JDBC in Java applications

Krzysztof Podlaski

Faculty of Physics and Applied Informatics,
University of Lodz,
Poland

AP Hogeschool Antwerpen, 19 November 2019

Agenda

1. Introduction
2. Query database
   - Statement
   - PreparedStatement
   - CallableStatement
3. Operate on DB results
4. Transactions
5. JDBC Annotations
6. Extensions

Introduction

Java **D**atabase **C**onnectivity

- API for relational databases

Introduction

Java **D**atabase **C**onnectivity

- API for relational databases
  - Independence on batabase engine

Introduction

Java **D**atabase **C**onnectivity

- API for relational databases
  - Independence on batabase engine
  - MySQL, PostgreSQL, Oracle, DB2, MSSQL, ...

Introduction

Java **D**atabase **C**onnectivity

- API for relational databases
  - Independence on batabase engine
  - MySQL, PostgreSQL, Oracle, DB2, MSSQL, ...
  - Even local databases like SQLite

Introduction

Java **D**atabase **C**onnectivity

- API for relational databases
  - Independence on batabase engine
  - MySQL, PostgreSQL, Oracle, DB2, MSSQL, ...
  - Even local databases like SQLite
  - One code many database implementations all dependency issues solved via DriverManager

Introduction

Java **D**atabase **C**onnectivity

- API for relational databases
  - Independence on batabase engine
  - MySQL, PostgreSQL, Oracle, DB2, MSSQL, ...
  - Even local databases like SQLite
  - One code many database implementations all dependency issues solved via DriverManager
- Based on ODBC (Open Data Base Conectivity)

JDBC versions used in development

JDBC versions used in development

- JDBC 3.0

JDBC versions used in development

- JDBC 3.0
- JDBC 4.0

JDBC versions used in development

- JDBC 3.0
- JDBC 4.0
  - Require Java 6+

JDBC versions used in development

- JDBC 3.0
- JDBC 4.0
  - Require Java 6+
  - Introduced annotations

JDBC versions used in development

- JDBC 3.0
- JDBC 4.0
  - Require Java 6+
  - Introduced annotations
  - Can work on XML like on Relational databases

JDBC versions used in development

- JDBC 3.0
- JDBC 4.0
  - Require Java 6+
  - Introduced annotations
  - Can work on XML like on Relational databases
- JDBC 4.1, 4.2, 4.3

JDBC versions used in development

- JDBC 3.0
- JDBC 4.0
  - Require Java 6+
  - Introduced annotations
  - Can work on XML like on Relational databases
- JDBC 4.1, 4.2, 4.3
  - Not all database engines provide such driver

What we need to use JDBC

Requirements:

What we need to use JDBC

Requirements:

- JDBC provides set of Interfaces : *java.sql, javax.sql*

What we need to use JDBC

Requirements:

- JDBC provides set of Interfaces : *java.sql, javax.sql*
- Drivers, specific database engine

What we need to use JDBC

Requirements:

- JDBC provides set of Interfaces : *java.sql, javax.sql*
- Drivers, specific database engine
  - (Type 3, 4) Pure java solutions

What we need to use JDBC

Requirements:

- JDBC provides set of Interfaces : *java.sql, javax.sql*
- Drivers, specific database engine
  - (Type 3, 4) Pure java solutions
  - (Type 1, 2) Based on ODBC and JNI

DB Connection

- *DriverManager* - a class used to connect to concrete database. All drivers are usually registers.

DB Connection

- *DriverManager* - a class used to connect to concrete database. All drivers are usually registers.
- Old approaches required loading driver class by hand:

```
1 Class.forName("com.mysql.jdbc.Driver").newInstance()
```

DB Connection

- *DriverManager* - a class used to connect to concrete database. All drivers are usually registers.
- Old approaches required loading driver class by hand:

```
1 Class.forName("com.mysql.jdbc.Driver").newInstance()
```

- Connection do database

```
1 Connection dCon =DriverManager.getConnection(
2 "jdbc:mysql:IP/DBNAME", USER_NAME, PASSWORD);
```

DB Connection

- *DriverManager* - a class used to connect to concrete database. All drivers are usually registers.
- Old approaches required loading driver class by hand:

```
1 Class.forName("com.mysql.jdbc.Driver").newInstance()
```

- Connection do database

```
1 Connection dCon =DriverManager.getConnection(
2 "jdbc:mysql:IP/DBNAME", USER_NAME, PASSWORD);
```

- *Connection* - an object that is responsible for connection between Java and database.

