

Mobile Programming

# Android 3 – Persistenz & Content Providers



Nicola Keller



# Inhalt

## ■ Lokale Persistenz

- Shared Preferences
- App spezifischer Speicher (intern + extern)
- Scoped Storage
- Datenbank (Room DB)

## ■ Content Providers

- Daten teilen: z.B. Contacts, SMS, Kalender, ...
- Intern oder mit anderen Apps

## ■ Permissions

- Aktionen, welche die Erlaubnis des Benutzers erfordern

# Lokale Persistenz: 3 Möglichkeiten

## ■ Preferences

- Schlüssel/Werte-Paare (Key, Value)
- Verwendung für kleine Datenmengen

## ■ Dateisystem, intern oder extern (SD-Karte)

- In App-Sandbox (privat) oder Scoped Storage (öffentlich)
- Verwendung für binäre Daten, grosse Dateien, Export

## ■ Datenbank (Room)

- SQLite + Object Relational Mapper (ORM)
- Verwendung für strukturierte Daten + Abfragen/Suche

Persistenz = Daten über Laufzeit der App erhalten

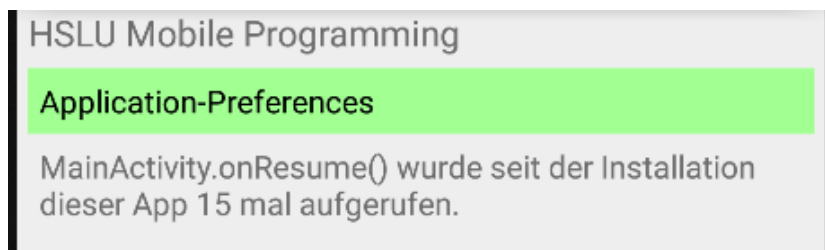
Lokal, daher ist z.B. Web-Storage (Cloud, Backend, etc.) noch kein Thema. Backend-Kommunikation schauen wir später im Modul an...



# Preferences

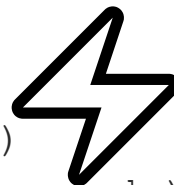
# Demo: Zustand persistieren

- Wir persistieren Anzahl Aufrufe von `onResume()` über die Lebenszeit der App hinaus



# SharedPreferences

- Persistente Einstellungen für Activity oder Applikation
  - Key-Value-Store («persistente Map»)
  - Für Applikation:
    - `activity?.getSharedPreferences (getString(R.string.preference_file_key), Context.MODE_PRIVATE)`
  - Preferences für Activity:
    - `activity?.getPreferences (Context.MODE_PRIVATE)`
    - Intern `getSharedPreferences (getLocalClassName(), mode);`
- Mögliche Datentypen für Preferences-Werte
  - String, Int, Long, Float, Boolean
  - Set<String> (mit separaten Werten)



# SharedPreferences: lesen & schreiben

- Mehrere Dateien pro Applikation sind möglich
  - Zugriff: `requireActivity.getSharedPreferences(name, mode)`
- Lesen
  - Methoden `preferences.getX()`
- Schreiben (immer mit Editor)
  1. `editor = preferences.edit()`
  2. `editor.putX(...)`
  3. `editor.apply()`
    - Persistiert asynchron, d.h. nicht-blockierende Methode
    - Falls synchron (blockierend) gewünscht (**Nie auf dem Main Thread**): `editor.commit()`

Unterschiedliche  
Datei-Namen

X = Typ, also z.B.  
String, Int,  
Boolean, ...

Damit werden  
Änderungen persistiert

# Demo: Zustand persistieren

- Wir persistieren Anzahl Aufrufe von onResume ( ) über die Lebenszeit der App hinaus
- Anzeige im Fragment

```
val preferences =  
    requireActivity().getSharedPreferences(SHARED_PREFERENCES_OVERVIEW,  
    Context.MODE_PRIVATE)  
val newResumeCount = preferences.getInt(COUNTER_KEY, 0) + 1  
val editor = preferences.edit()  
editor.putInt(COUNTER_KEY, newResumeCount)  
editor.apply()
```



# Demo: Preferences mit XML

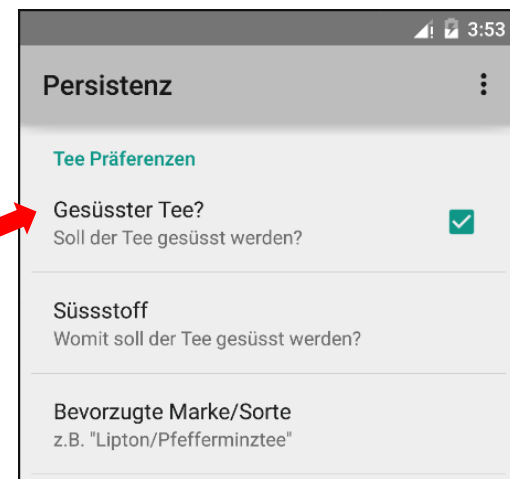
- Fragment für Tee-Präferenzen vom Benutzer:

- Boolean: „Mit Süsstoff?“
- List: „Süsstoff“
  - (Assugrin, Kristallzucker, Rohrzucker)
- String: „Bevorzugte Marke/Sorte“

- Preference-Bildschirm definieren in xml
  - `res/xml/preferences.xml`

Editierbarkeit abhängig vom Wert von „Mit-Süsstoff“

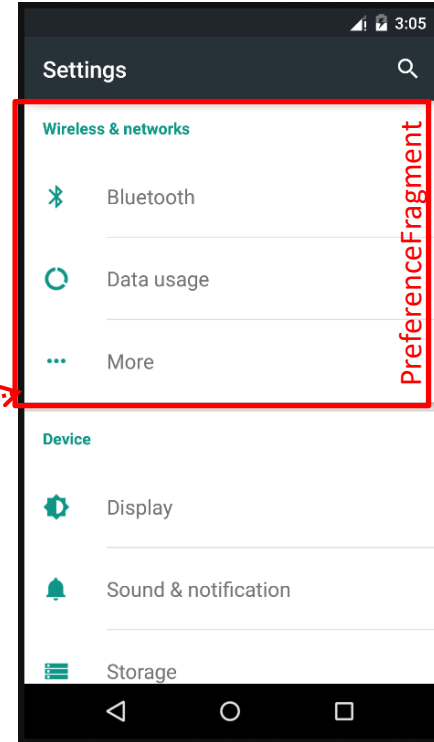
Ausgegraut wegen  
`android:dependency="teaWithSugar"`



```
<ListPreference
    android:dependency="teaWithSugar"
    android:entries="@array/teaSweetener"
```

# User-Preferences: Darstellung

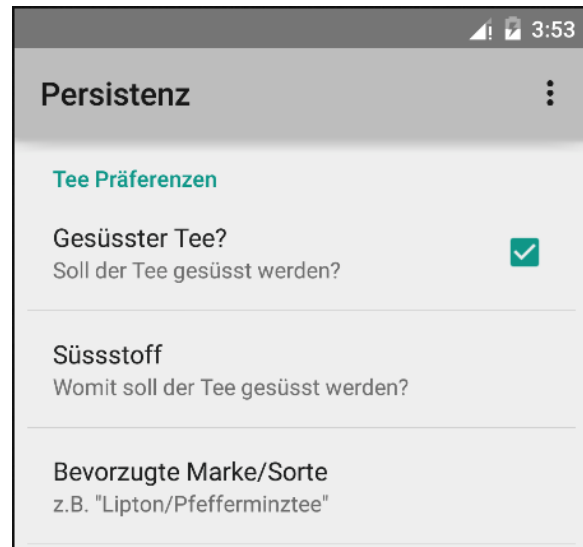
- Bekannt aus der Settings-App
  - Diesen Mechanismus können wir auch verwenden!
- Deklaration im XML = «Screen»
  - Darstellung „automatisch“ mit Hilfe von `PreferenceFragmentCompat`
  - Jeder Wertetyp hat eigenen Editor
- `PreferenceFragmentCompat` schreibt/liest grundsätzlich die `DefaultSharedPreferences`
  - Kann aber für anderen Preference-Store konfiguriert werden



# User-Preferences: Beispiel „Tee Präferenzen“

- Benutzer soll angeben...
  - Gesüsst: Ja / Nein (boolean)
    - `CheckBoxPreference`
  - Süsstoff: Auswahl aus Liste (string-array)
    - `ListPreference`
  - Bevorzugte Marke: Freitext (string)
    - `EditTextPreference`

