

DB Apps Introduction

Connecting via JDBC, Executing Statements, SQL Injection, Advanced Concepts



SoftUni Team

Technical Trainers



Software University



SoftUni Foundation



Software University

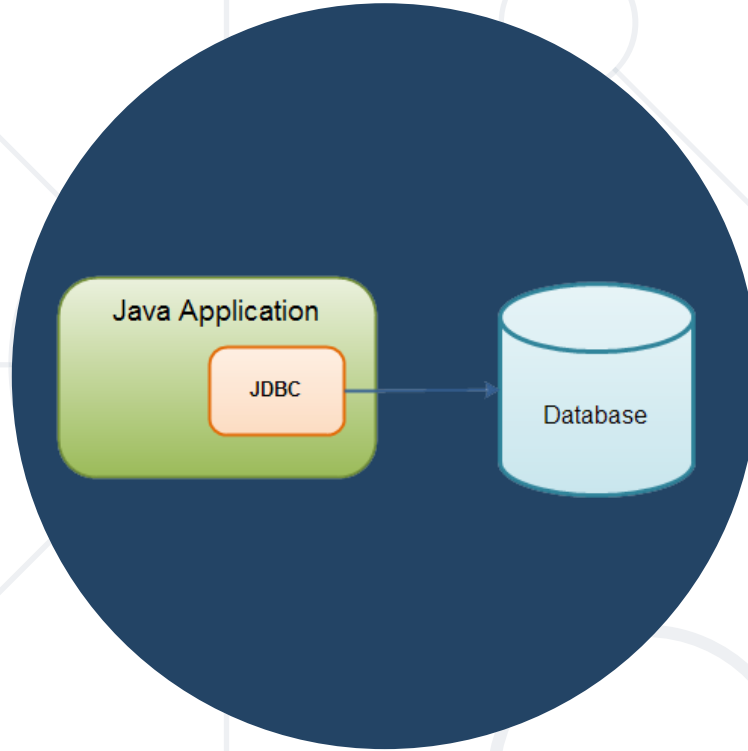
<http://softuni.bg>

Table of Content

1. Application to Database Connection.
2. Application to Database Connection Demo.
3. Java Database Connection.
4. JDBC Statements.
5. SQL Injection.
6. Advanced Concepts.



sli.do
#Java-DB



Application to Database Connection

Accessing data via client application

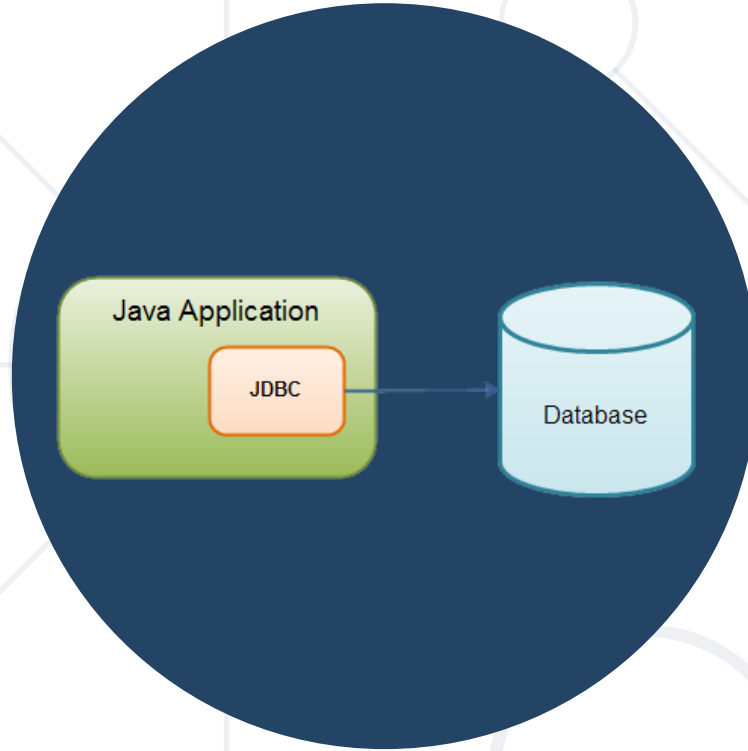
- In development programmers use **object relational mapping** frameworks.
 - Mapping Java classes and data types to **DB tables** and **SQL data types**
 - Generate SQL calls and **relieves** the developer from the **manual handling**
 - E.g. (pseudo-code)

```
User user = new User("Peter", 25);  
dbManager.saveToDB(user);
```

SQL Encapsulated
in method

- ORM frameworks **do not** drop the need to write SQL!
 - At some point you might need some **manual query optimization**
- ORM Frameworks **examples**:
 - Java – **Hibernate**, EclipseLink, TopLink...
 - .NET – Entity Framework, NHibernate...
 - PHP – Doctrine, Laravel(Eloquent)...