Query Using Statement

- Simple SQL query with use of *Statement*

```
1 Statement st = dCon.createStatement();
2 String sql = "SELECT␣*␣FROM␣Student␣WHERE␣id>12␣AND␣name='
      James'";
3 st.execute(sql);
```

Query Using Statement

- Simple SQL query with use of *Statement*

```
1 Statement st = dCon.createStatement();
2 String sql = "SELECT␣*␣FROM␣Student␣WHERE␣id>12␣AND␣name='
      James'";
3 st.execute(sql);
```

- We can use specific operations for different query types

Query Using Statement

- Simple SQL query with use of *Statement*

```
1 Statement st = dCon.createStatement();
2 String sql = "SELECT␣*␣FROM␣Student␣WHERE␣id>12␣AND␣name='
      James'";
3 st.execute(sql);
```

- We can use specific operations for different query types
  - SELECT operations:

```
1 st.executeQuery(sql)
```

Query Using Statement

- Simple SQL query with use of *Statement*

```
1 Statement st = dCon.createStatement();
2 String sql = "SELECT␣*␣FROM␣Student␣WHERE␣id>12␣AND␣name='
      James'";
3 st.execute(sql);
```

- We can use specific operations for different query types
  - SELECT operations:

```
1 st.executeQuery(sql)
```

  - UPDATE/INSERT/DELETE operations:

```
1 st.executeUpdate(sql)
```

Statement

Properties of the *Statement*:

Statement

Properties of the *Statement*:

- A simple class that allows query the database

Statement

Properties of the *Statement*:

- A simple class that allows query the database
- Used only with static query's

Statement

Properties of the *Statement*:

- A simple class that allows query the database
- Used only with static query's
- All parameters have to be hardcoded as Strings

Statement

Properties of the *Statement*:

- A simple class that allows query the database
- Used only with static query's
- All parameters have to be hardcoded as Strings
- Hand type control

Statement

Properties of the *Statement*:

- A simple class that allows query the database
- Used only with static query's
- All parameters have to be hardcoded as Strings
- Hand type control
  - Programmer have to know how to encode data for the specific database used

Statement

Properties of the *Statement*:

- A simple class that allows query the database
- Used only with static query's
- All parameters have to be hardcoded as Strings
- Hand type control
  - Programmer have to know how to encode data for the specific database used
  - For example Date object or String escape characters

Statement

Properties of the *Statement*:

- A simple class that allows query the database
- Used only with static query's
- All parameters have to be hardcoded as Strings
- Hand type control
  - Programmer have to know how to encode data for the specific database used
  - For example Date object or String escape characters
- Almost impossible to use with BLOB's CLOB's

Statement

Properties of the *Statement*:

- A simple class that allows query the database
- Used only with static query's
- All parameters have to be hardcoded as Strings
- Hand type control
  - Programmer have to know how to encode data for the specific database used
  - For example Date object or String escape characters
- Almost impossible to use with BLOB's CLOB's
- Vulnerable to SQLInjection

Query Using PreparedStatement

- Reusable SQL query with use of *PreparedStatement*

```
1 String sql = "SELECT␣*␣FROM␣Student␣WHERE␣id>?␣AND␣name=?";
2 PreparedStatement pst = dCon.prepareStatement(sql);
```

Query Using PreparedStatement

- Reusable SQL query with use of *PreparedStatement*

```
1 String sql = "SELECT␣*␣FROM␣Student␣WHERE␣id>?␣AND␣name=?";
2 PreparedStatement pst = dCon.prepareStatement(sql);
```

- Each ? represents a parameter of the query

Query Using PreparedStatement

- Reusable SQL query with use of *PreparedStatement*

```
1 String sql = "SELECT␣*␣FROM␣Student␣WHERE␣id>?␣AND␣name=?";
2 PreparedStatement pst = dCon.prepareStatement(sql);
```

- Each ? represents a parameter of the query
- Setting values of a parameter require type checking:

```
1 pst.setString(paramID,paramValue)
2 pst.setInt(paramID,paramValue)
```

Query Using PreparedStatement

- Reusable SQL query with use of *PreparedStatement*

```
1 String sql = "SELECT␣*␣FROM␣Student␣WHERE␣id>?␣AND␣name=?";
2 PreparedStatement pst = dCon.prepareStatement(sql);
```

- Each ? represents a parameter of the query
- Setting values of a parameter require type checking:

```
1 pst.setString(paramID,paramValue)
2 pst.setInt(paramID,paramValue)
```

- Query the database:

```
1 pst.execute();
2 pst.executeQuery();
3 pst.executeUpdate();
```

PreparedStatement

Properties of the *PreaparedStatement*:

PreparedStatement

Properties of the *PreaparedStatement*:

- Pre-compiled – faster execution than *Statement*

PreparedStatement

Properties of the *PreaparedStatement*:

- Pre-compiled – faster execution than *Statement*
- Queries are partially cached on database side – native binary communication is faster

PreparedStatement

Properties of the *PreaparedStatement*:

- Pre-compiled – faster execution than *Statement*
- Queries are partially cached on database side – native binary communication is faster
- Java checks types and their encoding

PreparedStatement

Properties of the *PreaparedStatement*:

- Pre-compiled – faster execution than *Statement*
- Queries are partially cached on database side – native binary communication is faster
- Java checks types and their encoding
- Possibility to work with BLOB's CLOB's

PreparedStatement

Properties of the *PreaparedStatement*:

- Pre-compiled – faster execution than *Statement*
- Queries are partially cached on database side – native binary communication is faster
- Java checks types and their encoding
- Possibility to work with BLOB's CLOB's
- Protection from SQLInjection

Query Using CallableStatement

- Call database stored procedures with *CallableStatement*

Query Using CallableStatement

- Call database stored procedures with *CallableStatement*
- suppose we have a stored procedure *some_proc* that takes three parameters on the database engine

```
1 String call = "{call␣some_proc(?,?,?)}";
2 CallableStatement cst = dCon.prepareCall(call);
```

Query Using CallableStatement

- Call database stored procedures with *CallableStatement*
- suppose we have a stored procedure *some_proc* that takes three parameters on the database engine

```
1 String call = "{call␣some_proc(?,?,?)}";
2 CallableStatement cst = dCon.prepareCall(call);
```

- Each ? represents a parameter of the procedure

Query Using CallableStatement

- Call database stored procedures with *CallableStatement*
- suppose we have a stored procedure *some_proc* that takes three parameters on the database engine

```
1 String call = "{call␣some_proc(?,?,?)}";
2 CallableStatement cst = dCon.prepareCall(call);
```

- Each ? represents a parameter of the procedure
- We use *CallableStatement* like *PrepareStatement*

CallableStatement

Properties of the *CallableStatement*:

CallableStatement

Properties of the *CallableStatement*:

- Used only for procedures stored on the database

CallableStatement

Properties of the *CallableStatement*:

- Used only for procedures stored on the database
- Three types of parameters IN, OUT, IN OUT

CallableStatement

Properties of the *CallableStatement*:

- Used only for procedures stored on the database
- Three types of parameters IN, OUT, IN OUT
  - OUT parameter has to be registered

```
1 cst.registerOutParameter(paramID, DataType);
```

CallableStatement

Properties of the *CallableStatement*:

- Used only for procedures stored on the database
- Three types of parameters IN, OUT, IN OUT
  - OUT parameter has to be registered

```
1 cst.registerOutParameter(paramID, DataType);
```

- High performance

Query results

- *ResultSet* – used to retrive data returned from *Statement*, *PerparedStatement or CallableSatement*

```
1 statement.getResultSet();
2 prepareStatement.getResultSet();
3 callableStatement.getResultSet();
```

Query results

- *ResultSet* – used to retrive data returned from *Statement*, *PerparedStatement or CallableSatement*

```
1 statement.getResultSet();
2 prepareStatement.getResultSet();
3 callableStatement.getResultSet();
```

- Fulfills Iterator pattern

```
1 ResultSet rs = st.getResultSet();
2 while (rs.next()){ ... }
```

Query results

- *ResultSet* – used to retrive data returned from *Statement*, *PerparedStatement or CallableSatement*

```
1 statement.getResultSet();
2 prepareStatement.getResultSet();
3 callableStatement.getResultSet();
```

- Fulfills Iterator pattern

```
1 ResultSet rs = st.getResultSet();
2 while (rs.next()){ ... }
```

- Each column element can be retrieved by name or id

```
1 rs.getString("name");
2 rs.getInt("height");
3 rs.getString(1);
```

ResultSet cont.

- Using retrieve result we can use fast row operations:

ResultSet cont.

- Using retrieve result we can use fast row operations:
  - Move to a row by its number

```
1 rs.absolute(12);
```

ResultSet cont.

- Using retrieve result we can use fast row operations:
  - Move to a row by its number

```
1 rs.absolute(12);
```

  - Update actual row

```
1 rs.updateString("name","Tintin");
2 rs.updateRow();
```

ResultSet cont.

- Using retrieve result we can use fast row operations:
  - Move to a row by its number

```
1 rs.absolute(12);
```

  - Update actual row

```
1 rs.updateString("name","Tintin");
2 rs.updateRow();
```

  - Insert a new row

```
1 rs.insertRow();
```

ResultSet cont.

