Advanced Programming Methods

Iuliana Bocicor
maria.bocicor@ubbcluj.ro

Babes-Bolyai University

2023

Overview

JDBC

- Connection
- Statements
- ResultSet
- Transactions
- Example

Java 8 Features

- Lambda Expressions
- Functional interfaces
- Method references

JDBC I

- Java Database Connectivity (JDBC) API defines a set of Java interfaces that encapsulate major database functionality:
    - Connection to a database.
    - Execution of queries and update statements to the database.
    - Retrieval and processing of results.
    - Use of configuration information.
- JDBC allows us to write one application that can send SQL statements to different data sources. It is not necessary to write separate applications to access different database systems.

JDBC II

- Main packages:
  - java.sql - contains classes and interfaces which allow accessing and processing of data stored in a database (relational database).
  - javax.sql - supplements java.sql, providing the API for server side data source access and processing. It is an extension of java.sql, its classes being based on the implementations in java.sql.

Establishing a connection

- This can be achieved in two ways:
  - Using the DriverManager class: it needs loading of a driver specific to the database and the connection is created using a URL.

```
jdbc.subprotocol.<database_name>
```

  - Using the DataSource interface: this is newer, more suitable for enterprise/web applications, it allows details about the underlying data source to be transparent to the application.
- Then the connection can be created.

Connection

- The Connection class represents a connection (session) with a specific database.
- Within its context, SQL statements are executed and results are returned.
- Notable methods:
  - *DatabaseMetaData getMetaData()*: retrieves a DatabaseMetaData object that contains metadata about the database.
  - *close(), isClosed(): boolean*
  - *Statement createStatement()*: creates a Statement object for the execution of SQL queries.
  - *PreparedStatement prepareStatement()*: creates a PreparedStatement object for the execution of SQL queries.
  - *rollback()*: drops all changes made since the previous commit/rollback.
  - *commit()*: saves the changes made since the previous commit/rollback.

Statement I

- The class Statement is used to execute static SQL statement and returning the results it produces.
- Notable methods:
  - *ResultSet executeQuery(String sql)*: for SELECT queries.
  - *int executeUpdate(String sql)*: for CREATE, DROP, INSERT, DELETE queries.
  - *boolean execute(String sql)*: for SQL statements that may return multiple results.
  - *int[] executeBatch()*: for batches of commands.

Statement II

- Examples:

```
// select
Statement s = conn.createStatement();
ResultSet rs =
    s.executeQuery("select * from books");
// process the result
rs.close();
s.close();

//delete
String delString =
    "delete from books where title='Open'";
Statement s = conn.createStatement();
s.executeUpdate(delString);
s.close();
```

PreparedStatement I

- As opposed to Statement, the PreparedStatement can be parametrized, allowing the execution of dynamic queries with parameter inputs.
- PreparedStatement is faster than Statement.
- It is helpful in preventing SQL injection attacks, as it automatically escapes the special characters.
- Example of SQL injection:

```
statement = "SELECT * FROM users
             WHERE name = '" + userName + "';"
```

PreparedStatement II

- If **userName** is set to be:

```
' OR '1'='1
```

  then the statement becomes:

```
SELECT * FROM users WHERE name = '' OR '1'='1';
```

- This will show the entire *users* table.
- But it can be even worse, **userName** can be set to:

```
a';DROP TABLE users;
SELECT * FROM userinfo WHERE 't' = 't
```

- Then the statement becomes:

```
SELECT * FROM users WHERE name = 'a';
DROP TABLE users;
SELECT * FROM userinfo WHERE 't' = 't';
```

PreparedStatement III

- Various *set* methods are used to set the values in a PreparedStatement:

```
PreparedStatement statement =
    conn.prepareStatement("INSERT INTO books
                             VALUES (?, ?, ?)");
statement.setString(1, "Andre Agassi");
statement.setString(2, "Open");
statement.setInt(3, 300);
statement.executeUpdate();
statement.close();
```

ResultSet I

- This class contains a table that represents the result of a SELECT instruction.
- It maintains a cursor pointing its current row of data.
- Initially the cursor is positioned before the first row. To move the cursor to the next row, use the method *next()*.
- When there are no more rows, this method returns *false*.
- A ResultSet object is not updatable.

ResultSet II

- Notable methods:
  - *boolean next(), previous(), first(), last()*: moves the cursor;
  - *boolean absolute(int row), relative(int row)*: moves the cursor to the specified row, either aboslute or relative.
  - *int getInt(int columnIndex), int getInt(String columnName)*: returns data from the specified column (by index or by name) as an integer.
  - Similar methods exist for various data types (*getLong, getDouble, getDate, getTime*, etc.).

Transactions I

- A transaction is a unit of work (a set of one or more instructions) that has to be executed as a whole. Either all instructions are executed or none of them is.
- E.g. Transferring money.
- If an instruction fails within the transaction, the transaction's entire effect should be reverted, as if it never happened.
- The default behaviour with regard to transactions can be modified from the Connection object, method *setAutoCommit()*.

Transactions II

- By default each individual SQL statement is treated as a transaction and is automatically committed right after it is executed.
- To allow grouping two or more statements in a transaction the auto-commit mode should be disabled. Then no SQL statements are committed until you call the Connection object's method *commit()* explicitly.
- The Connection object's method *rollback()* aborts a transaction and restores values to what they were before the attempted update.

