FACULTEIT INGENIEURSWETENSCHAPPEN
EN ARCHITECTUUR

Labo Gebruikersinterfaces
Reeks 6: Android & Kotlin

21-22 april 2020

This lab session will get you to grips with the basics of Android development, and with the debugging and compilation tools of the Android IDE. You will create a Quiz application consisting of two activities, experiment with the different types of resource files, use data binding and pass `Intents` between activities.

You will learn the following concepts:

- Developer tools and panes of the Android Studio IDE
- SDK versions: minimum, target and compile
- Support libraries
- The structure of an APK package
- Porting an APK package onto a real device
- Building layouts in XML: `LinearLayout` and `ConstraintLayout` and the view attributes
- use Databinding
- Handling click events in code
- Configuration-specific resources and resource types: layouts, strings and string-arrays
- Displaying a toast message
- Communication between activities: intents and results
- Up and backwards navigation

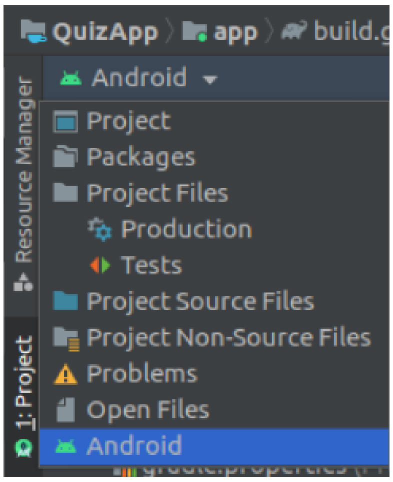
1 IDE: Android Studio

In this course we will use Android Studio, the official Android IDE. This program is installed and pre-configured on the lab PCs, together with the drivers for the mobile devices that are at your disposal during the lab sessions. To configure your own laptop, please refer to http://developer.android.com/sdk/index.html.

- Open Android Studio and start a new Android Studio project. First you need to select a project template, choose for **Empty activity**.
- Next, you have to decide which minimum SDK version you want to support. The minimum SDK version is a hard floor below which the OS will refuse to install the app. A low SDK version makes sure that most of the devices will be able to run your app, but this also means that you have to include *support libraries* that use certain new functionality. This means that your code may be more complex and that the resulting `.apk` package will increase in

size. In this course, we will only develop apps for phones and/or tablets that have SDK API 21 or higher (94.1% of the current active Android devices).

- Make sure that the selected language is **Kotlin**
- Get to know the project structure. You can change the view type of your app between 'Android' and 'Project'. The project view reflects the structure on disk (APK) and the Android view type is a flattened version of your project's structure that provides quick access to key source files. Some key files and directories in every Android application are:
  - **The Android manifest** (`AndroidManifest.xml`): This file describes the name and the components of the app. It also declares the permissions that the application must have in order to access protected parts of the API. This file is read by the Android runtime system when your app is executed.
  - **The Java folder**: This folder contains all the Kotlin files. There are three subfolders. the first folder contains all the Kotlin source code for your app. The other two folders contain code for (unit) testing.
  - **The Res folder**: Contains binary resources such as images as well as xml files defining strings, colors, ...
  - **Build.gradle**: Gradle is the build tool that is used to build the project and build.gradle allows you to configure the build process. Android Studio shows two `build.gradle` files which can be confusing at first. We will always work with the `build.gradle (Module: app)` file. An Android Studio project can have multiple modules (for example one main app and a library). Each module will have its own `build.gradle (Module: ...)` file that contains the specific configuration for that module while the `build.gradle (project: ...)` file contains all global settings, settings that apply to all modules in the project. The `build.gradle (Module: app)` file indicates the `compileSDKversion` (the version of the Android SDK that should be used to compile the source code), the `minSDKVersion` (the lowest version of the Android API that should be present to run the app) and the `targetSDKVersion` (the SDK version that the app was designed for).



