Hugbúnaðarverkefni 2 / Software Project 2

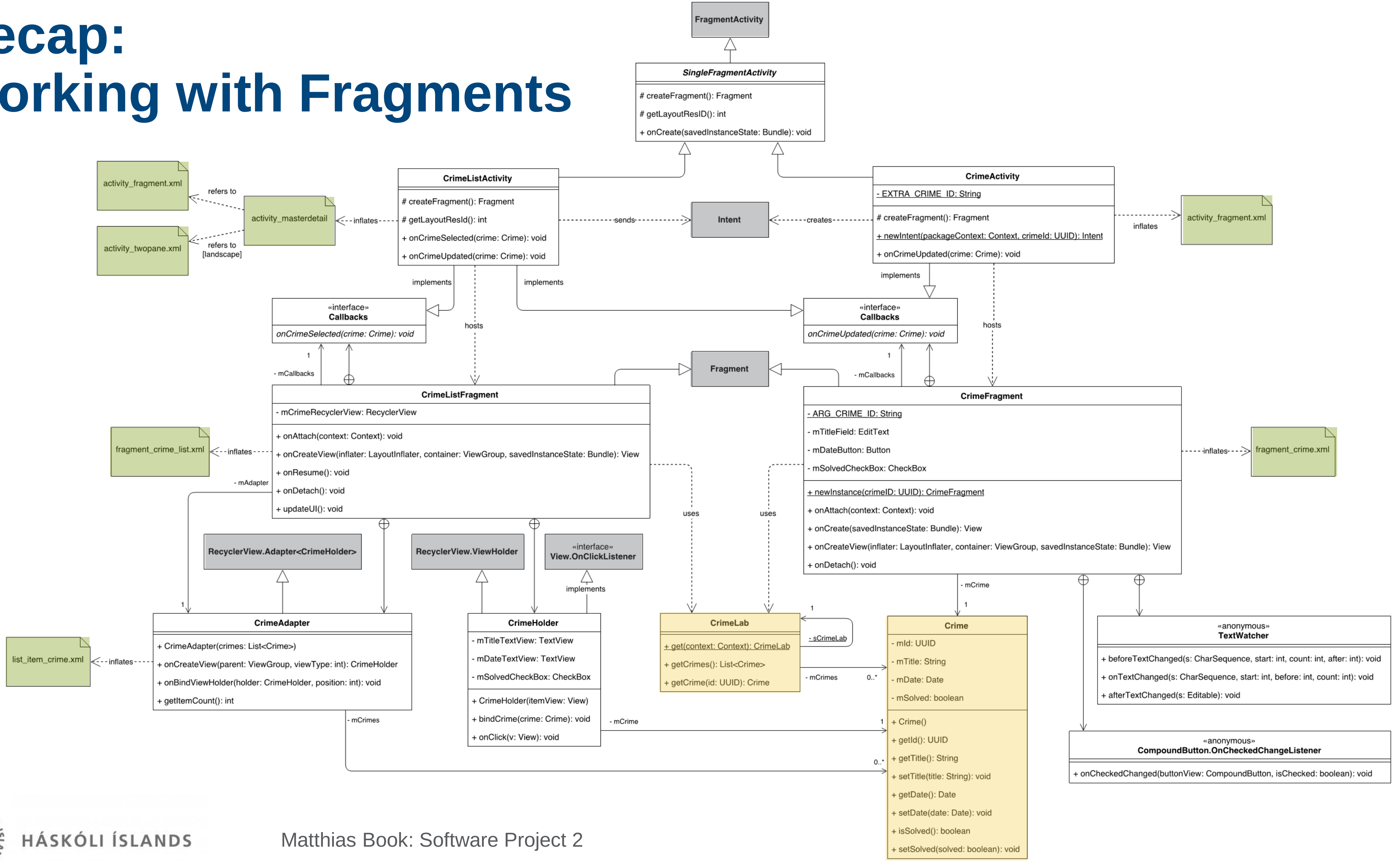
9. Data Storage in Android

HBV601G – Spring 2019

Matthias Book

HÁSKÓLI ÍSLANDS
VERKFRÆÐI- OG NÁTTÚRUVÍSINDASVIÐ
IÐNAÐARVERKFRÆÐI-, VÉLAVERKFRÆÐI-
OG TÖLVUNARFRÆÐIDEILD

Recap:
Working with Fragments



FragmentActivity

SingleFragmentActivity
createFragment(): Fragment
getLayoutResID(): int
+ onCreate(savedInstanceState: Bundle): void

CrimeListActivity
# createFragment(): Fragment
# getLayoutResId(): int
+ onCrimeSelected(crime: Crime): void
+ onCrimeUpdated(crime: Crime): void

CrimeActivity
- EXTRA_CRIME_ID: String
createFragment(): Fragment
+ newIntent(packageContext: Context, crimeId: UUID): Intent
+ onCrimeUpdated(crime: Crime): void

Intent

activity_fragment.xml
activity_masterdetail
activity_twopane.xml
refers to
refers to [landscape]
inflates
sends
creates
implements
hosts
uses
1
- mCallbacks

«interface»
Callbacks
onCrimeSelected(crime: Crime): void

«interface»
Callbacks
onCrimeUpdated(crime: Crime): void

Fragment

CrimeListFragment
- mCrimeRecyclerView: RecyclerView
+ onAttach(context: Context): void
+ onCreateView(inflater: LayoutInflater, container: ViewGroup, savedInstanceState: Bundle): View
+ onResume(): void
+ onDetach(): void
+ updateUI(): void

CrimeFragment
- ARG_CRIME_ID: String
- mTitleField: EditText
- mDateButton: Button
- mSolvedCheckBox: CheckBox
+ newInstance(crimeID: UUID): CrimeFragment
+ onAttach(context: Context): void
+ onCreate(savedInstanceState: Bundle): View
+ onCreateView(inflater: LayoutInflater, container: ViewGroup, savedInstanceState: Bundle): View
+ onDetach(): void

fragment_crime_list.xml
fragment_crime.xml
list_item_crime.xml
- mAdapter

RecyclerView.Adapter<CrimeHolder>
RecyclerView.ViewHolder
«interface»
View.OnClickListener

CrimeAdapter
+ CrimeAdapter(crimes: List<Crime>)
+ onCreateView(parent: ViewGroup, viewType: int): CrimeHolder
+ onBindViewHolder(holder: CrimeHolder, position: int): void
+ getItemCount(): int

CrimeHolder
- mTitleTextView: TextView
- mDateTextView: TextView
- mSolvedCheckBox: CheckBox
+ CrimeHolder(itemView: View)
+ bindCrime(crime: Crime): void
+ onClick(v: View): void

CrimeLab
+ get(context: Context): CrimeLab
+ getCrimes(): List<Crime>
+ getCrime(id: UUID): Crime
- sCrimeLab
- mCrimes
0..*
- mCrime

Crime
- mId: UUID
- mTitle: String
- mDate: Date
- mSolved: boolean
+ Crime()
+ getId(): UUID
+ getTitle(): String
+ setTitle(title: String): void
+ getDate(): Date
+ setDate(date: Date): void
+ isSolved(): boolean
+ setSolved(solved: boolean): void

«anonymous»
TextWatcher
+ beforeTextChanged(s: CharSequence, start: int, count: int, after: int): void
+ onTextChanged(s: CharSequence, start: int, before: int, count: int): void
+ afterTextChanged(s: Editable): void

«anonymous»
CompoundButton.OnCheckedChangeListener
+ onCheckedChanged(buttonView: CompoundButton, isChecked: boolean): void

In-Class Quiz 7: Activity & Fragment Communication

Solution

a) An activity can start another activity (3) by creating an intent and passing it to the ActivityManager, in order to let it start the desired activity.

b) An activity can start a fragment (6) by telling the FragmentManager to place the desired fragment in a container in the activity’s layout.

c) A fragment can start an activity (4) by sending a message to the fragment’s host activity, who can then invoke the desired activity.

d) An activity can send a message to a fragment it hosts (2) by calling a method of the fragment via the reference the activity holds to it.

e) A fragment can send a message to the activity hosting it (1) by calling a method on a callback interface defined by the fragment and implemented by the activity.

f) A fragment can send a message to another fragment (5) by sending a message to the fragment’s host activity, who can then send a message to the other fragment.

Resources

see also:

- Phillips et al.: Android Development, end of Ch. 2
- http://developer.android.com/guide/topics/resources/providing-resources.html

Overview: Where to Store Your App's Data?