☞ siehe XML-Datei nächste Folie

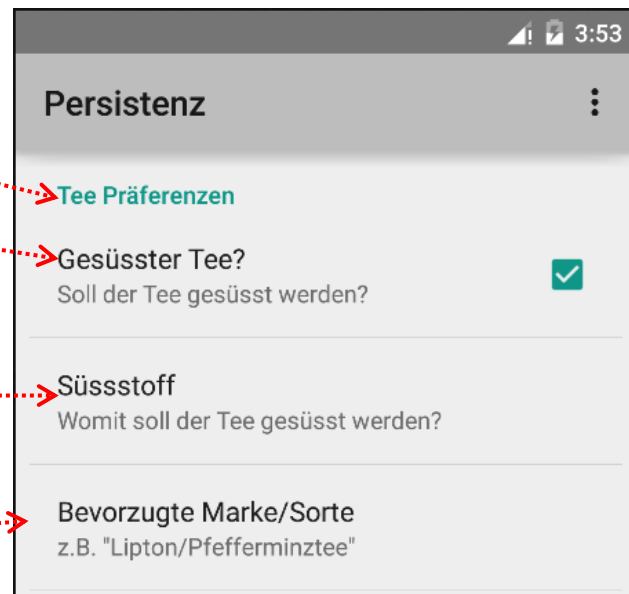


# User-Prefs: Deklaration

## ■ Deklaration von User-Preferences in XML

- Im Ordner `res/xml`, z.B. Datei `preferences.xml`

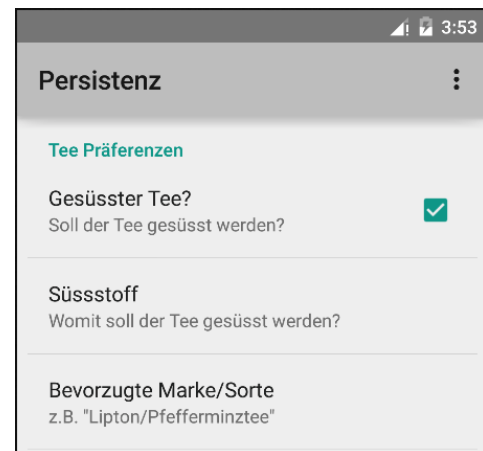
```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
  <PreferenceCategory
    android:key="teaPrefs"
    android:title="Tee Präferenzen">
    <CheckBoxPreference
      android:key="teaWithSugar"
      android:persistent="true"
      android:summary="Soll der Tee gesüsst werden?"
      android:title="Gesüsster Tee?" />
    <ListPreference
      android:dependency="teaWithSugar"
      android:entries="@array/teaSweetener"
      android:entryValues="@array/teaSweetenerValues"
      android:key="teaSweetener"
      android:persistent="true"
      android:shouldDisableView="true"
      android:summary="Womit soll der Tee gesüsst werden?"
      android:title="Süsstoff" />
    <EditTextPreference
      android:key="teaPreferred"
      android:persistent="true"
      android:summary="z.B. "Lipton/Pfefferminztee""
      android:title="Bevorzugte Marke/Sorte" />
  </PreferenceCategory>
</PreferenceScreen>
```



# User-Prefs: Erzeugung PreferenceFragmentCompat

- Als Beispiel hier im TeaPreferenceFragment

```
class TeaPreferenceFragment : PreferenceFragmentCompat() {  
  
    companion object {  
        fun newInstance(): TeaPreferenceFragment {  
            return TeaPreferenceFragment()  
        }  
    }  
  
    override fun onCreatePreferences(savedInstanceState: Bundle?, rootKey: String?) {  
        setPreferencesFromResource(R.xml.preferences, rootKey)  
    }  
}
```



Referenziert `res/xml/preferences.xml` aus voriger Folie

# Hinweis: Daten für ListPreference aus Arrays

- `res/xml/preferences.xml`

```
<ListPreference
    android:dependency="teaWithSugar"
    android:entries="@array/teaSweetener"
    android:entryValues="@array/teaSweetenerValues"
    android:key="teaSweetener"
    android:persistent="true"
    android:shouldDisableView="true"
    android:summary="Womit soll der Tee g
    android:title="Süsstoff" />
```

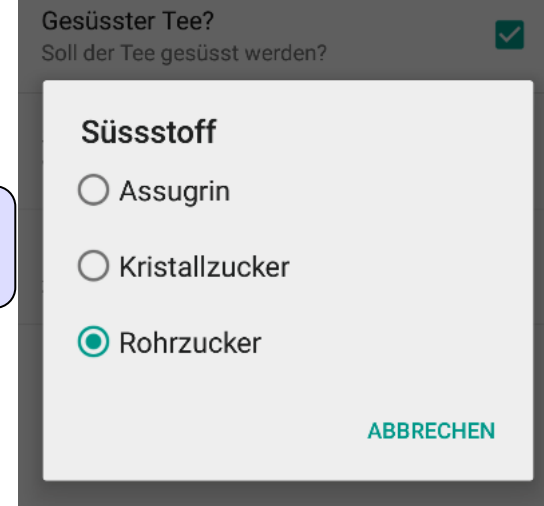
Entry = „Anzeigestring“,  
Übersetzbar

EntryValue = „Werte“,  
nicht übersetzt =  
technischer Key

- `res/values/arrays.xml`:

```
<resources>
    <string-array name="teaSweetenerValues">
        <item>artificial</item>
        <item>refined</item>
        <item>natural</item>
    </string-array>

    <string-array name="teaSweetener">
        <item>Assugrin</item>
        <item>Kristallzucker</item>
        <item>Rohrzucker</item>
    </string-array>
</resources>
```



# Demo: Präferenzen im Dateisystem inspizieren

Default Shared Preferences von dieser App, im Ordner /data/data/shared\_prefs

```
<map>
  <string name="teaPreferred">Hopfentee</string>
  <string name="teaSweetener">natural</string>
  <boolean name="teaWithSugar" value="true" />
</map>
```

# Demo: Tee-Präferenz programmatisch setzen

- Bei Klick auf Button sollen Tee-Präferenzen programmatisch auf fixe Werte gesetzt werden
  - Verwendung Default-Preferences:  
`PreferenceManager.getDefaultSharedPreferences(this)`  
dann: `Editor editor = prefs.edit()`, usw. (d.h. fixe Werte für die drei Einstellungen setzen und speichern)

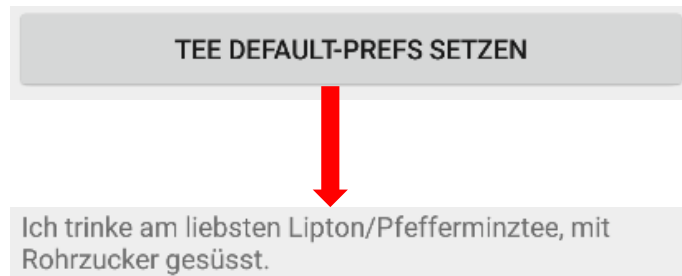


Diagram illustrating the XML representation of the preferences. A red double-headed arrow points from the text box in the previous diagram to this XML code block.

```
<map>
  <string name="teaPreferred">Lipton/Pfefferminztee</string>
  <string name="teaSweetener">natural</string>
  <boolean name="teaWithSugar" value="true" />
</map>
```





# App spezifischer Speicher

# App spezifischer Speicher

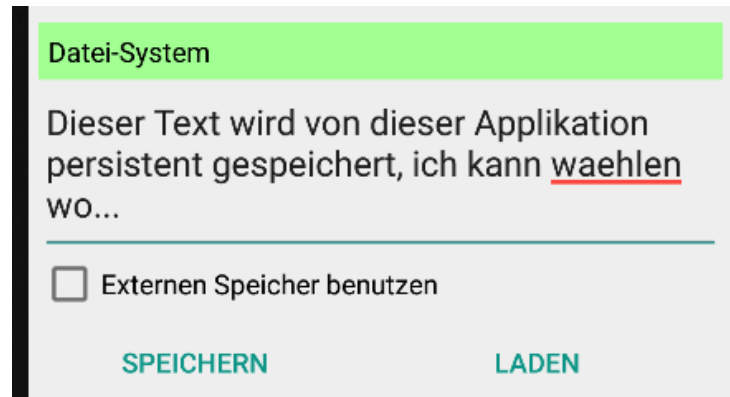
## – Interner Speicher

- persistente Daten
- Cache data
- Kein Zugriff durch andere Apps
- Ab Android 10 encrypted
- Deinstallieren der App entfernt alle Daten
- kleiner Speicher

## – Externer Speicher

- persistente Daten
- Cache data
- $\leq$  Android 9 andere App hat mit entsprechender Permission Zugriff
- Ab Android 10 haben andere Apps keinen Zugriff mehr (verwende scoped Storage)
- Deinstallieren der App entfernt alle Daten
- grosser Speicher

# Demo: Persistenz mit Datei



The screenshot shows a dialog box titled "Datei-System" in a green header bar. Below the title, the text reads: "Dieser Text wird von dieser Applikation persistent gespeichert, ich kann wählen wo...". The word "wählen" is underlined in red. A horizontal line separates this text from a checkbox labeled "Externen Speicher benutzen". At the bottom of the dialog, there are two buttons: "SPEICHERN" on the left and "LADEN" on the right, both in teal text.

