Block 29 – Datenbanken

- Einführung
- SQLite
  - Beispielanwendung
  - Datentypen
  - SQLite Klassen
  - SQLiteOpenHelper
  - Datenbank
  - Insert, Update, Delete, Select (CRUD)
  - Zusammenfassung Insert, Update, Delete, Select (CRUD)
  - Sanity-Checks
  - Datenbank exportieren
- Object-Relational-Mapping

Block 29 – Lernziele

In diesem Block werden Sie lernen, ...

- wie Daten einer mobilen Anwendung in einer lokalen Datenbank gespeichert werden können.
- wie SQLite als leichtgewichtigtes Datenbankmanagementsystem zum Einsatz kommen kann und was seine Eigenschaften sind.
- wie ein Datenbankhandler für den Zugriff auf eine Datenbank benutzt werden kann.
- wie Datenbankoperationen (Create, Read, Update, Delete) programmatisch auf der Datenbank ausgeführt werden können.
- welche Rolle Sanity-Checks im Zusammenhang mit Datenbanken spielen.
- wie eine Datenbank exportiert werden kann.
- was die objektrelationale Abbildung ist, und warum sie notwendig ist.

29.1 Einführung

Datenbanken erlauben die strukturierte Ablage und Abfrage von Daten.

Das Datenbankmanagementsystem (DBMS) steuert den Mehrbenutzerzugriff auf die Daten und stellt die Konsistenz u. Integrität der Daten sicher.

Abfragesprachen (z.B. SQL) erlauben die systematische Anfrage nach Datensätzen, die bestimmten Kriterien entsprechen.

Es gibt verschiedene Datenbankparadigmen (Hierarchisch, Relational, Objekt-orientiert, Graph-orientiert). Am weitesten verbreitet sind relationale Datenbanken.

29.2 SQLite

Leistungsstarke DBMS werden auf dedizierten Datenbankservern betrieben.

Für kleine Anwendungen genügt allerdings eine lokale Datenbank, die insbesondere auch in einem Offline-Zustand verfügbar ist.

SQLite (entworfen von Richard D. Hipp) verfolgt dieses Ziel.

Eine SQLite-Datenbank residiert in einer einzige Datenbankdatei.

Typische Anwendungsgebiete: Mobile Plattformen, Eingebettete System / Internet-of-Things, Webbrowser, alle Anwendung mit temporären Datenbeständen, ...

29.2 SQLite (Forts.)

Implementiert den größten Teil des SQL92-Standards

Unterstützt große Datenbestände (im Terabyte-Bereich) und Datenfelder im Gigabyte-Bereich

