Android Persistenz

Android Persistenz

Android bietet mehrere Möglichkeiten, Daten dauerhaft abzuspeichern:

- ***Shared Preferences***: Zur Abspeicherung kleiner, privater Daten bzw. Einstellungen in Form von Schlüssel/Wert-Paaren.
- ***Files (Dateien)*** : Unter der Verwendung der Java-API können auf (interne und externe) Dateien und Ordner zugegriffen werden. Es gilt dabei die Linux-Rechte.
- ***SQLite Datenbanken***: Strukturierte Daten können in einer Datenbank abgelegt werden.
- ***Netzwerk***: Daten können über den Web in einen eigenen Server abgelegt und gelesen werden.
- ***Content Provider***: Eine optionale Android-Komponente für Datenzugriffe und Manipulation über Anwendungsgrenze hinweg.

Shared Preferences

Shared Preferences: ein leichtgewichtiger Mechanismus auf `Activity`-Level zur Abspeicherung anwendungsbezogener Einstellungen, einfacher Benutzer-Informationen, Konfigurationsdaten usw. Alle Komponenten eines Pakets können diese Daten gemeinsam benutzen, sofern sie nicht privat sind. Die Daten werden lokal abgespeichert und bleiben über verschiedene Sitzungen hinweg persistent. Folgende elementare Datentypen werden unterstützt

- `Boolean`
- `Float`
- `Integer`
- `Long`
- `String`

Shared Preferences

Die Schnittstelle ***SharedPreferences*** (aus dem Paket `android.content`) bietet einen Framework zur Manipulation der Daten.

Typische Schritte:

- Ein `SharedPreferences`-Objekt wird angelegt bzw. geholt.
- ***`edit()`*** wird aufgerufen, um ein ***`SharedPrefences.Editor`***-Objekt zur Manipulation der Daten zu erhalten.
- Über den Editor werden Datenänderungen durchgeführt.
- Änderungen werden mit ***`commit()`*** bestätigt.

Shared Preferences

Erzeugung privater und gemeinsamer Preference-Dateien

Individuelle Activities können **private** Daten abspeichern, die nur für sie zugänglich sind:

```
import android.content.SharedPreferences;
…
SharedPreferences prefs = getPreferences(MODE_PRIVATE);
```

Shared Preferences müssen benannt sein. Es dürfen mehrere **gemeinsame** Preferences geben (es wird beim ersten Mal ein Objekt angelegt):

```
public static final String PREFS_NAME = "MyPrefsFile";
….
SharedPreferences settings = getSharedPreferences(PREFS_NAME, MODE_PRIVATE);
```

Bemerkung: Shared Preferences können zur Zeit nicht über Prozess-Grenze hinweg zugegriffen werden.

Shared Preferences

Benannte Preferences können folgende Attribute haben:

`MODE_PRIVATE`: Nur zugänglich für alle Activities derselben Applikation

`MODE_WORLD_READABLE` und `MODE_WORLD_WRITEABLE`:

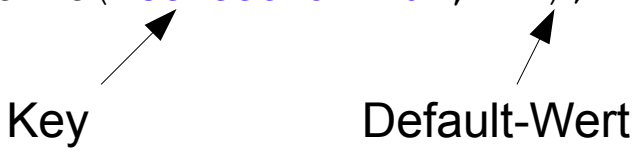
Zugänglich für aller Applikationen in demselben Prozess (sie stammen von demselben Anbieter). Allerdings benötigt die externe Anwendung einen gültigen Kontext, der auf das Paket zeigt, in dem die Preference-Datei vorher erzeugt wurde.

```
Context otherContext = createPackageContext("de.hs_kl.de", 0);
SharedPreferences prefs = otherContext.getSharedPreferences(PREF_NAME, 0);
Bzw.
prefs = PreferenceManager.getDefaultSharedPreferences(otherContext);
(siehe später)
```

Shared Preferences

Lesezugriff über die getter-Methoden:

```
int selectionStart = prefs.getInt("selectionStart", -1);
int selectionEnd = prefs.getInt("selectionEnd", -1);
```



Methoden

`getBoolean, getFloat, getInt, getLong, getString`

`Bundle getAll()` liefert einen Map für alle Schlüssel-Wert-Paare.

