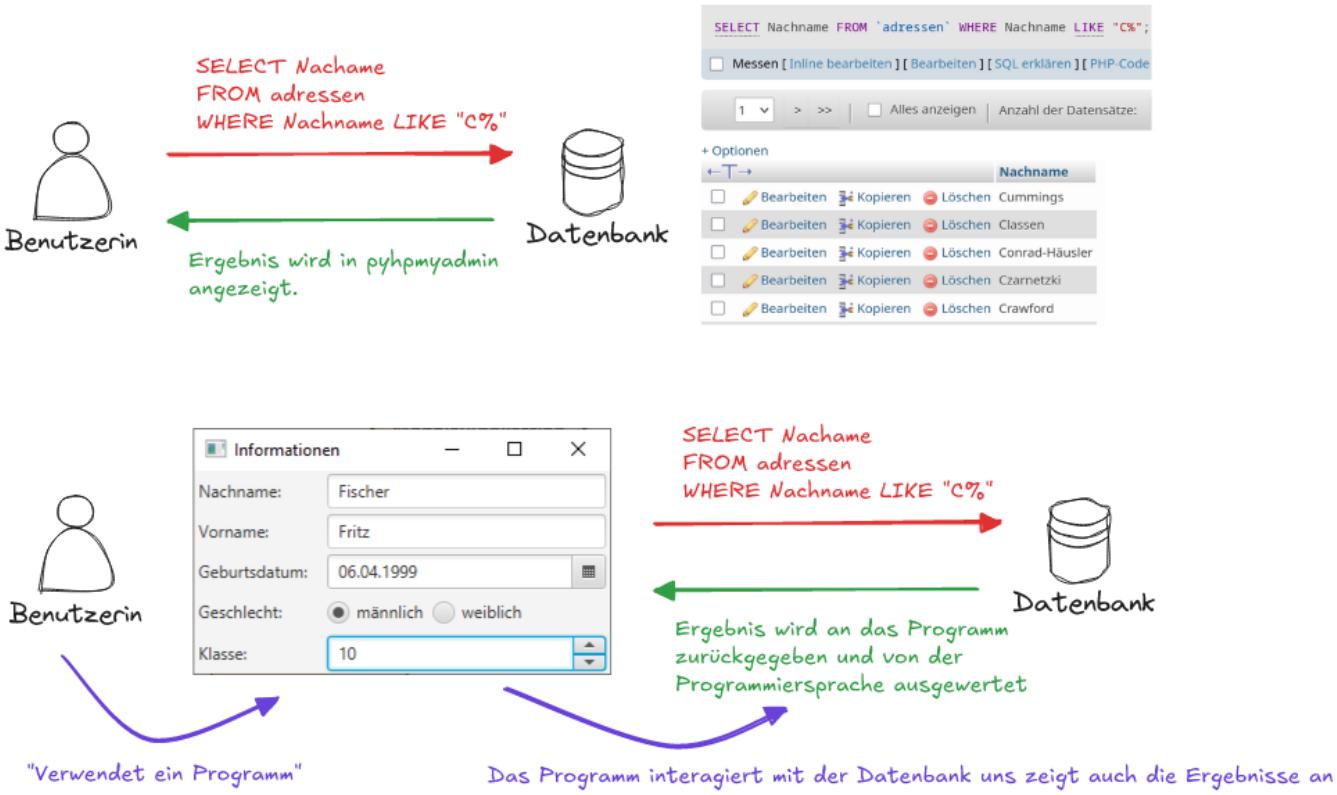
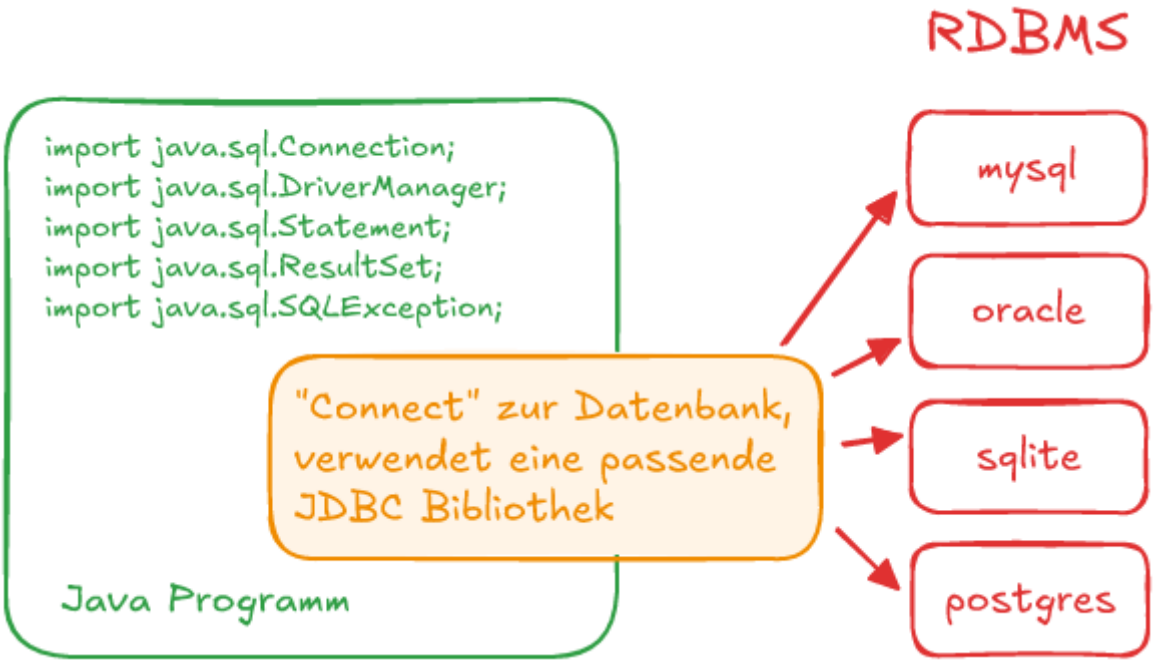
Datenbankzugriff mit Java