- `savedInstanceState` bundle is only appropriate for short-term data saving while an activity is inactive, and may be destroyed at any time to free memory

More persistent alternatives:

- Static media for integration in your app's user interface (e.g. images)
  - ➢ Resource directory of your project
- Persistent storage of simple data that only your app works with (e.g. game state)
  - ➢ `SharedPreferences` file with key/value map in your app's private internal storage directory
- Persistent storage of large data that only your app works with (e.g. cached data)
  - ➢ Files in your app's private internal storage directory
- Dynamic, unstructured data that you exchange with other apps (e.g. documents)
  - ➢ Files in the user's external storage directories
- Dynamic, structured data that only your app works with (e.g. a to-do list)
  - ➢ Local SQLite database in your app's internal storage directory
- Data that you want to share with other users (e.g. messages)
  - ➢ Central data storage on a remote server

Resource Directories

- Recap: A resource is any static piece of your app that is not code
  - i.e. an image, sound or XML file
  - Resources are static files, i.e. delivered with your app at deployment and not changed

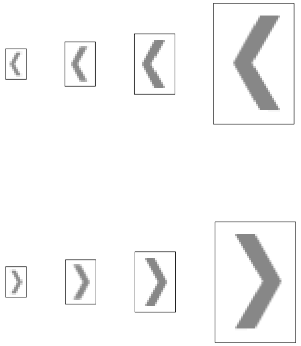
- Resources are stored in subfolders of your project's **`app/res`** directory, e.g.
  - **`animator/`** for XML files describing animations
  - **`color/`** for XML files describing color states
  - **`drawable/`** for PNG/JPG/GIF bitmap graphics or XML files describing vector graphics
  - **`layout/`** for XML files describing view layouts
  - **`menu/`** for XML files describing app menus
  - **`raw/`** for arbitrary static files in their original format
  - **`values/`** for XML files describing simple values such as strings
  - **`xml/`** for XML files containing arbitrary data (e.g. app-specific configurations)

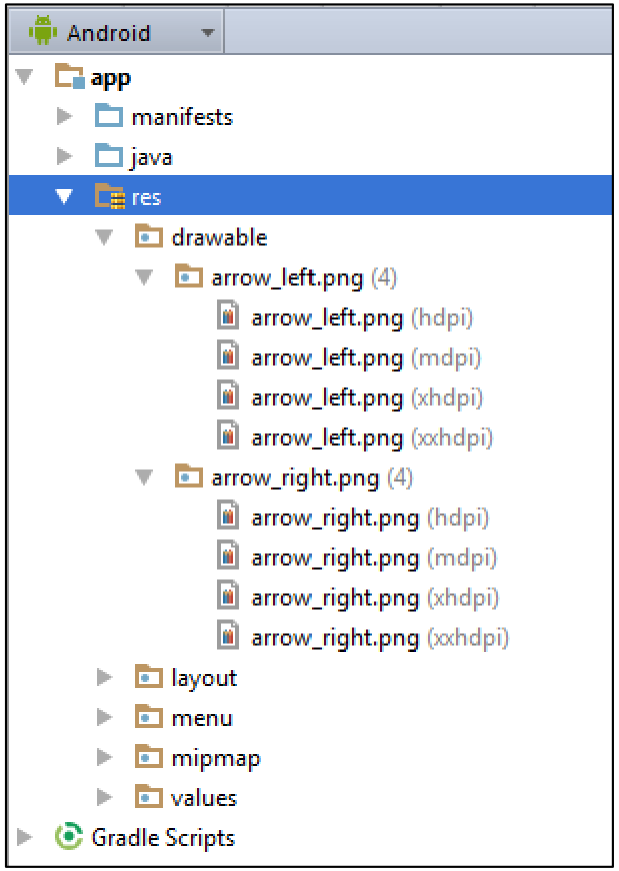
Configuration Qualifiers for Alternative Resources

- Recap: In different device configurations (i.e. orientation or language), your app may require different resources, e.g.
  - different layout files for portrait or landscape orientation
  - different string files for different languages
  - different image files for different screen resolutions
- Recap: “Configuration qualifiers” are suffixes added to resource directories to indicate in which configuration the resources should be used, e.g.
  - **`layout/`** for portrait layout, **`layout-land/`** for landscape layout
- For images, you need to provide different versions for different resolutions:
  - **`drawable-mdpi/`**, **`drawable-hdpi/`**, **`drawable-xhdpi/`**, **`drawable-xxhdpi/`** for screen pixel densities of 160, 240, 320 and 480 dpi, respectively
- Android will pick the most appropriate image at runtime and scale it as needed

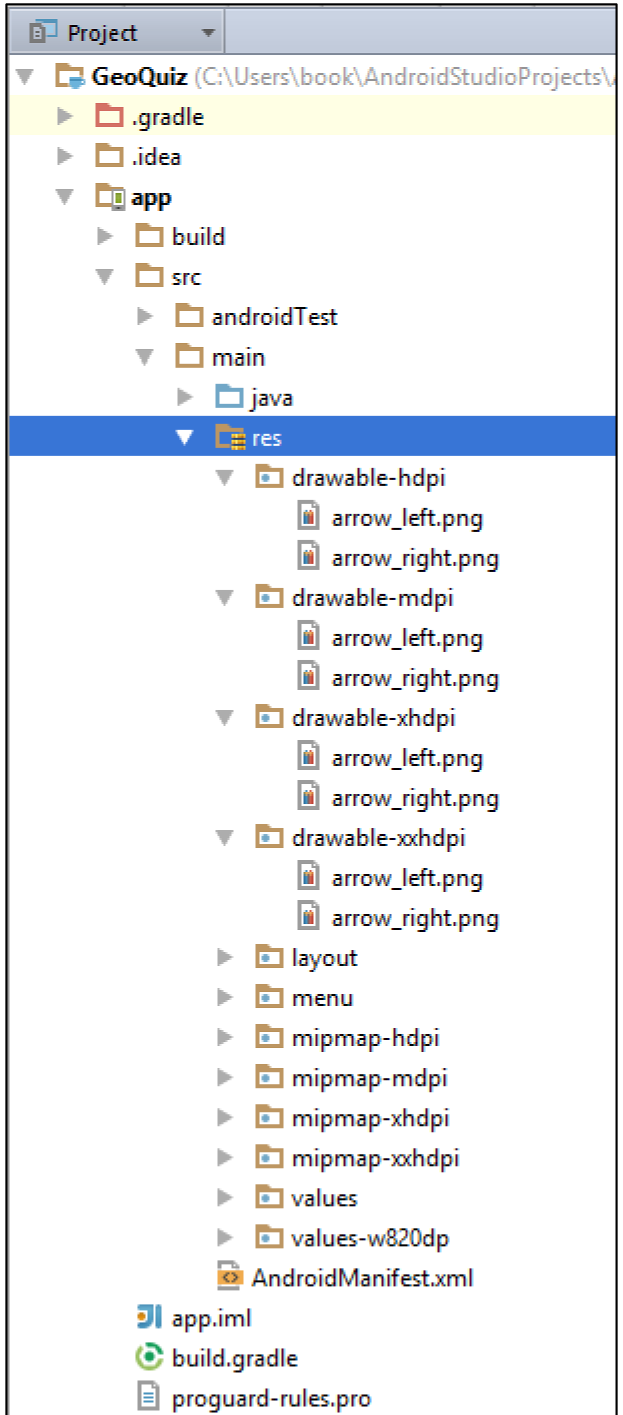
Providing Alternative Resources

- Example directory structure:

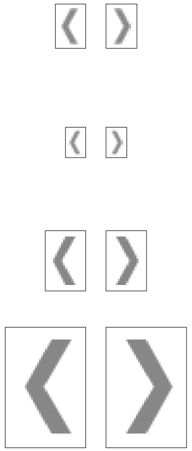




```
Android
▼ app
  ▶ manifests
  ▶ java
  ▼ res
    ▼ drawable
      ▼ arrow_left.png (4)
          arrow_left.png (hdpi)
          arrow_left.png (mdpi)
          arrow_left.png (xhdpi)
          arrow_left.png (xxhdpi)
      ▼ arrow_right.png (4)
          arrow_right.png (hdpi)
          arrow_right.png (mdpi)
          arrow_right.png (xhdpi)
          arrow_right.png (xxhdpi)
    ▶ layout
    ▶ menu
    ▶ mipmap
    ▶ values
▶ Gradle Scripts
```



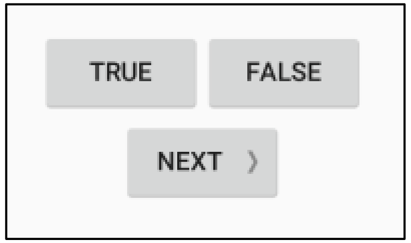
```
Project
▼ GeoQuiz (C:\Users\book\AndroidStudioProjects\
  ▶ .gradle
  ▶ .idea
  ▼ app
    ▶ build
    ▼ src
      ▶ androidTest
      ▼ main
        ▶ java
        ▼ res
          ▼ drawable-hdpi
              arrow_left.png
              arrow_right.png
          ▼ drawable-mdpi
              arrow_left.png
              arrow_right.png
          ▼ drawable-xhdpi
              arrow_left.png
              arrow_right.png
          ▼ drawable-xxhdpi
              arrow_left.png
              arrow_right.png
          ▶ layout
          ▶ menu
          ▶ mipmap-hdpi
          ▶ mipmap-mdpi
          ▶ mipmap-xhdpi
          ▶ mipmap-xxhdpi
          ▶ values
          ▶ values-w820dp
          AndroidManifest.xml
    app.iml
    build.gradle
    proguard-rules.pro
```



Accessing Resources

- Resource IDs are automatically created from resources' filenames and can be accessed in XML and Java
- Example: Adding the image **`arrow_right.png`** to a button

```
<Button
    android:id="@+id/next_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/next_button"
    android:drawableRight="@drawable/arrow_right"
    android:drawablePadding="4dp"/>
```



- In Java, a resource's ID can be obtained through the **`R`** class
  - e.g. **`R.drawable.arrow_right`** (Note: this is an **`int`** ID value, not the actual image)
- Note: No configuration qualifier is included in the reference!
  - Android automatically chooses the resource from the appropriate directory for the current device configuration

Shared Preferences

see also:

- https://developer.android.com/training/data-storage/shared-preferences

Storing App State vs. Activity State

- A `savedInstanceState` bundle is a volatile, activity-specific data storage
  - Can only be accessed by the activity that created it
  - May be destroyed by Android at any time to free up memory
  - ➢ Only suitable for maintaining activity state through a destroy-recreate lifecycle transition

- For reliable storage of small pieces of app state…
  - that shall be accessible by different activities of your app
  - that shall be guaranteed to persist independently of any activity’s lifecycle
  - that shall be deleted only upon uninstalling your app
- …use `SharedPreferences` instead.

- `SharedPreferences` are simple key/value pairs
  residing in a persistent file that is accessible only by your app.

