType`
    - Support of the `java.time` package
    - ...

JDBC Data Type Conversion

- See more in the specification Appendix B - *Data Type Conversion Tables*

JDBC Type	**Java Type**
CHAR	String
VARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BOOLEAN	boolean
INTEGER	int
REAL	float
FLOAT	double
DOUBLE	double
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

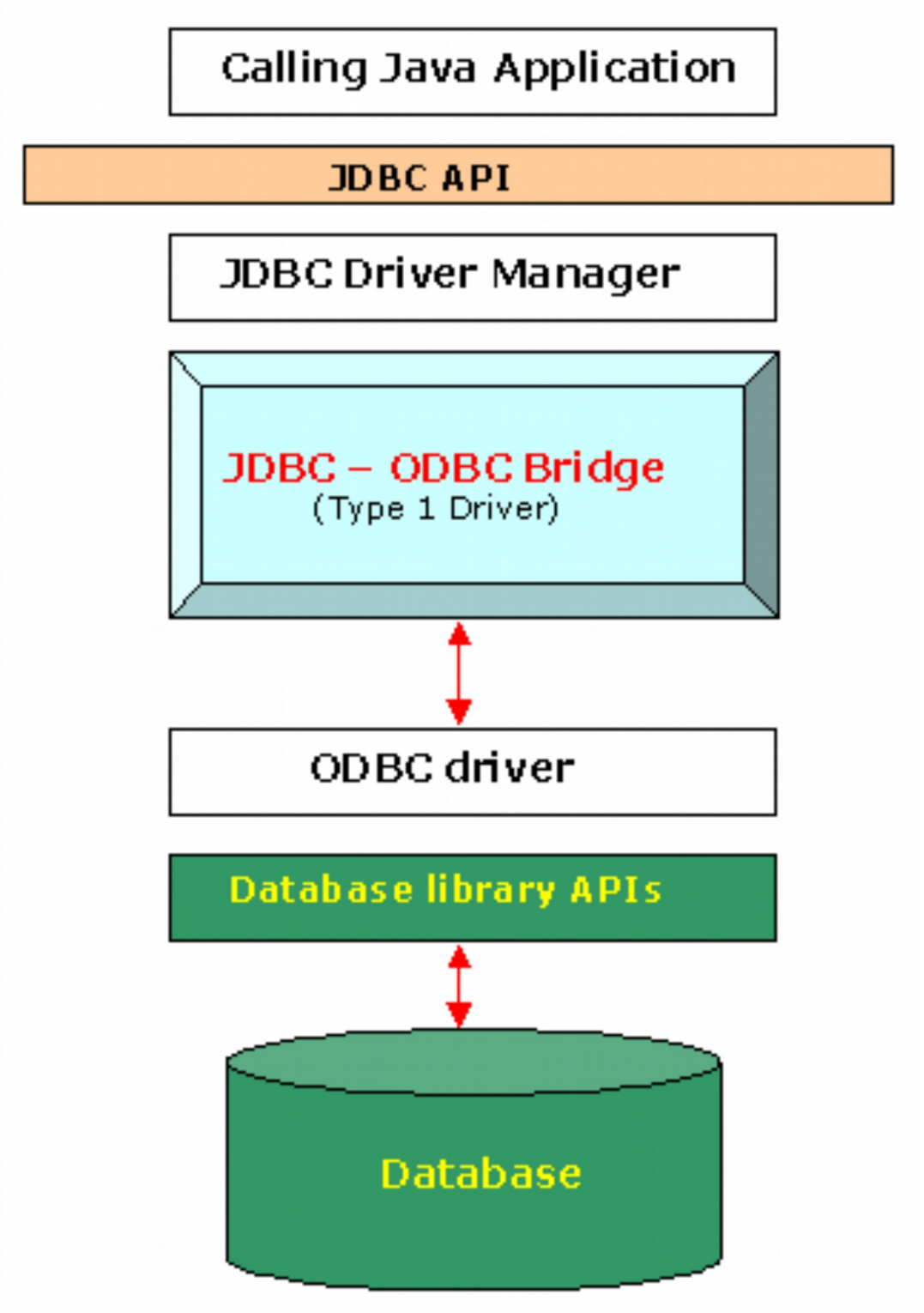
JDBC driver

- The software component that allows the Java application to establish contact to a database
  - An implementation of the JDBC API for a given data source
- The JDBC specification defines 4 different types of drivers

Type 1
JDBC-ODBC bridge

- Properties
  - JDBC requests turned into ODBC requests, which will be handled by the OBDC-driver
  - Client → JDBC driver → ODBC driver → database
- Advantages
  - Almost every database with an ODBC-driver can be reached
  - Ease of installation
- Disadvantages
  - Performance loss
  - The ODBC-driver has to be installed on the client
  - Inappropriate for applets/internet applications

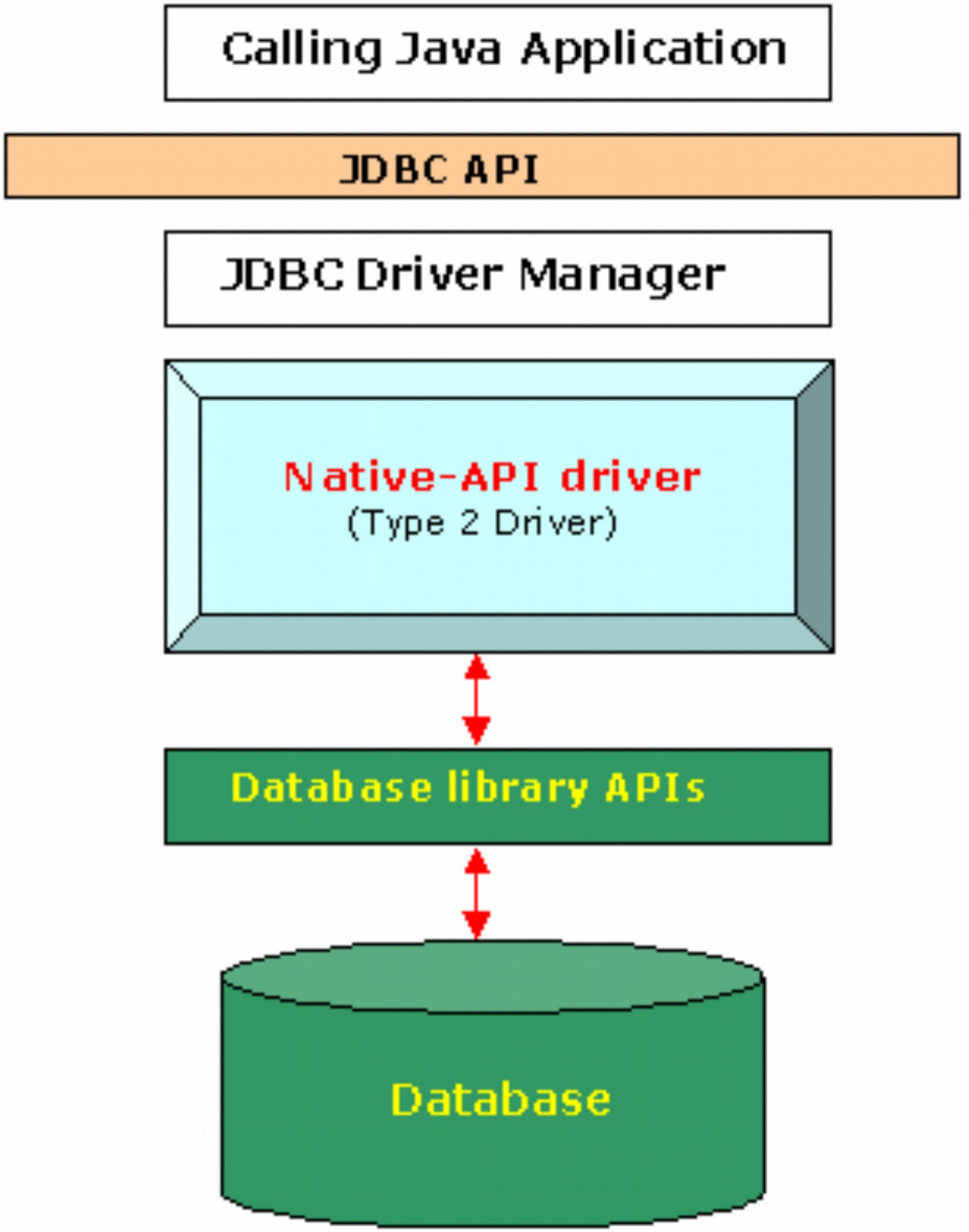
Type 1


Calling Java Application
JDBC API
JDBC Driver Manager
JDBC – ODBC Bridge
(Type 1 Driver)
ODBC driver
Database library APIs
Database

Type 2
Native API driver

- Properties
  - Client calls the API of the database
  - Client → JDBC driver → client side library of vendor → database
- Advantages
  - Better performance (no JDBC-ODBC translation)
- Disadvantages
  - Vendor's library has to be installed on client
  - Thus inappropriate for web apps
  - Not all databases have client side libraries