Application to Database Connection Demo

Connection to DB via Java app Demo

- Download the demo from the [course instance](#).
- You are given a simple application that:
 - Establishes connection with the "soft_uni" DB
 - Executes simple MySQL statement to retrieve the employees names by **given salary criteria**

- Lets analyze the program:
 - Connection to DB is established by asking the user to give credentials:

```
System.out.print("Enter username default (root): ");
```

```
String user = sc.nextLine();
```

```
user = user.equals("") ? "root" : user;
```

```
...
```

```
System.out.print("Enter password default (empty):");
```

```
String password = sc.nextLine().trim();
```

```
...
```

- Using an external library (**MySQL Connector/J**) we make a connection via a **DriverManager** and a **Connection** class.

```
Properties props = new Properties();  
    props.setProperty("user", user);  
    props.setProperty("password", password);  
  
Connection connection =  
    DriverManager.getConnection("jdbc:mysql://localhost:3306/s  
oft_uni", props);
```

Connection to DB via Java App Demo (2)

- We retrieve the result with the **ResultSet** and the **PreparedStatement** classes.

```
PreparedStatement stmt = connection.prepareStatement  
("SELECT * FROM employees WHERE salary > ?");
```

SQL Query

```
String salary = sc.nextLine();
```

Salary criteria by user
input

```
stmt.setDouble(1, Double.parseDouble(salary));
```

```
ResultSet rs = stmt.executeQuery();
```

Runs the SQL statement and
returns retrieved result

- Iterating over the result:

Retrieved data

```
while(rs.next()) {  
    System.out.printf("%s  %s",  
        rs.getString("first_name"),  
        rs.getString("last_name"));  
}
```

The ResultSet is a set of table
rows

Demo Conclusion

- We can access databases on a programmer level.
 - No manual actions needed
- In a bigger applications we can:
 - Encapsulate custom SQL logic in methods
 - Achieve database abstraction