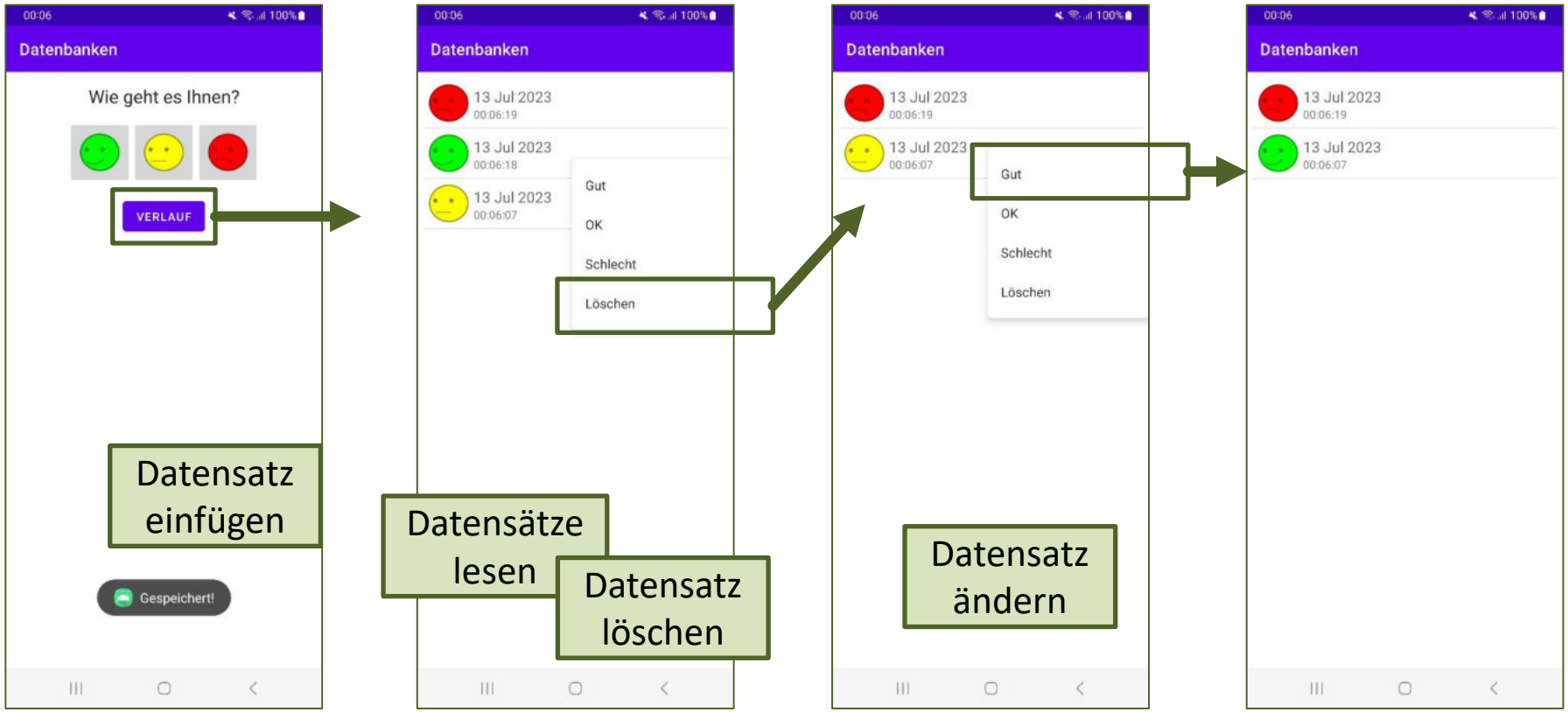
Performant im Vergleich zu anderen Systemen (Client/Server)

Verfügbar für viele Plattformen (Unix, Windows, Android, etc.)

SQLite (DBMS) hat eine Größe von ca. 0.5 MB

Erlaubt serialisierbare Transaktion (ACID)

29.2.1 Beispielanwendung (Mood)



29.2.2 SQLite - Datentypen

Integer: Vorzeichenbehafteter Integer mit max. 8 Byte

Real: Fließkommawert mit 8 Byte (EEE Standard 754)

Text: Textstring (im Format des eingestellten Datenbankencodings z.B. UTF-8, UTF-16BE, UTF-16LE) - VARCHAR

BLOB: Binary Large Objects zur direkten Speicherung von Binärdaten

Boolean: wird nicht unterstützt stattdessen (Integer `1=true` o. `0=false`).

29.2.3 SQLite - Klassen

`SQLiteOpenHelper`: Mit der Klasse `SQLiteOpenHelper` kann eine neue Datenbank angelegt oder eine bestehende Datenbank geöffnet werden. Eine Subklasse muss hierzu das individuelle Datenbankschema und weitere Definitionen enthalten.

`SQLiteDatabase`: Die Klasse `SQLiteDatabase` repräsentiert die Datenbank und ermöglicht die verschiedenen Datenbankoperationen (CREATE, READ, UPDATE, DELETE).

`SQLiteQueryBuilder`: Mit der Klasse `SQLiteQueryBuilder` können Abfragen an die Datenbank 'zusammengebaut' werden. Dabei ist nicht das vollständige SQL-Statement notwendig, sondern nur bestimmte Anteile.

`SQLiteCursor`: Mit der Klasse `SQLiteCursor` können die Ergebnisse der Abfragen iteriert werden.

