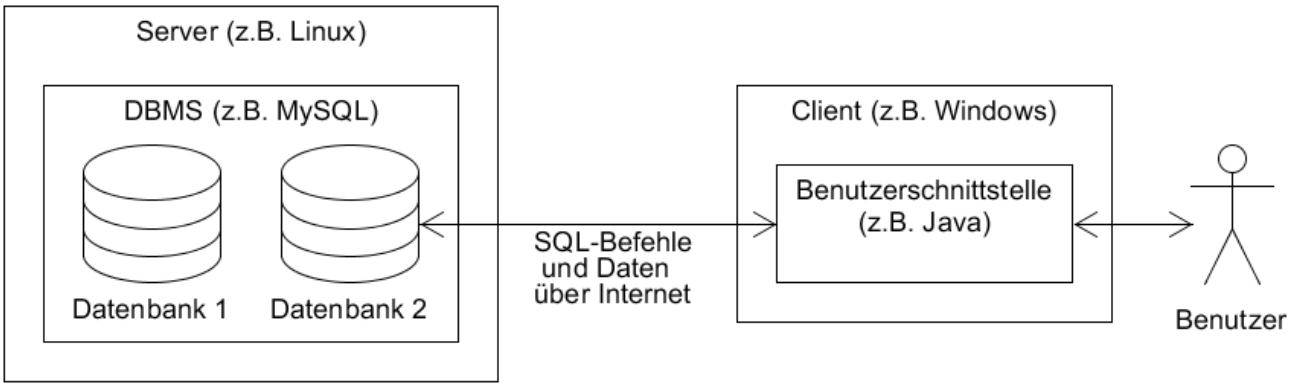
Eine Datenbank im Programm nutzen

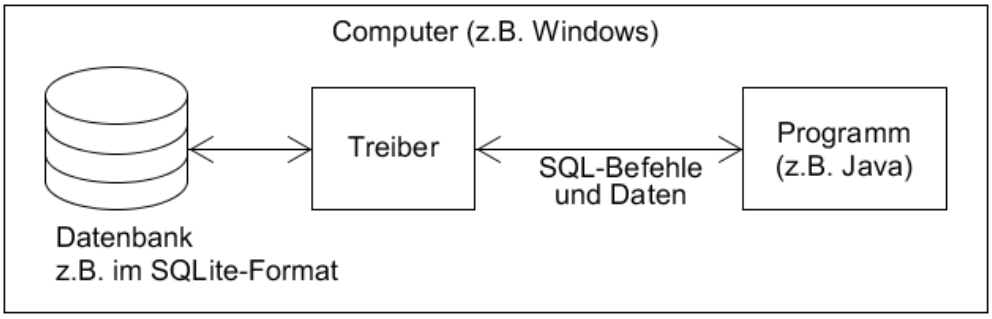
Benutzer, die mit Datenbanken arbeiten, tun das normalerweise nicht direkt über SQL – das wäre viel zu mühselig, und für einfache Anwender auch kaum möglich – SQL beherrschen in der Regel nur Datenbankentwickler. Stattdessen wird für Anwender eine Benutzerschnittstelle programmiert. Diese hat die Aufgabe, vom Benutzer gewünschte Informationen von der Datenbank zu holen und darzustellen. Der Benutzer wählt z.B. die Funktion „Gewinne des Monats“. Die Software erstellt die entsprechenden SQL-Abfragen, sendet sie an die Datenbank, erhält als Antwort die Ergebnisse, und stellt dieses dann in einer verständlichen Weise, z.B. als Grafik, dar.

Datenbanken werden in vielen Fällen von einem Datenbank-Management-System (DBMS), wie z.B. MySQL, verwaltet. Ein DBMS verwaltet eine oder mehrere Datenbanken. Eine Benutzerschnittstelle, die z.B. in Java programmiert ist, tauscht dann SQL-Abfragen und Daten mit dem DBMS aus. Die Benutzerschnittstelle und das DBMS müssen dabei nicht auf dem gleichen Computer laufen, die Verbindung kann auch über das Internet hergestellt werden. So können viele „Clients“ (also Computer für Benutzer) auf der ganzen Welt auf die gleiche Datenbank zugreifen.