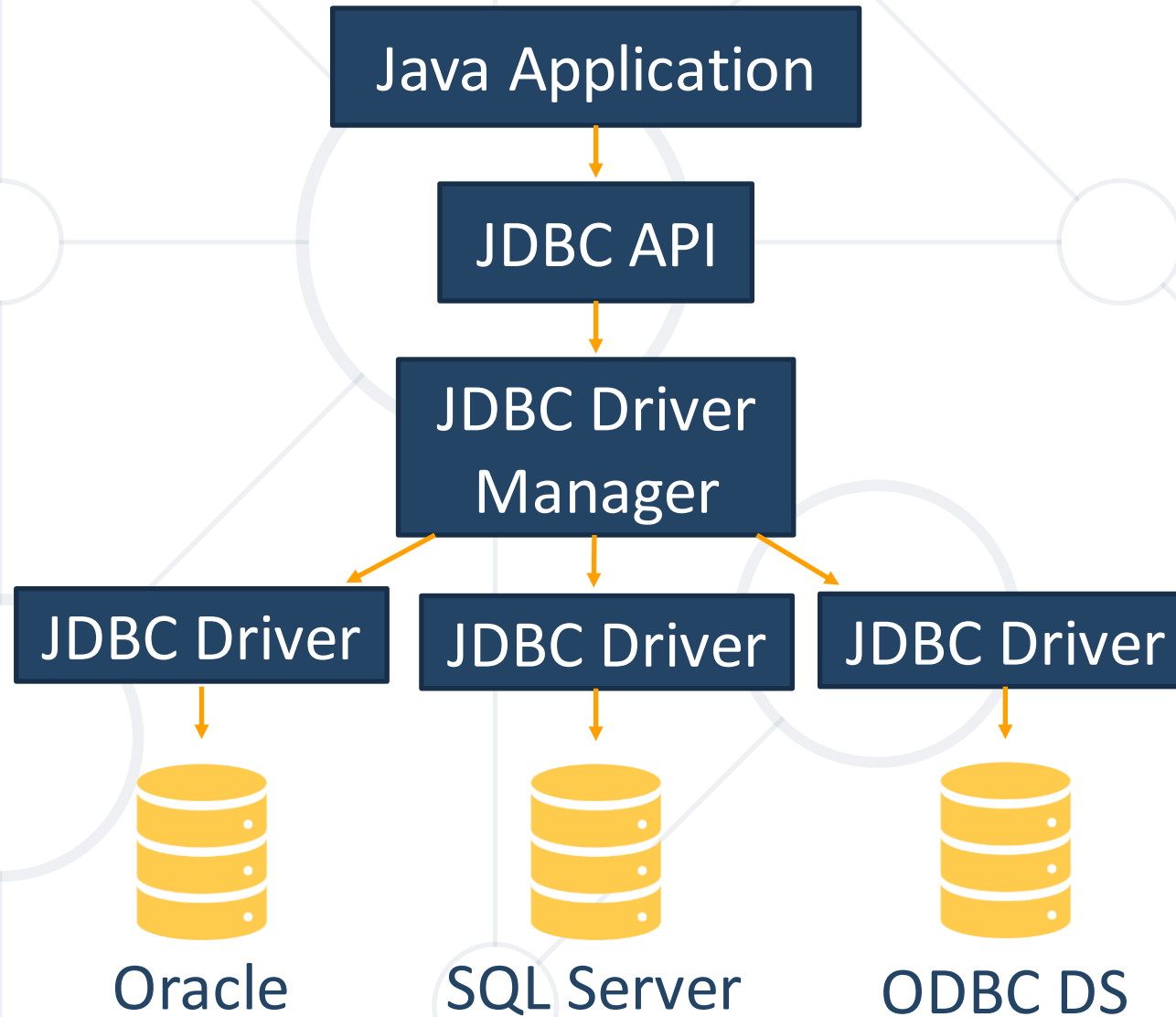




Java Database Connection

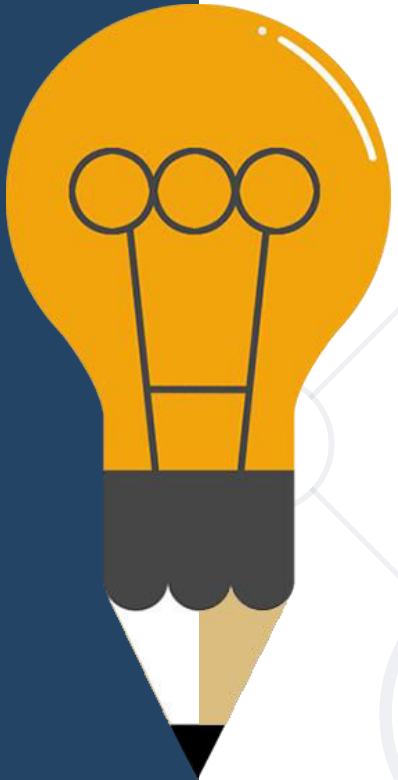
Client access to a database

- JDBC is a standard Java API for database-independent connectivity
- Includes APIs for:
 - Making a connection to a database
 - Creating and executing **SQL** queries in the database
 - Viewing & Modifying the resulting records



- JDBC **API** – provides the connection between the application and the driver manager
- JDBC **Driver Manager** – establishes the connection with the correct driver
 - Supports multiple drivers connected to different types of databases
- JDBC **Driver** - handles the communications with the database

- JDBC API provides several interfaces and classes:
 - **DriverManager** – matches requests from the application with the proper DB driver
 - **Driver** – handles the communication with the DB server
 - **Connection** – all methods for contacting a database
 - **Statement** – methods and properties that enable you to send SQL
 - **ResultSet** – retrieved data (set of table rows)
 - **SQLException**



- ResultSet maintains a **cursor** pointing to its **current row of data**
 - Not updatable
 - Iterable only once and only from the first row to the last row
- Provides getter methods for retrieving column values from the current row
 - E.g. from previous demo:

```
while(rs.next()) {  
    System.out.printf("%s %s", rs.getString("first_name"),  
rs.getString("last_name"));}
```

Getter method

Column name

- Retrieved information is reached by getter methods:
 - E.g.:
 - `getString('column_name')`
 - `getDouble('column_name')`
 - `getBoolean('column_name')` etc.
- The driver converts the underlying data to the Java type

- The java.sql package provides all previously mentioned JDBC classes
- In order to work with JDBC we need to download a MySQL Driver – Connector/J
 - It can be found on the following webpage:

<https://dev.mysql.com/downloads/connector/j/>

- Connection with the database is established via **connection string**
 - `jdbc:<driver protocol>:<connection details>`
 - E.g. connection from previous demo:

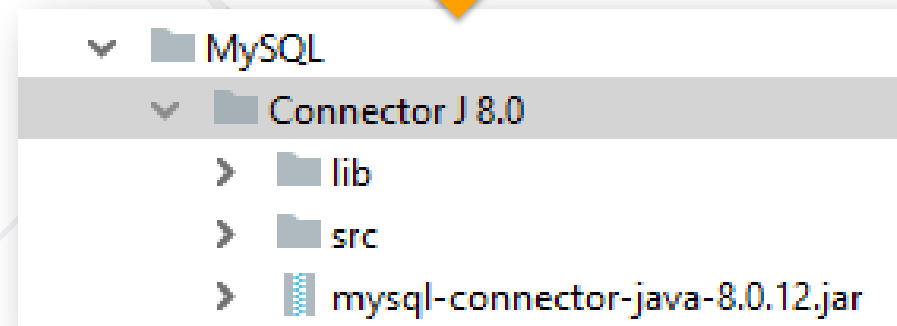
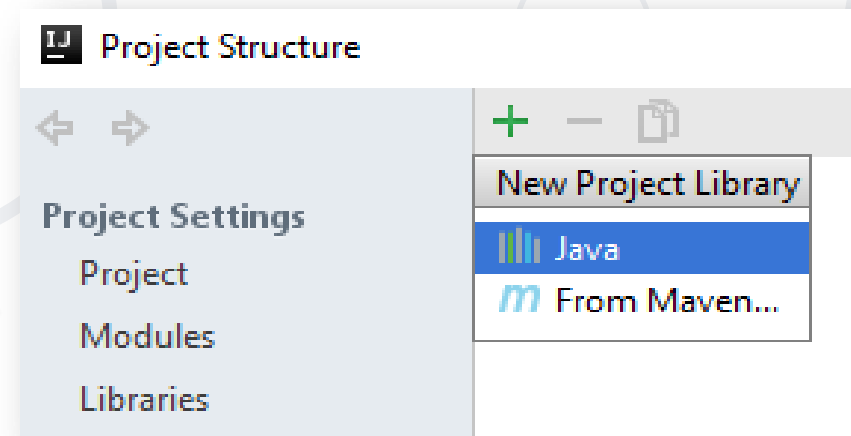
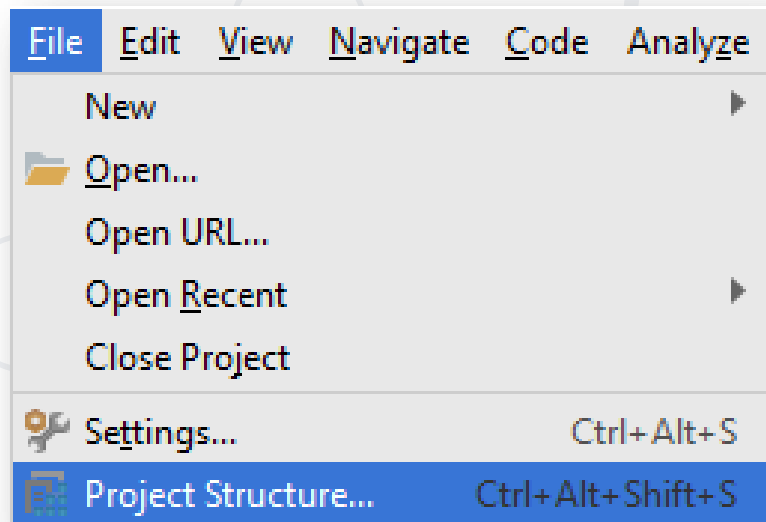
```
Connection c = DriverManager.getConnection(  
"jdbc:mysql://localhost:3306/soft_uni", props);
```

Database name

Credentials

Setting up the Driver in IntelliJ IDEA

- Add the driver as an external library:
 - "File" -> "Project Structure" -> "Libraries"





JDBC Statements

Statement, PreparedStatement, CallableStatement

- The JDBC **Statement interface** defines the methods and properties that enable you to send SQL commands to the database.