29.2.4 SQLiteOpenHelper

In der `MainActivity` wird mit `openHandler : OpenHandler = OpenHandler(this)` ein Handler für alle Datenbankoperation instanziiert:

```
class OpenHandler (context : Context): SQLiteOpenHelper
(context, DATABASE_NAME, null, DATABASE_VERSION) {

  companion object {
    // Name und Version der Datenbank
    private val DATABASE_NAME = "mood.db"
    private val DATABASE_VERSION = 1
    ...
  }
  ...
}
```

Über den Konstruktor der Klasse `SQLiteOpenHelper` kann eine neue Datenbank d.h. eine neue Datenbankdatei angelegt werden.

Hinweis: Die Datenbank ist dann erst mit dem ersten Zugriff auf dem Dateisystem zu sehen (z.B. .../data/user/0/com.example.datenbanken/databases/mood.db).

29.2.5 Datenbank

Mit dem Befehl `db: SQLiteDatabase = getWritableDatabase()` greift der `SQLiteOpenHelper` dann auf die Datenbank zu. Dabei wird jedes mal die `onCreate()`-Methode des `SQLiteOpenHelpers` aufgerufen.

```
class OpenHandler (context : Context): SQLiteOpenHelper
(context, DATABASE_NAME, null, DATABASE_VERSION) {

  companion object { ... val MOOD_MOOD = "mood"
    val MOOD_TIME = "timeMillis"}

  private val _ID = "_id"
  private val TABLE_NAME_MOOD = "mood"

  private val TABLE_MOOD_CREATE = ("CREATE TABLE "
    + TABLE_NAME_MOOD + " (" + _ID
    + " INTEGER PRIMARY KEY AUTOINCREMENT, " + MOOD_TIME
    + " INTEGER, " + MOOD_MOOD + " INTEGER);")
  ...
  override fun onCreate(db: SQLiteDatabase) {
    db.execSQL(TABLE_MOOD_CREATE)}
}
```

Die `onCreate()`-Methode erzeugt erstmalig aufgerufen die definierte Tabelle 'mood'.

29.2.6 Insert

Mit einem Klick auf einen der Smiley-Buttons wird die Methode `openHandler.insert(mood, System.currentTimeMillis())` aufgerufen:

```
fun insert(mood: Int, timeMillis: Long) {
  var rowId: Long = -1
  try {
    val db: SQLiteDatabase = getWritableDatabase()
    val values = ContentValues()
    values.put(MOOD_MOOD, mood)
    values.put(MOOD_TIME, timeMillis)
    rowId = db.insert(TABLE_NAME_MOOD, null, values)
  } catch (e: SQLiteException) {
    Log.e(TAG, "insert()", e)
  } finally {
    Log.d(TAG, "insert(): rowId=$rowId")
  }
}
```

Die Methode kennt über die Key/Value-Paare der `put()`-Methode die Spalten für die Daten.

29.2.6 Insert (Forts.)

Die Zeile `rowId = db.insert(TABLE_NAME_MOOD, null, values)` ersetzt dabei gleich zwei SQL-Statements:

```
INSERT INTO 'mood' (timeMillis, mood)
VALUES (1567183800509, 1);
```

und

```
SELECT max(_ID) FROM mood;
```

Die Methode `db.insert()` führt dazu, dass im Hintergrund die entsprechenden SQL-Statements generiert und auf der Datenbank ausgeführt werden. Alternativ können SQL-Statements aber auch manuell zusammengebaut werden:

```
db.execSQL("INSERT INTO 'mood' (timeMillis, mood)" +
  "VALUES (" + timeMillis + ", " + mood + ")");
```

29.2.7 Update

Bei dem Update soll ein ganz bestimmter Datensatz d. h. eine `id` und eine ganz bestimmte Spalte geändert werden d.h. die Tabelle `mood` geändert werden.

```
fun update(id: Long, smiley: Int) {
  val db: SQLiteDatabase = getWritableDatabase()
  val values = ContentValues()
  values.put(MOOD_MOOD, smiley)
  val numUpdated = db.update(TABLE_NAME_MOOD,
      values, "$_ID = ?", arrayOf(java.lang.Long.toString(id)))
  Log.d(TAG, "update(): id=$id -> $numUpdated")
}
```

Dabei erwartet die Methode `db.update()` den Tabellennamen, die neuen Werte (`ContentValues`), eine Where-Klausel (betroffene Datensätze i.d.R. ein Datensatz) und die Argumente der Where-Klausel.