Datenbankdateien

Datenbanken werden aber nicht immer von vielen Clients genutzt. Für viele Programme ist es sinnvoll, ihre Daten in einer Datenbank abzulegen, die nur von diesem Programm genutzt wird und auf demselben Computer liegt wie das Programm. Dafür braucht es dann kein aufwendiges DBMS. Die Datenbank liegt in einer Datei vor, z.B. im MS Access- oder SQLite-Format. Das Programm greift mithilfe eines „Treibers“ (also einer Programmbibliothek) auf die Datei zu.



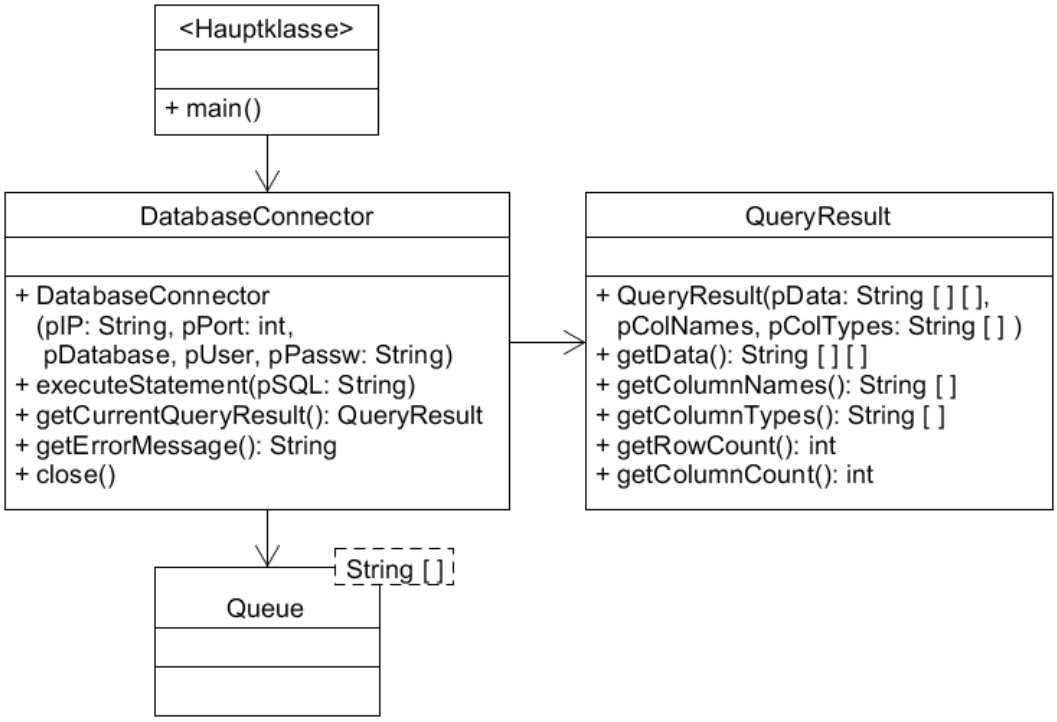
DatabaseConnector und QueryResult

Das Schulministerium NRW stellt für den Zugriff auf Datenbanken zwei Java-Klassen zur Verfügung, die auch in den Abiturklausuren verwendet werden:

Die Klasse **DatabaseConnector** stellt eine Verbindung zu einer Datenbank her und ermöglicht, SQL-Abfragen zu senden und die Ergebnisse in Tabellenform abzuholen. Es gibt drei Versionen dieser Klasse, je eine für Verbindungen mit dem MySQL-DBMS, mit einer SQLite- oder mit einer MS Access-Datenbankdatei.

Objekte der Klasse **QueryResult** enthalten je ein Ergebnis einer SQL-Abfrage in Form eines 2D-Arrays von Strings (also einer Tabelle von Texten).

Hinweis: Die Klasse DatabaseConnector nutzt die dir bekannte Klasse **Queue** bei der Zusammenstellung von QueryResult-Objekten. Daher muss die Klasse Queue auch im BlueJ-Projekt enthalten sein. Für die Hauptklasse, also für deinen Quellcode, ist das aber nicht relevant, du musst hier keine Queue-Objekt erzeugen.



Die einzelnen Methoden der Klasse DatabaseConnector und QueryResult werden im folgenden näher beschrieben.