Interfaces	Recommended use
Statement	For general-purpose access to your database and static SQL statements at runtime. Cannot accept parameters.
PreparedStatement	For SQL statements used many times. Accepts parameters.
CallableStatement	Used for stored procedures. Accepts parameters.

Statements Example

- Example(PreparedStatement) from previous demo:

```
PreparedStatement stmt =  
connection.prepareStatement("SELECT * FROM employees WHERE  
salary > ?");
```

SQL Query

Statements are created via
the connection

Query parameter

```
String salary = sc.nextLine();  
stmt.setDouble(1, Double.parseDouble(salary));
```

Parameter Index

Parameter value

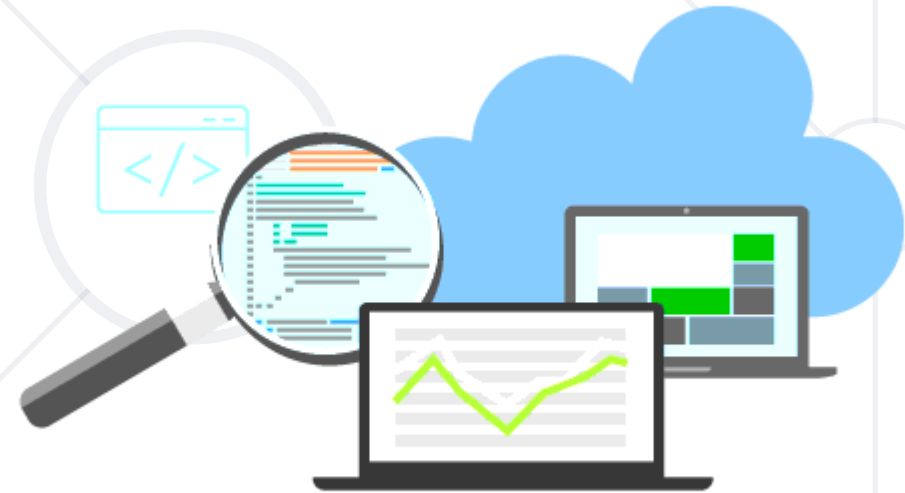


SQL Injection

How to prevent it?

What is SQL Injection?

- Placement of **malicious** code in SQL Statements
 - Usually done via user input
- To protect our data we can place parameters in our statements
 - We can do it by using **PreparedStatement**



SQL Injection Example: Login form input by user

- Ask the user to input username and password in fields
 - If we don't secure our statements, we risk SQL Queries to be written as an input
 - E.g. :
 - username: 'example_user'
 - password: '12345'
 - The following query will be built and executed to the data source:

```
SELECT id FROM users
WHERE username = 'example_user' AND password = '12345';
```

- In result the **id of the user** will be returned.
 - User will be authenticated to do actions in the application
- Without validating and securing our statements information might get exposed:
 - Value for password: "1' OR username = 'admin';"
 - The following query will be executed:

```
SELECT id FROM users  
WHERE username = 'pesho'  
AND password = '1' OR username = 'admin';
```

- In result the id **an admin** will be returned
 - Will permit actions to the user that can harm our application and database
- We can validate the input by setting rules
 - Length, special characters, digits etc.
 - Set up validation in our code in different layers (front-end, back-end etc.)



Advanced Concepts

Transactions and DAO Pattern

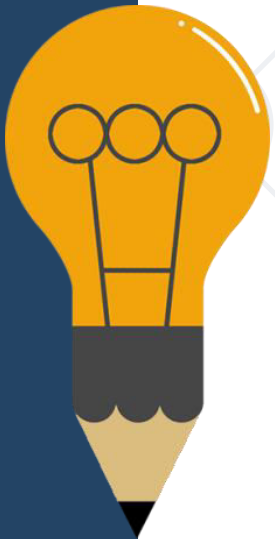
- Every JDBC Connection is set to **auto-commit** by default
 - SQL statements are committed on completion
- In bigger applications we want greater control
 - If and when changes are applied to the database
- Turn off auto-commit:

```
connection.setAutoCommit(false);
```

JDBC Transaction Pattern (2)

- Example (**pseudo code**):