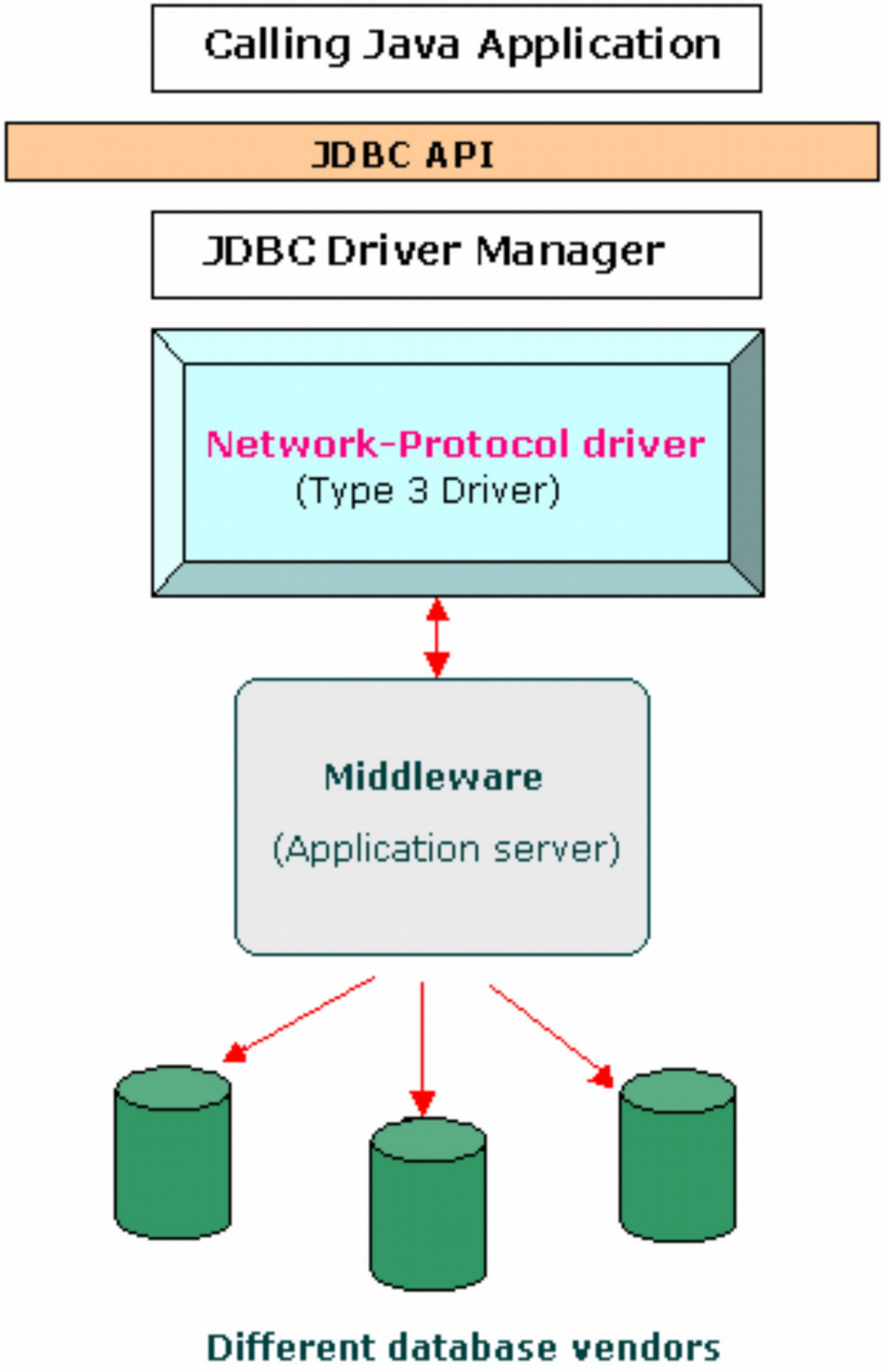
Type 2


Calling Java Application
JDBC API
JDBC Driver Manager
Native-API driver
(Type 2 Driver)
Database library APIs
Database

Type 3
Network Protocol Driver

- Properties
  - Written only in Java
  - Can be used for multiple databases (vendor independent)
  - Communication between the client and the middle layer is database independent
  - The middle layer translates to the language of the database
  - Client → JDBC driver → intermediate layer server → database
- Advantages
  - No need for library from vendor on client side
  - Changing between databases does not affect the connection between the client and the middle layer
  - The middle layer can help work with typical services such as caching, load balancing, logging, auditing, etc.
- Disadvantages
  - Database specific coding required on the middle layer
  - The new layer can prove to be a bottleneck regarding time

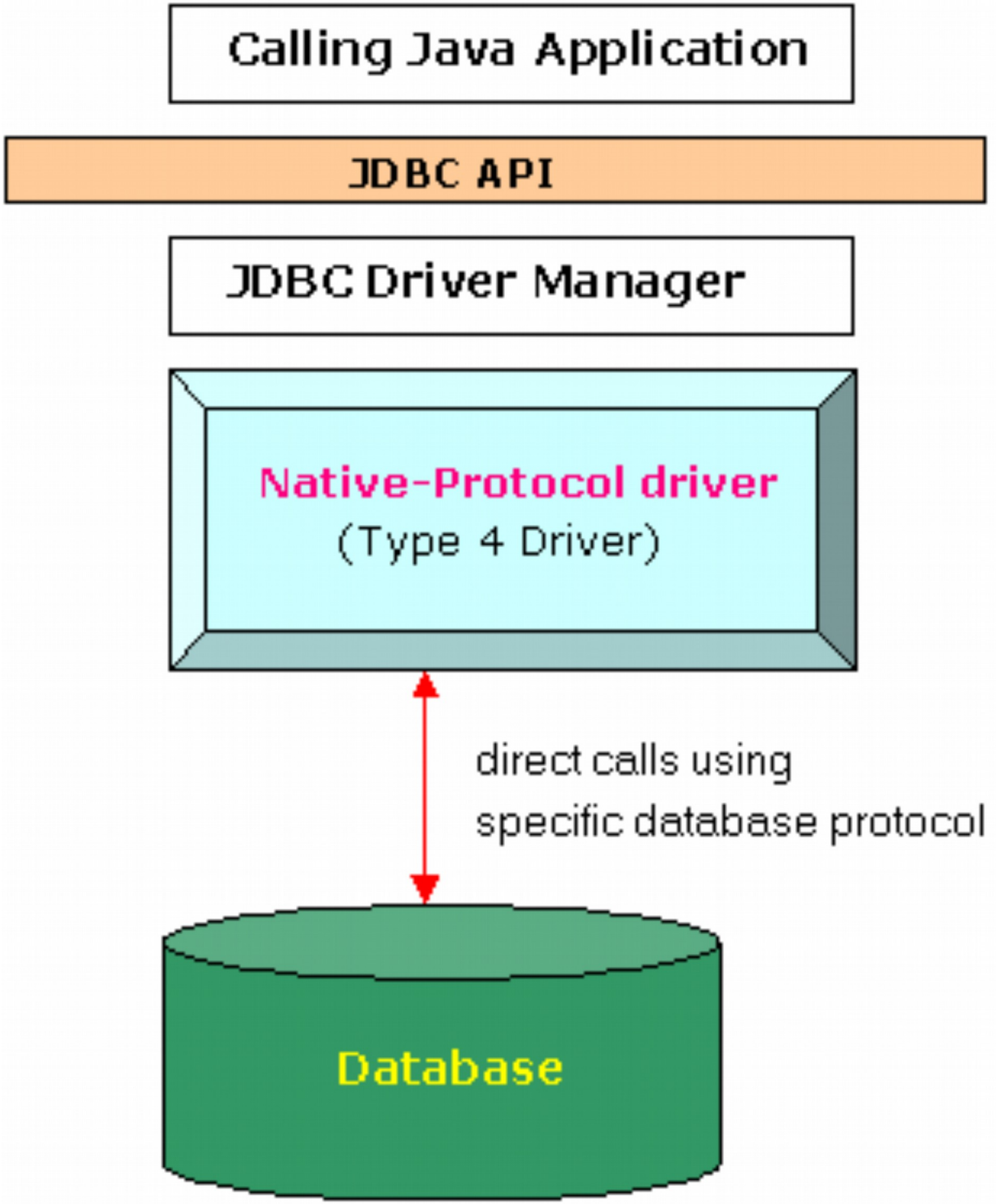
Type 3


Calling Java Application
JDBC API
JDBC Driver Manager
Network-Protocol driver
(Type 3 Driver)
Middleware
(Application server)
Different database vendors

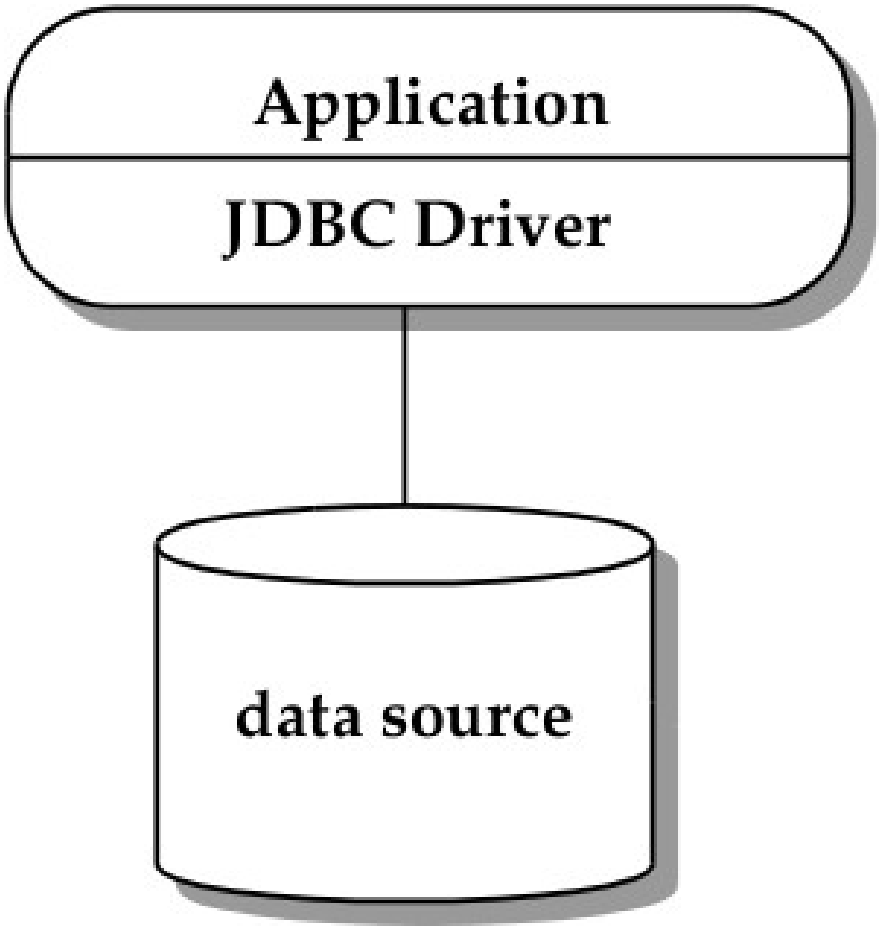
Type 4
Native Protocol Driver

- Properties
  - Written clearly in Java, direct communication with the databsae (socket)
  - The driver forms the JDBC calls to be appropriate for the vendor specific database protocol
  - Client → native protocol JDBC driver → database
- Advantages
  - No intermediate format, no intermediate layer – better performance
  - All aspects of the connection between the application and the database are handled in the JVM – easier debugging
- Disadvantages
  - All databases need different drivers on the client side