# Repetition: Streams, Reader & Co

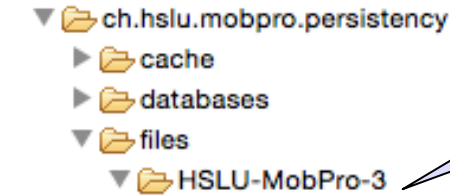
- Stream: Byte Datenstrom `[28, 11, 200, 255, 2, 15, 33]`
  - Auf File öffnen
    - `FileOutputStream, FileInputStream`
- Stream kann in Zeichenstrom `['h', 'a', 'l', 'l', 'o']` umgewandelt werden
  - `FileReader, FileWriter` + „Buffered“-Versionen
- Immer schliessen!
  - `stream.close(), reader.close()`
- Nicht vergessen: `try-catch-finally` implementieren

# Demo: Persistenz mit Datei

## ■ Text persistent speichern

```
var writer: Writer? = null
try {
    writer = BufferedWriter(FileWriter(getFile(useExtStorage)))
    writer.write(text)
    return true
} catch (ex: IOException) {
    //...
} finally {
    //...
}
```

## ■ Anschauen im „File Explorer“ („Android Device Monitor“, für Emulator)



### Datei-System

Dieser Text wird von dieser Applikation persistent gespeichert, ich kann wählen wo...

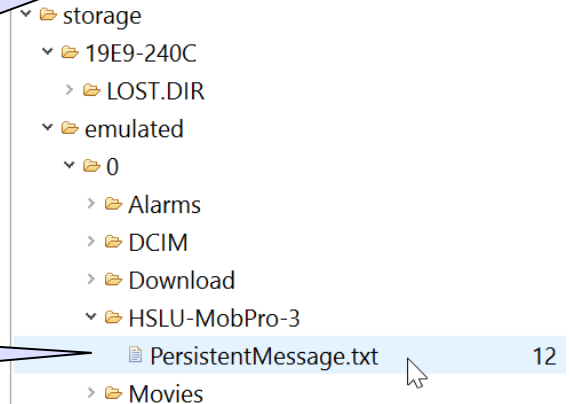
☐ Externen Speicher benutzen

SPEICHERN

LADEN

Pfad im Applikationsverzeichnis  
(d.h. privater Speicher)

Pfad auf emulierter SD-Karte  
(d.h. «externer»,  
öffentlicher Speicher)



# Zugriff Android Dateisystem

## ■ Grundsätzliche Unterscheidung: Dateien sind...

### – Interner Speicher

- `Context.getFilesDir()`

Für Emulator folgender Befehl im Terminal eingeben

```
adb shell sm set-virtual-disk true
```

### – Externer Speicher

- `context.getExternalFilesDir(null);`
- $\leq$  Android 9 ( $\geq$  4.4), können andere App mittels Permission auf diesen Speicher zugreifen
- $\geq$  Android 10 ist es nicht mehr möglich diesen Speicher zu lesen, dafür sollte Scoped Storage verwendet werden.

# SD Karte

- Es gibt Geräte mit der Option den Speicher mit einer SD Karte zu erweitern
- Das Gerät hat somit mehrere physikalische Volumen, welche als externen Speicher verwendet werden können.
- ```
val sdCardFolder =  
context.getExternalFileDirs (null)  
    .firstOrNull { it ->  
        Environment.isExternalStorageRemovable (it)  
    }
```



# Scoped Storage



# Scoped Storage

- Hier speichern wir Dateien, welche auch über andere App verwendet werden können
- Wird nicht gelöscht, wenn die App deinstalliert wird
- Folgende API existieren <https://developer.android.com/training/data-storage/shared>
  - Media content (Musik, Photos, ...)
  - Documents and other files (PDF, .txt , ..)
  - Datasets ab Android 11 → grosse Datensets, welche mehrere Apps nutzen können.