Abschließend liefert die Methode die Anzahl der geänderten Datensätze (ein Datensatz).

29.2.7 Update (Forts.)

Das intern generierte und dann ausgeführte SQL-Statement:

```
UPDATE mood
SET mood = '1'
WHERE _id = 1;
```

29.2.8 Delete

Ähnlich zu dem Update soll bei dem Delete ein ganz bestimmter Datensatz entfernt werden. Hier sind keine weiteren Angaben als die in der Where-Klausel (inkl. der Argumente) zu bestimmen.

```
fun delete(id: Long) {
  val db: SQLiteDatabase = getWritableDatabase()
  val numDeleted =
    db.delete(TABLE_NAME_MOOD, "$_ID = ?",
      arrayOf(java.lang.Long.toString(id)))
  Log.d(TAG, "delete(): id=$id -> $numDeleted")
}
```

Abschließend liefert die Methode die Anzahl der gelöschten Datensätze (ein Datensatz).

29.2.8 Delete (Forts.)

Das intern generierte und dann ausgeführte SQL-Statement:

```
DELETE FROM mood
WHERE _id = 1;
```

29.2.9 Select

Ein Select kann nicht mit einem `db.execSQl` erfolgen, da ein Select eine Menge von Datensätzen zurückliefert. Hierfür wird die `db.query()`-Methode genutzt, welche einen `Cursor` zurückliefert.

Der `Cursor` ist eine dynamische Zwischenstruktur, welche die mit der `db.query()`-Methode abgefragten Daten speichert und einen Zugriff ermöglicht.

```
fun query(): Cursor? {
  val db: SQLiteDatabase = getReadableDatabase()
  return db.query(TABLE_NAME_MOOD,
        null, null, null,
        null, null,
        "$MOOD_TIME DESC")
}
```

29.2.9 Select (Forts.)

Die Daten aus dem `Cursor` können nun ausgelesen (`MoodAdapter`) werden:

```
override fun bindView(view: View, context: Context?, cursor: Cursor)
  val ciMood: Int = cursor.getColumnIndex(OpenHandler.MOOD_MOOD)
  val ciTimeMillis: Int =
    cursor.getColumnIndex(OpenHandler.MOOD_TIME)
  val image: ImageView = view.findViewById(R.id.icon)
  val mood: Int = cursor.getInt(ciMood)
  if (mood == OpenHandler.MOOD_FINE) {
    image.setImageResource(R.mipmap.smiley_gut) ... }
  val textview1: TextView = view.findViewById(R.id.text1)
  val textview2: TextView = view.findViewById(R.id.text2)
  val timeMillis: Long = cursor.getLong(ciTimeMillis)
  date.setTime(timeMillis)
  textview1.setText(DF_DATE.format(date))
  textview2.setText(DF_TIME.format(date))
}
```

Hinweis: Der Cursor wird von der Superklasse `CursorAdapter` selbständig auf den nächsten Datensatz bewegt.

29.2.10 Zusammenfassung - Insert, Update, Delete, Select (CRUD)

```
long db.insert(table : String, nullColumnHack[6] : String ,
values : ContenValues)

long db.update(table : String, values : ContentValues,
whereClause : String, whereArgs : Array<String>)

long db.delete(table : String, whereClause : String,
whereArgs : Array<String>)

Cursor db.query(table String, columns Array<String>,
selection : String, selectionArgs : Array<String>, groupBy
: String, having : String, orderBy : String)
```

[6] Ist nur zu verwenden, falls die ContentValues keine Spaltennamen enthalten.