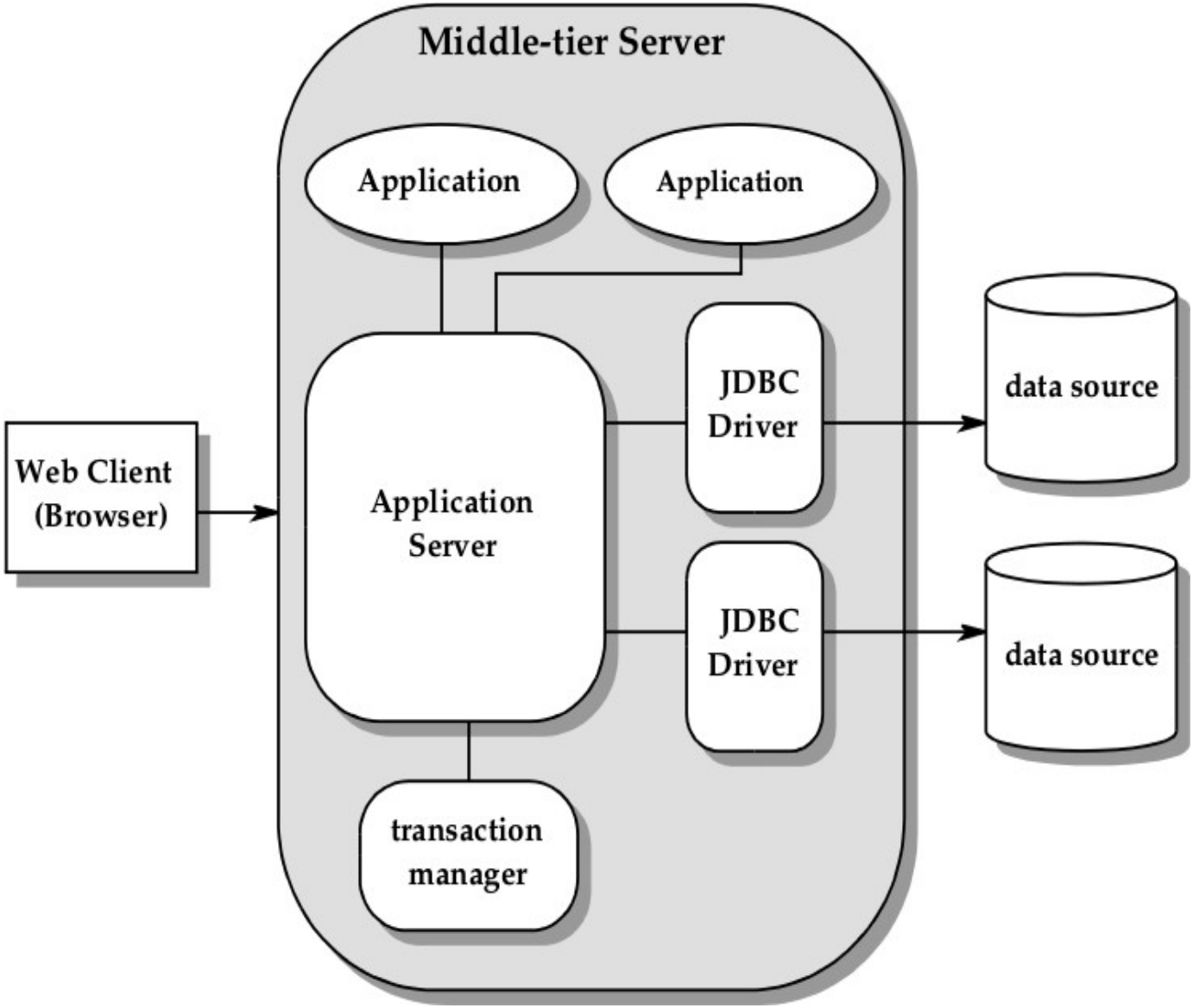
Type 4


Calling Java Application
JDBC API
JDBC Driver Manager
Native-Protocol driver
(Type 4 Driver)
direct calls using
specific database protocol
Database

2 layer model



3 layer model



The java.sql.Driver interface

- An interface representing the JDBC driver
- The implementation has to include a static initialization block that registers an instance of the class with the `java.sql.DriverManager`
  - For this the presence of the default constructor is mandatory

```
public class MyJDBCDriver implements java.sql.Driver {

    static {
        java.sql.DriverManager.registerDriver(new
                MyJDBCDriver());
    }

    // ...
}
```

Creating the database connection

- The `java.sql.Connection` interface represents a connection to the data source
  - An application can handle more than one connections for the same or for different data sources in the same time.
- A JDBC application can connect in two different ways to the data source:
  - By the use of the `java.sql.DriverManager` class
  - By the use of the `javax.sql.DataSource` interface (recommended)

Processing SQL statements

- The call of the methods of the `java.sql.Connection` interface is needed

The java.sql.DriverManager class (1)

- Handles the JDBC drivers available for clients
- The client uses the `getConnection()` method of the `DriverManager` class when it creates the connection. For the method an URL is passed that specifies the data source.
- The `DriverManager` looks for the proper JDBC driver from the available ones that creates the connection to the data source.
  - The class tries all the registered JDBC drivers and calls the connect methods of them passing the URL as a parameter.

The java.sql.DriverManager class (2)

- Form of the database URL:
  - `jdbc:sub`*`protocll`*`:`*`subname`*
  - Examples:
    - `jdbc:hsqldb:mem:testdb`
    - `jdbc:hsqldb:file:testdb`
    - `jdbc:hsqldb:file:/var/db/testdb`
    - `jdbc:hsqldb:hsql://localhost/testdb`
    - `jdbc:hsqldb:http://localhost/testdb`
    - `jdbc:oracle:thin:@db.inf.unideb.hu:1521:ora11g`

The java.sql.DriverManager class (3)

- During its initialization the class tries to load the drivers given as the values of the `jdbc.drivers` system property
- Before JDBC 4.0 the loading of the drivers was possible only in the following ways:

```
Class.forName("pkg.MyJDBCDriver");

    or

DriverManager.registerDriver(new pkg.MyJDBCDriver());
```

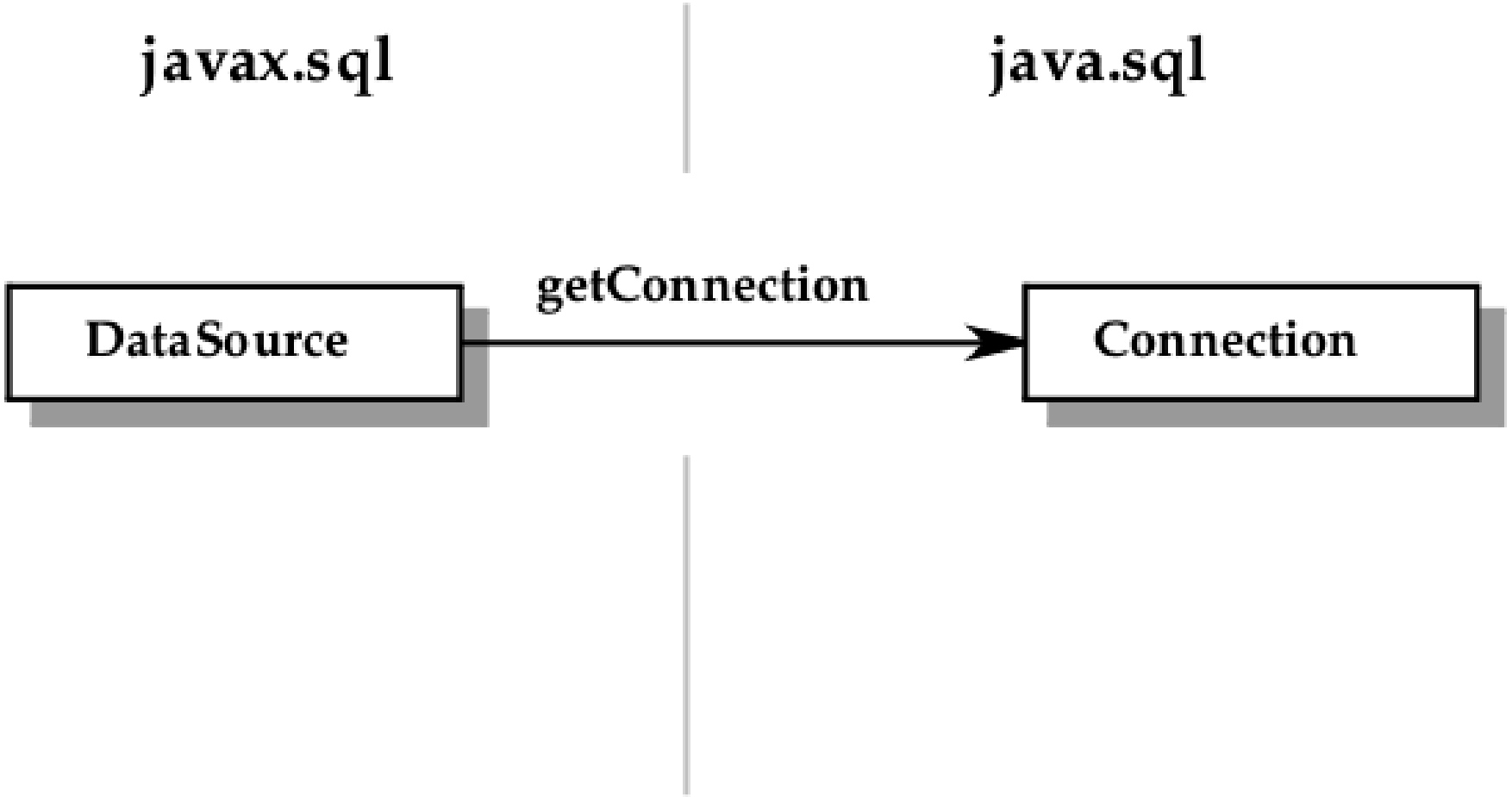
The java.sql.DriverManager class (4)

- After JDBC 4.0 the explicit loading of th driver is not needed. It is enough to add the JAR file to the *classpath*
  - The JAR file has to contain the fully qualified name of a class implementing the `java.sql.Driver` interface in the `META-INF/services/java.sql.Driver` file
    - See: *Service Provider* mechanism
      http://docs.oracle.com/javase/6/docs/technotes/guides/jar/jar.html#Service%20Provider
      http://docs.oracle.com/javase/6/docs/api/java/util/ServiceLoader.html

The javax.sql.DataSource interface (1)

- A recommended alternative of `java.sql.DriverManager` class when creating connections
- It increases portability of applications by using a logical name for data sources.
  - An object with a given logical name has to be registered and can be get from a JNDI name provider.
- It represents a physical data source which for it makes possible to connect.