## Persistenz: Datenbank (Room)

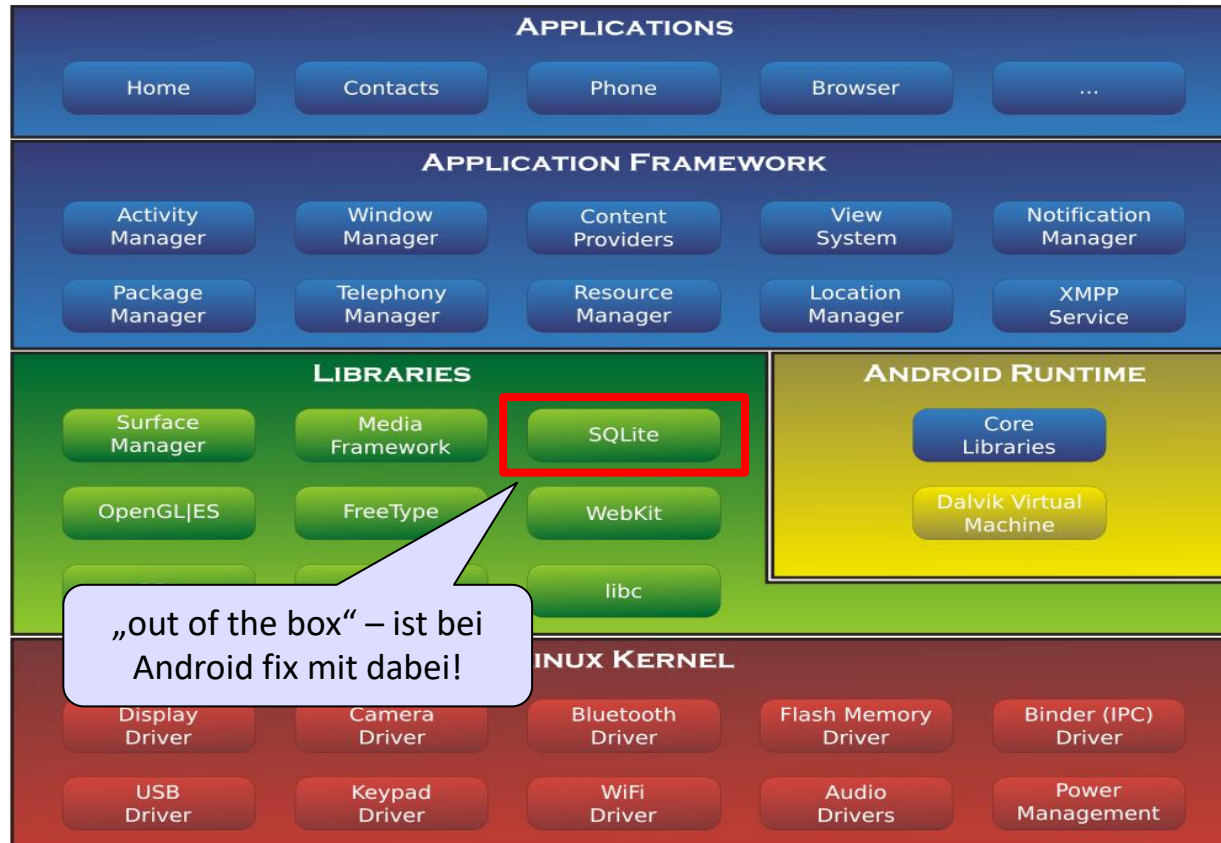
# SQLite + Room Database

Teil 1: «Low Level»,  
Abstraktionslayer selber  
implementieren

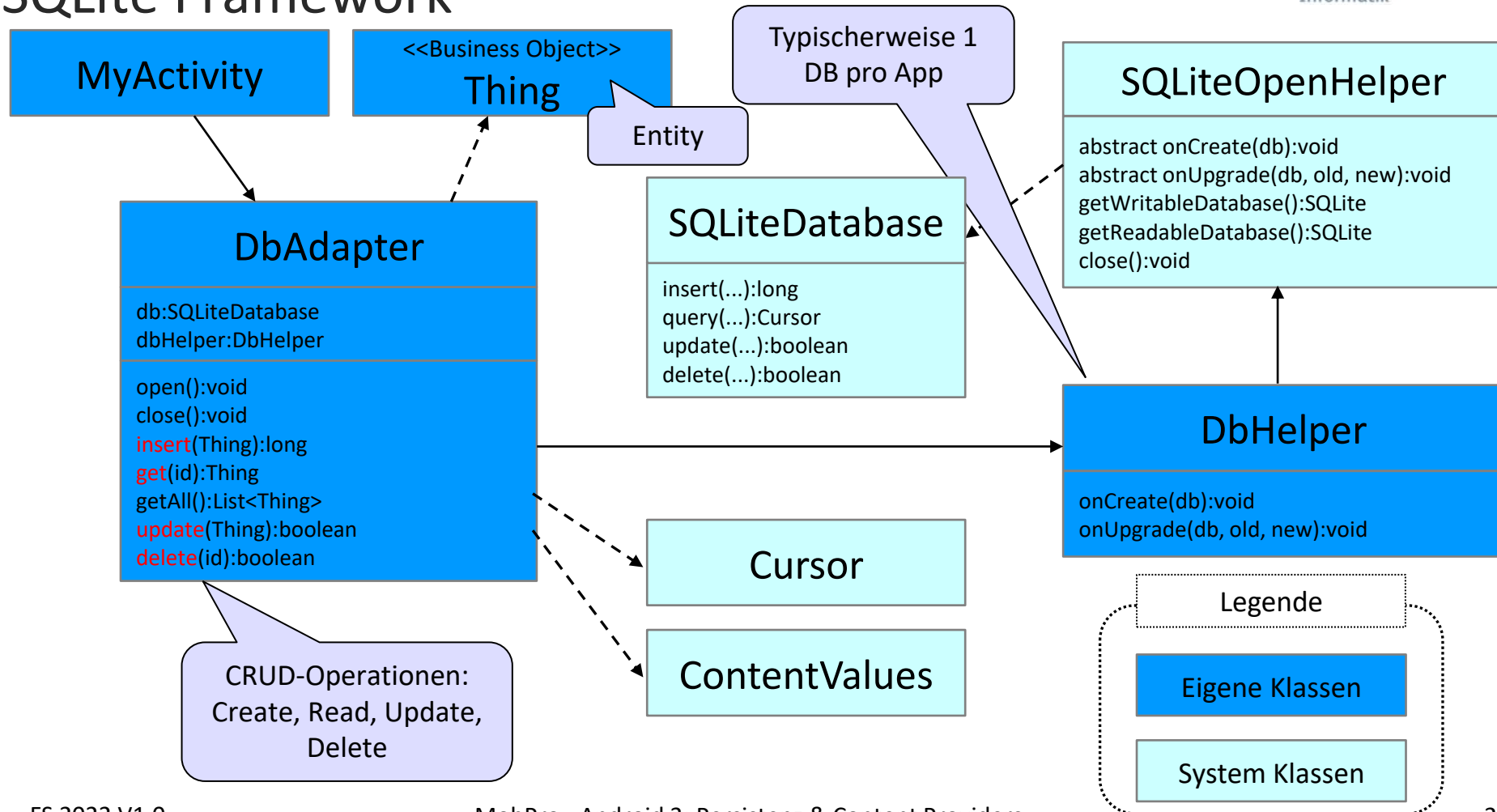
- SQLite: Relationales DBMS (Data Base Management System)
  - Optimiert für embedded/mobile, Teil von Android
  - Open Source, <http://www.sqlite.org>
  - Pro Applikation n Datenbanken (DB) = 1 Datei pro DB
- Room
  - Abstraktionslayer über SQLite
  - Mappt OO-Entities auf relationale Tabellen
  - Verwendet DAO-Pattern (Data Access Object)

Teil 2: Abstraktionslayer mittels  
Room, viel einfacher!

# Die Android-DB: SQLite



# SQLite Framework



# Die harte Tour: SQLite in Rohform

## Save data using SQLite

Saving data to a database is ideal for repeating or structured data, such as contact information. This page assumes that you are familiar with SQL databases in general and helps you get started with SQLite databases on Android. The APIs you'll need to use a database on Android are available in the `android.database.sqlite` package.



**Caution:** Although these APIs are powerful, they are fairly low-level and require a great deal of manual work.

- There is no compile-time verification of raw SQL queries. As your data graph grows, you may find yourself writing and testing SQL queries manually. This process can be time consuming and error prone.
- You need to use lots of boilerplate code to convert between SQL queries and data objects.

Wer's gerne aufwändig und fehleranfällig mag...

For these reasons, we **highly recommended** using the Room Persistence Library as an abstraction layer for accessing information in your app's SQLite databases.

# Room: Ein objektrelationaler Mapper

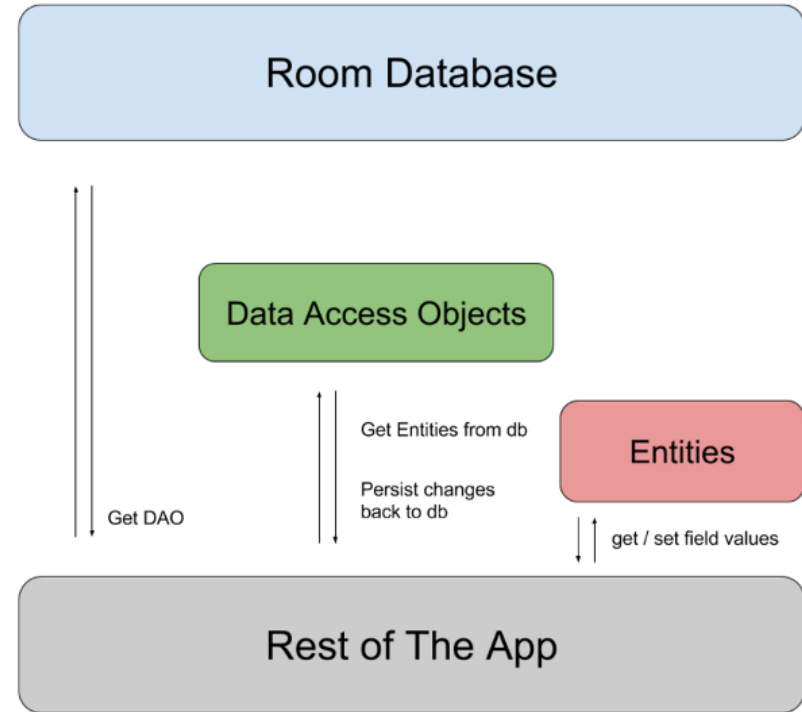
- Room ist ein ORM (Object Relational Mapping) für Android
  - Klassen werden auf relationale DB-Tabellen gemappt
  - Zugriff auf Datenbank wird abstrahiert
- Spezialfälle des *Room* ORM:
  - Datenzugriff über DAO: Queries werden als SQL-Statements in Annotationen definiert
  - Beziehungen zwischen Entitäten müssen manuell abgebildet werden (Performance!)
  - Nested Objects: Mehrere POJOs (Plain Old Java Object) in einer Tabelle
  - Einschränkungen für Datenzugriffe

Typischerweise werden SQL-Statements durch Methodenaufrufe gekapselt

Standardmässig nicht möglich im UI Thread. Nebenläufigkeit folgt!

# Die drei Room-Komponenten

- **Database**  
Abstraktion der Datenbankverbindung
- **Entity**  
Repräsentation einer Tabelle in der relationalen DB
- **DAO**  
(Data Access Object)  
Enthält Methoden für Datenzugriff





# Room – Code-Beispiele

```
@Dao
interface UserDao {
    @Query("SELECT * FROM user")
    fun getAll(): List<User>

    @Query("SELECT * FROM user WHERE uid IN (:userIds)")
    fun loadAllByIds(userIds: IntArray): List<User>

    @Query("SELECT * FROM user WHERE first_name LIKE :first AND " +
        "last_name LIKE :last LIMIT 1")
    fun findByName(first: String, last: String): User

    @Insert
    fun insertAll(vararg users: User)

    @Delete
    fun delete(user: User)
}
```

```
@Entity
data class User(
    @PrimaryKey val uid: Int,
    @ColumnInfo(name = "first_name") val firstName: String?,
    @ColumnInfo(name = "last_name") val lastName: String?
)
```

Entity: POJO mit  
Annotationen

DAO: Datenzugriff über Annotationen  
(teilweise mit SQL-Queries)

# Room – Code-Beispiele

Database: Subklasse von *RoomDatabase*, konfiguriert mit *Database* Annotation

Version ist wichtig für Migration!

```
@Database(entities = arrayOf(User::class), version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}
```

Eine Instanz der DB erzeugen

```
val db = Room.databaseBuilder(
    applicationContext,
    AppDatabase::class.java, "database-name"
).build()
```

# Daten mit Entitäten definieren

- POJO mit `@Entity` Annotation
- Primärschlüssel (wird in jeder Entität benötigt)
  - `@PrimaryKey` für einzelnes Feld...
  - ... optional mit `autoGenerate` Property
  - Für zusammengesetzte Primärschlüssel: `primaryKey` Property in `@Entity` Annotation
- Falls bestimmte Felder nicht gespeichert werden sollen
  - `@Ignore` Annotation für einzelnes Feld
  - Mit `ignoredColumns` Property in `@Entity` Annotation für mehrere Felder (v.a. von Superklassen)

# Code-Beispiel

```
@Entity(  
    primaryKeys = { "firstName", "lastName" },  
    ignoredColumns = "password, otherField"  
)  
public class User : Party {  
    @PrimaryKey(autoGenerate = true)  
    var id: Int = 0  
    var firstName: String? = null  
    var lastName: String? = null  
    @Ignore  
    var picture: Bitmap? = null  
}
```

## Achtung!

Dieses Code-Beispiel definiert  
mehrere Primärschlüssel  
und vermischt Ansätze zum  
Ignorieren von Feldern  
zwecks Syntax-Demonstration!