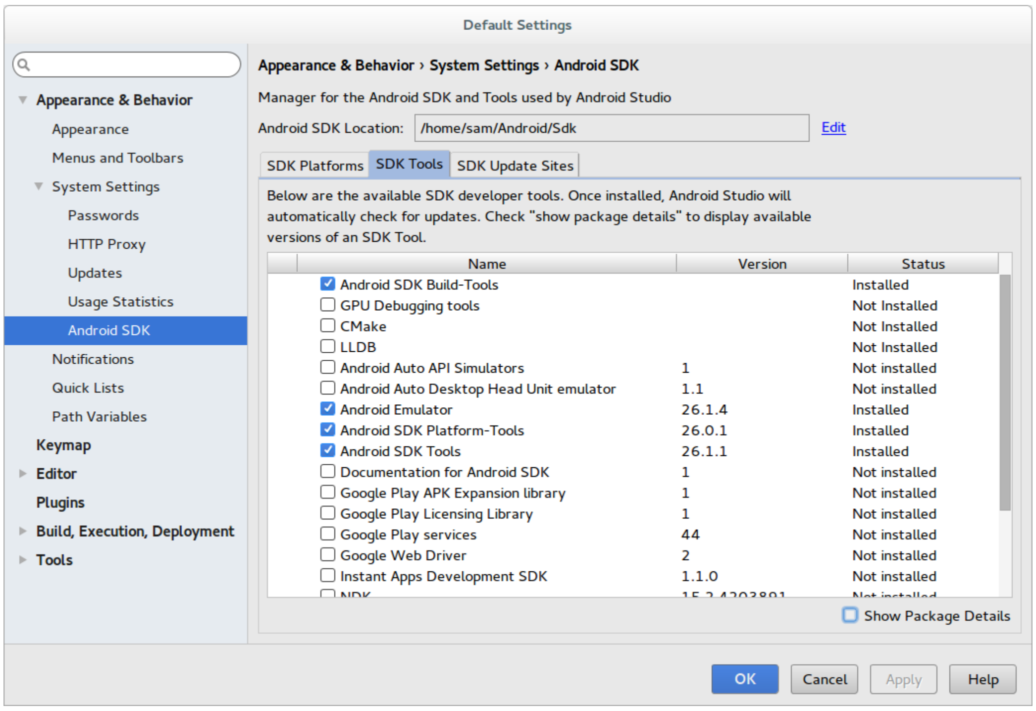
Figuur 1: Toggling between project views.

1.1 Android SDK Manager

The Android SDK bundles tools, platforms, and other components into packages you can download using the SDK Manager. When the SDK Tools or a new version of the Android

Platform is released, you can conveniently use the SDK Manager to update your environment.

- Open SDK Manager: **Tools > SDK Manager** or by clicking the corresponding menu bar icon.



Manager for the Android SDK and Tools used by Android Studio

Android SDK Location: /home/sam/Android/Sdk

Below are the available SDK developer tools. Once installed, Android Studio will automatically check for updates. Check "show package details" to display available versions of an SDK Tool.

	Name	Version	Status
✓	Android SDK Build-Tools		Installed
☐	GPU Debugging tools		Not Installed
☐	CMake		Not Installed
☐	LLDB		Not Installed
☐	Android Auto API Simulators	1	Not installed
☐	Android Auto Desktop Head Unit emulator	1.1	Not installed
✓	Android Emulator	26.1.4	Installed
✓	Android SDK Platform-Tools	26.0.1	Installed
✓	Android SDK Tools	26.1.1	Installed
☐	Documentation for Android SDK	1	Not installed
☐	Google Play APK Expansion library	1	Not installed
☐	Google Play Licensing Library	1	Not installed
☐	Google Play services	44	Not installed
☐	Google Web Driver	2	Not installed
☐	Instant Apps Development SDK	1.1.0	Not installed
☐	NDK	15.2.4203891	Not installed