The javax.sql.DataSource interface (2)



The javax.sql.DataSource interface (3)

- JDBC defines the following properties for the implementations of the `DataSource` (only `description` is mandatory):
  - The implementations provide getters and setters for the supported attributes

Attribute	Type	Description
`databaseName`	`String`	Database name
`dataSourceName`	`String`	Datasource name
`description`	`String`	Datasource description
`networkProtocol`	`String`	Network protocol for the communication with the server
`password`	`String`	password
`portNumber`	`int`	Port number
`roleName`	`String`	SQL role name
`serverName`	`String`	Database server name
`user`	`String`	username

The javax.sql.DataSource interface (4)

- Example implementations:
  - `org.postgresql.ds.PGSimpleDataSource`
    https://jdbc.postgresql.org/documentation/publicapi/org/postgresql/ds/PGSimpleDataSource.html
  - `org.apache.derby.jdbc.ClientDataSource`
    `https://db.apache.org/derby/docs/10.12/publishedapi/org/apache/derby/jdbc/ClientDataSource.html`
  - `org.hsqldb.jdbc.JDBCCommonDataSource`
    http://hsqldb.org/doc/src/org/hsqldb/jdbc/JDBCCommonDataSource.html

DataSource usage (1)

- *Java Naming and Directory Interface* (JNDI)
  http://www.oracle.com/technetwork/java/jndi/
  - *JNDI Interface-related APIs and Developer Guides*
    http://docs.oracle.com/javase/8/docs/technotes/guides/jndi/
  - *The Java Tutorials: Trail: Java Naming and Directory Interface*
    https://docs.oracle.com/javase/tutorial/jndi/
  - `javax.naming` csomag
    https://docs.oracle.com/javase/8/docs/api/javax/naming/package-summary.html

DataSource usage (2)

- Register a `DataSource` object at a JNDI name provider:

```
VendorDataSource vds = new VendorDataSource();
vds.setServerName("localhost");
vds.setDatabaseName("testDB");
vds.setDescription("Data source for testing");

Context ctx = new InitialContext();
ctx.bind("jdbc/testDB", vds);
```

DataSource usage (3)

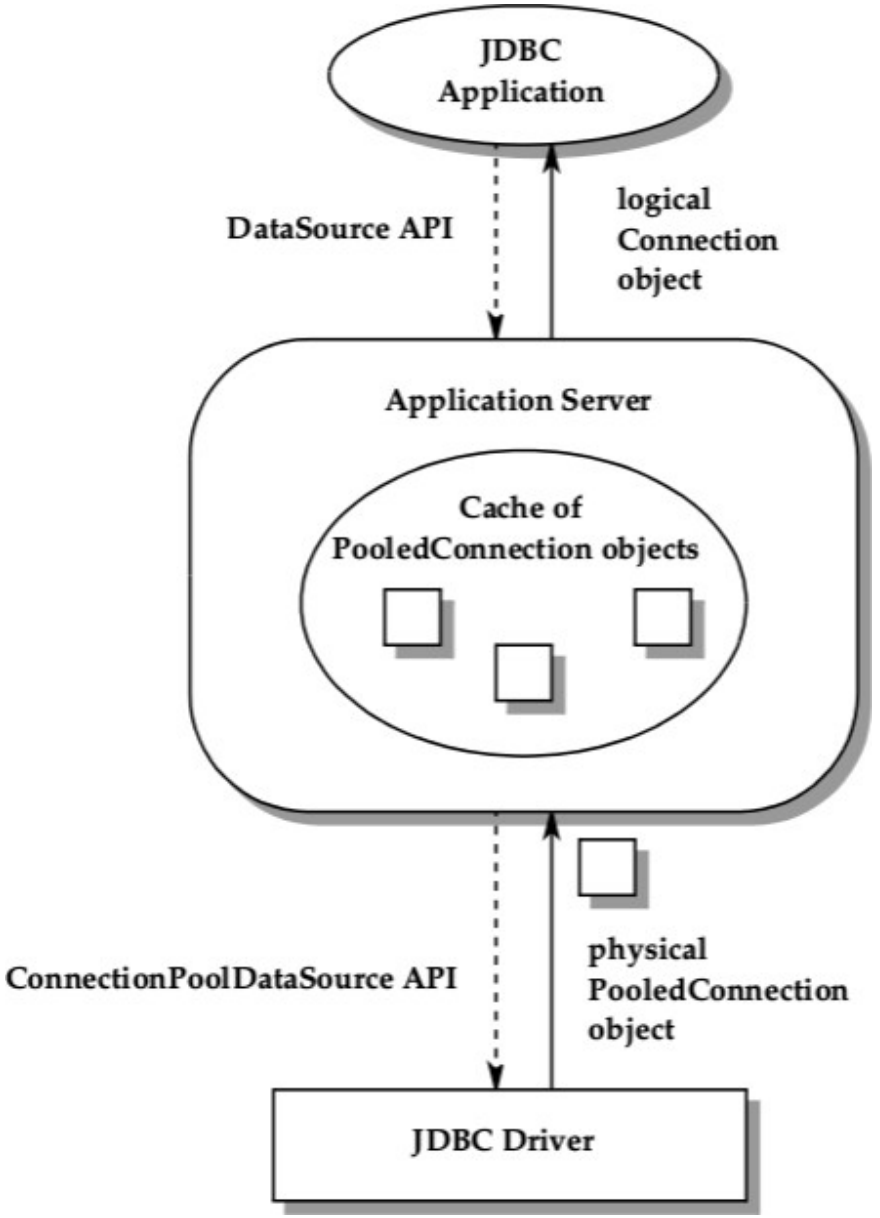
- Create a connection by the use of a `DataSource` object registered at a JNDI name provider:

```
Context ctx = new InitialContext();

DataSource ds = (DataSource) ctx.lookup("jdbc/testDB");
Connection con = ds.getConnection("tom", "secret");
```

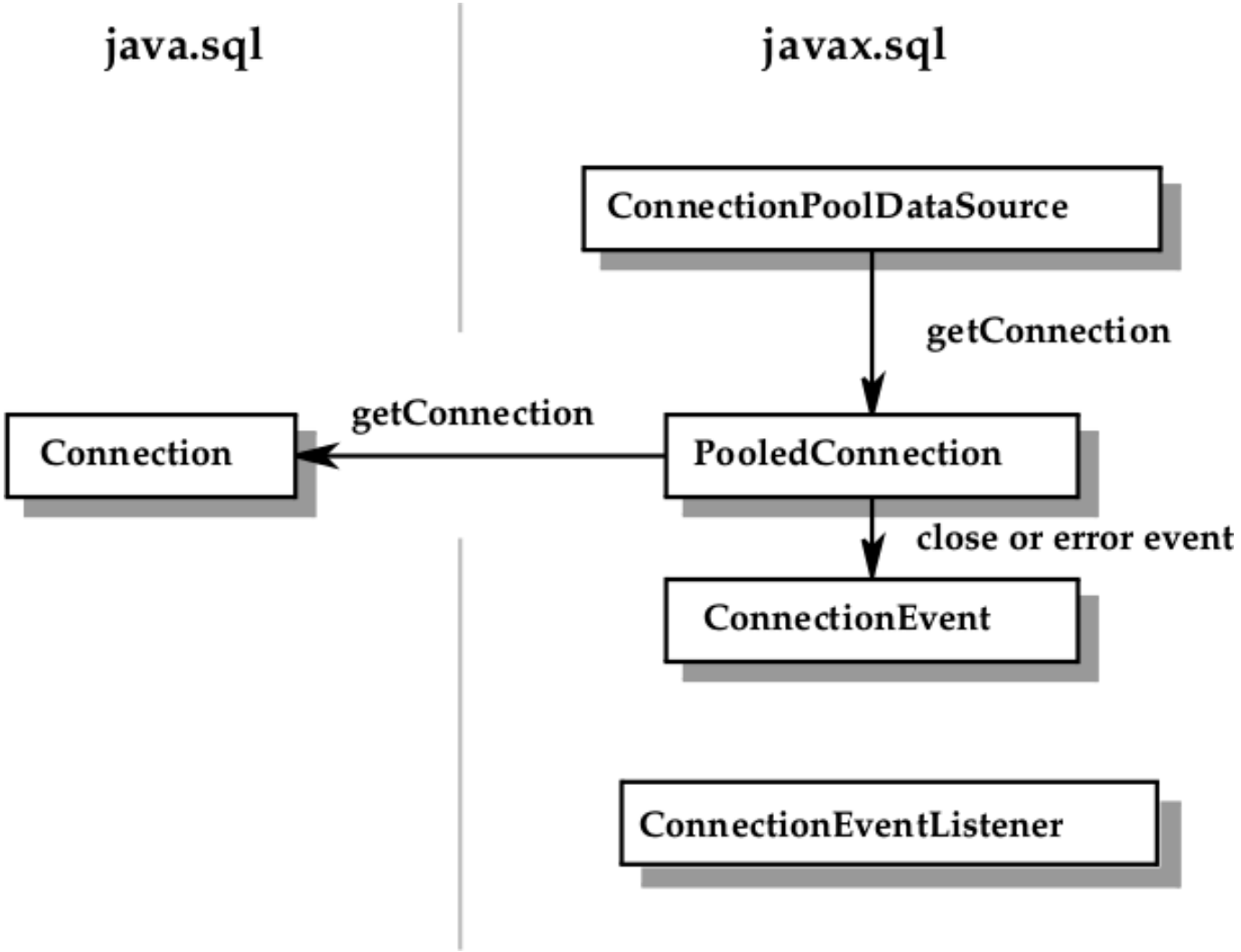
Connection Pooling (1)

- In a simple `DataSource` implementation there is a 1 to 1 mapping between the `Connection` object and the database connection
  - When the connection is closed the physical connection also closes
- The *connection pool* caches connections that can be used through multiple sessions
  - This increases performance

Connection Pooling (2)



Connection Pooling (3)



Connection Pooling (4)

- The `javax.sql.ConnectionPoolDataSource` interface:
  - It can be used to create `javax.sql.PooledConnection objects`
- The `javax.sql.PooledConnection` interface:
  - Represents a reusable physical connection to the data source
    - The connection is not closed when it is not needed anymore byy the client. It goes back to the *connection pool*
  - Application developers do not use it directly. It is handled by the application server.

