howest hogeschool

Object Oriented Architectures and Secure Development

JDBC: Java Database Connectivity

Arne Debou
Mattias De Wael
Frédéric Vlummens

What is JDBC?

- JDBC = Java Database Connectivity
- An API in the Java Standard Edition
- The JDBC API allows you to talk to (relational) databases from within your Java Program
- Native JDBC drivers exist, e.g. for MySQL
- Interested in more background info?
  https://docs.oracle.com/javase/tutorial/jdbc/index.html

In this course: MySQL

- We will be using MySQL
- Can be installed/obtained in multiple ways:
  - WampServer/LAMP/…
  - Standalone installer
  - …
- To download standalone MySQL, go to https://dev.mysql.com/downloads/
- We will be using a GUI administration client (such as SQLYog, MySQL Workbench or DataGrip)
  - SQLYog: https://github.com/webyog/sqlyog-community/wiki/Downloads
  - JetBrains DataGrip: https://www.jetbrains.com/datagrip/
- Make sure you don't forget your root password when installing!

Adding MySQL support to your project

- Go to www.mvnrepository.com.
- Look for **MySQL Connector/J** in the list of available artifacts.
- Make sure to copy-paste the correct version reference (depending on your MySQL version) into your **build.gradle.kts** file:

```
dependencies {
    testImplementation(platform("org.junit:junit-bom:5.9.1"))
    testImplementation("org.junit.jupiter:junit-jupiter")

    // https://mvnrepository.com/artifact/com.mysql/mysql-connector-j
    implementation("com.mysql:mysql-connector-j:8.0.32")
}
```

Opening the database connection

- We obtain a Connection instance in a **try-with-resources-block**.

```
private static final String URL = "jdbc:mysql://localhost/shop";
private static final String USERNAME = "myuser";
private static final String PASSWORD = "mypassword";

private static final Logger LOGGER = Logger.getLogger(ProductsDemo.class.getName());

public static void main(String[] args) {
    try (Connection con = DriverManager.getConnection(URL, USERNAME, PASSWORD)) {
        // use connection 'con'
    } catch (SQLException ex) {
        LOGGER.log(Level.SEVERE, "Unable to connect to database", ex);
        throw new ShopException("Unable to perform operation.");
    }
}
```

- Reason: when block closes, the resources (here: Connection) are automatically released (closed).
- In case of problem, SQLException is thrown, so we need to catch and process it appropriately.

Opening the database connection

- We obtain a Connection instance in a **try-with-resources-block**.

```
try (Connection con = DriverManager.getConnection(URL, USERNAME, PASSWORD)) {
      // use connection 'con'
   } catch (SQLException ex) {
      LOGGER.log(Level.SEVERE, "Unable to connect to database", ex);
      throw new ShopException("Unable to perform operation.");
   }
}
```

- Reason: when block closes, the resources (here: Connection) are automatically released (closed).
- In case of problem, SQLException is thrown, so we need to catch and process it appropriately.

The connection string

- In its most basic form, this is the format of a MySQL connection string:

```
jdbc:mysql://servername/dbname
```

- Some MySQL server instances/driver combinations require additional parameter(s).
- For example:

```
jdbc:mysql://localhost/mydatabase?serverTimezone=UTC
jdbc:mysql://localhost/mydatabase?useSSL=false&serverTimeZone=UTC
```

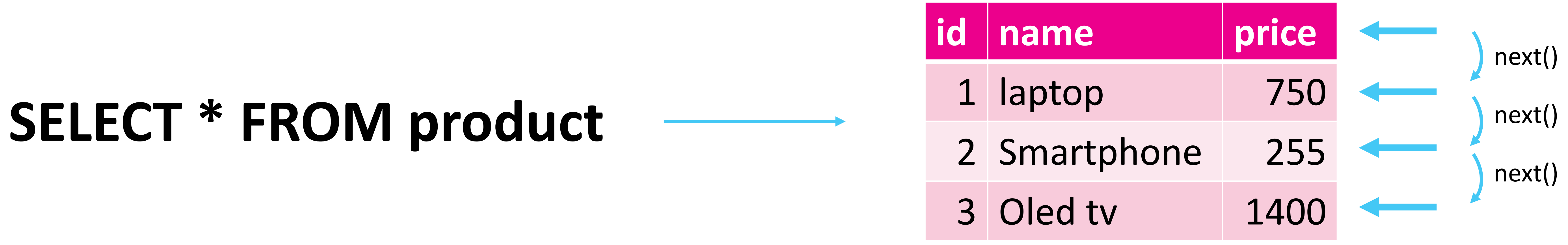
- When in doubt, consult the exception message.

Executing a query

- We distinguish two kinds of queries:
  - SELECT
  - INSERT, UPDATE, DELETE
- When executing a SELECT statement, we get a ResultSet as return value (=the resulting row(s)).
- When executing an INSERT, UPDATE or DELETE statement, we get an int as return value (=the number of affected rows).