`boolean contains(String key)` überprüft, ob ein Wert mit dem Schlüssel `key` gibt.

`Editor edit()` liefert einen Editor zum Manipulieren der Daten.

Shared Preferences

Hinzufügen, Änderung und Löschen

Über den erhaltenen Editor lassen sich Daten verändern:

```
SharedPreferences.Editor editor = getPreferences(MODE_PRIVATE).edit();
editor.putString("text", mSaved.getText().toString());
editor.putInt("selectionStart", mSaved.getSelectionStart());
editor.putInt("selectionEnd", mSaved.getSelectionEnd());
editor.commit();
```

Methoden

`putBoolean, putFloat, putInt, putLong, putString, commit`

`void remove(String key)` zum Löschen des entsprechenden Datums.

`void clear()` Löschen aller Daten.

Wichtig `void commit()` muss am Ende aufgerufen werden, damit die Änderungen wirksam werden.

Shared Preferences

Intern werden die Einstellungsparameter in XML-Format unter `/data/data/<package-name>/shared_prefs/<preferences filename>.xml`

abgespeichert. Etwa in der Form:

```
<?xml version="1.0" encoding="UTF-8" standalone="true"?>
<map>
    <int value="0" name="selectionStart"/>
    <string name="text">Das ist ein Test</string>
    <int value="16" name="selectionEnd"/>
</map>
```

Bemerkung: Häufig werden die Einstellungsparameter in `onResume` eingelesen und in `onPause` abgespeichert (Zeitspanne der Sichtbarkeit der Activity).

PreferenceFragment

PreferenceFragment wird verwendet zur Anzeige und Abspeicherung der Benutzereinstellungen (PreferenceActivity ist deprecated) .

- Mit einer speziellen XML-`Preference`-Hierarchie können die Schlüssel/Werte-Paare der Einstellungen dargestellt und automatisch abgespeichert werden.
- Die Wurzel der Hierarchie ist ein Objekt der Klasse ***`PreferenceScreen`***.
- Zur Einstellungen der einzelnen Daten werden ***`CheckBoxPreference`***, ***`ListPreference`***, ***`EditTextPreference`***, ***`MultiSelectListPrefererence`***, ***`RingtonePreference`*** usw. verwendet.
- Ein `PreferenceScreen` kann auch mehrere `PreferenceScreens` beinhalten, die nach Auswahl zur weiteren Einstellung geöffnet werden.
- Verschiedene Werte können in Kategorien (***`PreferenceCategory`***) eingeteilt werden.

PreferenceFragment

Allgemeine Attribute sind:

- ***android:key***: Der Schlüssel für die Option.
- ***android:title***: Titel für die Einstellung (wird groß dargestellt)
- ***android:summary***: Kurze Zusammenfassung der Option (wird unterhalb des Titels klein dargestellt).
- ***android:defaultValue***: Voreingestellter Wert.
- ***android:dependency***: Wird verwendet, um die Abhängigkeit (z.B. ein `PreferenceScreen` von einem `CheckBox`) darzustellen.