Connection Pooling (5)

- The *connection pool* manager is informed about events related to the `PooledConnection`, as a result of implementing the `javax.sql.ConnectionEventListener` interface and registers itself as the listener of the `PooledConnection` object
  - The registration is done by the `addConnectionEventListener()` method of the `PooledConnection` interface

Connection Pooling (6)

- Caching statements:
  - Just as connections `java.sql.PreparedStatement` objects can also be cached
  - This is transparent for the client

Connection Pooling (7)

- JDBC defines the following attributes for `ConnectionPoolDataSource` implementations:
  - The implementations provide getters and setters for the supported attributes

Tulajdonság	Típus	Leírás
`maxStatements`	`int`	Max number of open statements
`initialPoolSize`	`int`	Number of physical connections when creating the *connection pool*
`minPoolSize`	`int`	Min number of physical connections
`maxPoolSize`	`int`	Max number of physical connections
`maxIdleTime`	`int`	After how many secons a physical connection can be closed
`propertyCycle`	`int`	How many minutes has to pass before applying the policies above

Connection Pooling (8)

- implementations:
  - Apache Commons DBCP (licenc: Apache License v2)
    https://commons.apache.org/proper/commons-dbcp/
  - C3P0 (licenc: GNU LGPL v2.1)
    http://www.mchange.com/projects/c3p0/
  - HikariCP (licenc: Apache License v2)
    https://brettwooldridge.github.io/HikariCP/
  - Universal Connection Pool (UCP)
    www.oracle.com/technetwork/database/features/jdbc/

Connection pool usage (1)

- As a first step a `ConnectionPoolDataSource` object has to be registered at the JNDI naming provider:

```
com.acme.jdbc.ConnectionPoolDS cpds =
       new com.acme.jdbc.ConnectionPoolDS();
cpds.setServerName("bookserver");
cpds.setDatabaseName("booklist");
cpds.setPortNumber(9040);
cpds.setDescription("Connection pooling for bookserver");

Context ctx = new InitialContext();
ctx.bind("jdbc/pool/bookserver_pool", cpds);
```

Connection pool usage (2)

- After this a `DataSource` object has to be registered at the JNDI name provider which through the client can get logical connections from the pool

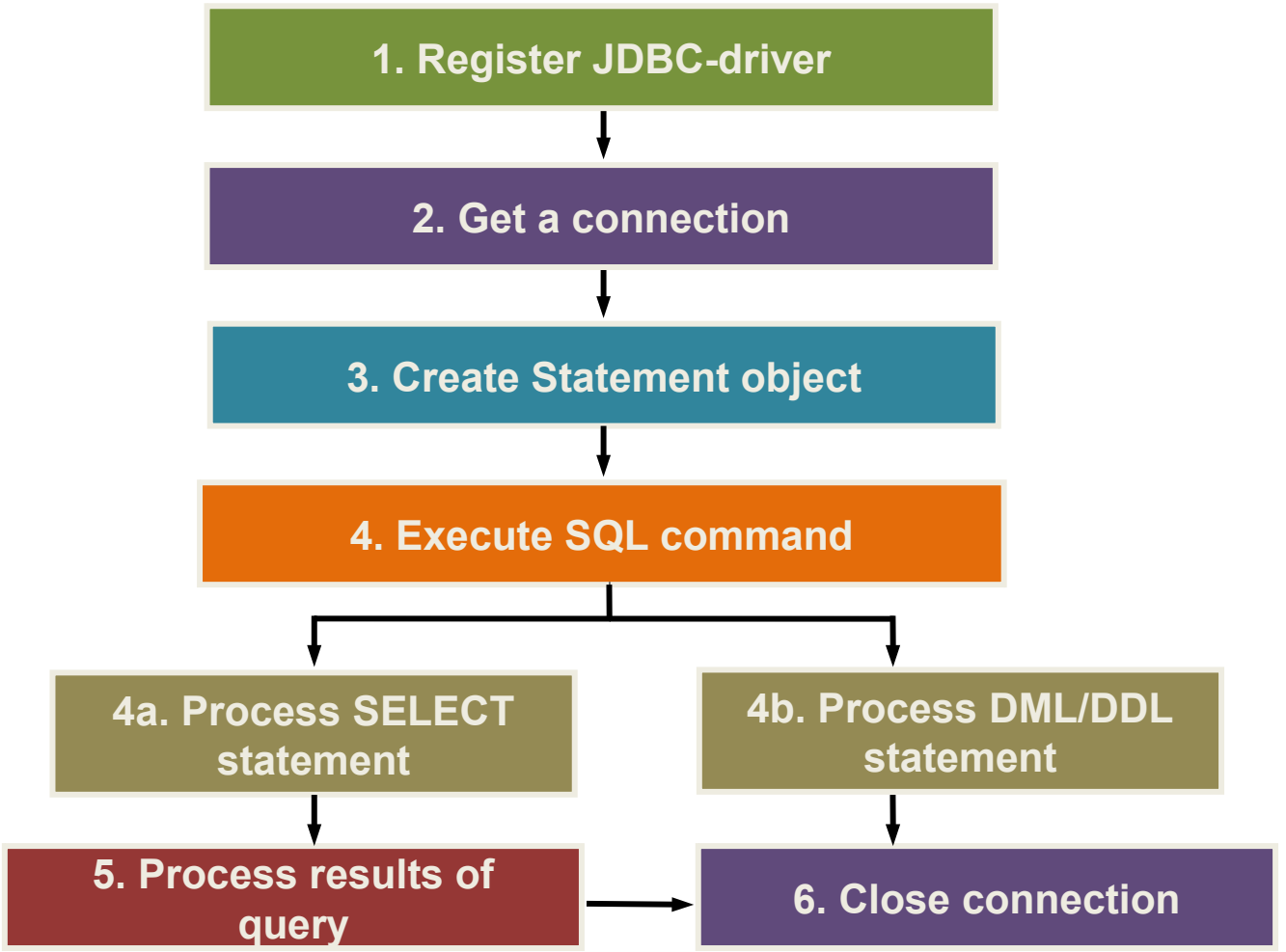
```
com.acme.appserver.PooledDataSource ds =
       new com.acme.appserver.PooledDataSource();
ds.setDescription("Datasource with connection pooling");
ds.setDataSourceName("jdbc/pool/bookserver_pool");

Context ctx = new InitialContext();
ctx.bind("jdbc/bookserver", ds);
```

Connection pool usage (3)

- The client creates a connection in the following way:

```
Context ctx = new InitialContext();
DataSource ds = (DataSource) ctx.lookup("jdbc/bookserver");
```

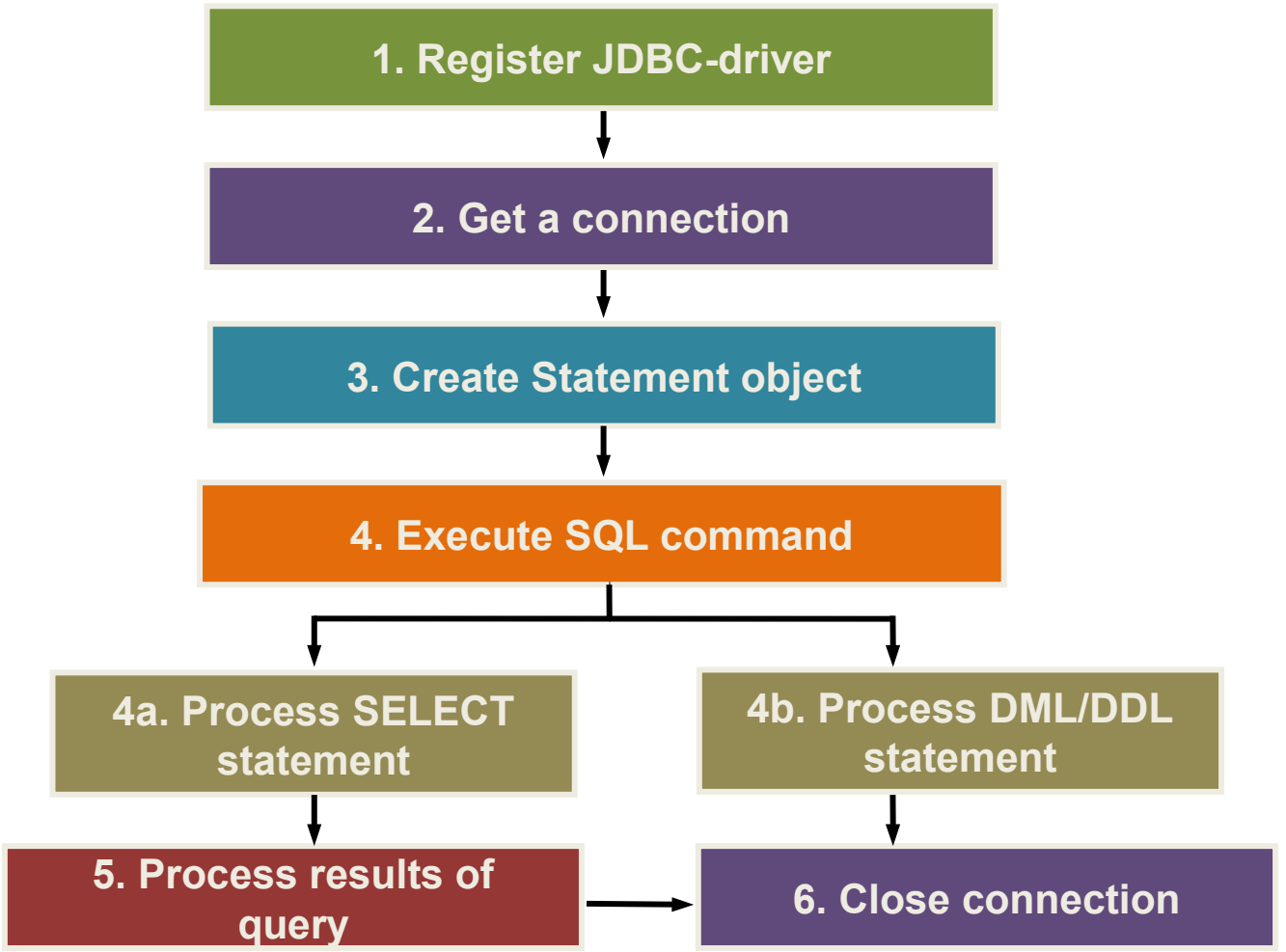
Steps of executing SQL commands



Step 1: registering the driver

- In code:
  - `DriverManager.registerDriver (new oracle.jdbc.OracleDriver());`
  - `Class.forName ("oracle.jdbc.OracleDriver");`
- When loading the class:
  - `java -D jdbc.drivers = oracle.jdbc.OracleDriver <ClassName>;`