Hinweis: Zur Verwendung der Klasse DatabaseConnector wird, je nach Version, ein passender **Treiber** benötigt. Siehe dazu die Erläuterungen auf der letzten Seite.

Klasse DatabaseConnector

Konstruktor:
```
DatabaseConnector(String pIP, String pPort, String pDatenbank,
                  String pBenutzer, String pPasswort)
```

Erzeugt eine Datenbankverbindung mit einem DBMS, das auf einem Computer mit der gegeben IP-Adresse und Port läuft, mit der gegebenen Datenbank, Benutzernamen und Passwort.
Falls dabei ein Fehler auftritt, kann man die entsprechende Meldung mit getErrorMessage() abfragen.

Beispiel **MySQL**:
```
dbCon = new DatabaseConnector("192.168.0.10", 3306, "terra",
                              "user1", "passw1");
```

Beispiel **SQLite**:
```
dbCon = new DatabaseConnector(null, 0, "terra.db", null, null);
```

Die Datenbankdatei „terra.db“ liegt hier im gleichen Ordner wie das Java-Programm.
SQLite unterstützt weder Verbindungen zu anderen Computern noch Benutzernamen / Passwörter.
Daher werden die **Parameter** für IP-Adresse, Port, Benutzernamen und Passwort auf null / 0 gesetzt.

Beachte: falls die Datei „terra.db“ **nicht existiert**, erzeugt der SQLite-Treiber eine neue, leere Datei.
Man kann diese dann mittels SQL mit neuen Tabellen füllen.
Da wir aber nur SELECT-Anfragen an eine bereits existierende Datenbank senden werden, deutet es auf einen Fehler hin, wenn die Datei nicht existiert (z.B. ein Tippfehler im Dateinamen).

Es ist daher sinnvoll, vor der Erzeugung des DatabaseConnectors zu prüfen, ob die Datei „terra.db“ existiert. Zu diesem Zweck kannst du die Klassen Files und Paths der Java-Klassenbibliothek nutzen:

```
if (Files.exists(Paths.get("terra.db")))
{
   // erzeuge DatabaseConnector
}
```

Um die Klassen File und Paths zu verwenden, musst du das Package java.nio.file.* importieren.

Beispiel **MS Access**:
```
dbCon = new DatabaseConnector(null, 0, "terra.mdb", "user1", "passw1");
```

Die MS Access-Datei „terra.mdb“ liegt hier im gleichen Ordner wie das Java-Programm.
IP-Adresse und Port werden für Access nicht benötigt, daher kann man sie auf null / 0 setzen.

Wie für SQLite gilt auch hier der Rat zur Prüfung, ob die Datei existiert.

`void` **`executeStatement`**`(String pSQL)`
Sendet eine SQL-Abfrage an die Datenbank. Beispiel:
```
dbCon.executeStatement("SELECT * FROM Kontinent");
```

`QueryResult` **`getCurrentQueryResult`**`()`	Gib das Ergebnis der letzten Abfrage zurück, die mit executeStatement() gesendet wurde.

`String` **`getErrorMessage`**`()`
Gibt eine eventuelle Fehlermeldung der zuletzt ausgeführten Methode zurück (z.B. Erzeugung durch den Konstruktor oder executeStatement()). Im Normalfall (kein Fehler) wird null zurückgegeben.

`void` **`close`**`()`	Beendet die Verbindung mit der Datenbank. Sollte am Ende des Programms aufgerufen werden.

Klasse QueryResult

Ein QueryResult-Objekt enthält eine Tabelle mit den Daten eines Abfrageergebnisses. Außerdem enthält es ein Array mit den Spaltennamen dieser Tabelle, sowie ein Array mit den Datentypen ihrer Spalten – daran kann man sehen, ob in einer Spalte z.B. Zahlen oder Texte enthalten sind. Da in QueryResult alle Daten in Textform vorliegen, müssen diese im Java-Programm ggf. noch in Zahlen umgewandelt werden.

Den **Konstruktor** `QueryResult(String[][], String[], String[])` brauchst du selbst nicht aufzurufen, da QueryResult-Objekte durch das DatabaseConnector-Objekt erzeugt werden. Nachdem man mit executeStatement() eine SELECT-Abfrage gesendet hat, liefert getCurrentQueryResult() das entsprechende QueryResult-Objekt.	
`String[] **getColumnNames**()`	Gibt die Spaltennamen des Abfrageergebnis zurück.
`String[] **getColumnTypes**()`	Gibt die Datentypen des Abfrageergebnis zurück.
`String[][] **getData**()`	Gibt die Daten des Abfrageergebnis zurück.
`int **getColumnCount**()`	Gibt die Anzahl von Spalten zurück, die getData() liefert.
`int **getRowCount**()`	Gibt die Anzahl von Zeilen zurück, die getData() liefert.