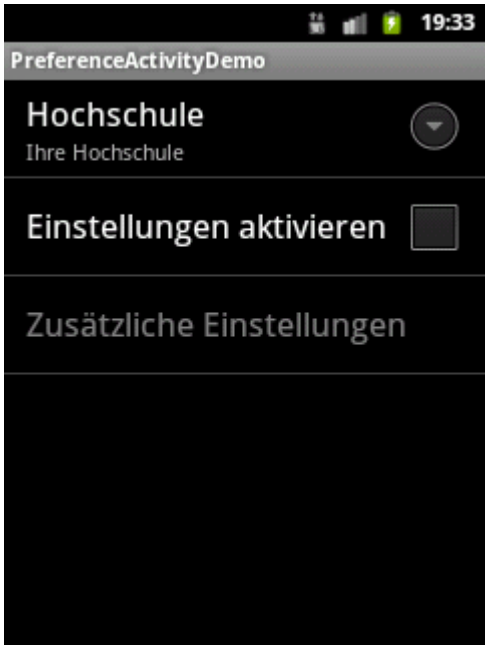
Für `ListPreference` kommen folgende Attribute auch häufig vor:

- ***android:entries***: Texte der Einträge (als Array) und ***android:entryValues***: Schlüssel der Einträge. Jeder Listeneintrag ist ein Schlüssel/Wert-Paar.
- ***android:dialogTitle***: Titel zur Anzeige der Liste als Dialog.

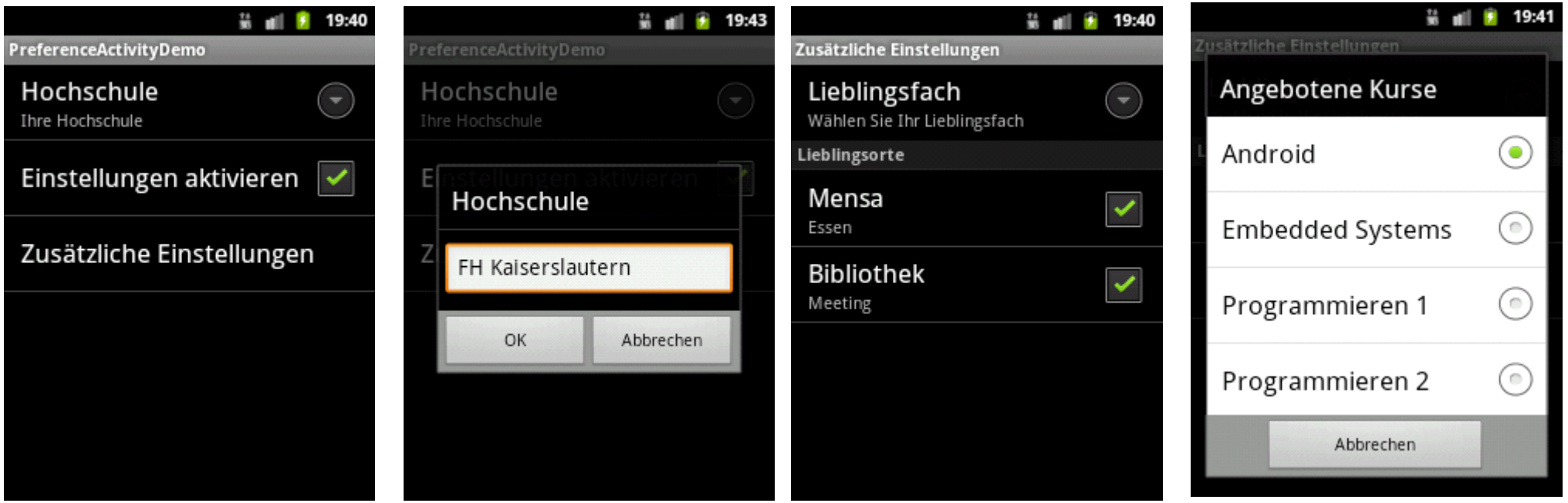
PreferenceFragment

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen
xmlns:android="http://schemas.android.com/apk/res/android" >

    <EditTextPreference
        android:defaultValue="FH Kaiserslautern"
        android:key="universityPref"
        android:summary="Ihre Hochschule"
        android:title="Hochschule" />

    <CheckBoxPreference
        android:defaultValue="false"
        android:key="morePrefCheckbox"
        android:title="Einstellungen aktivieren" />