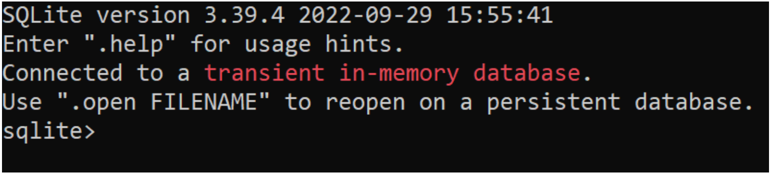
Example I

To be able to run this example, make sure to:

1. Download SQLite (sqlite-tools-win32-x86-3390400.zip) - the bundle of command-line tools for managing SQLite database files: link.
2. Unzip the folder in a path of your choice (e.g. "C:\sqlite").
3. Add the previous path to the **Path** system variable.

Example II

4. To test whether it was correctly installed: open a *Command.com* and execute: "sqlite3". You should be seeing something similar to the following:



```
SQLite version 3.39.4 2022-09-29 15:55:41
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite>
```

5. **Observation**: Steps 1-4 above are not necessary if you use IntelliJ IDEA Ultimate with the *Database Tools and SQL plugin* enabled (see here).

Example III

6. Download the sqlite driver. The latest version at the time of writing this is: 3.43.2.2. If you use this version, you alse need to download the slf4j jar file (Simple Logging Facade).
7. Copy them in a folder of your choosing (even in your current Java project folder).
8. Add both to classpath: in IntelliJ → File → Project Structure → Libraries → click the plus sign → Java → select the .jar file.

Example

jdbc.JDBC.

Java 8

- Java 8 was a revolutionary release, including huge upgrades to the Java programming model and a coordinated evolution of the JVM, Java language, and libraries.
- New features in Java 8:
  - Lambda expressions.
  - Pipelines and streams.
  - Date and time API.
  - Type annotations.
  - Default methods.
  - Parallel operations.
  - ... and others.

Lambda Expressions

- Java's first step into functional programming.
- A lambda expression is a function which can be created without belonging to any class and without being connected to an identifier.
- It can be passed around as if it were an object and executed on demand.
- The amount of code is reduced.

```
(<lambda_parameters>) -> lambda_body
```

Functional Interfaces

- A functional interface is an interface with just one abstract method.
- It can be annotated with the *@FunctionalInterface* annotation. However, this is not compulsory, it is more to avoid accidental addition of more abstract methods.
- Benefit: we can use lambda expressions to instantiate them. A functional interface can be implemented with a lambda expression.

```
@FunctionalInterface
public interface InterfaceName
{
   public Type function(<params>);
}
```

Build-in Functional Interfaces

- Java contains a set of functional interfaces designed for common use cases.
- All of them are found in package *java.util.function*.
- Examples:
  - Function
  - Predicate
  - UnaryOperator
  - BinaryOperator
  - Supplier
  - Consumer

Function

- The Function interface represents a function that takes a single parameter and returns a single value.

```
public interface Function<T,R> {
    public <R> apply(T parameter);
}
```

- The *apply* method needs to be implemented.
- The interface contains more methods that are default or static, so there is no need to implement them.

Predicates I

- The Predicate interface represents a function that takes a single parameter and returns true or false.

```
public interface Predicate<T> {
    boolean test(T t);
}
```

- The *test* method needs to be implemented.
- The interface contains more methods that are default or static, so there is no need to implement them.

Predicates II

- Notable methods:
  - *and(Predicate<? super T> other))*: returns a composed predicate that represents a short-circuiting logical AND of this predicate and another.
  - *or(Predicate<? super T> other)*: returns a composed predicate that represents a short-circuiting logical OR of this predicate and another.
  - *negate()*: returns a predicate that represents the logical negation of this predicate.

UnaryOperator and BinaryOperator

- The UnaryOperator interface represents an operation which takes a single parameter and returns a parameter of the same type.
- It can be used to represent an operation that takes a specific object as parameter, modifies that object, and returns it after the modification.
- The BinaryOperator interface is a functional interface that represents an operation which takes two parameters and returns a single value. Both parameters and the return type must be of the same type.

Supplier

- The Supplier interface represents a function that supplies some values.
- It can be regarded as a factory interface.

```
@FunctionalInterface
public interface Supplier<T> {
        T get();
}
```

Consumer

- The Consumer interface represents a function that consumes a value without returning any value.
- A consumer implementation could be printing out a value, or writing it to a file.

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
```

Method references

- Method references are a special type of lambda expressions.
- Types of method references:
  - Static methods.
  - Instance methods of particular objects.
  - Instance methods of an arbitrary object of a particular type.
  - Constructor.

Example

java8features.Examples

Other methods on collections

- forEach - used to perform the certain operation for each element in the collection.

```
String[] stringArray = { "Barbara", "James", "Mary"};
List<String> names = Arrays.asList(stringArray);
names.forEach(System.out::println);
```

- removeIf - remove all of the elements of the collection that satisfies a given predicate filter which is passed as a parameter to the method.

```
names.removeIf(x -> x.endsWith("a"));
```

Summary

- With JDBC API we can:
  - Connect to a database.
  - Execute queries and update statements.
  - Retrieve and process the results.
- Important updates in Java 8:
  - Lambda expressions.
  - Functional interfaces.
  - Method references.
- *Next week:*
  - Java 8 streams.
  - Graphical user interfaces.