# Mit DAOs auf Daten zugreifen

- Data Access Objects (DAOs) enthalten Methoden für den abstrahierten Datenbankzugriff
- Dies trägt zur *Separation of Concerns* bei und erhöht die Testbarkeit (DAOs können gemockt werden)
- DAOs werden als Interfaces oder abstrakte Klassen definiert → Room erzeugt passende Implementationen bei der Kompilierung!
- Zwei Möglichkeiten:
  - Convenience queries
  - **@Query** Annotation mit SQL-Statements

Typischerweise eine DAO-Klasse pro Entity, mit allen möglichen Operationen

# Convenience Queries

- Werden über Annotations für die jeweiligen Methoden definiert: `@Insert`, `@Update`, `@Delete`
- Alle Parameter müssen Klassen mit einer `@Entity` Annotation (oder Collections/Arrays) davon sein
- Rückgabewerte
  - Insert: `long` bzw. `long[]` bzw. `List<Long>`
  - Update / Delete: `int`

Liefert Row-Id(s) zurück

Anzahl modifizierte Tabelleneinträge

`@Insert`

```
fun insertUsersAndFriends(user: User, friends: List<User>): long[]
```

Parameter für Operation (Entities)

Optional: ID Rück-gabe  
(sonst void)

# Convenience Queries: Weitere Beispiele

```
@Dao
interface MyDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun insertUsers(vararg users: User)

    @Insert
    fun insertBothUsers(user1: User, user2: User)

    @Insert
    fun insertUsersAndFriends(user: User, friends: List<User>)

    @Update
    fun updateUsers(vararg users: User)

    @Delete
    fun deleteUsers(vararg users: User)
}
```

# Custom Queries mit @Query

- Die @Query Annotation kann für Schreib- und Lesevorgänge genutzt werden
- Jede @Query wird zur Kompilierzeit überprüft  
→ Kompilierfehler bei ungültigen Queries
- Für eine @Query kann eine beliebige Anzahl (0..n) Parameter verwendet werden
- Wenn nicht ganze Objekte benötigt werden, können durch die Verwendung von POJOs mit @ColumnInfo Annotationen Ressourcen gespart werden



# Custom Queries: Codebeispiele

```
@Dao
interface MyDao {
    @Query("SELECT * FROM user")
    fun loadAllUsers(): Array<User>

    @Query("SELECT * FROM user WHERE age > :minAge")
    fun loadAllUsersOlderThan(minAge: Int): Array<User>

    @Query("SELECT first_name, last_name FROM user WHERE region IN (:regions)")
    fun loadUsersFromRegions(regions: List<String>): List<NameTuple>
}
```

```
data class NameTuple(
    @ColumnInfo(name = "first_name") val firstName: String?,
    @ColumnInfo(name = "last_name") val lastName: String?
)
```

```
@Dao
interface MyDao {
    @Query("SELECT first_name, last_name FROM user")
    fun loadFullName(): List<NameTuple>
}
```

# Beziehungen modellieren

Define relationships between objects

Performanzgründe

Because SQLite is a relational database, you can specify relationships between objects. Even though most object-relational mapping libraries allow entity objects to reference each other, Room explicitly forbids this. To learn about the technical reasoning behind this decision, see [Understand why Room doesn't allow object references](#).

## ■ Nested Objekte

```
data class Address(  
    val street: String?,  
    val state: String?,  
    val city: String?,  
    @ColumnInfo(name = "post_code") val postCode: Int  
)
```

```
@Entity  
data class User(  
    @PrimaryKey val id: Int,  
    val firstName: String?,  
    @Embedded val address: Address?  
)
```

In der DB werden die Felder von Address gespeichert

# 1-1 Beziehungen

```
@Entity
data class User(
    @PrimaryKey val userId: Long,
    val name: String,
    val age: Int
)

@Entity
data class Library(
    @PrimaryKey val libraryId: Long,
    val ownerId: Long
)
```

Referenz zum  
anderen PrimaryKey

## Liste mit allen User und zugehöriger Library ?

```
data class UserAndLibrary(
    @Embedded val user: User,
    @Relation(
        parentColumn = "userId",
        entityColumn = "userOwnerId"
    )
    val library: Library
)
```

```
@Transaction
@Query("SELECT * FROM User")
fun getUsersAndLibraries():
    List<UserAndLibrary>
```

# 1-\* Beziehungen

Liste mit allen User und  
zugehörigen Playlists ?

```
@Entity
data class User(
    @PrimaryKey val userId: Long,
    val name: String,
    val age: Int
)

@Entity
data class Playlist(
    @PrimaryKey val playlistId: Long,
    val userCreatorId: Long,
    val playlistName: String
)
```

```
data class UserWithPlaylists(
    @Embedded val user: User,
    @Relation(
        parentColumn = "userId",
        entityColumn = "userCreatorId"
    )
    val playlists: List<Playlist>
)
```

```
@Transaction
@Query("SELECT * FROM User")
fun getUsersWithPlaylists():
    List<UserWithPlaylists>
```

# \*-\* Beziehungen

```
@Entity
data class Playlist(
    @PrimaryKey val playlistId: Long,
    val playlistName: String
)

@Entity
data class Song(
    @PrimaryKey val songId: Long,
    val songName: String,
    val artist: String
)

@Entity(primaryKeys = ["playlistId",
"songId"])
data class PlaylistSongCrossRef(
    val playlistId: Long,
    val songId: Long
)
```

# \*-\* Beziehungen

```
data class PlaylistWithSongs(  
    @Embedded val playlist: Playlist,  
    @Relation(  
        parentColumn = "playlistId",  
        entityColumn = "songId",  
        associateBy = @Junction(PlaylistSongCrossRef::class)  
    )  
    val songs: List<Song>  
)  
  
data class SongWithPlaylists(  
    @Embedded val song: Song,  
    @Relation(  
        parentColumn = "songId",  
        entityColumn = "playlistId",  
        associateBy = @Junction(PlaylistSongCrossRef::class)  
    )  
    val playlists: List<Playlist>  
)
```

# \*\_\* Beziehungen

```
@Transaction
@Query("SELECT * FROM Playlist")
fun getPlaylistsWithSongs(): List<PlaylistWithSongs>

@Transaction
@Query("SELECT * FROM Song")
fun getSongsWithPlaylists(): List<SongWithPlaylists>
```

# DB-Einträge in einer Liste darstellen

- Verschiedene Möglichkeiten, je nach Umfang / Komplexität der Datensätze:
  - ListView (sehr simple Listen)
  - RecyclerView (siehe nächste Folie)  
<https://developer.android.com/guide/topics/ui/layout/recyclerview>
  - Auch in Kombination mit ViewModel und LiveData  
<https://codelabs.developers.google.com/codelabs/android-room-with-a-view>
- In jedem Fall werden spezifische Adapter benötigt, um die Daten auf Views zu mappen



# DB-Einträge in einer RecyclerView darstellen

```
class UserAdapter(private val dataSet: Array<Note>, private val onItemClickListener: OnItemClickListener) :  
    RecyclerView.Adapter< UserAdapter.UserViewHolder>() {  
  
    class UserViewHolder(val view: View) : RecyclerView.ViewHolder(view)  
  
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): UserViewHolder {  
        val view = LayoutInflater.from(parent.context).inflate(R.layout.item_layout, parent, false)  
        return UserViewHolder(view)  
    }  
  
    override fun onBindViewHolder(holder: UserViewHolder, position: Int) {  
        val note = dataSet.get(position)  
        holder.view.note_title.text = note.title  
        holder.view.note_content.text = note.text  
        holder.view.setOnClickListener { onItemClickListener.onUserClicked(note.id) }  
    }  
  
    override fun getItemCount(): Int {  
        return dataSet.size  
    }  
  
    interface OnItemClickListener {  
        fun onUserClicked(id: Long)  
    }  
}
```