    <PreferenceScreen
        android:dependency="morePrefCheckbox"
        android:key="morePref"
        android:title="Zusätzliche Einstellungen" >
        ...
    </PreferenceScreen>
</PreferenceScreen>
```

Abhängigkeit



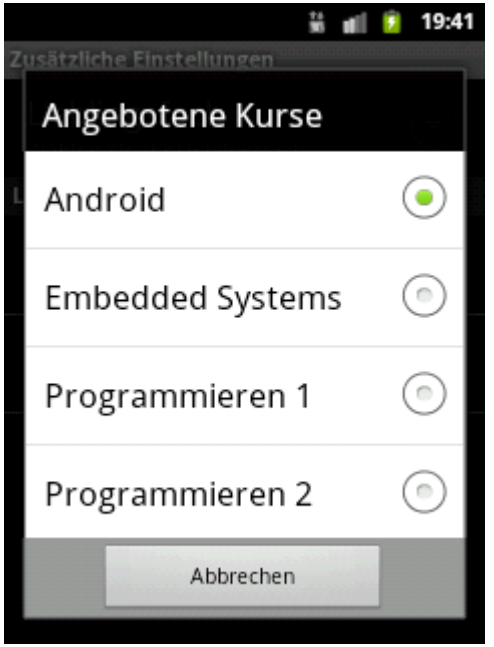
PreferenceFragment



Verschiedene Einstellungsbildschirme

PreferenceFragment

```
<PreferenceScreen ... >
    <ListPreference
        android:defaultValue="Android"
        android:dialogTitle="Angebotene Kurse"
        android:entries="@array/courses_names"
        android:entryValues="@array/courses_values"
        android:key="coursePref"
        android:summary="Wählen Sie Ihr Lieblingsfach"
        android:title="Lieblingsfach" />
    <PreferenceCategory android:title="Lieblingsorte" >
        <CheckBoxPreference
            android:defaultValue="true"
            android:key="mensaPref"
            android:summary="Essen"
            android:title="Mensa" />
        <CheckBoxPreference
            android:defaultValue="true"
            android:key="bibPref"
            android:summary="Meeting"
            android:title="Bibliothek" />
    </PreferenceCategory>
</PreferenceScreen>
```



PreferenceFragment

- `PreferenceScreen` ist **keine** `View`. Die zugehörigen XML- Dateien werden daher häufig unter `res/xml` platziert.
- Durch ***addPreferencesFromResource*** wird das entsprechende Fenster angezeigt:.

```
public class SettingsFragment extends PreferenceFragment {
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        addPreferencesFromResource(R.xml.settings);
    }
}
```

- Ein Zugriff auf `SharedPreferences` ist durch ***PreferenceManager*** möglich:

```
SharedPreferences settings
        = PreferenceManager.getDefaultSharedPreferences(this);
```

Zurücksetzen auf die voreingestellten Werte (nur über den Editor und die Methode ***setDefaultValue(Context c, int resId, boolean readAgain)***)

```
settings.edit().clear();
PreferenceManager.setDefaultValues(this, R.xml.settings, true);
```

PreferenceActivity

Bemerkung

Typischerweise ist eine `PreferenceActivity` nicht der Startpunkt der Anwendung. Daher werden beim Starten der Anwendung die Einstellungsparameter noch nicht geladen! Es ist daher wichtig, die Methode

```
setDefaultValue(Context c, int resId, boolean readAgain)
```

in der `onCreate`-Methode der main-Activity aufzurufen. Sie darf mehrmals aufgerufen werden und die voreingestellten Werte werden nicht wieder eingelesen.

Android Persistenz - Files

Es gibt verschiedene Möglichkeiten, der Anwendung statische Dateien als Ressourcen zur Verfügung zu stellen:

- Dateien nur mit dem lesenden Zugriff können in Verzeichnis `/res/raw` (z.B. `/res/raw/test.mp3`) abgelegt werden. Sie werden indiziert z.B. mit der ID `R.raw.test`. Mit