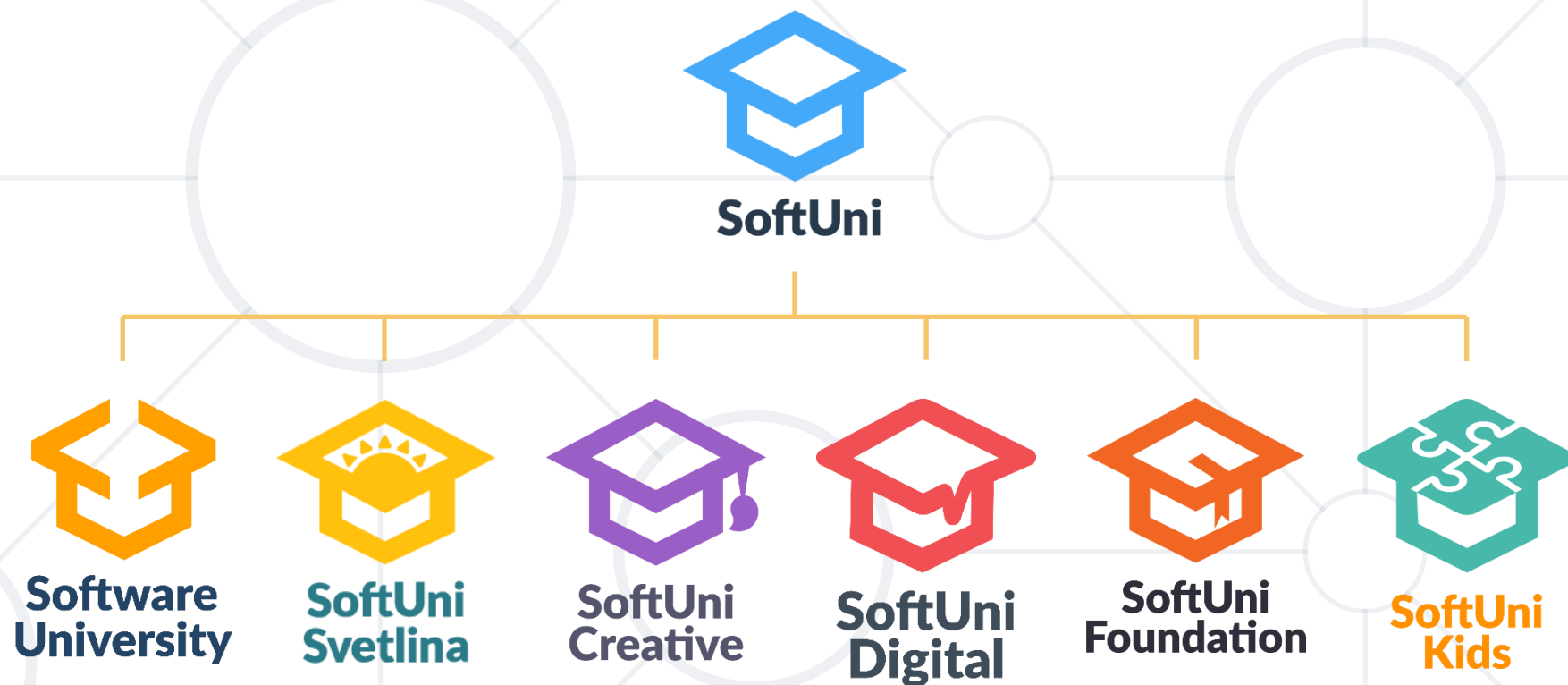
```
try {  
    connection.setAutoCommit(false);  
    Statement stmt = conn.createStatement();  
    String sql = "...";  
    stmt.executeUpdate(sql);  
    // If there is no error  
    connection.commit();  
} catch(SQLException se){  
    // If there is any error  
    conn.rollback();  
}
```



- ORM Frameworks map **Java objects** to **SQL entities**
- JDBC provides us **classes** for operating with a database
- SQL Injection can seriously harm our data source or expose it
 - Our application should secure the statements being sent



Questions?



SoftUni Diamond Partners



XSsoftware



SBTech



telenor



SoftwareGroup
doing it right

NETPEAK



SmartIT



Postbank

Решения за твоето утре

**SUPER
HOSTING
.BG**

INDEAVR

Serving the high achievers



INFRAGISTICS®



STEMO®
Computer Systems & Software



SoftUni Organizational Partners



OneBit
SOFTWARE



WORLD
OF
MYTHS

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education and Employment Opportunities
 - softuni.bg
- Software University Foundation
 - <http://softuni.foundation/>
- Software University @ Facebook
 - facebook.com/SoftwareUniversity
- Software University Forums
 - forum.softuni.bg



- This course (slides, examples, demos, videos, homework, etc.) is licensed under the "Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International" license