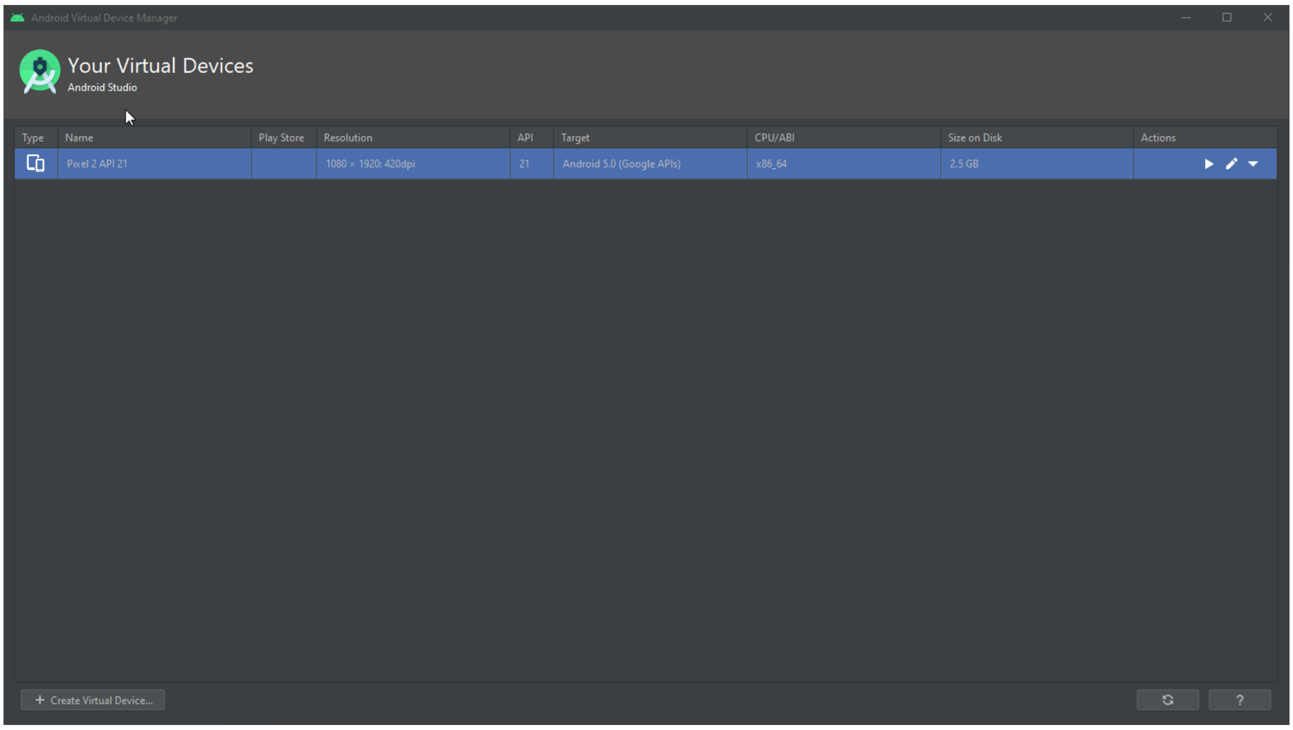
Figuur 2: The Android SDK Manager lists the status of each package: available, installed, or updateable.

- You will need the following packages
  - Android 5.0 (API 21) or higher
  - Android Emulator
  - Latest Android SDK Platform-tools

1.2 Android Virtual Device (AVD) Manager

The AVD Manager is a tool to create and manage Android Virtual Devices (AVDs), device images that are executed by the Android Emulator. This allows you to develop and test Android apps without having to install them on a physical device.

- Open AVD Manager: **Tools > AVD Manager** or click the corresponding icon in the toolbar.
- You will still need to create the virtual machine. Select Pixel 2 with API 21 or higher and system image x86(_64). The ARM images will run a lot slower since you are executing this on a CPU with x86 architecture.
- If it is the first time you use Android Studio you will need to download the system image. (A virtual device image is already downloaded on the lab PC's).
- Start an instance of this new device. This may take a while to complete, so keep this instance running for the entire lab session.



Type	Name	Play Store	Resolution	API	Target	CPU/ABI	Size on Disk	Actions
	Pixel 2 API 21		1080 × 1920: 420dpi	21	Android 5.0 (Google APIs)	x86_64	2.5 GB	

Figuur 3: The AVD Manager main screen provides an overview of all your virtual devices.

- Good to know: you can use the command line to control the advanced functionality of the emulator. https://developer.android.com/studio/run/emulator-commandline.html

Extra's before going on

Either you go on reading section 2, either you do one (or two) of the following before proceeding.

1. Read and follow the instructions in the online tutorial on codelabs.developers.google.com.
2. Solve the two reading exercices that you can find on Ufora. These are written in Dutch, and help you inspect the examples that were given in the theory course. You will learn *where* to write *what* part of the code.