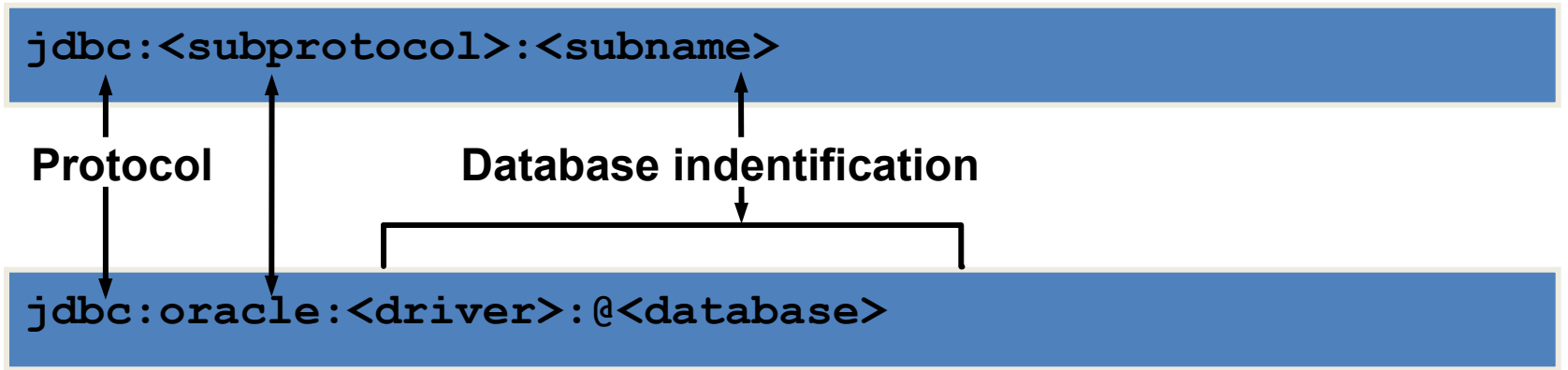
Steps of executing SQL commands



Step 2: getting a database connection

- JDBC 1.0: DriverManager.getConnection()
-
-
-
-
- JDBC URL structure
-

```
Connection conn =
DriverManager.getConnection(   ”jdbc:oracle:thin:@db.inf
.unideb.hu:1521:ora11g",
       ”user",”passwd");
```



JDBC URLs in Oracle

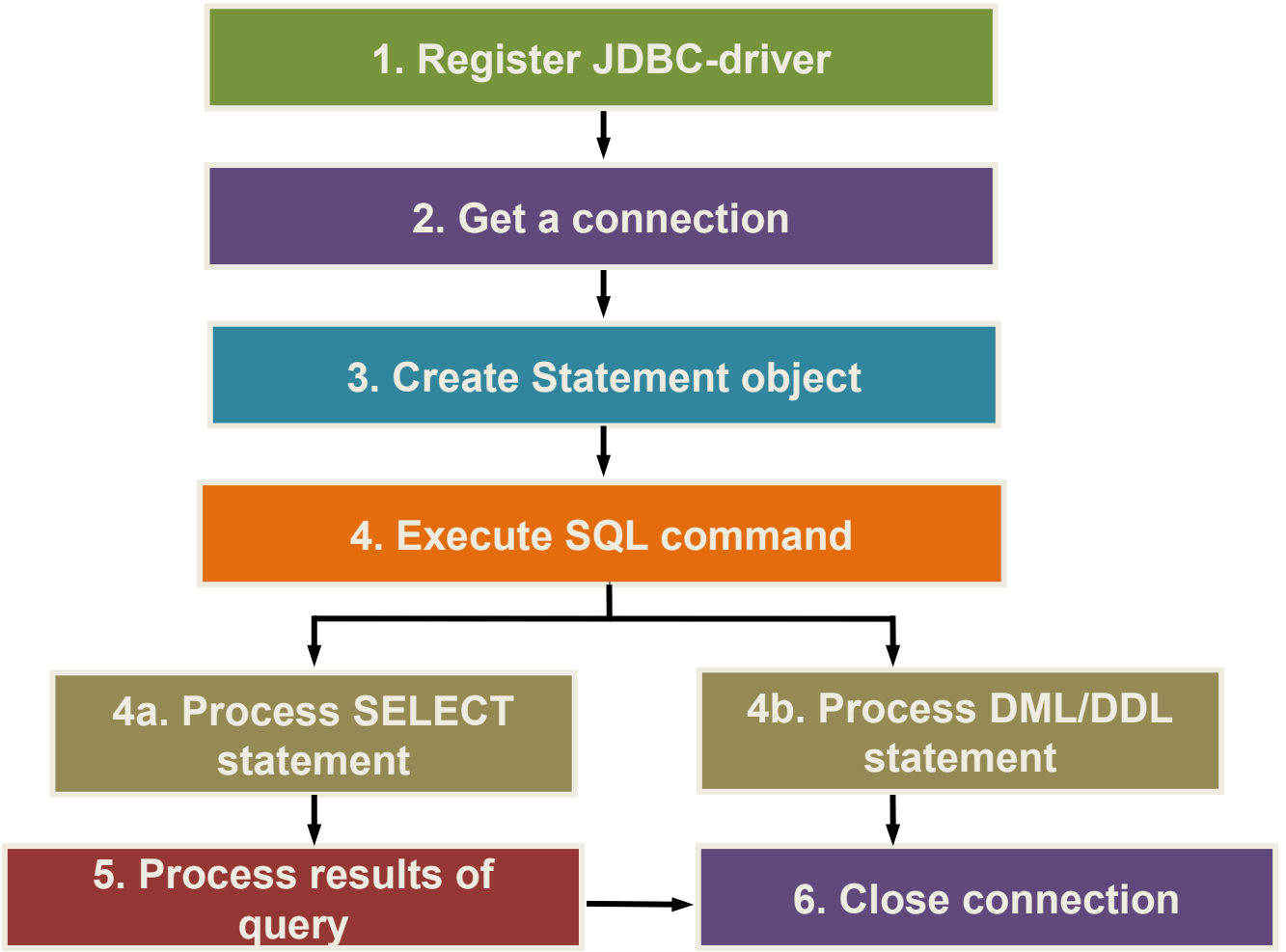
- Oracle Thin driver

```
Syntax:   jdbc:oracle:thin:@<host>:<port>:<SID>
Example:  "jdbc:oracle:thin:@myhost:1521:orcl"
```

- Oracle OCI driver

```
Syntax:   jdbc:oracle:oci:@<tnsname entry>
Example:  "jdbc:oracle:oci:@orcl"
```

