JDBC

Tamás Ambrus, István Gansperger

Eötvös Loránd University
ambrus.thomas@gmail.com

JDBC Interface

Java Database Connectivity (JDBC) is an application programming interface (API) for Java, which defines how a client may access a database.

It is part of the Java Standard Edition platform, from Oracle Corporation. It provides methods to query and update data in a database, and is oriented towards relational databases.

JDBC Interface

Java Database Connectivity (JDBC) is an application programming interface (API) for Java, which defines how a client may access a database.

It is part of the Java Standard Edition platform, from Oracle Corporation. It provides methods to query and update data in a database, and is oriented towards relational databases.

All the necessary classes are in the **java.sql** and **javax.sql** packages. As types in these packages are mostly interfaces, we will need to **add an implementation to the build path**.

Useful interfaces

- **Connection**: a connection (session) to a specific database. SQL statements are executed and results are returned within the context of a connection.
- **Statement**: the object used for executing a static SQL statement and returning the results it produces.
- **PreparedStatement**: an object that represents a precompiled SQL statement.
- **ResultSet**: a table of data representing a database result set, which is usually generated by executing a statement that queries the database.
- **SQLException**: an exception that provides information on a database access error or other errors.

Connection

So first we need to build a connection to a database. Obviously we cannot reach the database without username and password. Here's an example:

```
try (Connection conn =
   DriverManager.getConnection("jdbc:mysql://localhost/world?" +
   "user=tanulo&password=asd123&useSSL=false")) {

  // do something with conn

} catch (SQLException ex) {
    ex.printStackTrace();
}
```

Statement

```
try (Statement stmt = conn.createStatement()) {
  ResultSet rs = stmt.executeQuery("SELECT * FROM city");
  while (rs.next()) {
    String name = rs.getString("Name");
    int id = rs.getInt(1);
    System.out.println(id + " " + name);
  }
}
```

Statement

```
try (Statement stmt = conn.createStatement()) {
  ResultSet rs = stmt.executeQuery("SELECT * FROM city");
  while (rs.next()) {
    String name = rs.getString("Name");
    int id = rs.getInt(1);
    System.out.println(id + " " + name);
  }
}
```

By default, only one ResultSet object per Statement object can be open at the same time.

All execution methods in the Statement interface implicitly close a current ResultSet object of the statement if an open one exists.

Statement

Doc: https://docs.oracle.com/javase/8/docs/api/java/sql/Statement.html

Most important methods:

- executeQuery(sql: String): ResultSet
- executeUpdate(sql: String): int
- addBatch(sql: String): void
- executeBatch(): int[ ]

ResultSet

```java
try (Statement stmt = conn.createStatement()) {
  ResultSet rs = stmt.executeQuery("SELECT * FROM city");
  while (rs.next()) {
    String name = rs.getString("Name");
    int id = rs.getInt(1);
    System.out.println(id + " " + name);
  }
}
```

ResultSet

```
try (Statement stmt = conn.createStatement()) {
  ResultSet rs = stmt.executeQuery("SELECT * FROM city");
  while (rs.next()) {
    String name = rs.getString("Name");
    int id = rs.getInt(1);
    System.out.println(id + " " + name);
  }
}
```

A ResultSet object 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, and because it returns false when there are no more rows in the ResultSet object, it can be used in a while loop to iterate through the result set.

ResultSet

Doc: https://docs.oracle.com/javase/8/docs/api/java/sql/ResultSet.html

Most important methods:

- next(): boolean
- get*(columnIndex: int): get* (columnIndex starts from 1)
- get*(columnLabel: String): get*

* indicates a lot of types (basically all possible SQL types)

Example: filtering based on parameter

```java
private void selectPopulationForCityWrong(
    Connection conn, String city) throws SQLException {
  try (Statement stmt = conn.createStatement()) {
    ResultSet rs = stmt.executeQuery(
      "SELECT Population FROM city WHERE Name = '" + city + "'");

    while (rs.next()) {
      int population = rs.getInt(1);
      System.out.println(population);
    }
  }
}
```

Why is this approach wrong? Why shouldn't we get parameters from the user like this?