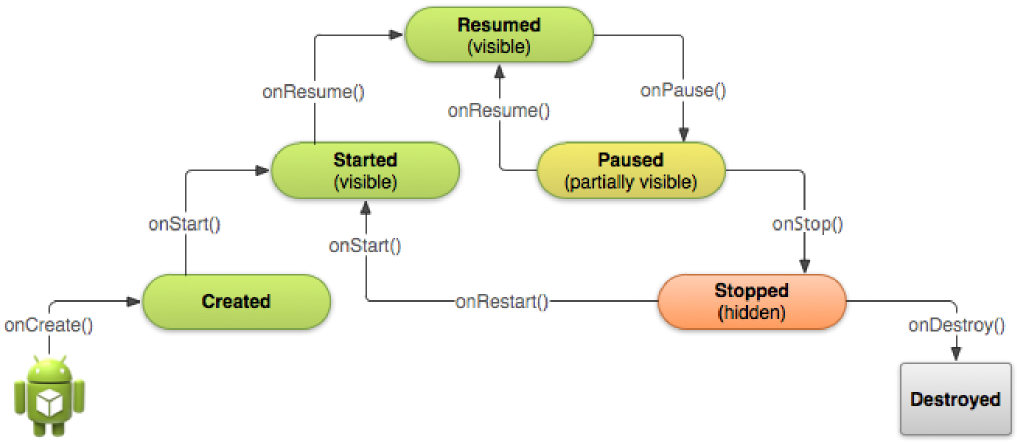
2 Debugging with adb logcat

Android Debug Bridge (`adb`) is a versatile command line tool to communicate with an emulated instance or a connected physical Android device. You can find the `adb` tool in `<sdk>/platform-tools/`.

- In order to use `adb` with a physical device connected over USB, you must enable **USB debugging** in the device system settings, under **Developer options**. On Android 4.2 and higher, the Developer options screen is hidden by default. To make it visible, go to **Settings** > **About phone** and tap **Build number** seven times. Return to the previous screen to find **Developer options** at the bottom.
- When you started a new project you chose the **Empty Activity** template. By selecting this template Android Studio added a java class which extends `AppCompatActivity`, a layout XML file in `res/layouts` and a reference to the `Activity` in your application's manifest file.

Make sure you understand the role of each file. The launcher or main activity is the default activity displayed when your app is started.

- Override all lifecycle callbacks in your `Activity` class. Use the automatic code generation via right mouse click on **Generate... > Override Methods...**. For the `OnCreate` callback, there is already an override method added. Figure 4 shows the different lifecycle callbacks.

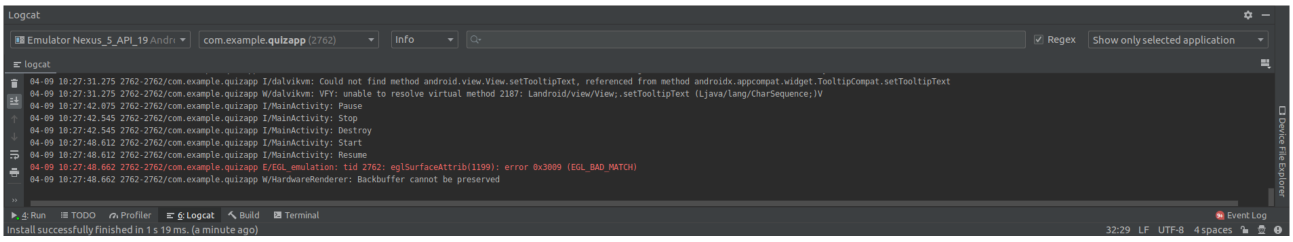


Figuur 4: The `Activity` lifecycle depicted as a step pyramid. For every callback used to take the activity a step toward the *Resumed* state at the top, there is a callback method that takes the activity a step down. The activity can also return to the resumed state from the *Paused* and *Stopped* state.

- Write log messages in each lifecycle callback. Experiment with different log levels.

To write log messages in your code, use the Log class. Log messages help you understand the execution flow by collecting the system debug output while you interact with your app. Log messages can tell you what part of your application failed. By default, the Android system sends `stdout` and `stderr` (`System.out` and `System.err`) output to /dev/null. On the emulator and some devices `stdout` and `stderr` gets redirected to logcat and printed using `Log.i()`. Make sure to always use the `Log` class for logging messages.