- Using retrieve result we can use fast row operations:
  - Move to a row by its number

```
1 rs.absolute(12);
```

  - Update actual row

```
1 rs.updateString("name","Tintin");
2 rs.updateRow();
```

  - Insert a new row

```
1 rs.insertRow();
```

  - Delete actual row

```
1 rs.deleteRow();
```

Transactions

- By default all queries are committed at once

Transactions

- By default all queries are committed at once
- Sometimes we need to guarantee all operations are successful

Transactions

- By default all queries are committed at once
- Sometimes we need to guarantee all operations are successful
  - All or None approach

Transactions

- By default all queries are committed at once
- Sometimes we need to guarantee all operations are successful
  - All or None approach
- Transactions procedure:

Transactions

- By default all queries are committed at once
- Sometimes we need to guarantee all operations are successful
  - All or None approach
- Transactions procedure:
  - Turn off AutoCommit

```
1 dCon.setAutoCommit(false);
```

Transactions

- By default all queries are committed at once
- Sometimes we need to guarantee all operations are successful
  - All or None approach
- Transactions procedure:
  - Turn off AutoCommit

```
1 dCon.setAutoCommit(false);
```

  - Plan all queries

Transactions

- By default all queries are committed at once
- Sometimes we need to guarantee all operations are successful
  - All or None approach
- Transactions procedure:
  - Turn off AutoCommit

```
1 dCon.setAutoCommit(false);
```

  - Plan all queries
  - Commit all planned queries at once

```
1 dCon.commit();
```

Savepoints and Rollback

- Transaction should be wrapped within *try catch*

Savepoints and Rollback

- Transaction should be wrapped within *try catch*
- If transaction fails *SQLException* is thrown

Savepoints and Rollback

- Transaction should be wrapped within *try catch*
- If transaction fails *SQLException* is thrown
  - In that case we should rollback all operations

```
1 dCon.rollback();
```

Savepoints and Rollback

- Transaction should be wrapped within *try catch*
- If transaction fails *SQLException* is thrown
  - In that case we should rollback all operations

```
1 dCon.rollback();
```

- We can define points to return in rollback

```
1 Savepoint spoint = dCon.setSavepoint();
2 ...
3 dCon.rollback(spoint);
```

Closing operations

- Java takes care of memory issues *Garbage Collector*

Closing operations

- Java takes care of memory issues *Garbage Collector*
- We have to care about closing operations

Closing operations

- Java takes care of memory issues *Garbage Collector*
- We have to care about closing operations
- All elements *ResultSet, Statement, PrepareStatement, CallableStatement, Connection* have *.close()* method.

Closing operations

- Java takes care of memory issues *Garbage Collector*
- We have to care about closing operations
- All elements *ResultSet, Statement, PrepareStatement, CallableStatement, Connection* have *.close()* method.
- We should close **all** elements in proper order.

Closing operations

- Java takes care of memory issues *Garbage Collector*
- We have to care about closing operations
- All elements *ResultSet, Statement, PrepareStatement, CallableStatement, Connection* have *.close()* method.
- We should close **all** elements in proper order.
- At least *Connection* object have to be closed.

Data types

- Java types are not exactly the same as database types

Data types

- Java types are not exactly the same as database types
- Basic types are easy to convert

Java	Database
String	Text, Varchar
int	bit, int, decimal
float,double	float, real

Data types

- Java types are not exactly the same as database types
- Basic types are easy to convert

Java	Database
String	Text, Varchar
int	bit, int, decimal
float,double	float, real

- Special types have *java.sql* counterparts

Data types

- Java types are not exactly the same as database types
- Basic types are easy to convert

Java	Database
String	Text, Varchar
int	bit, int, decimal
float,double	float, real

- Special types have *java.sql* counterparts
  - Date, Blob, Clob, NClob, ...

Data types

- Java types are not exactly the same as database types
- Basic types are easy to convert

Java	Database
String	Text, Varchar
int	bit, int, decimal
float,double	float, real

- Special types have *java.sql* counterparts
  - Date, Blob, Clob, NClob, ...
- Programmer can create own type mappings

Define Annotations

We can define interface that will do Query operation

Define Annotations

We can define interface that will do Query operation

- Define interface

```
1 public interface PersonQuery extebnds BaseQuery{
2 @Select(sql = "SELECT␣*␣FROM␣PERSON␣WHERE␣id=?1")
3 public DataSet<Person> getPersonById(int id);
4 }
```

Define Annotations

We can define interface that will do Query operation

- Define interface

```
1 public interface PersonQuery extebnds BaseQuery{
2 @Select(sql = "SELECT␣*␣FROM␣PERSON␣WHERE␣id=?1")
3 public DataSet<Person> getPersonById(int id);
4 }
```

- Usage example

```
1 PersonQuery pq = dCon.createQueryObject(PersonQuery.class);
2 Collection<Person> = pq.getPersonById(13);
```

What is next

- Relational objective mapping

What is next

- Relational objective mapping
  - JPA, Hibernate, ...

What is next

- Relational objective mapping
  - JPA, Hibernate, ...
- Data Access Objects (DAO)

What is next

- Relational objective mapping
  - JPA, Hibernate, ...
- Data Access Objects (DAO)
- MVC and MVC+