Ein Beispiel:

colNames	"ProductID"	"ProductName"	"Price"
colTypes	"VARCHAR"	"VARCHAR"	"DOUBLE"
data	"P002"	"Cheese"	"2.99"
	"P051"	"Fish Fingers"	"3.99"
	"P073"	"Roast Beef"	"5.99"

An dem Beispiel kannst du sehen, dass die Datentypen in SQL zum Teil anders heißen als in Java, daher hier eine kleine Übersicht. Bei den SQL-Datentypen gibt es eine ganze Reihe von Aspekten zu beachten, die jedoch für den Informatikunterricht kaum eine Rolle spielen. Bei Interesse kannst du z.B. nachlesen auf `https://www.w3schools.com/sql/sql_datatypes.asp`

Java-Datentyp	**SQL-Datentyp**
String	VARCHAR (d.h. die Anzahl der Buchstaben für Werte in dieser Spalte ist nicht fest vorgegeben – wohl aber die maximale Anzahl)
int	INT oder INTEGER
double	DOUBLE

Und zuletzt, um Zahlen aus einer SELECT-Abfrage zu verwenden und damit zu rechnen, kannst du die String-Werte, die du von getData() erhältst, in int- oder double-Werte umwandeln:

`double Double.**parseDouble**(String s)`	Wandelt einen String in einen double-Wert um.
`int Integer.**parseInt**(String s)`	Wandelt einen String in einen int-Wert um.

Programmbeispiel

Das folgende Beispiel verwendet die SQLite-Datenbankdatei terra.db.
Diese speichert verschiedene Daten wie Städte, Sprachen usw. zu den Ländern der Erde.

Am Ende der Methode printResult() wird auch gezeigt, wie man String-Daten umwandeln kann in Zahlen im double-Format. Das Rechenbeispiel (Mittelwert der Einwohnerzahlen aller Länder) hätte man auch einfach mit der SQL-Aggregatfunktion AVG berechnen können, aber es eignet sich hier als Beispiel.

```
import java.nio.file.*;
import console.*;

public class DBBeispiel
{
   private DatabaseConnector dbCon;
   private String dateiname = "terra.db";

   public DBBeispiel()    // dbCon wird in main() erzeugt
   { }

   public void main()
   {
      QueryResult result;

      // Falls die Datenbankdatei nicht gefunden wird, breche ab
      if (!Files.exists(Paths.get(dateiname)))
      {
         Console.println("Datei " + dateiname + " existiert nicht.");
         return;
      }

      // Öffne Verbindung mit der Datenbank
      dbCon = new DatabaseConnector("", 0, dateiname, "", "");

      // Falls Verbindung nicht aufgebaut werden kann, breche ab
      String fehler = dbCon.getErrorMessage();
      if (fehler != null)
      {
         Console.println(fehler);
         return;
      }

      // Sucht Länder mit Namen und Einwohnern
      dbCon.executeStatement("SELECT Name, Einwohner FROM Land;");
      result = dbCon.getCurrentQueryResult();

      // Falls kein Ergebnis zurückkommt, breche ab
      if (result == null)
      {
         Console.println("Kein Abfrageergebnis");
         return;
      }
```

```
      // alle Fehler ausgeschlossen
      int zeilen              = result.getRowCount();
      int spalten             = result.getColumnCount();
      String[]   spaltenNamen = result.getColumnNames();
      String[]   datentypen   = result.getColumnTypes();
      String[][] daten        = result.getData();
      int s, z;

      // Überschriften ausgeben
      for (s = 0; s < spalten; s++)
      {
         Console.print(spaltenNamen[s] + " ");
      }
      Console.println();

      // Datentypen ausgeben
      for (s = 0; s < spalten; s++)
      {
         Console.print(datentypen[s] + " ");
      }
      Console.println();

      // Daten ausgeben
      for (z = 0; z < zeilen; z++)
      {
         for (s = 0; s < spalten; s++)
         {
            Console.print(daten[z][s] + " ");
         }
         Console.println();
      }

      // Durchschnittliche Einwohnerzahl ausrechnen
      double summe = 0.0, mittel;
      for (z = 0; z < zeilen; z++)
      {
         summe += Double.parseDouble(daten[z][1]);
      }
      mittel = summe / zeilen;
      Console.println("Mittlere Einwohnerzahl: " + mittel);
   }
}
```