Obtaining a `SharedPreferences` File

- To obtain a **`SharedPreferences`** key/value map for your app, call the

  `SharedPreferences getSharedPreferences(String name, int mode)`

  method of any **`Context`** (i.e. **`Activity`**) of your app.
  - **`name`** is a filename of your choice (in case you want to maintain several shared pref. files)
  - **`mode`** must be **`MODE_PRIVATE`** (all other options are deprecated)

- Example:

Obtaining the context e.g. from within a fragment

`SharedPreferences` are not readable by any other apps

```
Context context = getActivity();
SharedPreferences sharedPref = context.getSharedPreferences(
          getString(R.string.preference_file_key), Context.MODE_PRIVATE);
```

Good practice to use string constants everywhere to avoid bugs by typos

Reading Key/Value Pairs from `SharedPreferences`

- The **`SharedPreferences`** instance we obtained provides a variety of methods for reading key/value pairs of different types
  - All require the **`key`** to look up, as well as a **`default`** value to return if the key wasn't found
  - e.g. **`int getInt(String key, int default)`**
  - similar for **`boolean`**, **`float`**, **`String`** etc.

- Example:

```
int highScore = sharedPref.getInt(getString(R.string.saved_high_score), 0);
```

Writing Key/Value Pairs to `SharedPreferences`

- To write data, we first need to call the **`edit()`** method on the **`SharedPreferences`** we obtained previously
- This will give us a **`SharedPreferences.Editor`** that provides methods for writing and manipulating key/value pairs of different types
  - e.g. **`putInt(String key, int value)`** to set/update the **`value`** for the **`key`**
    - similar for **`boolean`**, **`float`**, **`String`** etc.
  - **`remove(String key)`** to remove the value for the **`key`**
- Finally, call **`apply()`** to write all changes to the persistent file (important!)
  - The **`edit`**, **`put`** and **`remove`** methods all return **`Editor`** instances to enable call chaining
- Example:

```
sharedPref.edit()
          .putInt(getString(R.string.saved_high_score), newHighScore)
          .putFloat(getString(R.string.saved_balance), newBalance)
          .apply();
```

File Storage

see also:

- https://developer.android.com/training/data-storage/files
- Phillips et al.: Android Development, beginning of Ch. 14 (internal)
- Phillips et al.: Android Development, middle of Ch. 16 (external)

Internal vs. External Storage

Internal Storage

- Built-in, always available
- Files are accessible only by your app
- Does not require special permission
- All files deleted upon uninstalling app

➢ **Suitable for files that should be access-restricted**

  - e.g. app state, sensitive data

External Storage

- Might be SD card unmounted by user
  - highly unlikely in modern devices
- Files accessible by all of user’s apps
- Requires permission upon installation
- “Private” files deleted upon uninstalling app, “public” files remain

➢ **Suitable for files that are intended for sharing/outliving your app**

  - e.g. created media/documents

Files in Internal Storage

- Every app has a “sandbox” directory that is only accessible by the app itself
  - Not by other apps, and not by the user (unless the device is rooted or in an emulator)
- Located in the device’s `/data/data/<app package>` directory
  - e.g. `/data/data/is.hi.hbv601g.geoquiz`
- `Context` (i.e. `Activity`) objects provide methods for internal storage access:
  - `File getFilesDir()` returns handle of the app’s internal storage directory
  - `FileInputStream openFileInput(String name)` opens existing file for reading
  - `FileOutputStream openFileOutput(String name, int mode)` opens file for writing, creating it if necessary
  - `File getDir(String name, int mode)` returns handle of a subdirectory of the app’s internal storage directory, creating it if necessary
  - `String[] fileList()` returns list of file names in app’s internal storage directory
- Use the Java class library’s standard `java.io` classes for file and stream ops

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/package-summary.html

Example: Writing a File to Internal Storage

```
String filename = "myfile";
String s = "Hello world!";
FileOutputStream outputStream;

try {
  outputStream = openFileOutput(filename, Context.MODE_PRIVATE);
  outputStream.write(s.getBytes());
  outputStream.close();
} catch (Exception e) {
  e.printStackTrace();
}
```

Create file in the app’s internal storage directory and get a stream to write into it

`MODE_PRIVATE`: recreate file even if it exists
`MODE_APPEND`: if file exists, append to it, otherwise create it

Write bytes into the file

Cache Directory in Internal Storage

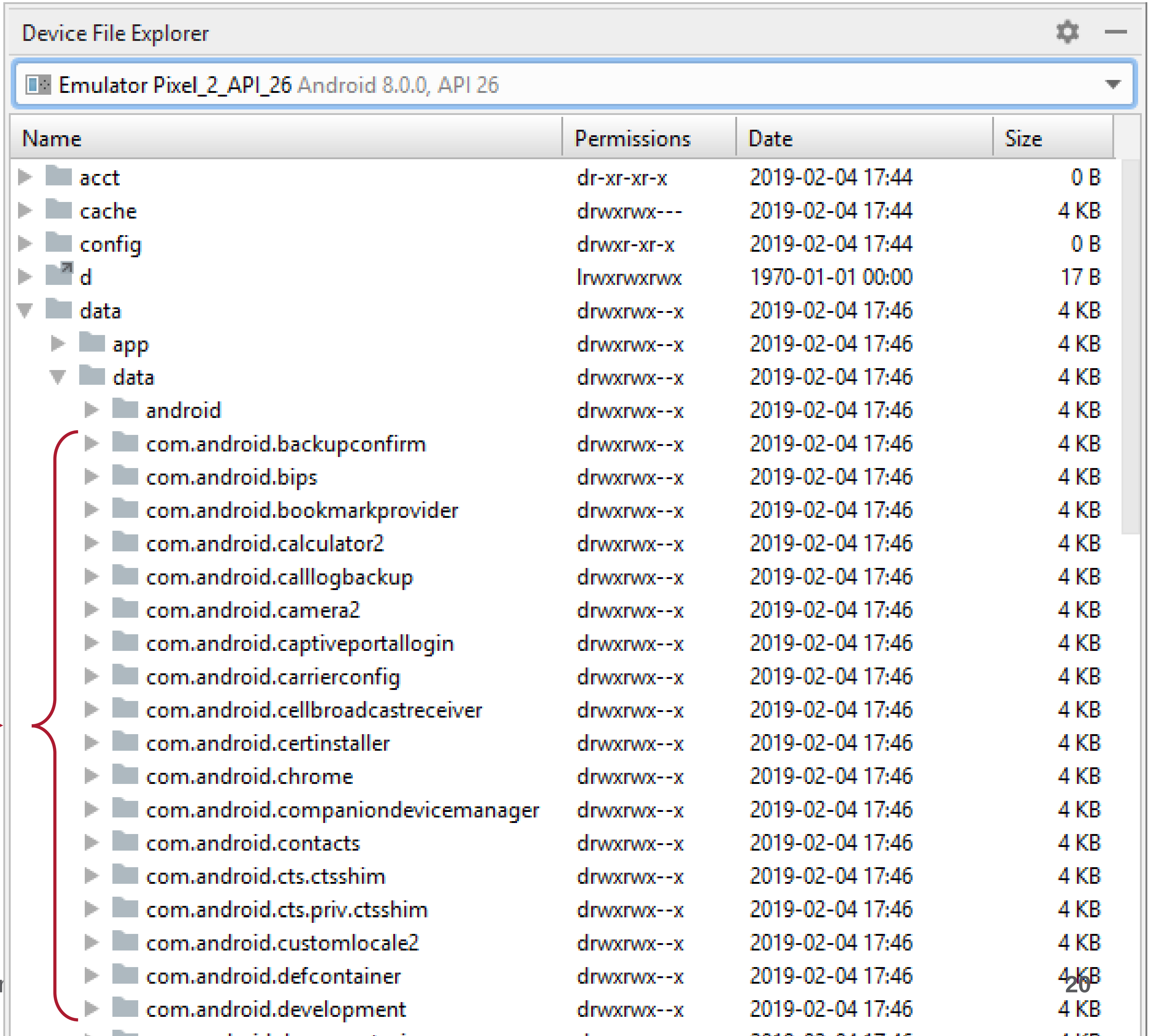
- Besides files that are intended for persistent local storage, an app may choose to store data on the device for caching purposes
  - e.g. to avoid having to repeatedly request it from a remote server
- Cached data is special insofar as
  - it typically gets stale (and thus useless) after some time
  - it is not an authoritative information source
    - i.e. if it is missing, the original source can simply be queried again

- To reflect these properties, cache data should be stored in a special directory:
  - **`File getCacheDir()`** returns handle of the app's cache directory in internal storage
- Android can then delete cache data automatically if file storage space runs low
  - But there is no guarantee of deletion (except upon uninstalling the app)
  - ➢ App should ensure the cache directory does not grow massively (< 1 MB is a good idea)

Location of Internal Storage Directories

- In Android Studio, open View > Tool Windows > Device File Explorer to look at emulated device's file system

- `/data/data` contains sandbox directories for various apps
  - (only visible on rooted or emulated devices)



Device File Explorer

Emulator Pixel_2_API_26 Android 8.0.0, API 26

Name	Permissions	Date	Size
acct	dr-xr-xr-x	2019-02-04 17:44	0 B
cache	drwxrwx---	2019-02-04 17:44	4 KB
config	drwxr-xr-x	2019-02-04 17:44	0 B
d	lrwxrwxrwx	1970-01-01 00:00	17 B
data	drwxrwx--x	2019-02-04 17:46	4 KB
app	drwxrwx--x	2019-02-04 17:46	4 KB
data	drwxrwx--x	2019-02-04 17:46	4 KB
android	drwxrwx--x	2019-02-04 17:46	4 KB
com.android.backupconfirm	drwxrwx--x	2019-02-04 17:46	4 KB
com.android.bips	drwxrwx--x	2019-02-04 17:46	4 KB
com.android.bookmarkprovider	drwxrwx--x	2019-02-04 17:46	4 KB
com.android.calculator2	drwxrwx--x	2019-02-04 17:46	4 KB
com.android.calllogbackup	drwxrwx--x	2019-02-04 17:46	4 KB
com.android.camera2	drwxrwx--x	2019-02-04 17:46	4 KB
com.android.captiveportallogin	drwxrwx--x	2019-02-04 17:46	4 KB
com.android.carrierconfig	drwxrwx--x	2019-02-04 17:46	4 KB
com.android.cellbroadcastreceiver	drwxrwx--x	2019-02-04 17:46	4 KB
com.android.certinstaller	drwxrwx--x	2019-02-04 17:46	4 KB
com.android.chrome	drwxrwx--x	2019-02-04 17:46	4 KB
com.android.companiondevicemanager	drwxrwx--x	2019-02-04 17:46	4 KB
com.android.contacts	drwxrwx--x	2019-02-04 17:46	4 KB
com.android.cts.ctsshim	drwxrwx--x	2019-02-04 17:46	4 KB
com.android.cts.priv.ctsshim	drwxrwx--x	2019-02-04 17:46	4 KB
com.android.customlocale2	drwxrwx--x	2019-02-04 17:46	4 KB
com.android.defcontainer	drwxrwx--x	2019-02-04 17:46	4 KB
com.android.development	drwxrwx--x	2019-02-04 17:46	4 KB