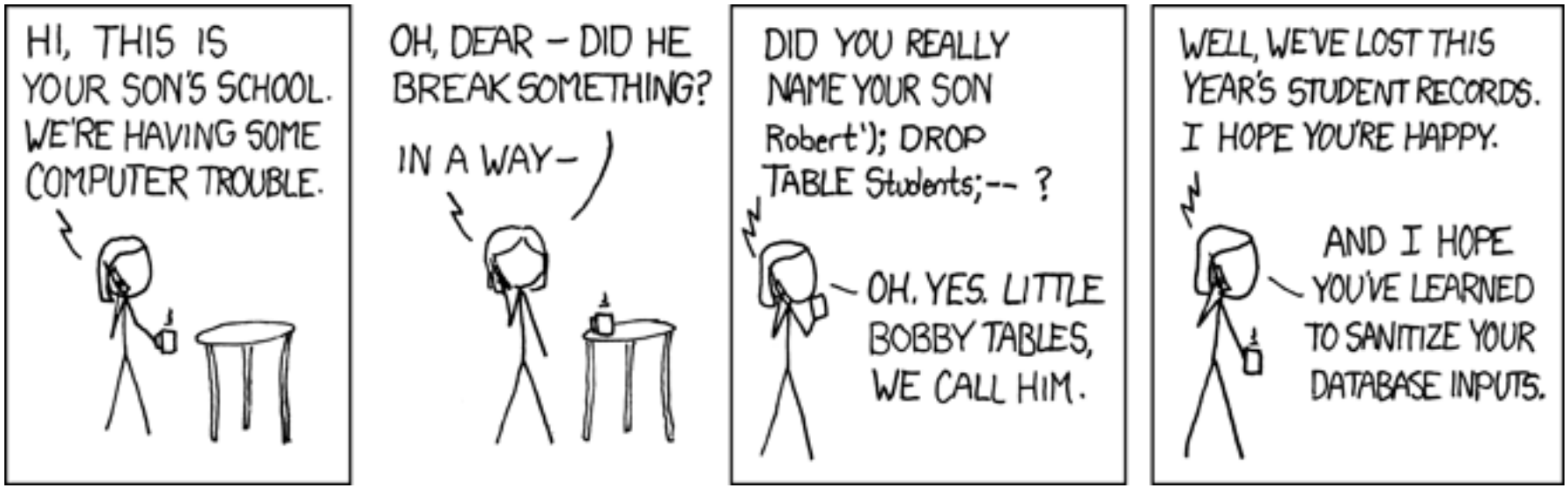
29.2.11 Sanity-Checks

In der `db.execSQL`-Variante des `db.input` wurde der Wert von `mood` direkt aus vom Benutzer eingegeben Parameter `mood` übernommen:

```
db.execSQL("INSERT INTO 'mood' (timeMillis, mood)" +
  "VALUES (" + timeMillis + ", " + mood + ");")
```

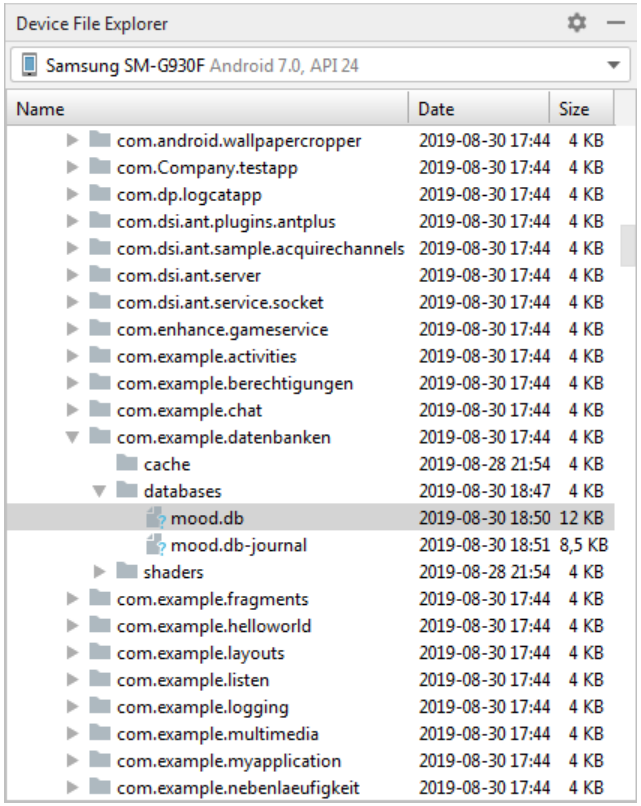
Was wäre wenn der Benutzer statt `'1'` (mood) den Wert `'1); DROP TABLE mood;'` eingeben hätte?