Im Fragment

```
val recyclerView = note_recycler  
val notes = notesDb.noteDao().loadAllNotes()  
val notesAdapter = NotesAdapter(notes, this)  
recyclerView.adapter = notesAdapter
```

# Room: Weitere Themen

- Weitere Themen, die für Android Apps mit Room relevant sein könnten:
  - Queries in Klassen kapseln (Views)  
<https://developer.android.com/training/data-storage/room/creating-views>
  - Observable Queries mit Live Data  
<https://developer.android.com/training/data-storage/room/accessing-data#query-observable>
  - Datenbank migrieren (z.B. bei App Updates)  
<https://developer.android.com/training/data-storage/room/migrating-db-versions>
  - Datenbank testen  
<https://developer.android.com/training/data-storage/room/testing-db>
  - TypeConverter: Objekt-Referenzen in der Datenbank  
<https://developer.android.com/training/data-storage/room/referencing-data>

# Android Permission-Model



# Permissions

- Gewisse Operationen von Apps benötigen eine Permission, die vom Benutzer erteilt werden muss
  - Zugriff auf Kontakte, Internet, SD-Karte, Kamera, SMS, Telefonieren, Apps deinstallieren, etc.
- Erforderliche Permissions werden von der App im Manifest deklariert

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
    package="ch.hslu.mobpro.persistence"
    xmlns:android="http://schemas.android.com/apk/res/android">

    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.READ_SMS" />
    <uses-permission android:name="android.permission.WRITE_SMS" />

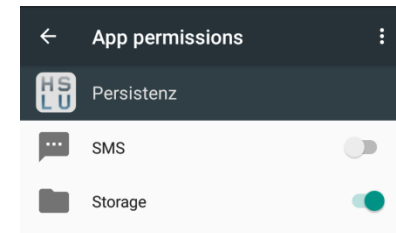
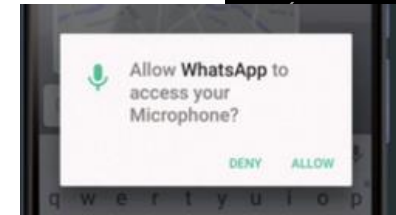
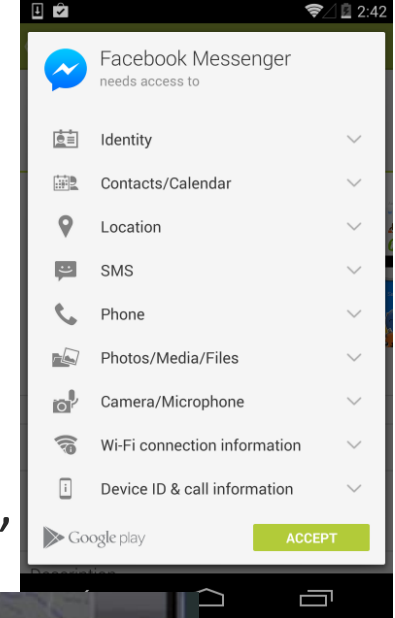
</manifest>
```

- Klasse `android.Manifest.permission`
  - Auflistung aller Permissions, Groups, Protection Level

Normal,  
Dangerous,  
Signature,  
System

# Permissions vor/nach Android 6

- In Android API < 23 (d.h. vor Android 6 / M) mussten alle Rechte vom Benutzer bei der Installation gewährt werden.
  - Alles oder nichts
- Seit API 23 werden bei der Installation keine 'dangerous' Permissions mehr gewährt (nur unkritische)
  - App muss für jede kritische Permission beim Benutzer nachfragen (wenn zum 1. Mal benötigt)
  - Permissions können einzeln abgelehnt oder entzogen werden (Settings > Apps > ... > Permissions)
  - Konsequenz: Apps müssen mit teilweise gewährten Permissions umgehen können!



# Permissions

## ■ Arten von Permissions

– normal, dangerous, signature

Wird bei  
Installation  
automatisch  
erlaubt

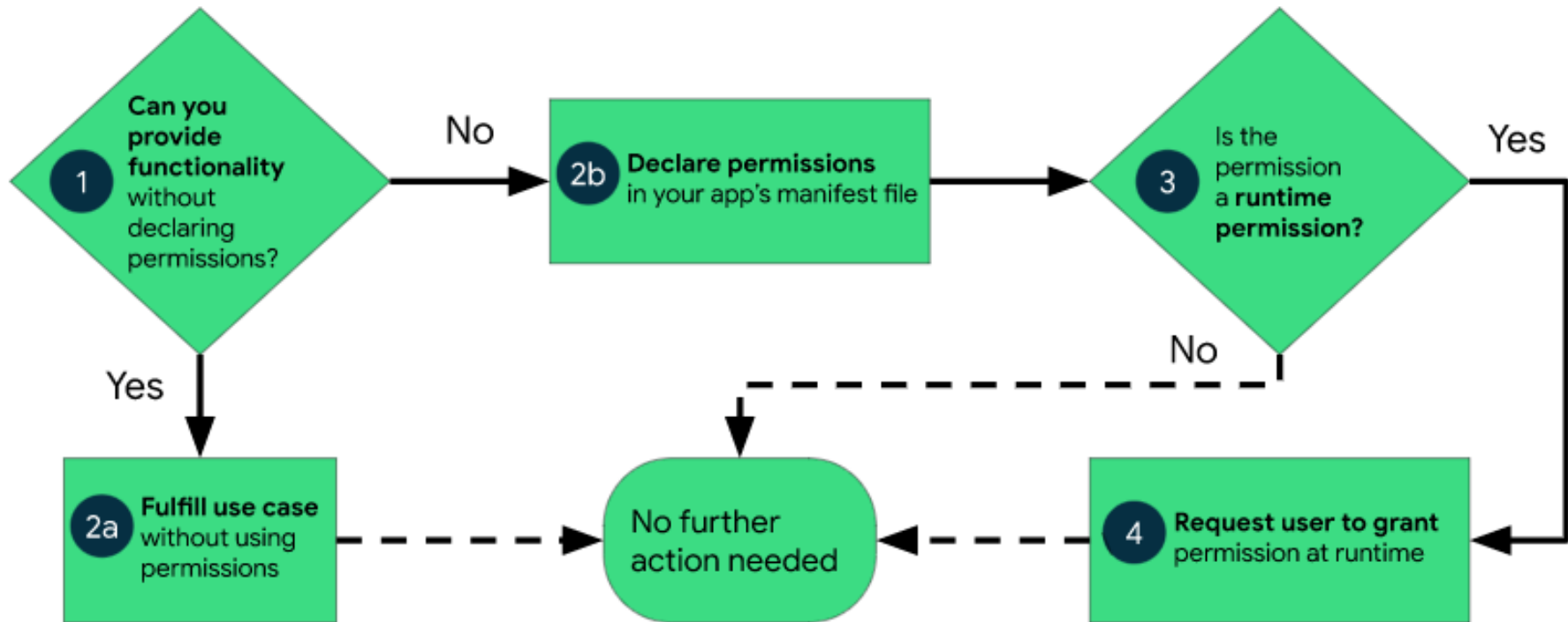
Muss von User erlaubt  
werden (und kann wieder  
entzogen werden)

Runtime Permissions

Wird automatisch erlaubt, wenn App, welche  
Permission definiert, von gleichem Hersteller,  
wie App, die Permission beanträgt. Sonst kann  
diese nicht erlaubt werden.

Liste mit allen «normal» Permissions (u.a. **INTERNET**, BLUETOOTH, NFC, VIBRATE, ...):  
<https://developer.android.com/guide/topics/permissions/normal-permissions.html>

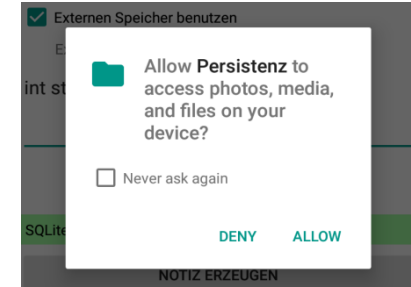
# Permissions



<https://developer.android.com/guide/topics/permissions/overview>

# Runtime Permissions

- Konzept: [.../guide/topics/security/permissions.html](http://developer.android.com/guide/topics/security/permissions.html)
- Howto: <http://developer.android.com/training/permissions/index.html>

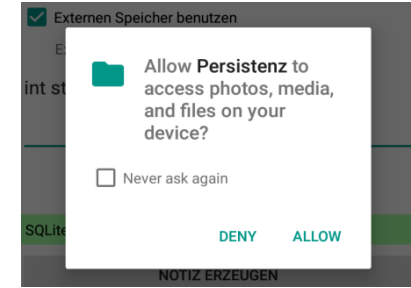