Files in External Storage

- External storage used to be a removable SD card in early devices, but is today usually emulated by just another folder in the device’s built-in flash memory
  - This is so-called “primary” external storage
  - Today’s devices can have “additional” external storage in the form of a removable SD card

- Still, it’s good practice to check whether external storage is available before attempting to use it:
  - **`Environment.getExternalStorageState()`** should return **`MEDIA_MOUNTED`**

Private vs. Public External Storage

- **“Private” directories in external storage**
  - `File` handle for directory obtained through `getExternalFilesDir(String type)` instance method of a `Context` object (typically, an `Activity`)
  - Files created here are readable/writable by other apps, but your app is in charge of the folder
    - This could e.g. be data that you publish for consumption by other apps, or vice versa
  - The directory is deleted when your app is uninstalled
- **“Public” directories in external storage**
  - `File` handle for directory obtained through static `getExternalStoragePublicDir(String type)` method of the `Environment` class
  - Files created here are readable/writable by other apps, and the user is in charge of the folder
    - This is typically data “owned” by the user, i.e. the user’s photos, music, documents etc.
  - The directory remains on the device when your app is uninstalled
- Note: On a multi-user device, apps only have access to the external storage of the user they are running as (i.e. one user can’t see another’s photos)

External Storage Directory Types

- The **`String type`** parameter passed to the previously shown methods indicates the type of file you want to access/store
- This determines which external storage directory will be provided to you, e.g.:
  - **`DIRECTORY_PICTURES`** for photos
  - **`DIRECTORY_MOVIES`** for video files
  - **`DIRECTORY_MUSIC`** for audio files containing music (for use by media players)
  - **`DIRECTORY_RINGTONES`** for audio files containing ringtones (not for use by media players)
  - **`DIRECTORY_DOCUMENTS`** for documents

- It's recommended to use the above constants (defined in **`Environment`**)
  - In private external storage, you could also use your own directory names
  - In public external storage, using your own names is discouraged so you don't clutter up the user's directory structure

Example: Creating a Photo Album

- **In private external storage:**

```
public File getAlbumStorageDir(Context context, String albumName) {
    File file = new File(context.getExternalFilesDir(
            Environment.DIRECTORY_PICTURES), albumName);
    if (!file.mkdirs()) {
        Log.e(LOG_TAG, "Directory not created");
    }
    return file;
}
```

Get handle for app’s private external storage directory for pictures

Try creating the subdirectory **`albumName`** in there

- **In public external storage:**

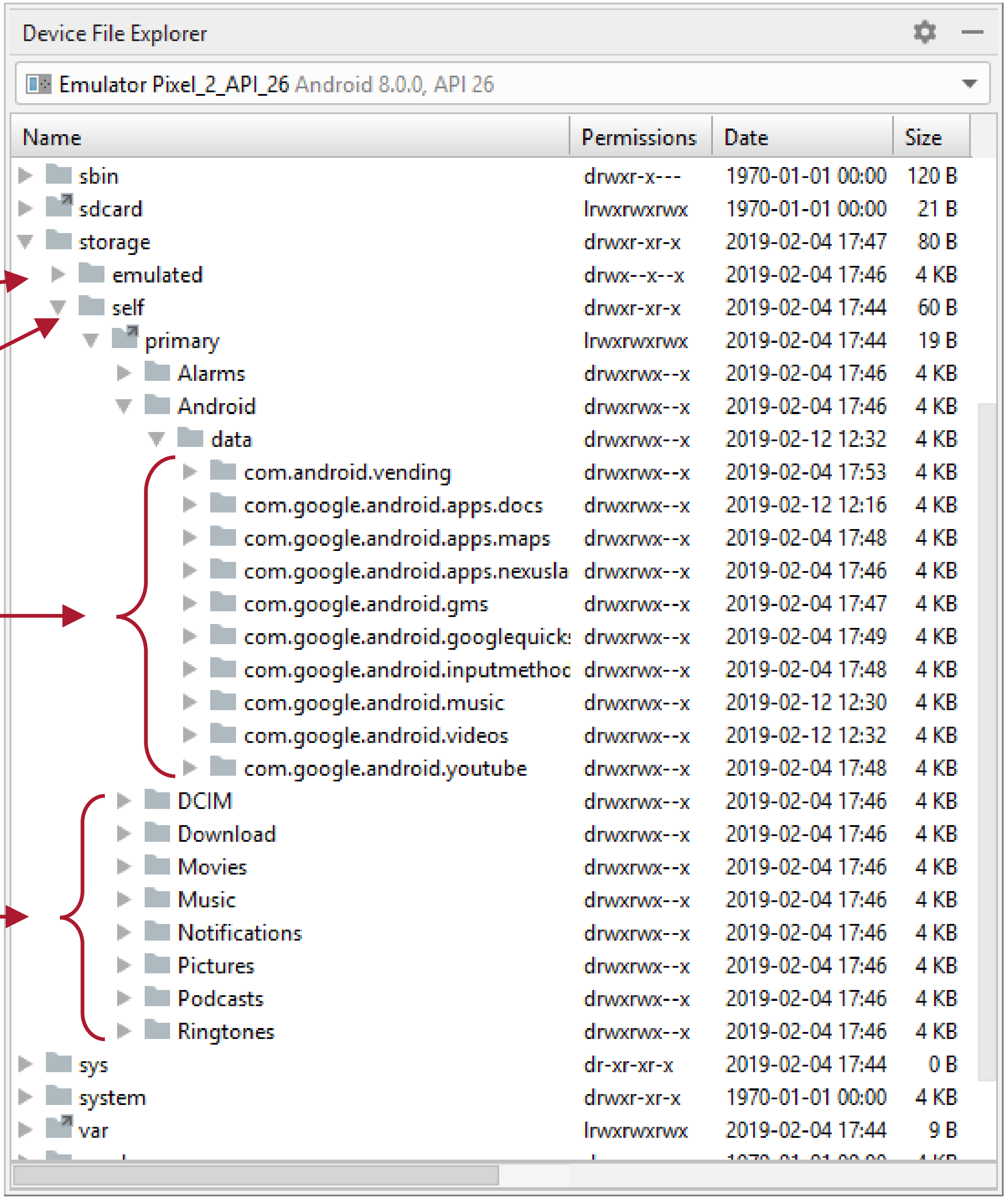
```
public File getAlbumStorageDir(String albumName) {
    File file = new File(Environment.getExternalStoragePublicDirectory(
            Environment.DIRECTORY_PICTURES), albumName);
    if (!file.mkdirs()) {
        Log.e(LOG_TAG, "Directory not created");
    }
    return file;
}
```

Get handle for user’s public external storage directory for pictures

After this, use the Java class library’s standard **`java.io`** classes for file and stream operations

Location of External Storage Directories

- Primary external storage is emulated on this device
  - i.e. in built-in flash memory, not on a removable SD card
  - Storage for current user
- Private external storage
  - Managed by individual apps
  - Deleted when uninstalled
  - Readable by any app
- Public external storage
  - Managed by the user
  - Outlives any app
  - Structured by file type



Device File Explorer

Emulator Pixel_2_API_26 Android 8.0.0, API 26

Name	Permissions	Date	Size
sbin	drwxr-x---	1970-01-01 00:00	120 B
sdcard	lrwxrwxrwx	1970-01-01 00:00	21 B
storage	drwxr-xr-x	2019-02-04 17:47	80 B
emulated	drwx--x--x	2019-02-04 17:46	4 KB
self	drwxr-xr-x	2019-02-04 17:44	60 B
primary	lrwxrwxrwx	2019-02-04 17:44	19 B
Alarms	drwxrwx--x	2019-02-04 17:46	4 KB
Android	drwxrwx--x	2019-02-04 17:46	4 KB
data	drwxrwx--x	2019-02-12 12:32	4 KB
com.android.vending	drwxrwx--x	2019-02-04 17:53	4 KB
com.google.android.apps.docs	drwxrwx--x	2019-02-12 12:16	4 KB
com.google.android.apps.maps	drwxrwx--x	2019-02-04 17:48	4 KB
com.google.android.apps.nexusla	drwxrwx--x	2019-02-04 17:46	4 KB
com.google.android.gms	drwxrwx--x	2019-02-04 17:47	4 KB
com.google.android.googlequick	drwxrwx--x	2019-02-04 17:49	4 KB
com.google.android.inputmethoc	drwxrwx--x	2019-02-04 17:48	4 KB
com.google.android.music	drwxrwx--x	2019-02-12 12:30	4 KB
com.google.android.videos	drwxrwx--x	2019-02-12 12:32	4 KB
com.google.android.youtube	drwxrwx--x	2019-02-04 17:48	4 KB
DCIM	drwxrwx--x	2019-02-04 17:46	4 KB
Download	drwxrwx--x	2019-02-04 17:46	4 KB
Movies	drwxrwx--x	2019-02-04 17:46	4 KB
Music	drwxrwx--x	2019-02-04 17:46	4 KB
Notifications	drwxrwx--x	2019-02-04 17:46	4 KB
Pictures	drwxrwx--x	2019-02-04 17:46	4 KB
Podcasts	drwxrwx--x	2019-02-04 17:46	4 KB
Ringtones	drwxrwx--x	2019-02-04 17:46	4 KB
sys	dr-xr-xr-x	2019-02-04 17:44	0 B
system	drwxr-xr-x	1970-01-01 00:00	4 KB
var	lrwxrwxrwx	2019-02-04 17:44	9 B