  ```
  FileInputStream fis = getResources().openRawResource(R.raw.test);
  ```

  kann man die Datei einlesen.

- Geschützte, read-only Dateien/Ordner (die nicht indiziert werden) unter `assets`-Ordner können ähnlich geöffnet und eingelesen werden:

  ```
  InputStream is = getAssets().open("read_asset.txt");
  ```

Android Persistenz - Files

- Jede Anwendung ist ein Linux-Benutzer mit dem eigenen Home-Bereich. Sie kann/darf interne Dateien unter dem Ordner
  /data/data/<package-name>/files
  abspeichern.
- Zugriffe auf Dateien benötigen ein `Context`-Objekt (`this` – die `Activity` selbst bzw. `getApplicationContext()`)

```
FileOutputStream fos = getApplicationContext().openFileOutput(FILENAME,
                                                Activity.MODE_PRIVATE);
```

- Wichtige Methoden: `openFileInput, openFileOutput, deleteFile, String[] fileList()`: Für die Liste aller Dateien in `/files`
  `File getFilesDir()`: Für den absolute Pfad zu dem Ordner `/files`
  `File getCacheDir()`: Für den absolute Pfad zu dem Cache-Ordner.
  `File getDir(String name, int mode)`: Suchen und ggf. Erzeugen eines Unterordners für die Anwendung.

Android Persistenz - Files

Bemerkungen

- Angabe der Dateinamen nur relativ möglich. Ansonsten wird eine Exception ausgeworfen.
- Modi: `MODE_PRIVATE` oder `MODE_APPEND`.
- Bei `openFileOutput` mit `MODE_PRIVATE` wird die Datei erzeugt (falls nicht vorhanden) bzw. gelöscht geöffnet.
- `FileInputStream` und `FileOutputStream` arbeiten mit Bytes (binären Daten). Für Texte können die Klasse `Scanner` und `PrintWriter` verwendet werden.

```
FileOutputStream fos = openFileOutput(FILENAME,
                                      Activity.MODE_PRIVATE);
  PrintWriter out = new PrintWriter(fos);
  ...
  out.print(data);
  out.close();
```

Android Persistenz - Files

Externe Dateisysteme

- Für die Zugriffe auf die SD-Karte muss die Anwendung das Erlaubnis `android.permission.WRITE_EXTERNAL_STORAGE` im Manifest deklarieren.
- Der Pfad ist durch `Environment.getExternalStorageDirectory()` gegeben. Vor der Verwendung sollte der Zustand mit `Environment.getExternalStorageState()` überprüft werden.

```
// Check if external storage is usable
if (!Environment.getExternalStorageState().
                                  equals(Environment.MEDIA_MOUNTED)) {
  .. // Exception handling
}
// Create a new directory on external storage
File rootPath = new File(Environment.getExternalStorageDirectory(),DNAME);
if (!rootPath.exists()) { rootPath.mkdirs(); }

// Create the a object
File dataFile = new File(rootPath, FILENAME);
```

SQLite Datenbanken

SQLite

- ist eine quelloffene Bibliothek, die ein relationales Datenbanksystem beinhaltet (siehe *www.sqlite.org*).
- ist weitgehend konform zum Standard SQL-92.
- wird für mobile Anwendungen optimiert (eine Datenbank ist eine Datei) , ist in Apple iPhone, Symbian Phones, Mozilla Firefox, Skype etc. zu finden.
- braucht keine Installation bzw. zentrale Konfiguration.
- erlaubt die zeitgleiche Nutzung einer Datenbank durch mehrere Anwendungen.
- bietet Einbindung über verschiedene Bibliotheken.

Die Datenbank einer Anwendung wird im Verzeichnis

`/data/data/<package-name>/databases`

abgelegt.

DDL (Data Definition Language) Anweisungen

Eine SQLite-Datenbank besteht aus mehreren Tabellen. Eine Tabelle kann etwa durch