- View log messages in the Logcat tab and adjust the log level.
  - ▸ Start your application
  - ▸ Click **Android monitor** at the bottom of the screen to open the *Android DDMS* tool window.
  - ▸ If the system log is empty in the Logcat view, click **Restart**.



```
04-09 10:27:31.275 2762-2762/com.example.quizapp I/dalvikvm: Could not find method android.view.View.setTooltipText, referenced from method androidx.appcompat.widget.TooltipCompat.setTooltipText
04-09 10:27:31.275 2762-2762/com.example.quizapp W/dalvikvm: VFY: unable to resolve virtual method 2187: Landroid/view/View;.setTooltipText (Ljava/lang/CharSequence;)V
04-09 10:27:42.075 2762-2762/com.example.quizapp I/MainActivity: Pause
04-09 10:27:42.545 2762-2762/com.example.quizapp I/MainActivity: Stop
04-09 10:27:42.545 2762-2762/com.example.quizapp I/MainActivity: Destroy
04-09 10:27:48.612 2762-2762/com.example.quizapp I/MainActivity: Start
04-09 10:27:48.612 2762-2762/com.example.quizapp I/MainActivity: Resume
04-09 10:27:48.662 2762-2762/com.example.quizapp E/EGL_emulation: tid 2762: eglSurfaceAttrib(1199): error 0x3009 (EGL_BAD_MATCH)
04-09 10:27:48.662 2762-2762/com.example.quizapp W/HardwareRenderer: Backbuffer cannot be preserved
```

Install successfully finished in 1 s 19 ms. (a minute ago)   32:29 LF UTF-8 4 spaces

Figuur 5: The system log in Logcat.

- Use the log messages to verify which lifecycle methods are called in the scenarios below. Make sure you know how to simulate these scenarios in the emulator as well.
  - ▸ The user pressed the home button.

    - The user pressed the back button.
    - The user removed a task from the recent apps.
    - The user changed the device orientation.

  Hint: If you override a lifecycle method, do not forget to call the **`super`** method.

- In which of the above scenarios is the onSaveInstanceState() method called ?

3 First Android application: QuizApp!

We will now start the development of a quiz application. Our quiz application has two screens. On the first screen, the user is presented with a question. He can press an OK button to check his answer, or he can press a button to ask for a hint. This hint is presented in a second activity. In the second activity, the user is presented with three buttons: "OK","Text Hint" and "Image Hint". If the user presses "Text Hint", a textual hint is shown. Accordingly, pressing "Image Hint" will reveal a more visual hint. The user returns to the first activity by pressing the "OK" button or the back button in the navigation bar.

3.1 The model

The model of our application consists of two classes: `Question` and `QuizMaster`. Given that the model is not the focus of this lab, we have kept these classes rudimentary. For instance, the list of questions is hardcoded instead of fetched from a database. In later lab sessions, you will learn how to use repositories.

- The Kotlin files can be downloaded from Ufora.
    - Create a Kotlin package "model" as a sub-package of your app package. right click on com.example.quizapp > new > package
    - Import both classes in the newly created package.
    - Make sure to set the correct **`package`** declarations in both classes.

4 The main activity of our quiz application

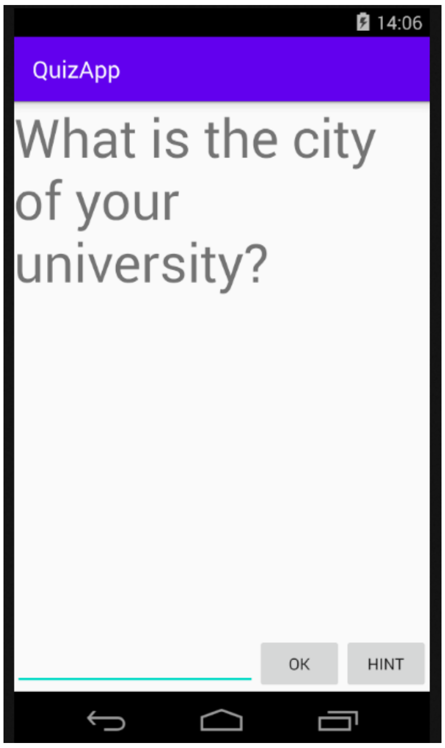
We will now declare the layout and code the main activity.