Obtaining Permission to Access External Storage

- In order to be able to access external storage, your app must obtain permission from the user upon installation
- To ask for the required permission, include one of the following lines in your manifest file (**`app/src/main/AndroidManifest.xml`**):
  - **`<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />`**
  - **`<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />`**
- Write permission implies read permission
- Read permission is currently not required by Android, but may be in later release
  - So if your app needs it, declare it already now so your app will keep working

Local Databases

Using the Room Persistence Library

see also:

- https://developer.android.com/training/data-storage/room

SQLite and Room in Android

- Use a local database for easy access to any structured data that
  - can be kept in its entirety on the client device
  - needs to be accessed only by that device's user
- Use a server-side database for any structured data that
  - is shared between users
  - is too large for storage on the device

Technology for client-side database:

- SQLite, a relational database storing data in simple files (www.sqlite.org)
- Room, a persistence library providing an abstraction layer over SQLite
- Android includes the SQLite library in its standard library
- Room can easily be added to an app

Preparation: Adding Architecture Components

- Add Google’s Maven repository to your <u>project’s</u> **`build.gradle`** file:
    - i.e. e.g. for GeoQuiz, to **`GeoQuiz\build.gradle`**

```
allprojects {
    repositories {
        google()
        jcenter()
    }
}
```

Details: https://developer.android.com/jetpack/androidx/releases/room

- Add the dependencies for Room to your <u>app’s</u> **`build.gradle`** file:
    - i.e. e.g. for GeoQuiz, to **`GeoQuiz\app\build.gradle`**

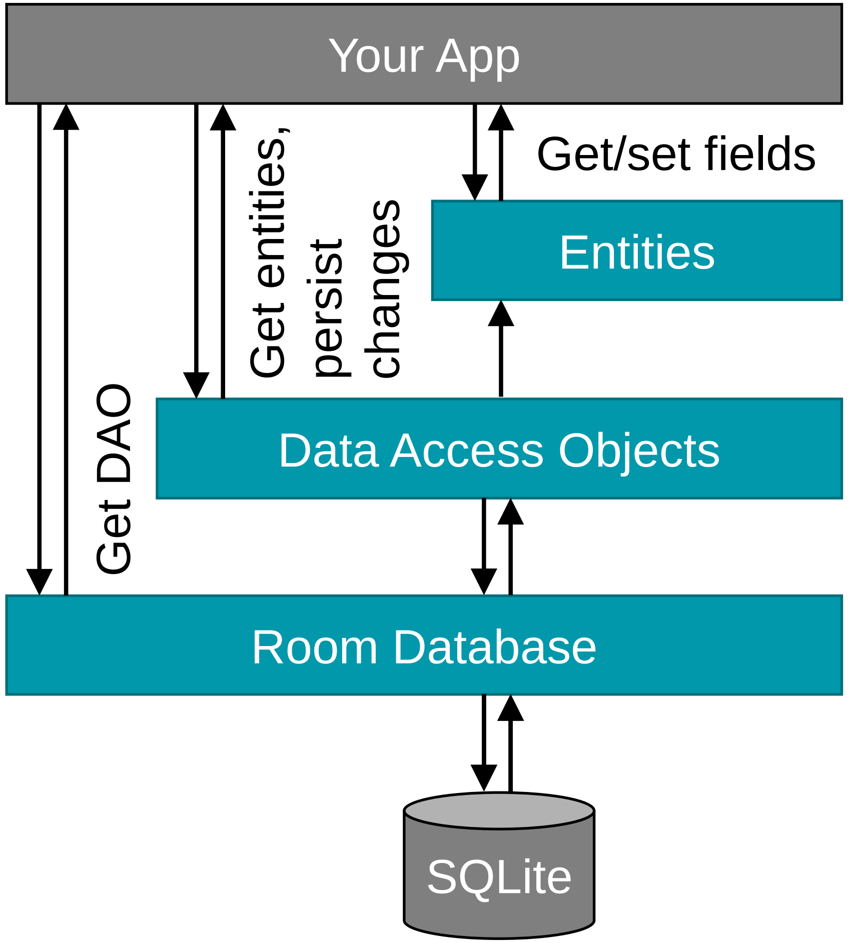
```
dependencies {
    def room_version = "2.1.0-alpha04"

    implementation "androidx.room:room-runtime:$room_version"
    annotationProcessor "androidx.room:room-compiler:$room_version"

    // Test helpers
    testImplementation "androidx.room:room-testing:$room_version"
}
```

Room Persistence Library Architecture

- **`@Database`** provides an abstraction of the underlying SQLite database containing the app's persisted relational data

- **`@Dao`** instances contain methods for getting entities from the database and saving changes back to the database

- **`@Entity`** instances represent tables within the database and provide access to its columns



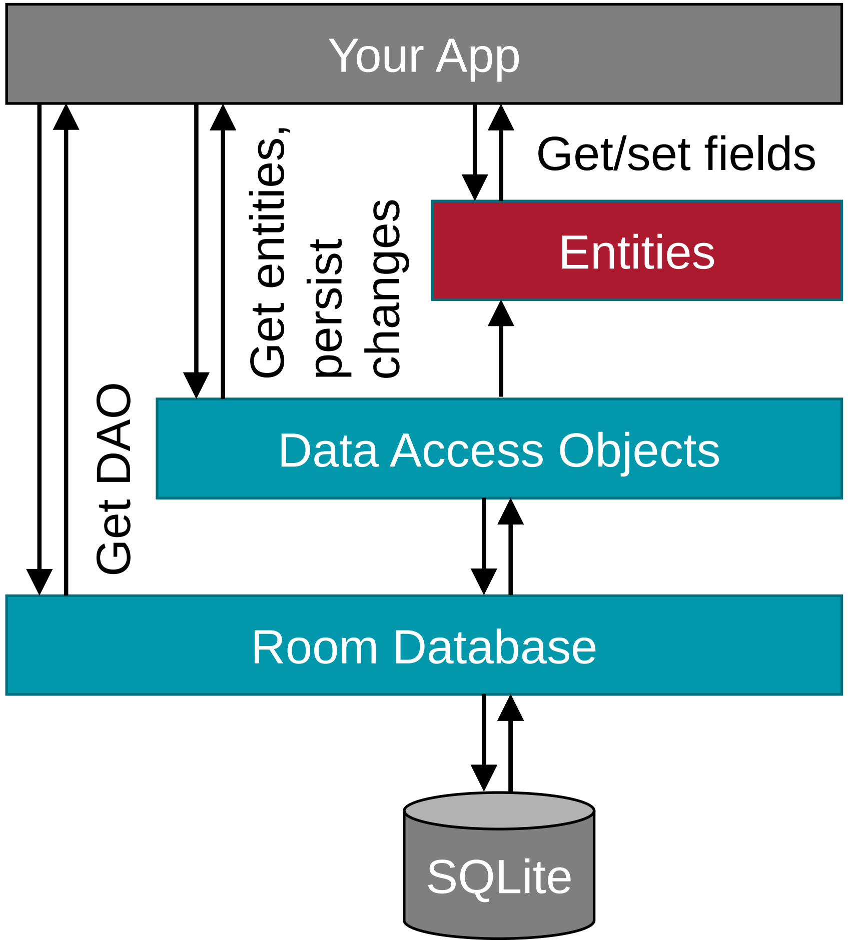
Example: Storing User Objects
User Entity

```
@Entity
public class User {
    @PrimaryKey
    public int id;

    @ColumnInfo(name = "first_name")
    public String firstName;

    @ColumnInfo(name = "last_name")
    public String lastName;
}
```

Required for at least one attribute

Optional, using attribute name otherwise

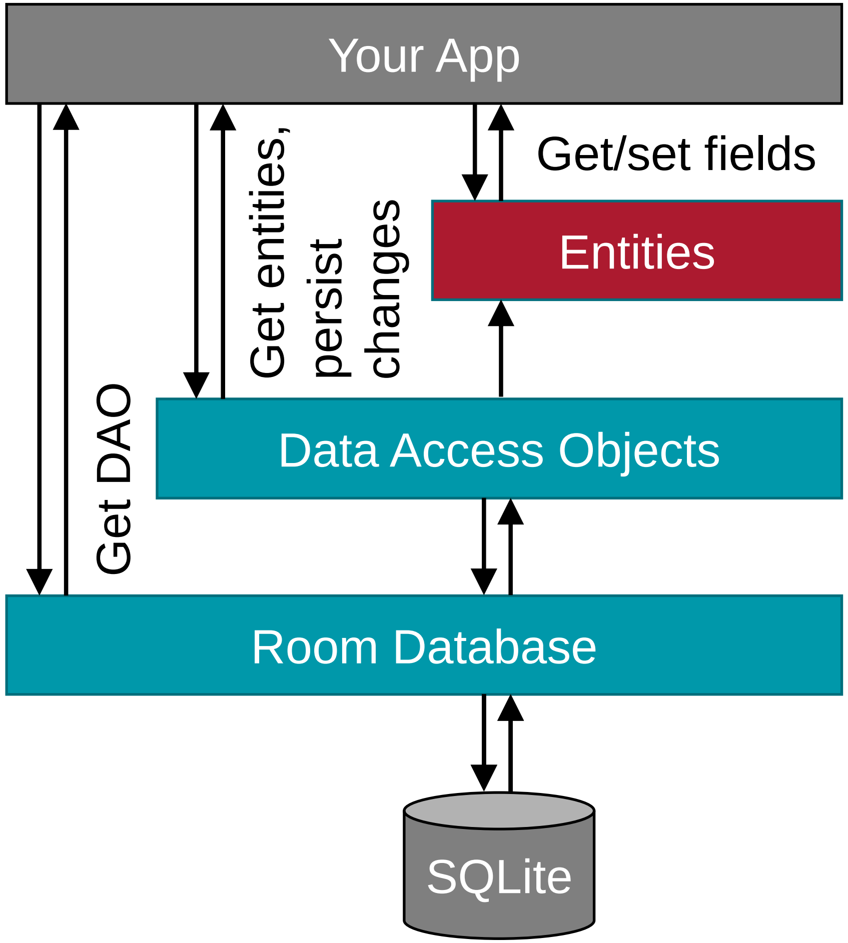
- Note: Contrary to other object-relational mappers, Room forbids object references between entities for performance reasons
  - Eager loading can waste memory
  - Lazy loading can cost time at inopportune moments