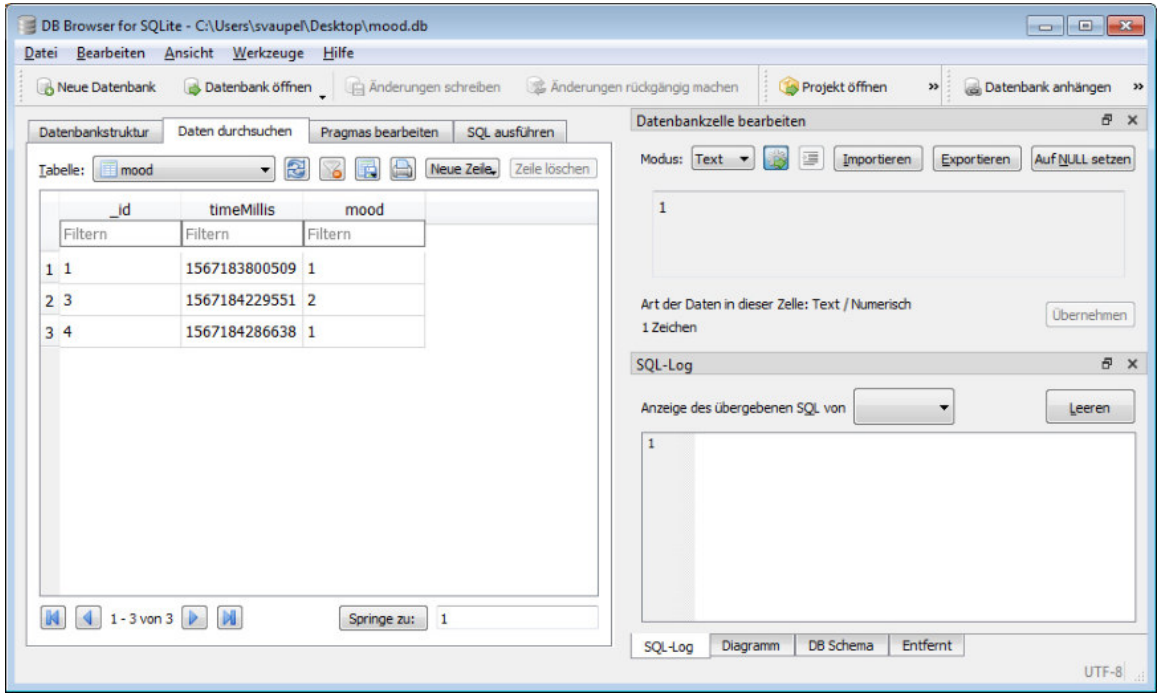
29.2.11 Sanity-Checks (Forts.)



https://xkcd.com/327/

29.2.12 Datenbank exportieren





http://sqlitebrowser.org/

29.3 Object-Relational-Mapping

Die Arbeit mit einem `Cursor` gestaltet sich eher umständlich, da Feld für Feld ausgelesen werden muss.

```
override fun bindView(view: View, context: Context?, cursor: Cursor)
  val ciMood: Int =
    cursor.getColumnIndex(OpenHandler.MOOD_MOOD)
  val ciTimeMillis: Int = cursor.getColumnIndex(OpenHandler.MOOD_TIME
  val mood: Int = cursor.getInt(ciMood)
  val timeMillis: Long = cursor.getLong(ciTimeMillis) ...
}
```

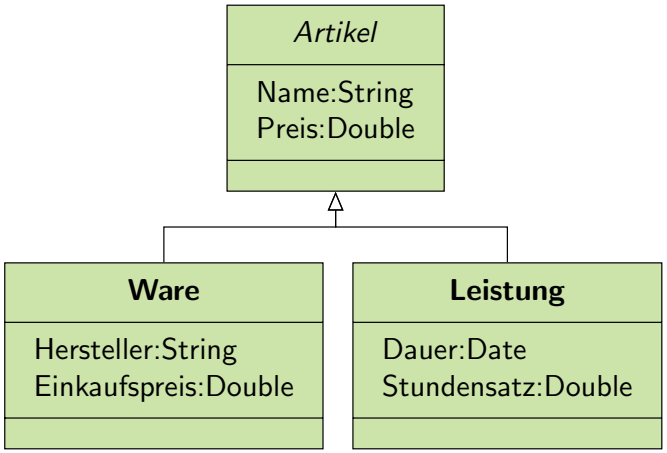
Optimaler wäre es, wenn Java-Klassen speicherbar bzw. wieder abrufbar wären.