4.1 Layout declaration

The Empty Activity template creates a XML layout file in the `app/res/layout` folder with a `ConstraintLayout` root view. When you open a layout file in Android Studio, you are first presented the **Design Editor**. In the upper right corner you find three icons, you can use these icons to switch between the **Design**, **Code** or **Split** views. Figure 6 shows an example layout.

To create this layout, you have to:

- Modify the layout file and use a vertically oriented LinearLayout container as root container in your layout instead of the ConstraintLayout. You can't do this using the design editor.
- Include a `TextView` to show the question, an `EditText` field where the user can write his answer and two buttons ("OK" and "Hint"). You can use Layout elements as children of another Layout element.



Figuur 6: Example layout of the main activity

- Make sure that the `TextView` uses all available space, using the `layout_weight` attribute.
- Use the correct layout parameters for each component so you get a layout like in figure 6.
- Increase the font size of the `TextView`. Keep in mind that `px` (pixel) is not used as a measure for size in Android, because the actual size of a pixel depends on screen density and resolution. Instead, use `sp` (scaled independent pixels) for fonts. Density-independent pixels (`dp`) are similar but are not used for fonts because font sizes can be scaled by the user's font size preferences.
- It is bad practice to hard code the string values in your app. Android includes a convenient way of storing these values in XML resource files (`values` folder in your project). Create `string` resources for the labels of both buttons. The label of a button is defined in the `text` attribute. The light bulb in the left margin of the layout editor can be used to generate a `string` resource entry from hard coded strings in code or XML.
- Translate your application by adding new strings resource files with a language qualifier to the `values` folder. This will create a new directory and files in the Project view with a name akin to `values-nl/strings.xml`. You can use the resource new file wizard to see all qualifiers and their values. Use the "Custom locale" app on the emulator to try out a different language setting or change the locale in the designer view of the layout file. Strings which are not translated will use the value in the default strings resource file.

Good to know:

- Use <ctrl> + <space> to open suggestions based on the location of the cursor. Other shortcuts can be found here: https://developer.android.com/studio/intro/keyboard-shortcuts.
- Missing attributes or methods can be automatically generated by moving the cursor onto the desired location or method.
- All attributes like "text" are defined in the **"android:" namespace**. There is also a "tools" namespace that contains attributes with the same name but with a different

purpose. The “tools” attributes are only used to render the layout in the preview window at design time. They are not used in the final compiled app. In this application we can use "tools:textäs a placeholder for the question. In most cases you will set the “android” attributes.

4.2 Showing the question manually

For years the default approach to set attributes of Views was to first obtain a reference to the view object with `findViewById(R.id....)and then to call a method on this object.`

- Write the code in MainActivity to :
  - Use the Kotlin object `QuizMaster`.
  - load a random question from the `QuizMaster` and show the text of that question to the user.

4.3 Use data binding to show the question

Showing the text manually is easy but not very convenient if you want to set multiple attributes or if you want to update the property regularly. Instead the new Android Architecture Components introduce the Data Binding Library that allows you to bind UI components and data declaratively (in XML) instead of programmatically (in Kotlin). For more information see https://developer.android.com/topic/libraries/data-binding/expressions.

- Configure your build.gradle file (app module) to enable databinding:

```
android {
    ...
    dataBinding {
        enabled = true
    }
}
```

Also add: `apply plugin: 'kotlin-kapt'` under the other applies and `kapt 'com.android.databinding:compiler:3.1.4'` in the dependecies section.

- Convert your layout XML to data binding layout. This means that the root element of your XML should be a `<layout>` element and has a XML child `<data>` element with `<variable>` elements for each data object.

- Create a `<variable>` property that stores the `QuizMaster` object in your XML.

- Create a binding in the XML, using the @ syntax between the `text` attribute of our `TextView` and the `currentQuestion.question` attribute of the `QuizMaster` data binding variable.

- In the `onCreate()` method of the activity, obtain an instance of the binding class and assign the QuizMaster instance to the binding class. To use auto-completion try building your project so the binding class is generated.