SQL Injection

SQL injection is a technique where malicious users can inject SQL commands into an SQL statement, via user input. Injected SQL commands can alter SQL statement and compromise the security of an application.

Example:

```
selectPopulationForCityWrong(conn, "Budapest");
```

SQL Injection

SQL injection is a technique where malicious users can inject SQL commands into an SQL statement, via user input. Injected SQL commands can alter SQL statement and compromise the security of an application.

Example:

```
selectPopulationForCityWrong(conn, "Budapest");

selectPopulationForCityWrong(conn,
  "'; DROP TABLE City; SELECT * FROM Country WHERE Name = '123");
```

SQL Injection

SQL injection is a technique where malicious users can inject SQL commands into an SQL statement, via user input. Injected SQL commands can alter SQL statement and compromise the security of an application.

Example:

```
selectPopulationForCityWrong(conn, "Budapest");

selectPopulationForCityWrong(conn,
  "'; DROP TABLE City; SELECT * FROM Country WHERE Name = '123");
```

```
"SELECT Population FROM city WHERE Name = '" + city + "'"
becomes
"SELECT Population FROM city WHERE Name = '';
DROP TABLE City; SELECT * FROM Country WHERE Name = '123'"
```

Solution: PreparedStatement

```java
private void selectPopulationForCity(
    Connection conn, String city) throws SQLException {
  try (PreparedStatement stmt = conn.prepareStatement(
        "SELECT Population FROM city WHERE Name = ?")) {
    stmt.setString(1, city);

    ResultSet rs = stmt.executeQuery();
    while (rs.next()) {
      int population = rs.getInt(1);
      System.out.println(population);
    }
  }
}
```

Prepared SQL statements are compiled and cached by the DB, later on one just needs to send the parameters to it.

Transactions

We use an RDBMS mainly because we want **transactions** and their **ACID** property in other words.

A (atomiticity): "all or nothing"
C (consistency): valid state $\rightarrow$ valid state
I (isolation): how much concurrency is allowed
D (durability): after commit results remain forever

More about ACID: `https://dev.mysql.com/doc/refman/5.7/en/mysql-acid.html`

Transactions with JDBC

Transactions are either fully executed or rolled back. The programmers control the success of a transaction by calling 'commit()' or 'rollback()'.

Transactions with JDBC

Transactions are either fully executed or rolled back. The programmers control the success of a transaction by calling 'commit()' or 'rollback()'.

By default JDBC sets auto-commit to disabled, meaning we do a commit after each statement execution (so there's no transaction). For turning it off we need to call the 'setAutoCommit' method with 'false'.

Transactions with JDBC

There are 5 Transaction Isolation Levels defined in JDBC:

- TRANSACTION_NONE
- TRANSACTION_READ_UNCOMMITTED
- TRANSACTION_READ_COMMITTED
- TRANSACTION_REPEATABLE_READ
- TRANSACTION_SERIALIZABLE

TRANSACTION_REPEATABLE_READ is usually the default, but the database driver decides it.
More about isolation levels:
https://dev.mysql.com/doc/refman/5.7/en/innodb-transaction-isolation-levels.html

Transaction Example

```
conn.setAutoCommit(false);
conn.setTransactionIsolation(Connection.???);
try (Statement stmt = conn.createStatement()) {
  ResultSet rs = stmt.executeQuery(
    "SELECT Population FROM city WHERE Name = 'Budapest'");
  rs.next();
  System.out.println(rs.getInt(1));

  stmt.execute(
    "UPDATE city SET Population = 500 WHERE Name = 'Budapest'");
  stmt.execute(
    "UPDATE country SET Population = Population + 500 - " +
            population + " WHERE Name = 'Budapest'");

  rs = stmt.executeQuery(
    "SELECT Population FROM city WHERE Name = 'Budapest'");
  rs.next();
  System.out.println(rs.getInt(1));
  conn.commit();
} // maybe in case of some exception we roll back
```

References

- https://stackoverflow.com/questions/23845383/
  what-does-it-mean-when-i-say-prepared-statement-is-pre