Bislang haben wir Datenbanken aus einer sehr technischen Sicht betrachtet: Mit phpmyadmin konnten wir SQL-Befehle ausführen, um MySQL/MariaDB Datenbanken abzufragen, Tabellen zu erstellen oder Daten zu verändern.

Dazu war die Kenntnis von SQL unerlässlich. Die meisten Anwendungen, die auf Datenbanken operieren sind jedoch für Endanwender ohne detaillierte Kenntnisse von SQL gedacht - diese sollen sich um SQL-Statements keine Gedanken machen, sondern in einer verständlichen Benutzeroberfläche mit der Datenbank interagieren.



Viele Programmiersprachen bieten **Bibliotheken** an, mit Hilfe derer man auf Datenbanken zugreifen kann. Damit wird das Programm von der Datenbank abstrahiert - es spielt keine Rolle, welches Datenbankmanagementsystem zum Einsatz kommt, man muss lediglich mit Hilfe der passenden Bibliothek eine Vebindung zum DBMS herstellen und kann dann aus dem Programm heraus Abfragen auf der Datenbank ausführen.



Java verwendet hier **JDBC ("Java Database Connectivity")**, für JDBC gibt es Treiber für alle gängigen DMBS. Wir verwenden zunächst SQlite als DMBS, da man dafür keinen gesonderten Datenbankserver benötigt - die Datenbanken liegen bei SQLite einfach als Dateien vor.

- Beispiel 1: Schülerliste

Erstes Beispiel

Im ersten Beispiel werden wir ohne grafische Benutzerschnittstelle (GUI) auf eine SQLite Datenbank zugreifen, um uns zunächst auf die Datenbankfunktionalität zu konzentrieren.

Du findest das BlueJ-Projekt auf Codeberg. Klone das Projekt mit git - du benötigst später auch die anderen Branches. `git clone https://codeberg.org/info-bw-wiki/bluej-db-adressen`. Öffne es mit BlueJ.

Das Beispiel bringt eine SQLite Datenbank `schueler.db` mit.

(A1)

Untersuche die Datenbank, indem du die Datei mit dem DB Browser for SQLite öffnest. Notiere für die Schülerdatenbank die Namen der Tabelle(n) sowie Namen und Typen der Attribute.