- By default, the name of the `DataBinding` class is based on the name of the layout file, converting it to Pascal case and adding the Binding suffix to it. For instance, if your layout filename is `activity_main.xml` so the corresponding generated class is `ActivityMainBinding`.

- If your `DataBinding` class is not generated, there is likely a syntax error in your XML. Check this first. Also check if you have the correct name for the correct class (see the naming rules above). You can find the generated class by switching to Project mode and navigate to `<module>\build\generated\source\kapt\debug\<package>`. If you are sure there are no errors, you can try to clear Android Studio's cache by Build > Rebuild project. In rare cases, only restarting Android Studio seems to help.

4.4 Handling events

Currently, pressing the buttons in our layout has no effect. To execute a method when a certain event (such as a click) occurs you have different options:

- The "old-fashioned" way of attaching a method of your activity to the event:
  - In XML: `android:onclick="clickHandler"`. This requires a public method "click-Handler"which accepts a View parameter in the corresponding Activity class.
  - From code: `val b: Button = findViewById(R.id.btn) b.setOnClickListener (...)`
- With Databinding:
  - Method references
  - Listener bindings

Here we will use Data binding to write expressions in the XML of how events should be handled that are dispatched from the views. Event attribute names are determined by the name of the listener method (with a few exceptions). For example, `View.OnClickListener` has a method `onClick()`, so the attribute for this event is `android:onClick`.

We will use Method references for handling click events on the Hints button and Listener bindings for the OK button.

4.4.1 The Hints button

In your binding expression, you can only reference methods that conform to the signature of a listener method:

- `public` access modifier
- `void` return value
- a single `View` object parameter (this will be the `View` that was clicked)

- Create a method with the correct signature in the Activity and show a `Toast` when the button is pressed. You can find more information on https://developer.android.com/guide/topics/ui/notifiers/toasts.
- Add a handler data variable of the MainActivity class in XML and configure this data binding object in the `onCreate()` of the Activity.
- In your XML, set the `android:onClick` attribute of the Hint button to this method, using a method reference. See: https://developer.android.com/topic/libraries/data-binding/expressions.

4.4.2 The OK button: first version with one-way binding

When pressing the OK button, we should validate the answer of the user, give some feedback and show the next question. As a first step, we will only focus on how a new question is shown to the user when pressing the OK button. Validation of user input and giving feedback on the answer will be handled in the next step.

To realize this functionality, we need the following flow: the button press must be communicated to the `Activity`, which contacts the model (`QuizMaster`) to select a new question. Then, the member variable of `QuizMaster` is updated with the new question, which should in turn trigger an update of the `android:text` attribute of our `TextView`.

One straightforward way could be to implement a listener method in the `Activity`, using the same approach as we did for the Hints button. In this listener method, we would then call a method of the `QuizMaster` to update the question. Then, our listener method should get the updated text from the `QuizMaster` and update the `TextView`. There is however a much cleaner way to implement this behavior, using listener bindings and `ObservableData`. Using an XML-defined listener binding, we will directly call a method in the `MainActivity`. The update of the question will automatically update the text shown by the `TextView`.

We will first create the binding:

- Create a **listener binding** method in your MainActivity which is called when the Ok button is pressed. Call the `nextQuestion()` method of `QuizMaster` inside this method. In a full-fledged app, we would validate the answer of the user, update the score and select a new question. As said, we will keep things simpler: in this method you should only ask the model to select a new question and update your member variable that holds the question text. In method reference bindings (the approach we used for the Hint button), the parameters of the method must match the parameters of the event listener. In listener bindings, only your return value must match the expected return value of the listener (unless it is expecting void). You can check the documentation here: https://developer.android.com/topic/libraries/data-binding/expressions.
- Define the correct **listener binding expression** in your layout XML which uses the MainActivity handler to call the previous created method.
- Launch the app and confirm using log statements that the `nextQuestion()` method is called and that your member variable holding the question is updated. However, the `TextView` will not be updated.