Steps of executing SQL commands

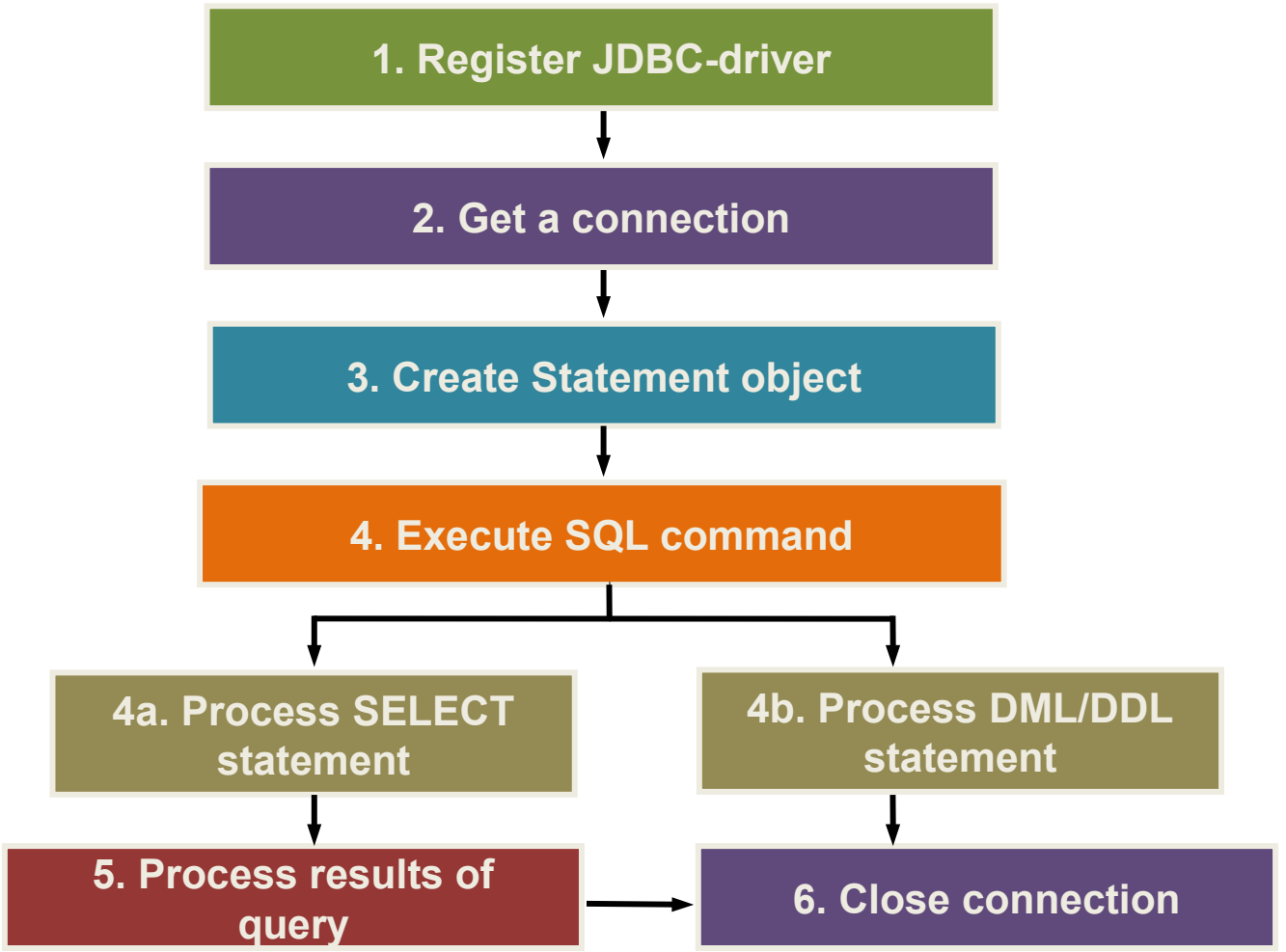


Step 3: Creating a Statement object

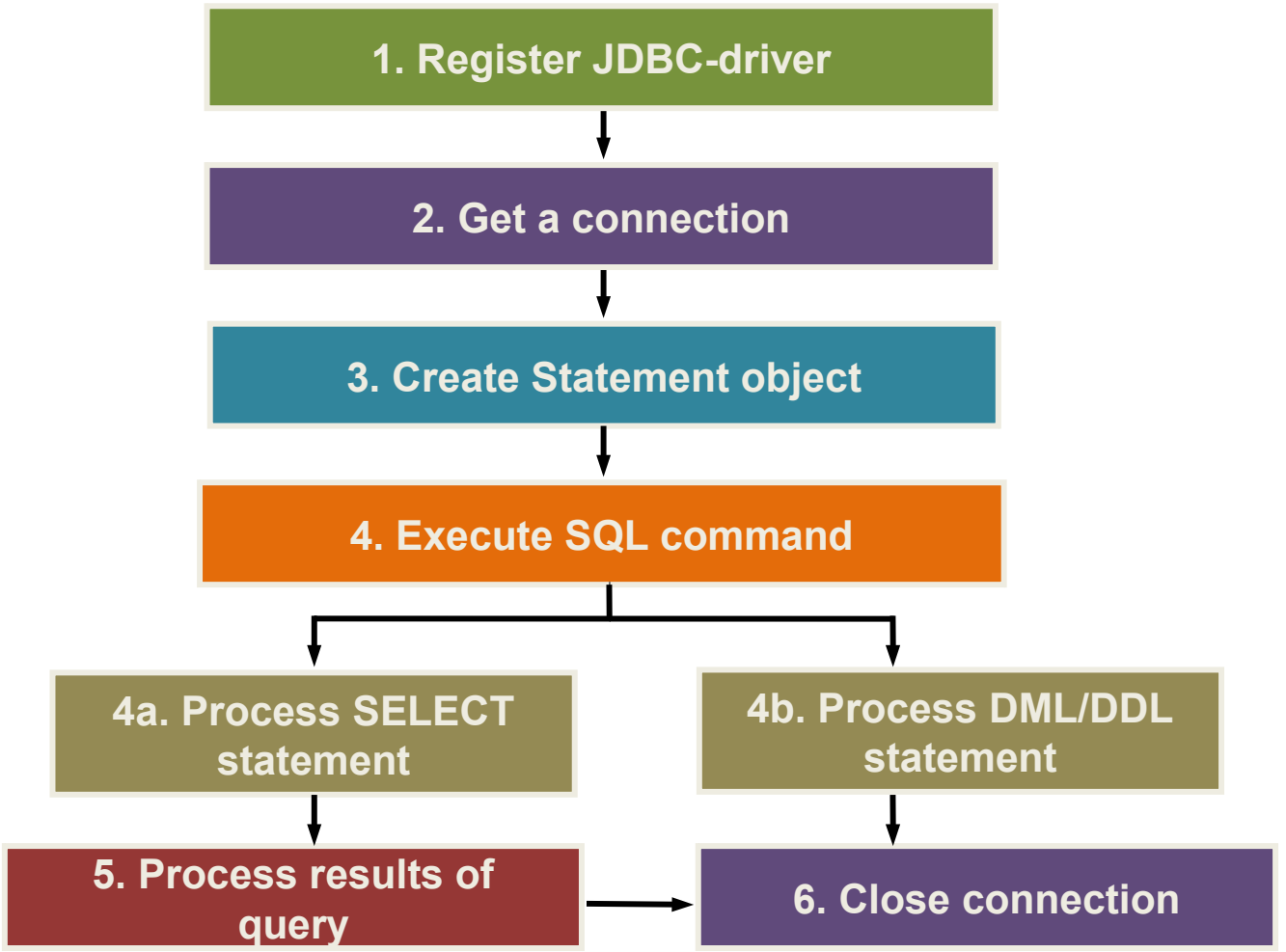
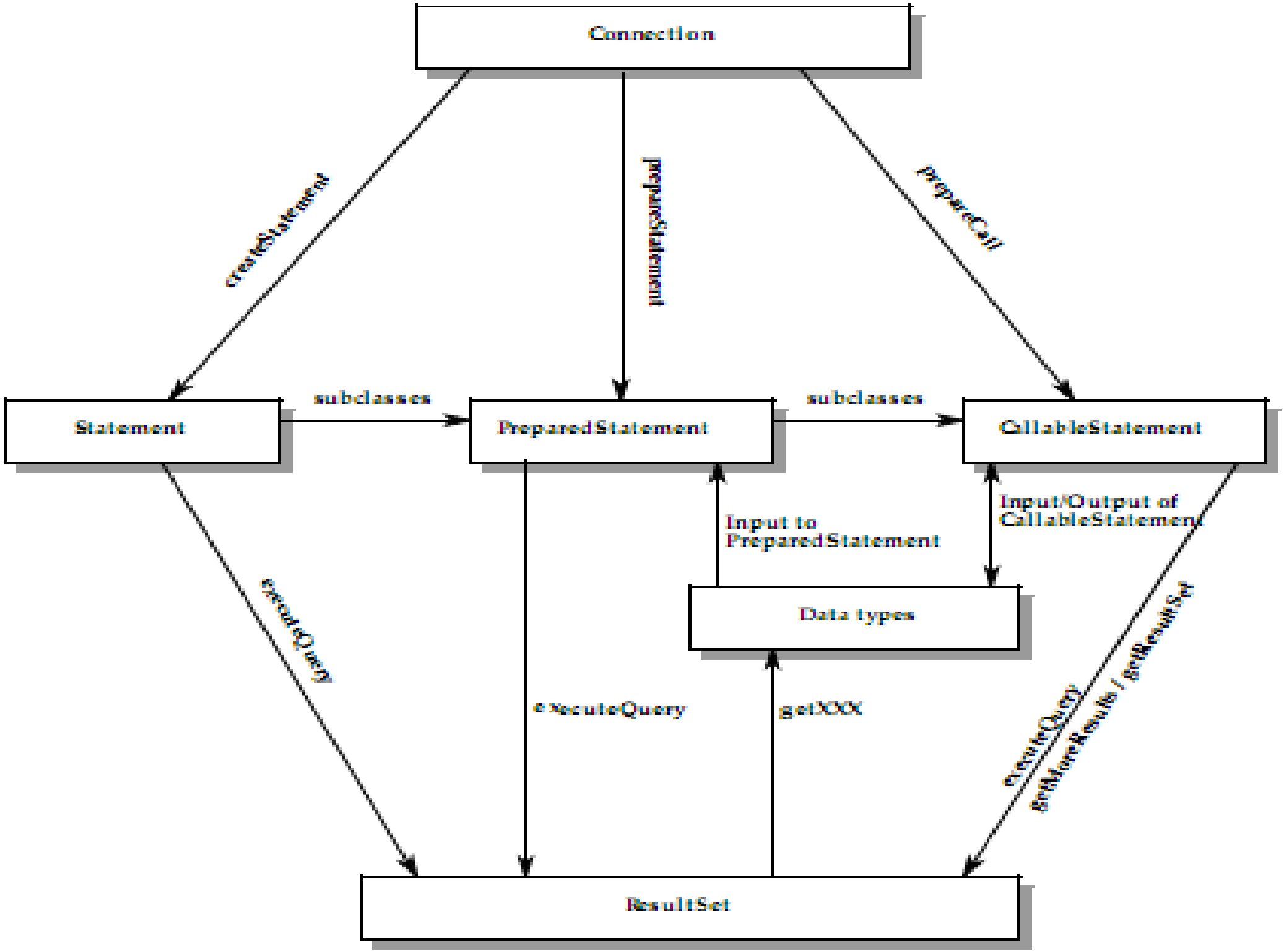
- Based on the Connection instance
- ```
  Statement stmt = conn.createStatement();
  ```
- Methods of the Statement interface:
  - For `SELECT` execution:

    ```
    ResultSet executeQuery(String sql);
    ```
  - For execution of other DML or DDL commands:

    ```
    int executeUpdate(String sql);
    ```
  - For executing arbitrary SQL commands:

    ```
    boolean execute(String sql);
    ```
    ```

Steps of executing SQL commands



Step 4a: Executing a query

- Define the text of the query (without semicolon!)
-
-

```
ResultSet rset = stmt.executeQuery
   ("SELECT ename FROM emp");
```

- The ResultSet interface
  - Represents the resultset of a query (the resulting relation)
  - Shows the actual row with a cursor (its initial position is before the first row)
  - Cannot be modified by default, and can only be navigated forward, can only be went through once
    - Rewritable since JDBC 2

Step 5: Going through a resultset

- With the next() and getXXX() methods of ResultSet
-
-

```
while (rset.next())
  System.out.println (rset.getString(1));
```

  - solution: use getBigDecimal()!

- Lezárás

```
rset.close();
stmt.close();
```

Step 4b: executing a DML command

- ```
  int rowcount = stmt.executeUpdate
    ("DELETE FROM order_items
        WHERE order_id = 2354");
  ```

```
int rowcount = stmt.executeUpdate (
   "INSERT INTO pictures (id) " +
      "SELECT region_id FROM regions");
```

```
int rowcount = stmt.executeUpdate
  ("UPDATE employees SET salary = 1.1 * salary
WHERE dept_id = 20");
```
```

Step 4b: executing a DDL command

- ```
  int rowcount = stmt.executeUpdate
    ("CREATE TABLE temp (col1 NUMBER(5,2),
     col2 VARCHAR2(30))");
  ```
  ```

Steps of executing SQL commands



Step 6: closing the connection

-
```
Connection conn = ...;
Statement stmt = ...;
ResultSet rset = stmt.executeQuery(
           "SELECT ename FROM emp");
...
// clean up
rset.close();
stmt.close();
conn.close();
...
```

Handling unknown commands

```
boolean isQuery = stmt.execute(SQLstatement);
if (isQuery) { // query
  ResultSet r = stmt.getResultSet(); ...
}
else { // other DML or DDL command
  int count = stmt.getUpdateCount(); ...
}
```


Connection
createStatement
prepareStatement
prepareCall
Statement
subclasses
PreparedStatement
subclasses
CallableStatement
Input to
PreparedStatement
Input/Output of
CallableStatement
Data types
executeQuery
executeQuery
getXXX
executeQuery
getMoreResults / getResultSet
ResultSet