Example: Storing User Objects
Book Entity

```
@Entity(foreignKeys = @ForeignKey(
    entity = User.class,
    parentColumns = "id",
    childColumns = "user_id",
    onDelete = CASCADE))
public class Book {
    @PrimaryKey
    public int bookId;

    public String title;

    @ColumnInfo(name = "user_id")
    public int userId;
}
```

Delete all of a user's books if the user is deleted

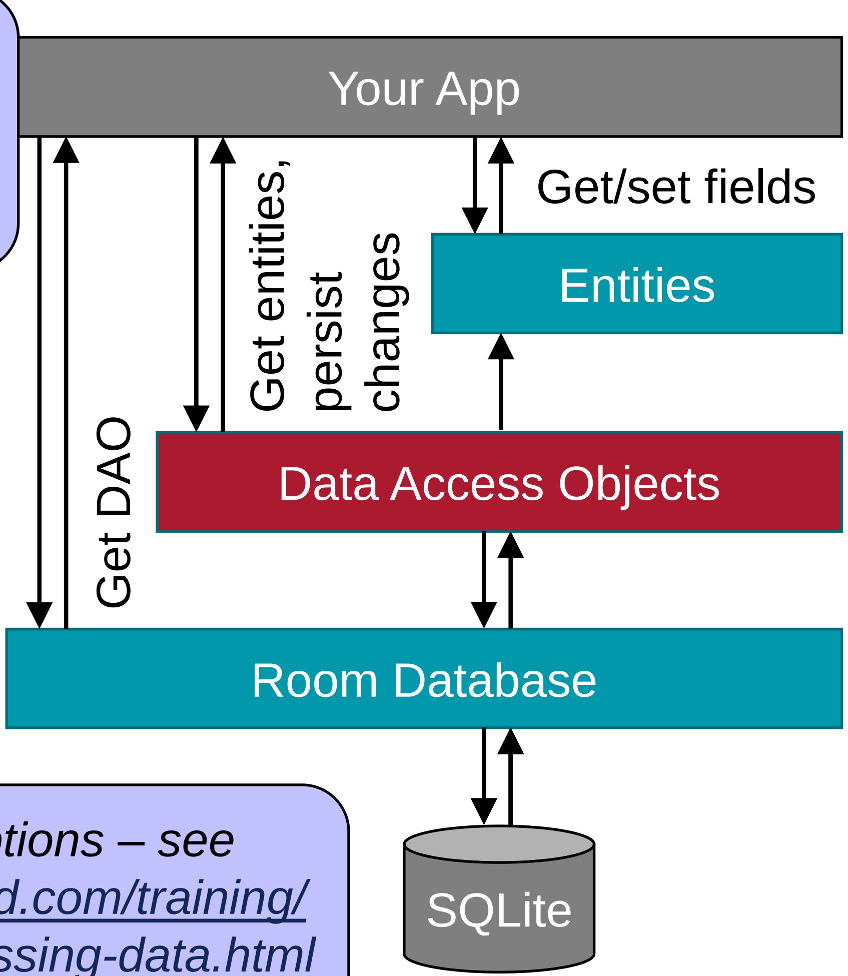


- Solution: Define foreign key relationships
  - To request dependent data yourself when needed
  - To let SQLite take care e.g. of cascading deletions

Example: Storing User Objects
User Data Access Object

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

    @Query("SELECT * FROM user " +
           "WHERE id IN (:userIds)")
    List<User> loadAllByIds(int[] userIds);

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

    @Insert
    void insertAll(User... users);

    @Delete
    void delete(User user);
}
```

Compile-time error if bind parameter does not match the name of a method parameter

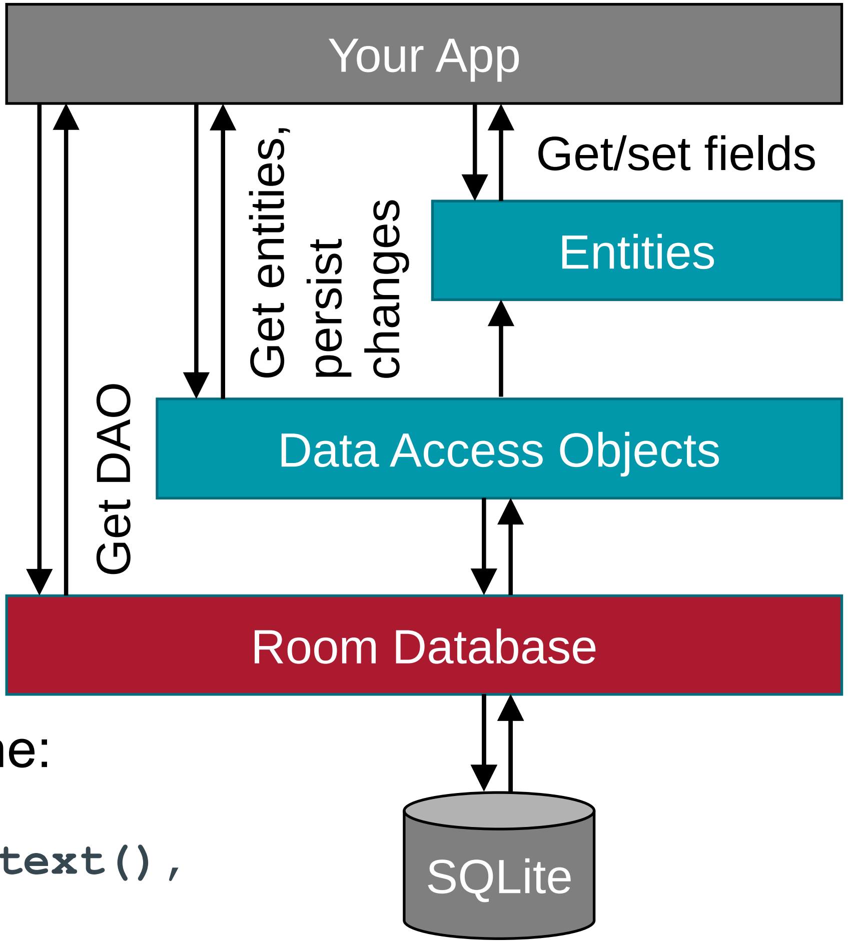


Many more query options – see https://developer.android.com/training/data-storage/room/accessing-data.html for details

Example: Storing User Objects
Database

- Declaring the database abstraction:

```
@Database(entities = {User.class}, version = 1)
public abstract class AppDatabase
                extends RoomDatabase {
    public abstract UserDao userDao();
}
```



- Obtaining an instance of the database:
  - Note: Wrap this in a singleton pattern since the instance is expensive, and you'll rarely need more than one:

```
AppDatabase db = Room.databaseBuilder(getApplicationContext(),
        AppDatabase.class, "database-name").build();
```

Example: Storing User Objects
Accessing Entities via the DAO

- Obtain an instance of the database:

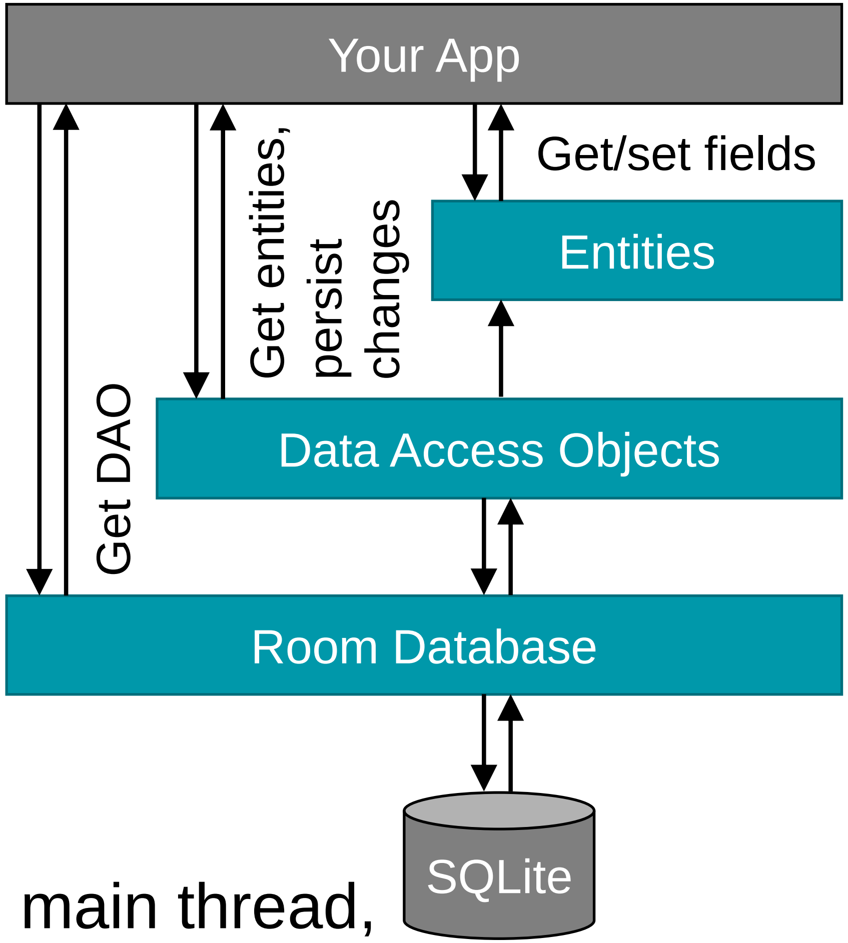
```
AppDatabase db = Room.databaseBuilder(
        getApplicationContext(),
        AppDatabase.class, "database-name").build();
```

- Obtain an instance of the DAO:

```
UserDao userDao = db.userDao();
```

- Query database via DAO to receive entities:

```
List<User> users = userDao.getAll();
```



- Note: Room does not allow database access on the main thread, as it may delay user interface responses

Need to run queries in a separate thread instead – details in next class

Summary: Where to Store Your App's Data?

- Static media for integration in your app's user interface (e.g. images)
  - ➢ Resource directory of your project
- Volatile short-term storage of little data while activity is inactive (ie. activity state)
  - `savedInstanceState` bundle of an activity
- Persistent storage of simple data that only your app works with (e.g. game state)
  - ➢ `SharedPreferences` file in your app's private internal storage directory
- Persistent storage of large data that only your app works with (e.g. cached data)
  - ➢ Files in your app's private internal storage directory
- Dynamic, unstructured data that you exchange with other apps (e.g. documents)
  - ➢ Files in the user's external storage directories
- Dynamic, structured data that only your app works with (e.g. a to-do list)
  - ➢ Local SQLite database in your app's internal storage directory
- Data that you want to share with other users or devices (e.g. messages)
  - ➢ Central data storage on a remote server *(next class)*

Local Databases

Appendix:

Low-level Alternative:
Using the SQLite API

(Legacy slides from 2017)

see also:

- Phillips et al.: Android Development, Ch. 14
- https://developer.android.com/training/data-storage/sqlite.html

SQLite in Android

- Use a local database for easy access to any structured data that
  - can be kept in its entirety on the client device
  - needs to be accessed only by that device's user

- Use a server-side database for any structured data that
  - is shared between users
  - is too large for storage on the device

Technology for client-side database:

- SQLite, a relational database storing data in simple files (www.sqlite.org)
- Android includes the SQLite library in its standard library

Describing the Database Schema in Java

```
package com.bignerdranch.android.criminalintent.database;

 public class CrimeDbSchema {
   public static final class CrimeTable {
     public static final String NAME = "crimes";
     public static final class Cols {
       public static final String UUID = "uuid";
       public static final String TITLE = "title";
       public static final String DATE = "date";
       public static final String SOLVED = "solved";
     }
   }
 }
```

- This is just a fancy way of organizing the string constants for your table and column labels, but helps to avoid bugs when you are constructing SQL queries:
  - `import com.bignerdranch.android.criminalintent.database.CrimeDBSchema.CrimeTable;`
  - You can now refer to database columns as `CrimeTable.Cols.TITLE` etc.

Managing the Database Schema

- When starting an app relying on a local database, you always need to
  1. Check to see if the database already exists
  2. If it does not, create the database and the tables and initial data you need
  3. If it does, open it and see which version its schema has
  4. If the schema was created by an older version of your app,
     upgrade it to suit the schema your app now requires
- The upgrade functionality is particularly important for client-side databases:
  - You may want to roll out upgrades of an app to devices where it's already installed
  - The upgraded version may have different storage requirements and thus a different schema
  - Users don't want to lose their data in the transition from one app version to the next
  - ➢ It's your responsibility to provide algorithms for automatic migration of your app's data from any old version to the current one!
    - Requires discipline in versioning schemas carefully
      and providing robust, reliable upgrade logic that maintains database consistency

Managing the DB Schema Using `SQLiteOpenHelper`

- **`SQLiteOpenHelper`** is a convenience class implementing the previously mentioned setup steps
- You just need to fill in the logic for creating and upgrading DBs in a subclass, e.g.:

```java
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import com.bignerdranch.android.criminalintent.database.CrimeDbSchema.CrimeTable;

public class CrimeBaseHelper extends SQLiteOpenHelper {
  private static final int VERSION = 1;
  private static final String DATABASE_NAME = "crimeBase.db";

  public CrimeBaseHelper(Context context) {
    super(context, DATABASE_NAME, null, VERSION);
  }

  @Override
  public void onCreate(SQLiteDatabase db) { ... }

  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {...}
}
```

Tell **`SQLiteOpenHelper`** the filename of your database and the version of your DB schema in these constants

Creating the Database (`CrimeBaseHelper.java`)

- The **`onCreate`** method simply constructs an SQL CREATE statement and executes it on the given database:

```
@Override
public void onCreate(SQLiteDatabase db) {
    db.execSQL("create table " + CrimeTable.NAME + "(" +
            " _id integer primary key autoincrement, " +
            CrimeTable.Cols.UUID + ", " +
            CrimeTable.Cols.TITLE + ", " +
            CrimeTable.Cols.DATE + ", " +
            CrimeTable.Cols.SOLVED +
            ")"
    );
}
```

No column types specified here for simplicity, but would be a good idea to use them

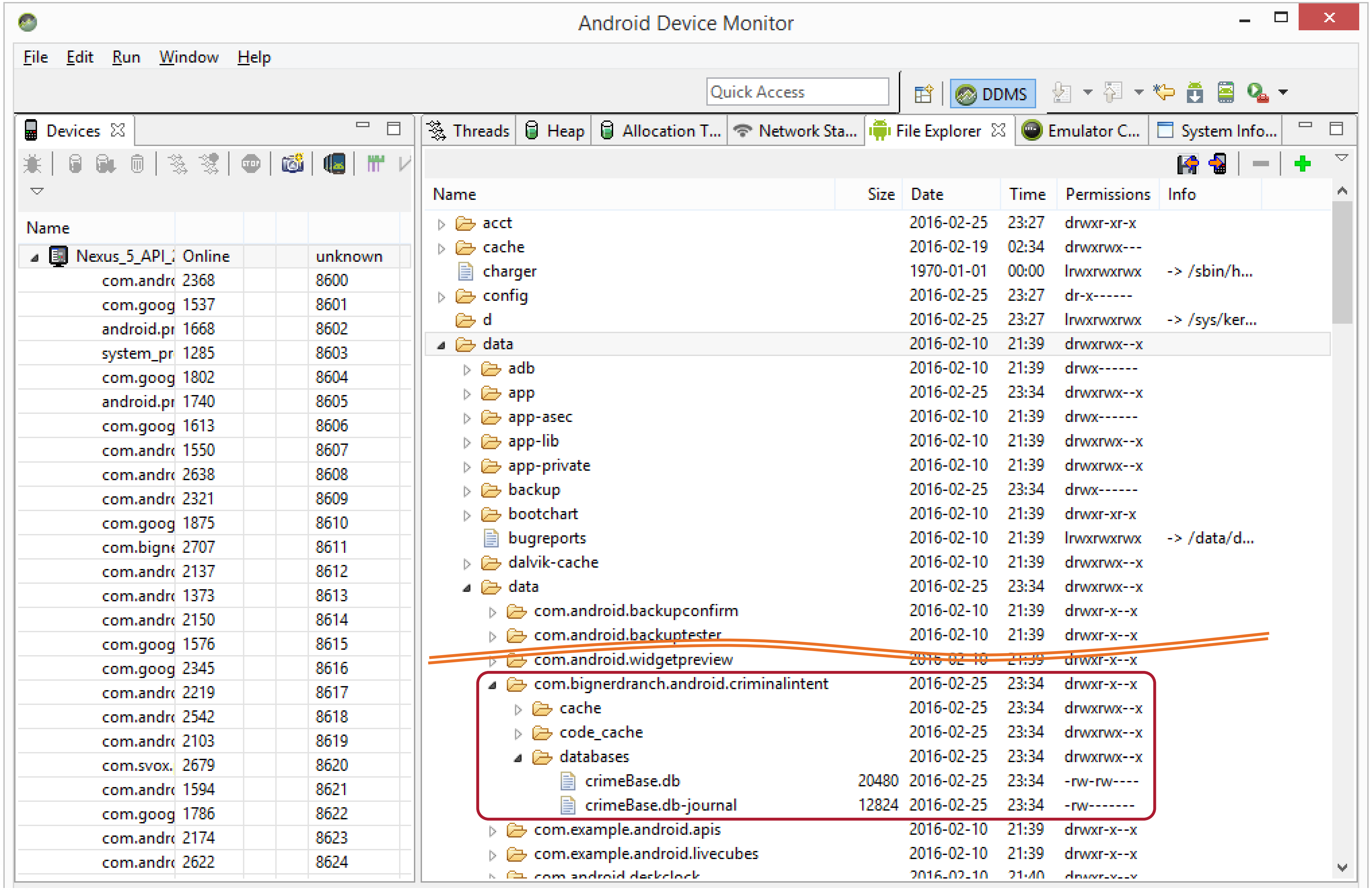
Updating the Database Schema

- While you are still developing the app:
  - Just uninstall the app from your development device (this will delete the DB as well)
  - Make any necessary changes to your **`onCreate`** method
  - Reinstall the app on your development device (this will recreate a fresh DB)

- When you are publishing an upgraded version of your app in the Play store:
  - Increment the version number in your **`VERSION`** constant
  - Implement logic in your **`onUpgrade`** method that migrates DB schema and contents from previous versions to your current version (Caution: this can get complex over time!)
    - Use e.g. SQL ALTER TABLE statements to modify your DB schema
  - Publish the new version of your app in the Play store
  - Upgrading an installed version of the app on a device will now automatically update its DB

Location of DB

- Resides in the app’s private internal storage folder
  - ➢ Not accessible by other apps
  - ➢ Deleted upon app uninstall



Name		
Nexus_5_API_2	Online	unknown
com.andr	2368	8600
com.goog	1537	8601
android.pr	1668	8602
system_pr	1285	8603
com.goog	1802	8604
android.pr	1740	8605
com.goog	1613	8606
com.andr	1550	8607
com.andr	2638	8608
com.andr	2321	8609
com.goog	1875	8610
com.bigne	2707	8611
com.andr	2137	8612
com.andr	1373	8613
com.andr	2150	8614
com.goog	1576	8615
com.goog	2345	8616
com.andr	2219	8617
com.andr	2542	8618
com.andr	2103	8619
com.svox.	2679	8620
com.andr	1594	8621
com.goog	1786	8622
com.andr	2174	8623
com.andr	2622	8624

Name	Size	Date	Time	Permissions	Info
acct		2016-02-25	23:27	drwxr-xr-x	
cache		2016-02-19	02:34	drwxrwx---	
charger		1970-01-01	00:00	lrwxrwxrwx	-> /sbin/h...
config		2016-02-25	23:27	dr-x------	
d		2016-02-25	23:27	lrwxrwxrwx	-> /sys/ker...
data		2016-02-10	21:39	drwxrwx--x	
adb		2016-02-10	21:39	drwx------	
app		2016-02-25	23:34	drwxrwx--x	
app-asec		2016-02-10	21:39	drwx------	
app-lib		2016-02-10	21:39	drwxrwx--x	
app-private		2016-02-10	21:39	drwxrwx--x	
backup		2016-02-25	23:34	drwx------	
bootchart		2016-02-10	21:39	drwxr-xr-x	
bugreports		2016-02-10	21:39	lrwxrwxrwx	-> /data/d...
dalvik-cache		2016-02-10	21:39	drwxrwx--x	
data		2016-02-25	23:34	drwxrwx--x	
com.android.backupconfirm		2016-02-10	21:39	drwxr-x--x	
com.android.backuptester		2016-02-10	21:39	drwxr-x--x	
com.android.widgetpreview		2016-02-10	21:39	drwxr-x--x	
com.bignerdranch.android.criminalintent		2016-02-25	23:34	drwxr-x--x	
cache		2016-02-25	23:34	drwxrwx--x	
code_cache		2016-02-25	23:34	drwxrwx--x	
databases		2016-02-25	23:34	drwxrwx--x	
crimeBase.db	20480	2016-02-25	23:34	-rw-rw----	
crimeBase.db-journal	12824	2016-02-25	23:34	-rw-------	
com.example.android.apis		2016-02-10	21:39	drwxr-x--x	
com.example.android.livecubes		2016-02-10	21:39	drwxr-x--x	
com.android.deskclock		2016-02-10	21:40	drwxr-x--x	

Obtaining the Database (`CrimeLab.java`)

- So far, we have just created the database, but we still need to give our model-layer classes a way to access it.
- In the Criminal Intent example, **`CrimeLab`** is the class managing the collection of all crimes, so it's the natural place to communicate with the database:

```
public class CrimeLab {

  private Context mContext;
  private SQLiteDatabase mDatabase;

  // ...

  private CrimeLab(Context context) {
    mContext = context.getApplicationContext();
    mDatabase = new CrimeBaseHelper(mContext).getWritableDatabase();
  }

  // ...

}
```

Remember **`CrimeLab`** is a singleton with a static factory method, that's why the constructor is private

As a singleton, CrimeLab will exist longer than any activity. That's why we are using the application context instead of the activity context here.

This is an **`SQLiteDatabase`** instance on which we can call **`execSQL`** etc.

`getWritableDatabase` is a method provided by **`SQLiteOpenHelper`** that will internally use our **`onCreate`** or **`onUpgrade`** methods to ensure a database with the required schema exists

Converting a Model Class’ Attributes to SQL Values (`CrimeLab.java`)

- For SQL INSERT and UPDATE operations, you’d usually have to provide a list of column names and values corresponding to all attributes of the model classes
  - Tedious and error-prone to keep converting this
  - Use a convenience method in **`CrimeLab`** to create a **`ContentValues`** map instead:

```java
private static ContentValues getContentValues(Crime crime) {
    ContentValues values = new ContentValues();
    values.put(CrimeTable.Cols.UUID, crime.getId().toString());
    values.put(CrimeTable.Cols.TITLE, crime.getTitle());
    values.put(CrimeTable.Cols.DATE, crime.getDate().getTime());
    values.put(CrimeTable.Cols.SOLVED, crime.isSolved() ? 1 : 0);
    return values;
}
```

- Essentially, this creates just a hashmap,
  but one that SQLite is prepared to work with

Inserting and Updating Database Rows (`CrimeLab.java`)

- Using the **`ContentValues`**, the INSERT and UPDATE operations are simple:

```
public void addCrime(Crime c) {
    ContentValues values = getContentValues(c);
    mDatabase.insert(CrimeTable.NAME, null, values);
}
```

Insert **`values`** into named table

Identify the item we want to update

```
public void updateCrime(Crime crime) {
    String uuidString = crime.getId().toString();
    ContentValues values = getContentValues(crime);
    mDatabase.update(CrimeTable.NAME, values,
            CrimeTable.Cols.UUID + " = ?",
            new String[] { uuidString });
}
```

Copy the item’s attributes into the hashmap

WHERE clause template, where all **`?`** characters will be replaced by strings passed in the following array

Array from which to replace the **`?`** occurrences in previous argument, *bound as strings*

That’s the reason for using this convoluted construct – it ensures the string is not executed as an SQL command, thus preventing SQL injection attacks

Keeping the Database Up-to-Date

- When should we call the **`updateCrime`** method to update the database?

- In the one-pane layout of the Criminal Intent app, we could call it whenever we are leaving the crime detail screen,
  i.e. when the activity hosting the **`CrimeFragment`** is paused:

```
@Override
public void onPause() {
    super.onPause();
    CrimeLab.get(getActivity()).updateCrime(mCrime);
}
```

- In the two-pane layout, the **`CrimeFragment`** is always visible (i.e. never left), so we could update the DB before each update of the list screen's UI
  - But also consider performance implications of too-frequent updates

Querying the Database

- The DB can be queried using **`SQLiteDatabase`**’s **`query`** method (with parameters reflecting the clauses of an SQL SELECT statement):

```
public Cursor query(
  String table,
  String[] columns,
  String where,
  String[] whereArgs,
  String groupBy,
  String having,
  String orderBy,
  String limit)
```

- The resulting **`Cursor`** object can be used to
  - step through the rows of the result set (iterator-like)
  - obtain the values for individual cells in the current row
    - which could then be used one by one to populate the attributes of corresponding model classes

Constructing Data Objects from SQL Query Results (CrimeCursorWrapper.java)

- Since piecing together an object from the cells in an SQL query result is tedious, we wrap the cursor in a class that provides this conversion for us:

```
public class CrimeCursorWrapper extends CursorWrapper {
    public CrimeCursorWrapper(Cursor cursor) {
        super(cursor);
    }

    public Crime getCrime() {
        String uuidString = getString(getColumnIndex(CrimeTable.Cols.UUID));
        String title = getString(getColumnIndex(CrimeTable.Cols.TITLE));
        long date = getLong(getColumnIndex(CrimeTable.Cols.DATE));
        int isSolved = getInt(getColumnIndex(CrimeTable.Cols.SOLVED));

        Crime crime = new Crime(UUID.fromString(uuidString));
        crime.setTitle(title);
        crime.setDate(new Date(date));
        crime.setSolved(isSolved != 0);

        return crime;
    }
}
```

`CursorWrapper` provides all of `Cursor`’s methods for iterating over an SQL result set

Get all values out of current result row

Create a data object and store the values in its attributes

Querying the Database and Providing Wrapped Results (`CrimeLab.java`)

- For our query purposes in the Criminal Intent app, we implement a convenience method that simplifies the interface of the `query` method and provides the results through a `CrimeCursorWrapper`:

```java
private CrimeCursorWrapper queryCrimes(String whereClause, String[] whereArgs) {
    Cursor cursor = mDatabase.query(
            CrimeTable.NAME,
            null, // Columns - null selects all columns
            whereClause,
            whereArgs,
            null, // groupBy
            null, // having
            null  // orderBy
    );
    return new CrimeCursorWrapper(cursor);
}
```

- This allows us to send simple queries for all items or one particular item
    - *(see following slides)*

Retrieving All Items (`CrimeLab.java`)

- To retrieve all items, we call **`queryCrimes`** with an empty WHERE clause...
- …and then iterate over the results (using the wrapped cursor) to populate a **`List`** of **`Crime`**s:

```
public List<Crime> getCrimes() {
    List<Crime> crimes = new ArrayList<>();

    CrimeCursorWrapper cursor = queryCrimes(null, null);

    cursor.moveToFirst();
    while (!cursor.isAfterLast()) {
        crimes.add(cursor.getCrime());
        cursor.moveToNext();
    }
    cursor.close();

    return crimes;
}
```

We want all items → empty WHERE clause

Iterate over rows in result set

Get current result row (our **`getCrime`** method takes care of turning the values into a **`Crime`** instance) and add it to the **`List`** of **`Crimes`**

Important – otherwise your app may run out of file handles and crash eventually

Retrieving a Particular Item (`CrimeLab.java`)

- To retrieve a certain item, we identify it by its UUID in the WHERE clause…
- …and then return the first result (if our query found the item)

```
public Crime getCrime(UUID id) {
    CrimeCursorWrapper cursor = queryCrimes(
            CrimeTable.Cols.UUID + " = ?", new String[] { id.toString() }
    );

    try {
        if (cursor.getCount() == 0) {
            return null;
        }
        cursor.moveToFirst();
        return cursor.getCrime();
    } finally {
        cursor.close();
    }
}
```

Creates the clause **`WHERE uuid = <UUID>`**, while preventing SQL injection

Return **`null`** if no item with the given UUID was found…

…or return the first item (which should be the only one if we rely on the UUIDs to be indeed unique)

Ensure we close the handle for the database file