Aufgabe

Verwende die ausgeteilte Vorlage. Sie enthält die Klassen DatabaseConnector, QueryResult und Queue, sowie die SQLite-Datenbankdatei „terra.db“.

Die Datenbank hat die folgende Struktur (ein ER-Modell findest du auf der folgenden Seite):

Kontinent (KNR, Name, Flaeche)
Land (LNR, Name, Einwohner, Flaeche, ↑KNR, ↑HauptONR) (HauptONR für die Hauptstadt)
Ort (ONR, Name, Landesteil, Einwohner, Breite, Laenge, ↑LNR)
Sprache (SNR, Name)
gesprochen (↑SNR, ↑LNR, Anteil) (Bevölkerungsanteil, der die Sprache als Muttersprache hat)

Implementiere die Hauptklasse entsprechend des Klassendiagramms auf S. 2.
Orientiere dich am Programmbeispiel auf den vorigen Seiten.

a) Die main-Methode prüft, ob die Datei terra.db existiert und baut eine Datenbankverbindung mit der Datei auf. Falls ein Fehler auftritt, wird eine Meldung ausgeben und die Methode bricht ab. Sonst werden die Methoden aus b) und c) aufgerufen.

b) Die Methode europa() sendet eine Abfrage an die Datenbank, um alle Länder des Kontinents Europa mit Name, Einwohnern, Fläche und Hauptstadt zu erhalten.
Sie gibt die Überschriften und Datentypen der Spalten der Ergebnistabelle sowie die erhaltenen Daten auf der Konsole aus.

c) Die Methode englisch() ermittelt die Anzahl Menschen, die die Sprache Englisch als Muttersprache sprechen und gibt diese auf der Konsole aus.

   In der dazu nötigen Abfrage bilde einen Verbund der Tabellen Sprache, gesprochen und Land und filtere nach „Englisch“. Bearbeite die Ergebnistabelle dann mithilfe einer for-Schleife und berechne für jedes Land (anhand der Einwohnerzahl und des Anteils der Sprache) die Anzahl Personen, die die Sprache sprechen, und addiere diese Werte.

   Hinweise:

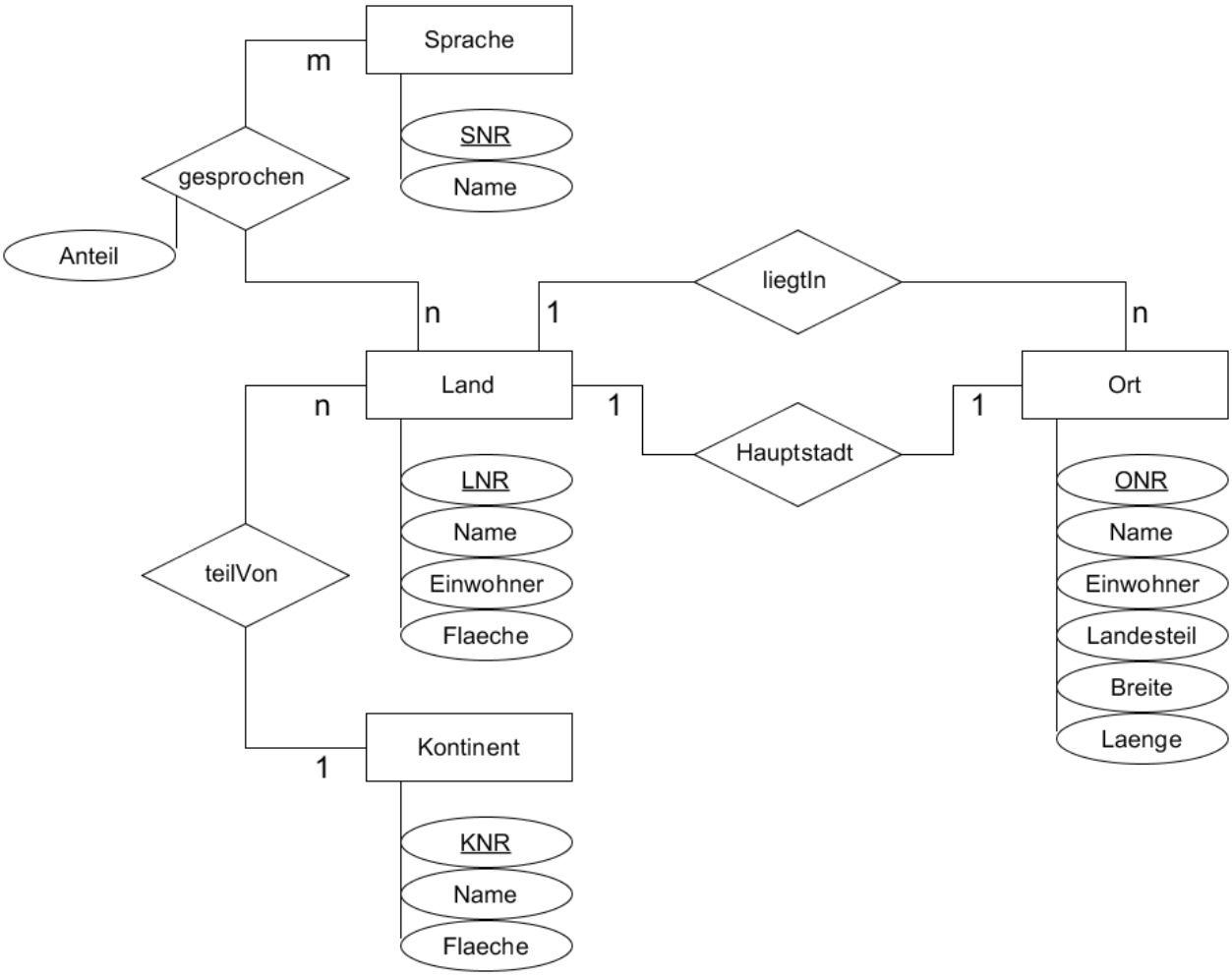
   i. Die Spalte „Einwohner“ der Tabelle Land enthält Double-Werte mit der Einheit „Millionen Einwohner“. Ein Wert von 82.26 bedeutet also 82.26 Millionen.
   Die Spalte „Anteil“ der Tabelle gesprochen enthält Integer-Werte mit der Einheit Prozent. Ein Wert von 100 bedeutet also 100%.

   ii. Einige Anteile in der Tabelle „gesprochen“ haben den Wert NULL (gerade im Fall von Englisch, das in vielen Ländern zwar Amtssprache, aber nicht Muttersprache ist).
   Diese sollte deine SQL-Abfrage herausfiltern.

   iii. Das Ergebnis sollte ca. 322 Millionen betragen. Dieser Wert stimmt nicht unbedingt mit der Realität überein. Die Daten für die Datenbank wurden zu Übungszwecken zusammengetragen und sind daher nicht unbedingt vollständig oder ganz korrekt.

   iv. Die Berechnung könnte man auch über SQL-Aggregatfunktionen lösen.
   Es geht hier um eine Übung, Zahlen aus einem Abfrageergebnis in Java zu verwenden.

ER-Modell zur Aufgabe:



Treiber für die Klasse DatabaseConnector

In der Schule sind die Treiber für die Klasse DatabaseConnector für MySQL / SQLite / MS Access in BlueJ bereits installiert. Du kannst diese Klasse also in deinem Projekt verwenden.
Für den Gebrauch zu Hause werden dir die Treiber von der Lehrkraft zur Verfügung gestellt.
Sie müssen dann ins BlueJ-Programmverzeichnis kopiert werden, z.B. in `C:\Programme\BlueJ\lib\userlib`.

Du findest die Treiber auch im Web:

- MySQL: `https://dev.mysql.com/downloads/connector/j/`
  (oder suche nach „mysql java driver“)
- SQLite: `https://github.com/xerial/sqlite-jdbc/releases`
  (oder suche nach „sqlite java driver“)
- MS Access: `https://sourceforge.net/projects/ucanaccess/`
  (oder suche nach „ucanaccess“)