PreparedStatement

- Derived from Statement; for storing precompiled SQL
  - commands
- Used when a command is to be executed multiple times
- Parametrizable: actual parameters have to be defined when executed

PreparedStatement

After registering the driver and establishing the connection:

- ```
  PreparedStatement pstmt =
    conn.prepareStatement
  ("UPDATE emp SET ename = ? WHERE empno = ?");
  ```

```
PreparedStatement pstmt =
  conn.prepareStatement
("SELECT ename FROM emp WHERE empno = ?");
```

```
pstmt.setXXX(index, value);
```

```
pstmt.executeQuery();
pstmt.executeUpdate();
```
```

PreparedStatement - example

- After registering the driver and establishing the connection:

```
int empNo = 3521;
PreparedStatement pstmt = conn.prepareStatement(
    "UPDATE emp SET ename = ? WHERE empno = ? ");
pstmt.setString(1, "DURAND");
pstmt.setInt(2, empNo);
pstmt.executeUpdate();
```

- Setting a NULL value:

```
pstmt.setNull(1, java.sql.Types.VARCHAR);
```

ResultSetMetaData

```
PreparedStatement pstmt = conn.prepareStatement(
                "SELECT * FROM CATALOG");
ResultSetMetaData rsmd = pstmt.getMetaData();
int colCount = rsmd.getColumnCount();
int colType;
String colLabel;
for (int i = 1; i <= colCount; i++) {
    colType = rsmd.getColumnType(i);
    colLabel = rsmd.getColumnLabel(i);
    ...
}
```

Exception handling

```
 try  {
     rset = stmt.executeQuery("SELECT empno, name FROM
emp");
}
   catch (java.sql.SQLException e)
   { ... /* handle SQL errors */ }
...
   finally {  // clean up
   try {
     if (rset != null) rset.close();
   } catch
   { ... /* handle closing errors */ }
...
```

Transaction handling

- Autocommit mode is enabled by default
  This can be disabled with a `conn.setAutoCommit(false)` call
  In the latter case
  - `conn.commit()`: commits
  - `conn.rollback()`: rolls back

  The closing of the connection means commiting

JDBC in Java EE platform

- When Java EE components – e.g. JSP or EJB components – use the JDBC API, the container handles the transactions and their data sources
  - In this case the developer does not use the transaction and connection handling facilities of the JDBC API

Database metadata (1)

- The `java.sql.DatabaseMetaData` interfaceis used to get the attributes of the datasource
  - It has more than 150 methods!
  - Example usage:

```
Connection conn;

DatabaseMetaData dmd = conn.getMetaData();
int maxStatementLength = dmd.getMaxStatementLength()
```

Database metadata (2)

- **Getting general information**:
  - E.g.: `getDatabaseProductName()`, `getDatabaseProductVersion()`, `getDriverMajorVersion()`, `getDriverMinorVersion()`, ...
- **Check the availability of possibilities**: `supports`*`XXX`*`()` methods
  - E.g. `supportsGroupBy()`, `supportsOuterJoins()`, `supportsStoredProcedures()`, ...
- **Getting limits of datasources**: `getMax`*`XXX`*`()` methods
  - E.g. `getMaxRowSize()`, `getMaxColumnNameLength()`, `getMaxConnections()`, ...
- **Getting SQL objects and their attributes**:
  - E.g. `getSchemas()`, `getTables()`, `getPrimaryKeys()`, `getProcedures()`, ...
- **Getting properties of transaction handling**:
  - E.g. `getDefaultTransactionIsolation()`, `supportsMultipleTransactions()`

Exceptions (1)

- The `java.sql.SQLException` class and its subclasses provide information about exceptions while using the data source
  - In case of more than one exceptions they are linked. The next element of the chain can be get by the `getNextException()` method

Exceptions (2)

- Processing the linked exceptions:

```
try {
    // ...
} catch(SQLException ex) {
    while(ex != null) {
        System.out.println("SQLState: " + ex.getSQLState());
        System.out.println("Error Code: " + ex.getErrorCode());
        System.out.println("Message: " + ex.getMessage());
        Throwable t = ex.getCause();
        while(t != null) {
            System.out.println("Cause: " + t);
            t = t.getCause();
        }
        ex = ex.getNextException();
    }
}
```

Exceptions (3)

- Processing the linked exceptions:

```
try {
    // ...
} catch(SQLException ex) {
    for(Throwable e : ex) {
        System.out.println("Error encountered: " + e);
    }
}
```

Exceptions (4)

- The `java.sql.SQLWarning` class represents warnings (a subclass of the `java.sql.SQLException` class)
  - The methods of the `java.sql.Connection`, `java.sql.Statement` and `java.sql.ResultSet` interfaces can result in warnings. These can be get by the `getWarnings()` method.
    - Warnings can be linked just like exceptions.

Exceptions (5)

- The `java.sql.BatchUpdateException` class (subclass of `java.sql.SQLException`) provides information about exceptions during bulk processing of statements

Popularity of database management systems

- *DB-Engines Ranking*
  http://db-engines.com/en/ranking
  - *DB-Engines Ranking of Relational DBMS*
    http://db-engines.com/en/ranking/relational+dbms
  - *Popularity of open source DBMS versus commercial DBMS*
    http://db-engines.com/en/ranking_osvsc

Relational DBMS-s

- Commercial:
  - Microsoft SQL Server
  - MySQL Enterprise Edition, MySQL Cluster CGE
  - Oracle Database
  - …
- Free and open-source:
  - MySQL Community Server
  - PostgreSQL
  - H2
  - HSQLDB
  - JavaDB
  - ...

Commercial relational DBMS-s (1)

- Microsoft SQL Server
  http://www.microsoft.com/sqlserver/
  - Developer: Microsoft
  - Platform: Windows
    - Linux support:
      - *Announcing SQL Server on Linux* (March 7, 2016)
        http://blogs.microsoft.com/blog/2016/03/07/announcing-sql-server-on-linux/
      - *SQL Server on Linux* http://www.microsoft.com/sqlserveronlinux
  - Programming language: C, C++

Commercial relational DBMS-s (2)

- MySQL https://www.mysql.com/
  - Developer: *Oracle Corporation*
  - Platform: Linux, Solaris, FreeBSD, Mac OS X, Windows
  - Programming languages: C, C++
  - Versions:
    - *MySQL Enterprise Edition* *http://www.mysql.com/products/enterprise/* *http://www.oracle.com/us/products/mysql/mysqlenterprise/overview/*
    - *MySQL Standard Edition* http://www.mysql.com/products/standard/
    - *MySQL Classic Edition* http://www.mysql.com/products/classic/
    - *MySQL Cluster CGE* https://www.mysql.com/products/cluster/

Commercial relational DBMS-s (3)

- Oracle Database https://www.oracle.com/database/
  - Developer: *Oracle Corporation*
  - Platform: Unix-szerű, Windows
  - Programming language: Assembly, C, C++
  - Versions:
    - *Oracle Database 12c Enterprise Edition*
    - *Oracle Database 12c Standard Edition 2*
    - *Oracle Database 11g Express Edition*
      - Free to use with limited facilities
        - Max 11 GB data, 1 GB memory, 1 CPU

Free and open-source relational DBMS-s (1)

- MySQL Community Server
  http://dev.mysql.com/downloads/mysql/
  - Developer: *Oracle Corporation*
  - Platform: Linux, Solaris, FreeBSD, Mac OS X, Windows
  - Programming language: C, C++
  - License: GNU GPL v2

Free and open-source relational DBMS-s (2)

- PostgreSQL http://www.postgresql.org/
  - Developer: *PostgreSQL Global Development Group*
  - Platform: Unix-szerű, Windows
  - Programming language: C
  - License: *PostgreSQL License* http://www.postgresql.org/about/licence/

Free and open-source relational DBMS-s (3)

- H2 http://www.h2database.com/
  - Platform: Java
  - Programming language: Java
  - License: *Mozilla Public License v2*, *Eclipse Public License 1.0*
  - Usage: *embedded*, server
  - JDBC compatibility: 4.0

Free and open-source relational DBMS-s (4)

- HSQLDB http://hsqldb.org/
  - Developer: *The HSQL Development Group*
  - Platform: Java
  - Programming language: Java
  - License: *BSD License* http://hsqldb.org/web/hsqlLicense.html
  - Usage: embedded *in-process*, server
  - JDBC compatibility: 4.0

Free and open-source relational DBMS-s (5)

- JavaDB http://docs.oracle.com/javadb/
  - Developer: *Apache Software Foundation*
    - JavaDB is a mirror copy of Apache Derby https://db.apache.org/derby/
  - Platform: Java
  - Programming language: Java
  - License: *Apache License v2*
  - Usage: *embedded*, server
  - Availibility: part of JDK, contained by the db/ folder of JDK installation
  - JDBC compatibility: 4.2

Availibility of JDBC drivers (1)

- Oracle Database:
  - *JDBC and Universal Connection Pool (UCP*) http://www.oracle.com/technetwork/database/features/jdbc/index-091264.html
  - *Get Oracle JDBC drivers from the Oracle Maven Repository* https://blogs.oracle.com/dev2dev/entry/oracle_maven_repository_instructions_for
- Microsoft SQL Server:
  - *Download Microsoft JDBC Driver for SQL Server* https://msdn.microsoft.com/en-us/library/mt683464%28v=sql.110%29.aspx

Availibility of JDBC drivers (2)

- Maven:

Product	Maven coordinates
MySQL	`mysql:mysql-connector-java:6.0.2`
PostgreSQL	`org.postgresql:postgresql:9.4.1208.jre7`
H2	`com.h2database:h2:1.4.191`
HSQLDB	`org.hsqldb:hsqldb:2.3.3`
JavaDB	`org.apache.derby:derby:10.12.1.1`

JDBC driver classes

Product	JDBC driver classy	JDBC compatibility
Oracle Database	`oracle.jdbc.OracleDriver`	4.2
Microsoft SQL Server	`com.microsoft.sqlserver.jdbc.SQLServerDriver`	4.2
MySQL	`com.mysql.jdbc.Driver`	4.0
PostgreSQL	`org.postgresql.Driver`	4.2
H2	`org.h2.Driver`	4.0
HSQLDB	`org.hsqldb.jdbc.JDBCDriver`	4.0
JavaDB	`org.apache.derby.jdbc.EmbeddedDriver` `org.apache.derby.jdbc.ClientDriver`	4.2

Recommended reading

- *The Java Tutorials: Trail: JDBC Database Access*
  https://docs.oracle.com/javase/tutorial/jdbc/