```
CREATE TABLE meineTabelle (
_id INTEGER PRIMARY KEY AUTOINCREMENT,
   name TEXT,
   phone TEXT );
```

erzeugt werden. Hier haben wir eine Tabelle mit drei Spalten:

Spalte 1: Die ID ist der primäre Schlüssel, identifiziert eindeutig den Datensatz (Datenzeile/ data record) (Angabe: `INTEGER PRIMARY KEY`), wird automatisch inkrementiert (Angabe: `AUTOINCREMENT`). Laut Konvention soll die erste Spalte immer `_id` heißen und sie wird im Zusammenhang mit `ContentProvider` benötigt (siehe später) .

Spalte 2 und 3: Für `name` und `phone` vom Typ Zeichenketten.

SQLite – Eine Einführung

- SQLite nimmt **keine Rücksicht auf die Längenangabe des Textes** (d.h. VARCHAR(5) wird wie TEXT behandelt). Das führt zur optimalen Ausnutzung des vorhandenen Speichers auf Kosten der Ineffizienz.
- SQLite unterscheidet nicht zwischen Groß- und Kleinschreibung der Befehle (**case insensitive**).
- Die Typenangaben der Spalten sind nur Hinweise. D.h. man kann z.B. eine Zeichenkette in eine Spalte für Integer schreiben und umgekehrt. Die SQLite Autoren betrachten dies als Feature und keinen Bug.
- Eine Anwendung kann entweder den vollen oder keinen Zugriff auf die Datenbank haben. Es gibt keine Datenbank-Nutzerkonten mit verschiedenen Rechten.

DML (Data Manipulation Language)

Verschiedene Anweisungen erlauben das Einfügen, Löschen und Abändern von Datensätzen (für Details siehe *www.sqlite.org*).

Beispiel

- Einfügen
```
INSERT INTO meineTabelle VALUES(NULL,'Mustermann',
'06332-12345');
```
Für _id wird NULL eingegeben, weil sie automatisch vergeben wird.

- Abfragen
```
SELECT * FROM meineTabelle WHERE _id=3;
```

- Löschen
```
DELETE FROM meineTabelle WHERE (name='Schmidt');
```

- Abändern
```
UPDATE meineTabelle SET phone='06332-54321' WHERE _id=3;
```

SQLite in Android

Zur Demonstration wird ein kleiner Event-Logger implementiert. Der Event-Logger kann Events mit einem Zeitstempel in eine Datenbank speichern.

Schritt 1: Definition einer Schnittstelle für die Spaltennamen. Diese Schnittstelle erweitert normalerweise die `android.provider.BaseColumns` (wo der Name _ID definiert ist).

```
import android.provider.BaseColumns;

public interface ColumnConstants extends BaseColumns
{
   public static final String TABLE_NAME = "EventDB";
   // 2. und 3. Spalten der Tabelle - 1. Spalte ist _ID
   public static final String TIME ="time";
   public static final String EVENT = "event";
}
```

SQLite in Android

Schritt 2: Definition einer Hilfsklasse, die eine Subklasse von `SQLiteOpenHelper` ist. Diese Klasse ist zuständig für die Verwaltung der DB-Erzeugung und -update. Zu implementieren sind normalerweise der Konstruktor und die Methoden `onCreate` und `onUpgrade`.