```
val granted =
    requireActivity().checkSelfPermission(Manifest.permission.WRITE_EXTERNAL_STORAGE)
if (granted != PackageManager.PERMISSION_GRANTED) {
    requestPermissionLauncher.launch(permissions)
} else {
    //call some method
}
```

Runtime-Check, ob benötigte  
Permission(s) vorhanden, sonst  
Permission(s) anfragen



# Runtime Permissions

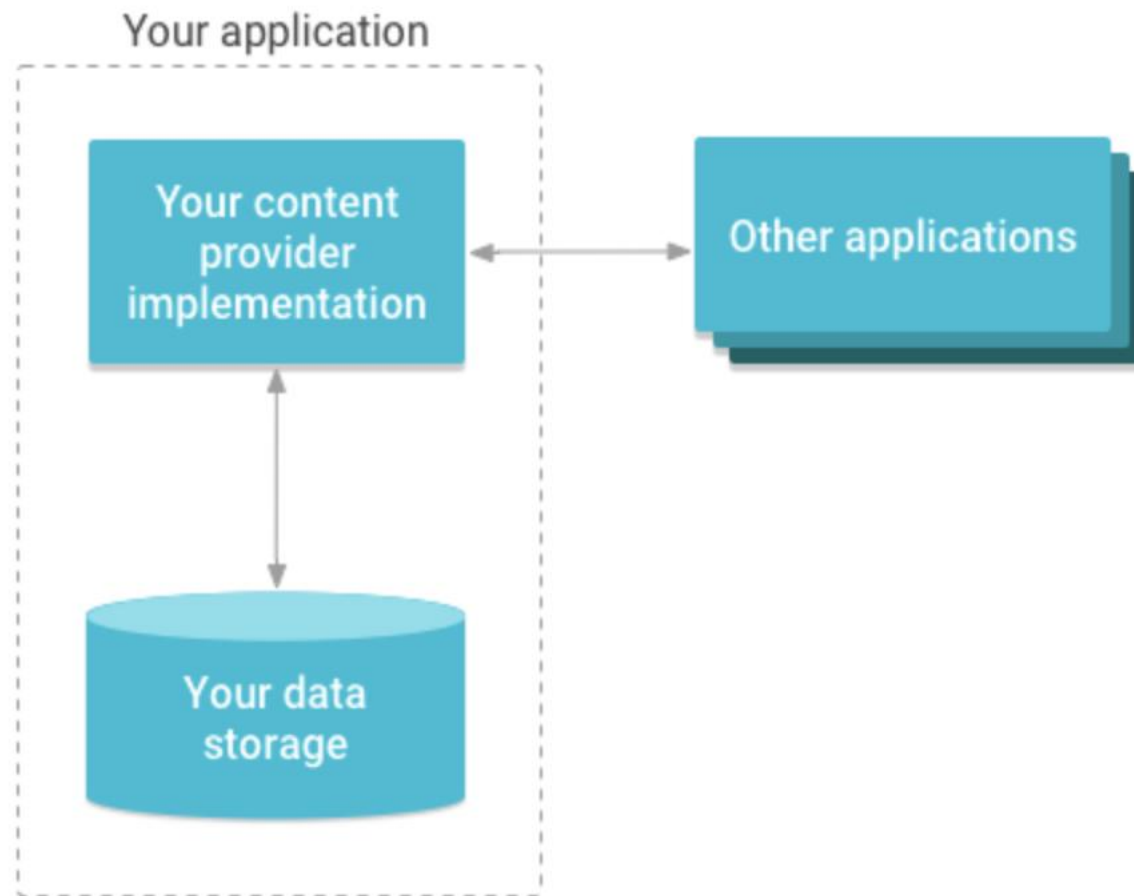


```
private val requestPermissionLauncher: ActivityResultLauncher<String> =
    registerForActivityResult(
        ActivityResultContracts.RequestPermission(),
        object : ActivityResultCallback<Boolean> {
            override fun onActivityResult(result: Boolean) {
                if (!result) {
                    Toast.makeText(context, "Permission denied!", Toast.LENGTH_SHORT).show()
                    return
                } else {
                    readSms()
                }
            }
        }
    )
```

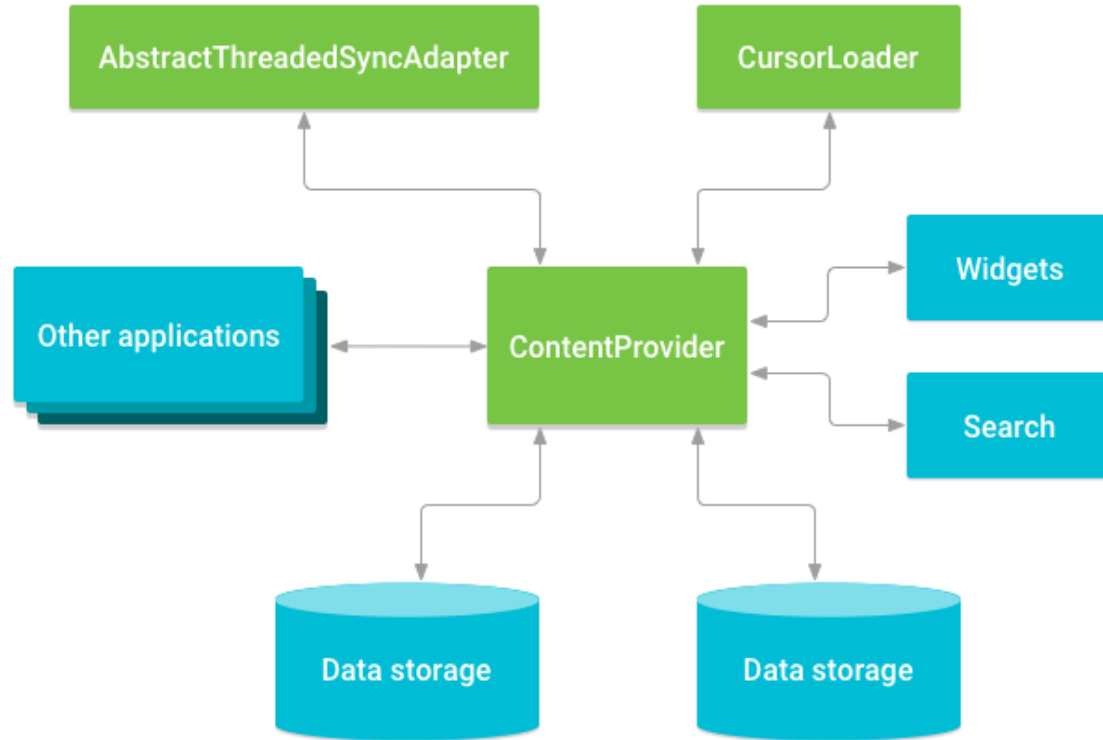
Callback aus Permission-Anfrage



# Content Providers



# Content Provider: Anwendung



# Content Provider

- Content Provider stellen für andere Applikationen Daten bereit
  - Daten stammen aus einer gekapselten DB oder aus dem privaten Dateisystem oder werden on-the-fly erzeugt
  - Zugriff auf Daten über URI (Uniform Resource ID), z.B.:

`content://hslu_notes/notes/4`

Scheme  
(immer gleich)

Authority  
(sollte FQN sein)

Path  
(Welche Daten)

Optional: Item  
(Datensatz)

(Fully Qualified Name)

- Zwei Arten von URIs:
  - Pfad (Bezeichnet Datenmenge, vgl. Verzeichnis mit Dateien)
  - Item (Einzelnes Datenelement, vgl. einzelne Datei)

# Standard Content Providers

- Im Android-System gibt es bereits einige Content Providers, die benutzt werden können:
  - Kontakte: Namen, Telefon-Nummern, Emails, Adressen, etc.
  - SMS/MMS: Erhaltene/Gesendete/Draft SMS/MMS
  - Media Store: Auf Device gespeicherte Audio-, Video-, Bilder-Daten
  - Settings: Einstellungen für das Gerät
  - Kalender: Kalender, Events, Erinnerungen, Teilnehmer, etc.
- Daten sind meist in mehreren Tabellen abgelegt

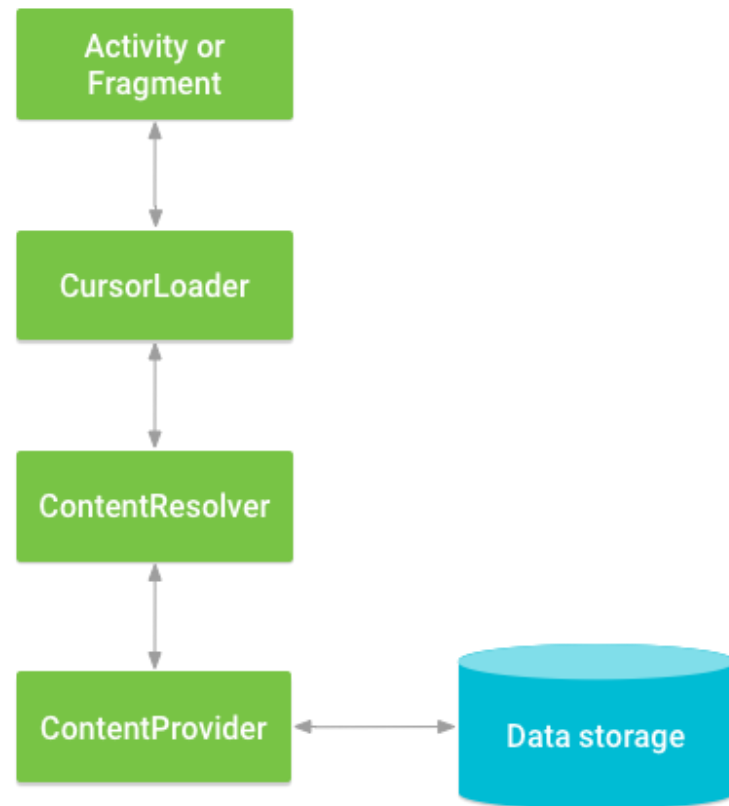
# Content Resolver & Content Provider

- Zugriff auf einen Content Provider erfolgt über einen Content Resolver
  - `Context.contentResolver()`
  - Bietet DB-Methoden und Zugriff auf Content via Streams
    - `insert()` / `query()` / `update()` / `delete()`
    - `openInputStream(uri)` / `openOutputStream(uri)`
  - Ein Content Resolver ist ein Proxy, der...
    - URI auflöst und zuständigen Content Provider sucht/findet
    - Interprozess-Kommunikation behandelt (aufrufende App ist meist in einem anderen Package als der aufgerufene CP)
- Achtung: Permissions müssen u.U. gesetzt werden!  
`<uses-permission android:name="android.permission.READ_CALENDAR" />`

# Zugriff auf Daten

## Über Content Resolver + Query:

```
// Queries the user dictionary and returns results
cursor = contentResolver.query(
    UserDictionary.Words.CONTENT_URI,
    // The content URI of the words table
    projection,
    // The columns to return for each row
    selectionClause,
    // Selection criteria
    selectionArgs.toArray(),
    // Selection criteria
    sortOrder
    // The sort order for the returned rows
)
```





# Zugriff auf Daten

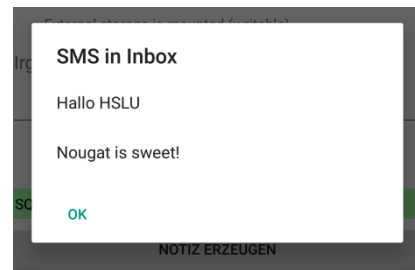
## Vergleich ContentProvider Query und SQL Query Parameter:

Content Provider Query	SQL SELECT Query	Notes
contentUri	FROM table_name	contentUri maps to the table in the provider named table_name.
projection	Col, col, col,...	projection is an array of columns that should be included for each row retrieved.
selection	WHERE col = value	selection specifies the criteria for selecting rows.
selectionArgs	(No exact equivalent. Selection arguments replace ? placeholders in the selection clause.)	-
sortOrder	ORDER BY col,col,...	sortOrder specifies the order in which rows appear in the returned Cursor.

# Beispiel: SMS Provider

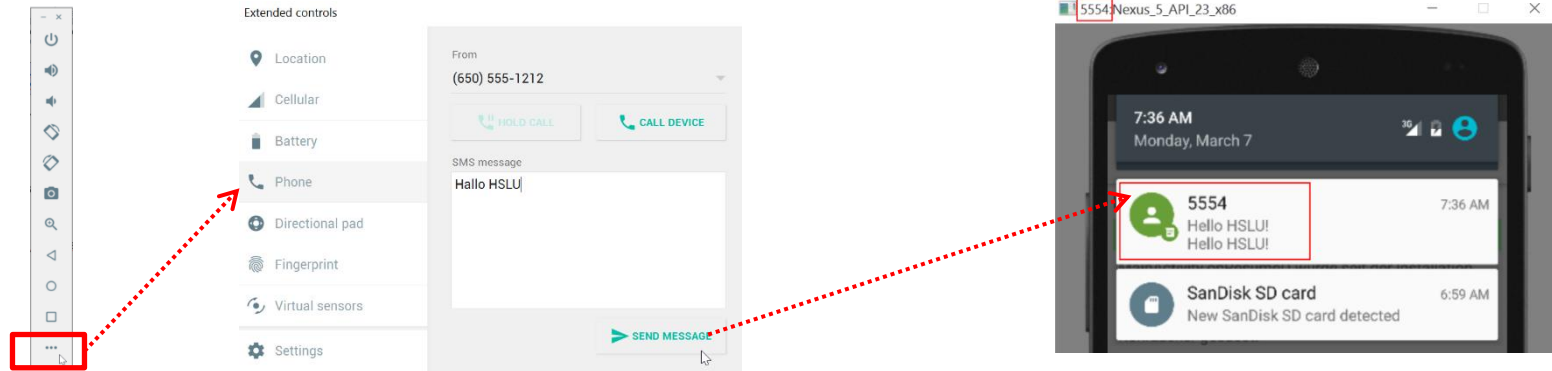
- SMS des Systems sind über Content-Provider zugänglich
  - `android.provider.Telephony.Sms`
    - «Sub Providers» für *Sent*, *Inbox*, *Draft*, etc.
  - Im Package `android.provider.*` finden wir „Contract Klasse“ `Telephony.Sms` mit Hilfsklassen `BaseColumns` und `Telephony.TextBasedSmsColumns`
    - Hier finden wir Content-URI und Spalten-Namen für Projections
- Anwendungsbeispiel : Alle Sms mit Text anzeigen

```
val cursor = requireActivity().contentResolver.query(Telephony.Sms.Inbox.CONTENT_URI,
    arrayOf(Telephony.Sms.Inbox._ID, Telephony.Sms.Inbox.BODY), // projection
    null, // selection
    null, // selection args
    null // sort order
)
AlertDialog.Builder(context)
    .setTitle("SMS in Inbox")
    .setCursor(cursor, null, Telephony.TextBasedSmsColumns.BODY)
    .setNeutralButton("Ok", null)
    .create()
    .show()
```



# SMS an Emulator schicken

- Bei Ausführung des Beispielcodes sehen wir im Emulator nichts. Grund: keine SMS vorhanden
- Mit den *Extended Controls* können SMS an den Emulator geschickt werden. Nummer angeben!



Tipp: Wird als Absender eine Emulatornummer verwendet, so können SMS zwischen zwei Emulatoren verschickt werden (Reply-Funktion)

# Einen Content Provider verwenden

API? Datenstruktur?

- Wo fange ich an?
  - Jeder Content Provider hat zwar ein Standard-API, aber woher weiss ich die Content-URI, Projection, etc.?
- Dokumentation?
  - In Android Doku sind Zugriff auf Kontakte und Kalender gut dokumentiert (weil eher kompliziertes Modell)
    - <http://developer.android.com/guide/topics/providers/calendar-provider.html>
    - <http://developer.android.com/guide/topics/providers/contacts-provider.html>
  - Einstiegspunkt = `Package android.provider.*`
    - <http://developer.android.com/reference/android/provider/package-summary.html>
    - Ausgangspunkt: «Contract Klassen» mit Content-URI und Column-Constants

Tutorials

Für andere Provider

# Eigener Content Provider

- Einen eigenen Content Provider zu schreiben ist nicht so schwer
- Die eigene Klasse muss von der abstrakten Klasse `android.content.ContentProvider` ableiten
- Wird bei Start der App hochgefahren und bleibt aktiv
  - In `onCreate()` kann eine Initialisierung vorgenommen werden (einzige Lifecycle-Methode)
- CRUD-Methoden: create, retrieve, update, and delete
  - Nicht alle müssen implementiert werden

So kann z.B. ein read-only Content Provider angelegt werden



## Übung 3

# Zur Übung 3

- Preferences
  - Resume Counter
  - Tee Präferenzen inkl. Default
- Dateisystem
  - Text speichern / laden
  - intern / extern
- Content Provider
  - SMS anzeigen
- SQLite DB / Room
  - Notizen erfassen & anzeigen

Optionaler Teil

D.h. müssen Sie nicht  
vorzeigen für's Testat.  
- Ist aber natürlich  
trotzdem Prüfungsstoff!