Executing a SELECT (1)

- We ask the connection to **prepare** a statement, based on our SQL
  = **PreparedStatement**
- Next, the prepared statement is **executed**, and we obtain a pointer to the resulting rows.
  = **ResultSet**
- The pointer to the resulting rows starts **before the first row** and can be queried using **next()**, returning a boolean (true → points to a row).
- Therefore, you need to execute next() once before getting access to the first row.



SELECT * FROM product

id	name	price
1	laptop	750
2	Smartphone	255
3	Oled tv	1400

Executing a SELECT (2)

```
private static final String SQL_SELECT_ALL_PRODUCTS = "select * from product";

private List<Product> getProducts() {
    try (Connection connection = DriverManager.getConnection(URL, USERNAME, PASSWORD);
         PreparedStatement prep = connection.prepareStatement(SQL_SELECT_ALL_PRODUCTS);
         ResultSet rs = prep.executeQuery()) {

        List<Product> products = new ArrayList<>();

        while (rs.next()) {
            products.add(resultSetToProduct(rs));
        }

        return products;
    } catch (SQLException ex) {
        LOGGER.log(Level.SEVERE, "A database error occured retrieving all products.", ex);
        throw new ShopException(”Unable to retrieve products.");
    }
}
```

Executing a SELECT (3)

```
private Product resultSetToProduct(ResultSet rs) throws SQLException
{
    return new Product(rs.getInt("id"), rs.getString("name"),
            rs.getDouble("price"));
}
```

- Keyword **throws** will allow the SQLException that can be thrown by **rs.getXXX** methods to be caught by the calling method (i.e. getProducts).

Executing a SELECT – with parameters

```
private static final String SQL_SELECT_PRODUCTS = "select * from product where price > ?";

private List<Product> getProducts() {
    try (Connection connection = DriverManager.getConnection(URL, USERNAME, PASSWORD);
         PreparedStatement prep = connection.prepareStatement(SQL_SELECT_PRODUCTS)) {

         prep.setDouble(1, 300);

         try (ResultSet rs = prep.executeQuery()) {
            ...

            return products;
         }
    } catch (SQLException ex) {
       // ...
    }
}
```

Specify parameters in SQL query using question marks ?

Set values using the PreparedStatement's **setXXX** methods

Index-based, starting at 1

Executing an INSERT/UPDATE/DELETE

```
private static final String SQL_ADD_PRODUCT =
                             "insert into product(name, price) values(?, ?)";

private void addProduct() {
    try (Connection connection = DriverManager.getConnection(URL, USERNAME, PASSWORD);
         PreparedStatement prep = connection.prepareStatement(SQL_ADD_PRODUCT)) {
        prep.setString(1, "tablet");
        prep.setDouble(2, 499);

        prep.executeUpdate();
    } catch (SQLException ex) {
        LOGGER.log(Level.SEVERE, "A database error occured adding a product.", ex);
        throw new RuntimeException(”Unable to add product.");
    }
}
```

Security attention points

- Always use prepared statements **with parameters**.
- They help you avoid SQL injection (and in certain situations can also enhance the application’s performance).
- **Never ever** concatenate Strings to build your SQL!

- Do not connect to the database using root credentials.
- Give your database user the lowest privileges necessary for the application to function.

Throwing custom exceptions and logging details

```
} catch (SQLException ex) {
    logger.log(Level.SEVERE, "DB error", ex);
    throw new ShopException("A database error occured.");
}
```

- We **do not** pass confidential information in our own exceptions.
  - Oracle secure coding guideline 2.1.
  - https://www.oracle.com/technetwork/java/seccodeguide-139067.html#2-1
- Using loggers
  - We can log more details about what goes wrong.
  - Take into account Oracle secure coding guideline 2.2.
  - https://www.oracle.com/technetwork/java/seccodeguide-139067.html#2-2

More advanced JDBC topics (1)

- Requesting the AUTOINCREMENT value immediately after an INSERT operation:

```
private void addProduct(Product product) {
    try (Connection connection = DriverManager.getConnection(URL, USERNAME, PASSWORD);
         PreparedStatement prep = connection.prepareStatement(SQL_ADD_PRODUCT,
                 Statement.RETURN_GENERATED_KEYS)) {
        prep.setString(1, product.getName());
        prep.setDouble(2, product.getPrice());

        prep.executeUpdate();

        try (ResultSet autoId = prep.getGeneratedKeys()) {
            autoId.next();
            product.setId(autoId.getInt(1));
        }
    } catch (SQLException ex) {
        LOGGER.log(Level.SEVERE, "A database error occured adding a product.", ex);
        throw new ShopException(”Unable to add product.");
    }
}
```

More advanced JDBC topics (2)

- Working with transactions:

```
connection.setAutoCommit(false);

Savepoint trx = connection.setSavepoint();

try {
    // execute SQL1
    // execute SQL2
    connection.commit();
} catch (SQLException ex) {
    connection.rollback(trx);
} finally {
    connection.setAutoCommit(true);
}
```

- https://docs.oracle.com/javase/tutorial/jdbc/basics/transactions.html