```
public class EventDatabase extends SQLiteOpenHelper implements
ColumnConstants
{
   private static final String DATABASE_NAME = "eventlogger.db";
   private static final int DATABASE_VERSION = 1;

   public EventDatabase(Context ctx)
   {
      super(ctx, DATABASE_NAME, null, DATABASE_VERSION);

   }
```

In der Praxis soll die Versionsnummer bei jeder Änderung des Datenbankschemas erhöht werden.

SQLite in Android

Wenn die Datenbank zum ersten Mal zugegriffen wird, wird die `onCreate`-Methode aufgerufen. SQL-Anweisung wird als Parameter der `execSQL`-Methode übergeben:

```
public void onCreate(SQLiteDatabase db)
{
   db.execSQL("CREATE TABLE " + TABLE_NAME + " (" + _ID
   + " INTEGER PRIMARY KEY AUTOINCREMENT, " + TIME
   + " INTEGER," + EVENT + " TEXT NOT NULL);");
}
```

Wenn Android erkennt, dass aufgrund der Versionsnummer eine alte Datenbank referenziert wird, wird die Methode `onUpgrade` aufgerufen. Für einfache Anwendung wird die alte Tabelle gelöscht und eine neue angelegt.

```
public void onUpgrade(SQLiteDatabase db, int oldVersion,
                      int newVersion)
{
   db.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);
   onCreate(db);
}
```

SQLite in Android

Weitere Methoden für SQLite-Datenbanken

- `void close(), void beginTransaction(), void endTransaction(), int getVersion(), boolean isOpen()`

Zugriff auf SQLite-Datenbanken

Methode	Beschreibung
query	SQL-Anfrage. Als Parameter werden die Bestandteile übergeben
rawQuery	SQL-Anfrage mit SQL-Befehl in Form einer Zeichenkette als Parameter
insert	Fügt einen neuen Datensatz
update	Ändert Attribute eines vorhandenen Datensatzes
delete	Löschen eines Datensatzes anhand des Schlüsselwertes
execSQL	Führt eine SQL-Anweisung aus

SQLite in Android

Insert, Update und Delete-Anweisungen sind immer möglich mittels `execSQL`. Dabei müssen die kompletten SQL-Anweisungen als String aufgebaut werden. Alternativ dazu ist die Verwendung von ***`ContentValues`*** (Verwendung wie ein Map).

Beispiel:

Verbindung zum Datenbank-Helfer

```
    private EventDatabase mEventDatabase;
...
        mEventDatabase = new EventDatabase(this);
```

Hinzufügen eines neuen Datensatzes

```
// fügt einen neuen Datensatz in die Datenbank
SQLiteDatabase db = mEventDatabase.getWritableDatabase();
ContentValues values = new ContentValues();
values.put(TIME, System.currentTimeMillis());
values.put(EVENT, eventText);
db.insert(TABLE_NAME, null, values); // null column hack
```

SQLite in Android

Löschen eines Datensatzes

```
// fügt einen neuen Datensatz in die Datenbank
SQLiteDatabase db = mEventDatabase.getWritableDatabase();
db.delete(TABLE_NAME,
        "_id=?", // WHERE _ID =
         new String[]{"" +id }); // Bedingungen als String-Array
```

Ändern eines Datensatzes

```
public void updateEvent(String eventText, long id) {
   // fügt einen neuen Datensatz in die Datenbank
   SQLiteDatabase db = mEventDatabase.getWritableDatabase();
   // Neuer Wert
   ContentValues values = new ContentValues();
   values.put(TIME, System.currentTimeMillis());
   values.put(EVENT, eventText);
   // Update
   db.update(TABLE_NAME, values,
             "_id=?", // WHERE _ID =
             new String[]{"" +id }); // Bedingungen als String-Array
}
```

Transaktionen

Eine Transaktion wird im Erfolgsfall abgeschlossen. Im Fehlerfall wird die Datenbank wieder auf den Zustand vor dem Transaktionsbeginn zurückgesetzt. Der Aufbau hat die folgende Gestalt:

```
db.beginTransaction();
try {
   …
   Folge von DB-Änderungen als atomare Einheit
   …
   db.setTransactionSuccessful(); // commit
} finally {
  db.transActionEnd();// Transaktion auf jeden Fall beenden
}
```

SQLite in Android

Anfragen

Verbindung zum Datenbank-Helfer