Lösung

- Die Schülerdatenbank hat nur eine Tabelle, diese heißt `schueler`.
- Als Spalten/Attribute finden sich `SVorname`, `SNachname`, `SGeschlecht`, `SEmail` (alle vom Typ `varchar`), `SGeburtsdatum` (Typ `date`) und `SKlasse` (Typ `integer`).

Bibliotheken einbinden

Die Bibliotheken für den Datenbankzugriff befinden sich im Unterverzeichnis `+libs` des BlueJ Projekts. Es sind die Bibliotheken für mysql und sqlite im Projekt dabei: `mysql-connector-java-8.0.29.jar` und `sqlite-jdbc-3.49.1.0.jar`.

Der Ordner `+libs' ist bei PlueJ Projekten automatisch im Suchpfad für Bibliotheksfunktionen - um auf SQL Datenbanken zuzugreifen kann man am Beginn der Java-Klassendatei die nötigen Bibliotheken importieren und diese werden dann gefunden:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.SQLException;
```

Das sieht so ähnlich aus, wie wenn wir Array-Lists verwenden wollen, nur dass die Bibliotheksfunktionen für SQL eben nicht (wie die für die ArrayList) mit Java "mitgeliefert" werden, sondern gesondert in `+libs` zur Verfügung gestellt werden müssen.

Der gesamte Code sieht so aus:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class Adressen
{
    Connection DBConnection;
    public Adressen()
    {
        System.out.print('\u000C');
        String uri = "jdbc:sqlite:schueler.db";
        connect(uri);

        try {
            Statement statement = DBConnection.createStatement();
            ResultSet result = statement.executeQuery("SELECT * FROM
schueler");
            while(result.next()) {
                String nachname = result.getString("SNachname");
                String vorname = result.getString("SVorname");
                System.out.print(String.format("%-12s", nachname) + "\t");
                System.out.print(String.format("%-12s", vorname) + "\t");
```

```
                System.out.println();
            }
        } catch (SQLException e) {
            System.out.println(e.getMessage());

        }

    }

    public void connect(String DBUri) {
        try {
            DBConnection = DriverManager.getConnection(DBUri);
            System.out.println("Verbindung mit " + DBUri + " erfolgreich
hergestellt");

        } catch (SQLException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Fehlerbehandlung

Alle Datenbankoperationen müssen in `try-catch`-Blöcken ausgeführt werden. Das ist ein Mechanismus zur **Fehlerbehandlung**, im Code kann man gut erkennen, was dabei passiert. Zunächst wird im `try`-Block die gewünschte Operation programmiert. Wenn diese fehlerfrei durchgeführt wird, wird der `catch`-Block übersprungen.

Tritt jedoch bei den Operationen im `try`-Block ein Fehler auf, etwa weil das SQL-Statement fehlerhaft ist, eine Datenbank oder Tabelle nicht existiert o.ä. wird der `catch`-Block ausgeführt. Im einfachsten Fall, so wie hier im Beispiel, wird einfach der Fehler ausgegeben.

(A2)

Compiliere das Programm und erzeuge eine Instanz der Klasse `SchuelerListe`. Vollziehe nach, was auf der Java Konsole ausgegeben wird. Informiere dich über die `String.format()` Methoden und mache dir klar, was sie in unserem Beispiel machen, einen Einstieg findest du hier.

Baue Fehler ein und teste, wie sich das Programm verhält:

- Ändere den Dateiverweis `jdbc:sqlite:schueler.db`, der auf die Datenbank `schuler.db` zeigt so ab, dass die angegebene Datei nicht existiert - an welcher Stelle des Ablaufs tritt ein Fehler auf?

- Baue Fehler in das SQL-Stzatement ein, das die Datensätze ermittelt.
- Probiere weitere Fehler aus.

> Hier erkennt man, warum die Fehlerbehandlung wichtig ist: Der Java Compiler weiß nichts über deine Datenbanken, er kann beim Übersetzen des Programmcodes also auch in keiner Weise beurteilen, ob das Programm im Zusammenspiel mit der angesprochenen Datenbank sinnvolle Ergebnisse liefert - das muss alles die Programmiererin sinnvoll gestalten!

Datenbankzugriff Schritt für Schritt

Der Zugriff auf die Datenbank erfolgt in mehreren Schritten:

- Datenbankverbindung herstellen
- Auf dieser Verbindung ein `Statement` oder ein `Prepared Statement` instanziieren
- Mit Hilfe des Statements kann eine SQL Abfrage an die Datenbank gesendet werden
- Das Ergebnis ist vom Typ `ResultSet` und kann mit Hilfe eines Iterators verarbeitet werden.

Am Beispiel:

- Die Klasse hat ein Attribut `DBConnection`. In der NMethode `connect(String DBUri)` wird versucht, eine Vebindung zu der in der Variablen `DBUri` übergebenen Datenbank herzustellen. Wenn das gelingt, ist der erste Schritt abgeschlossen, über `DBConnection` besteht jetzt eine Verbindung zur Datenbank.
- Unter Verwendung dieser Verbindung kann jetzt ein Objekt des Typs `Statement` erzeugt werden:

```
Statement statement = DBConnection.createStatement();
```

- Auf diesem `Statement`-Object kann jetzt eine Abfrage ausgeführt werden, wobei das Ergebnis vom Typ `ResultSet` ist:

```
ResultSet result = statement.executeQuery("SELECT * FROM schueler");
```

- Über die Elemente des ResultSets kann mit einem Iterator iteriert werden, das entspricht in etwa einem "foreach" für die Zeilen des Abfrageergebnisses:

```
while(result.next()) {

}
```

- Der Zugriff auf die Spalten eines Ergebnis-Datensatzes kann entweder über den Feldnamen oder die laufende Nummer[1)] erfolgen:

```
String nachname = result.getString("SNachname");
Integer klasse = result.getInt("SKlasse");
```

> Hier muss die Programiererin wieder genau aufpassen und das Programm passend zur Datenbank programmieren: Die abgefragten Datenbankfelder haben innerhalb der Datenbank einen Daten-Typ. z.B. `varchar(60)`, also eine Zeichenkette mit bis zu 60 Zeichen. Wenn man diese Felder in eine Java-Variable ausliest, muss die Java-Variable die gelesenen Werte auch speichern können, im Falle des `varchar`-Attributs sollte also ein String verwendet werden und zum Auslesen die Methode `result.getString(…)`
>
> Als Variablentypen sollten auch bei den primitiven Datentypen wie `int`, `char` die Wrapper-Klassen verwendet werden - also `Integer` oder `Character`.

Einen Überblick über die Methoden findest du in der Java-Dokumentation:
https://docs.oracle.com/en/java/javase/11/docs/api/java.sql/java/sql/ResultSet.html

---

(A3)

Ergänze das Programm so, dass außer Nach- und Vorname der Schülerinnen auch die Klasse abgefragt und ausgegeben wird. Verwende für die Abfrage der Klasse als Java-Variablentyp `Integer` und als Methode `result.getInt(…)`. Gib zwei weitere Spalten aus, einmal die Klasse in diesem Schuljahr und um eins erhöht die Klasse des kommenden Schuljahrs.

Lösungsvorschlag

```
[....]
while(result.next()) {
                String nachname = result.getString("SNachname");
                String vorname = result.getString("SVorname");
                Integer klasse = result.getInt("SKlasse");
                System.out.print(String.format("%-12s", nachname) + "\t");
                System.out.print(String.format("%-12s", vorname) + "\t");
                System.out.print(klasse + "\t");
                System.out.print(klasse + 1 + "\t");
                System.out.println();
            }
[....]
```

(A4)

Ändere den Code aus Aufgabe 3 so ab, dass du die Klasse in eine Java-Variable des Typs `String` ausliest und dazu die Methode `result-getString(…)` verwendest, lass den restlichen Code unverändert. Was fällt dir auf?

Lösungshinweis

Da das Feld jetzt als String ausgelesen wird, verkettet Java das auch als String, es wird nicht mehr korekt addiert. Wenn man nicht auf die korrekte Passung der Datentypen achtet, können seltsame Dinge geschehen.

(A5)

Was passiert, wenn du versuchst, den Vor oder Nachnamen in eine Java-Variable des Typs `Integer` auszulesen?

Lösungshinweis

1)
Startet bei 1…