SQLite ist jedoch relational, während Java-Klassen einem Objekt-orientieren Paradigma folgen. Wie geht beides zusammen?

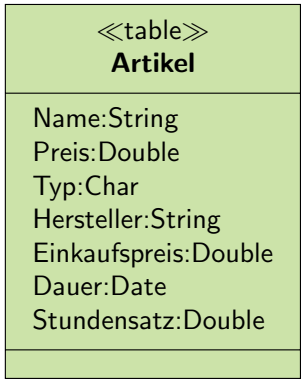
Die Techniken des Object-Relational-Mapping erlaubten zumindest die strukturelle Abbildung der objekt-orientierten Klassen auf die relationalen Tabellen.

29.3 Object-Relational-Mapping (Forts.)

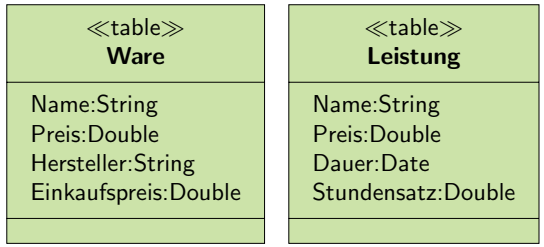
Beispiel (OO-Datenmodell – Relationales Modell; Vgl. Ambler 2012 [1]):



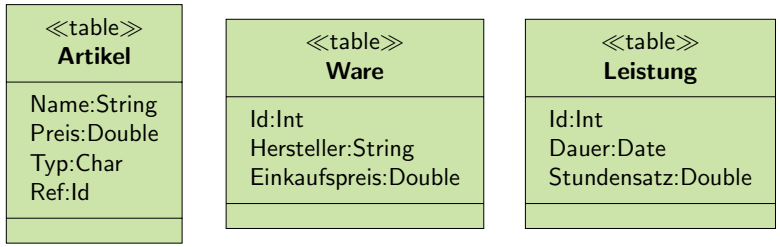
a) OO-Datenmodell



b) Implementierung 1



c) Implementierung 2



d) Implementierung 3

□

Block 29 – Zusammenfassung

- **Datenbanken** sind ein wichtiger Bestandteil zur dauerhaften lokalen Datenspeicherung und Abfrage
- **SQLite** ist ein leichtgewichtiges DMBMs, welches besonders gut für mobile Plattformen geeignet ist
- Der Umgang mit SQLite erlaubt die Speicherung und Abfrage in Anlehnung zu **SQL** bzw. die direkte Ausführung von SQL-Statements ist auch möglich
- Bei Benutzereingaben i. Z. mit Datenbanken müssen **Sanity-Checks** vorgenommen werden
- Der Umgang mit **Cursor**n ist ggf. unpraktisch, da alle Datenfelder ausgelesen werden müssen
- **Object-Relationale-Mapper** erlaubt es, direkt auf der objekt-orientierten Struktur zu arbeiten

Block 29 – Weitere Aufgaben

- Erstellen Sie eine einfache Tabelle um Personalien (z. B. Anrede, Name, Vorname, Geburtsdatum, Titel, etc.) zu erfassen. Achten Sie auf passende Datentypen.
- Erstellen Sie eine Vererbungshierarchie aus verschiedenen Klassen. Bilden Sie diese Vererbungshierarchie mit einem objektrelationalen Abbildungsmuster ab. Testen Sie die Speicherung von Objekten dieser Vererbungshierarchie in der/den Datenbanktabellen. Test Sie auch das Auslesen der gespeicherten Objekte.
- Testen Sie die Transaktionsicherheit des Datenbankmanagementsystems, indem Sie konkurierrende Zugriffe (z. B. über Threads) auf der Datenbank ausführen.

Block 29 – Literatur I

[1] S. Ambler. *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. Wiley, 2012.