```
SQLiteDatabase db = mEventDatabase.getReadableDatabase();
```

Anfragen

Drei Möglichkeiten

- ***query***: **`Cursor ergebnis = db.query(anfrageparameter)`**
  Kaum Kenntnisse von SQL nötig, einfach zu verwenden.
- ***rawQuery***:
  `Cursor ergebnis = rawQuery(SQL-Anweisung, Arguments)`
  SQL-Anweisung als Zeichenkette ohne ein abschließendes Semikolon, geeignet für komplexe Anfragen.
- Mit ***SQLiteQueryBuilder***: Für komplexe Anfragen (z.B. mit **joins**) ist ein Objekt von diesem Typ hilfreich.

SQLite in Android

- **query-Methode**

Es sind verschiedene Formen. Hier wird nur eine gängige betrachtet:

```
public Cursor query (
 boolean distinct,        // true => Duplikate werden eliminiert
 String table,            // Tabellennamen
 String[] columns,        // Projektion – Spaltennamen der Anfrage
 String selection,        // Bedingung für die Anfrage „?“ als Platzhalter
 String[] selectionArgs,// Arguments (ersetzt ? in selection)
 String groupBy,          // Spaltenname zur Gruppierung (SQL GROUP BY)
 String having,           // Gruppenbedingung
 String orderBy,          // Sortierordnung (SQL ORDER BY-Klausel)
 String limit)            // Max. Anzahl der zurückgelieferten Datensätze
```

SQLite in Android

Cursor

- Anfrage liefert ein Objekt vom Typ Cursor zurück.
- Ein Cursor ist ein Zeiger auf einen Datensatz.
- Mit dem Cursor kann man
  - innerhalb der Ergebnismenge navigieren,
  - die Datensätze auslesen,
  - die Anzahl der zurückgelieferten Datensätze bestimmen (`getCount`).

Cursor-Navigation

```
boolean moveToFirst(), boolean moveToNext(),
boolean moveToPrevious(), boolean moveToLast(),
boolean moveToPosition(int)
```

Cursor-Navigation

```
boolean isFirst(), boolean isLast(),

boolean isBeforeFirst(), boolean istAfterLast(),

int getPosition()
```

Einlesen der Datensätze

Um den Wert einer Spalte des Datensatzes der aktuellen Cursorposition auszulesen, werden benötigt:

- der Datentyp der Spalten
- die Spaltennummer.

Dazu wird die Methode
`int getColumnIndex(String colName)`
verwendet.

SQLite in Android

Einlesen der Datensätze

Beispiel:

```
int index = getColumnIndex(Colums.AUTHOR);

String author = cur.getString(index);
```

Typische Schleife:

```
if (cur.moveToFirst()) {
   int nameIdx = …. // Notwendige Spaltenindexen lesen
   while (! cur.isAfterLast()) {
       … // Daten einlesen
       cur.moveToNext();
   }
}
```

SQLite in Android

Loader und LoaderManager

- Eingeführt ab Honeycomb (Android 3)
- Ziel: asynchrones Laden und "Caching"/Verwaltung der Daten.
- Jede Activity/jedes Fragment hat einen Loadermanager.
- Ladevorgang wird in einem **separaten Thread** ausgeführt.
- Um einen asynchronen Ladevorgang zu starten, wird `initLoader` aufgerufen

```
getLoaderManager().initLoader(0, null, this);
```

`0` eindeutige ID des Loaders

`null` Argument für den Loader – hier nicht verwendet

`this` LoaderCallbacks wird vom LoaderManager aufgerufen.

SQLite in Android

LoaderCallbacks<T>

- wird vom LoaderManager asynchron im Main-Thread-Kontext aufgerufen
- zu implementierende Methoden

```
public Loader<T> onCreateLoader(int id, Bundle args);
public void onLoadFinished(Loader<T> loader, T data);
public void onLoaderReset(Loader<T> loader)
```

Häufig wird `CursorLoader` verwendet (=> siehe Beispiel).