To have our view automatically updated when the data in our `ViewModel` changes, we need to refactor our member variable to an observable:

- Change the type of the `currentQuestion` variable from `Question` to `ObservableField< Question>`. (It is possible to postpone the initialisation to the `init`-method, but there is less reason to do so. Initialisation of an object of type `ObservableField<...>` is possible with the default constructor.) **Why should we change the type from `Question` to `ObservableField<Question>`?**

  Solution: If we don't do this, the UI element will not be updated when the value of `currentQuestion` changes. The values of the properties are read when the binding is established.

- In the `nextQuestion()` method, update the value of the `ObservableField` using `set` . You can find the documentation here: https://developer.android.com/topic/libraries/data-binding/observability.
- Make sure that you instantiate `currentQuestion` by calling the constructor before setting the variable.

4.4.3 The OK button: second version with two-way binding

When the user presses a button, we should not only show a new question, but also provide feedback to the user of the answer he entered in the `EditText`. One way to realize this would be to use a method reference binding, and implement a `onOK` listener method in our `Activity`. This method would then fetch the current text in the `EditText`, call a `validateUserInput(String)` method and use the result of that method call to show feedback to the user (e.g. in a Toast).

While the above implementation is perfectly valid, we will show a different approach, using two-way binding. With one-way binding, the UI is automatically notified (and updated) when the data of our `Observable` changes. This is called one-way binding: from the `Model` to the `View`. With two-way binding, updates also flow in the opposite direction. If the user changes something in the UI (e.g. entering username, password, or a quiz answer), the `Model` is automatically informed about the new data.

- Add a new member variable `userAnswer` to your `QuizMaster` that will hold the content of the `EditText`. This member variable must be of type `String` or `ObservableField<String>` if we want to clear the text and update the UI.
- Realize two-way binding between this member variable and the `android:text` attribute of your `EditText`, by modifying your XML as follows: `android:text="@={quizMaster.userAnswer}"`. Notice the "=" in the XML! Every time the user enters text, the `userAnswer` is automatically updated.
- Validate the answer of the user (using `validateAnswer` of `Quizmaster`) and only call `nextQuestion` if the answer was correct.
- Clear the text in `EditText` after validating the input. Also show an appropriate message in a Toast to give feedback to the user about the given answer.

5 The Hints activity

We will now implement the second screen of our quiz app that shows hints. We will follow the same methodology as for the implementation of the main activity: declare a layout, implement the `HintActivity` with a shared model and use data binding to interact.

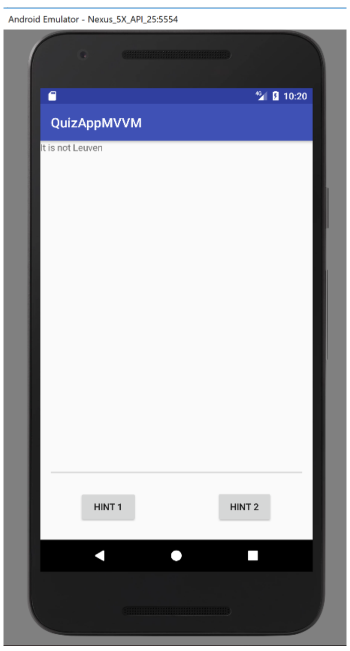
The activity contains two buttons: one button to show a textual hint, and one button to show a visual hint. Pressing the back button or up button will return the user to the main activity informing it if one of the hints has been seen.

In this lab session, you will learn the following concepts

- Intent-based navigation between activities, returning a result, backwards navigation, up navigation
- Constraintlayout
- New types of View objects: ImageView
- Normal permissions (not requiring user approval)
- Advanced data binding concepts: BindingAdapter, import statements in XML, dynamic visibility of View components

5.1 Declaring the layout

We will realize the layout shown in Figure 7.



Figuur 7: Layout of the Hints activity (left) and example constraints (right).

- Add a new **empty** `Activity` to your project to show the hints.
- Use a `ConstraintLayout` for this activity. In such a layout, each `View` must be constrained vertically **and** horizontally by at least one constraint. This constraint can be to another `View`, or to a guideline.
  - All relevant documentation can be found here: https://developer.android.com/training/constraint-layout/.
  - Add a vertical guideline, exactly in the middle of the screen.
  - Add a first button. This button must be centered horizontally between the left of the screen and the vertical guideline, and vertically aligned to the bottom of the parent (screen).
  - Add a second button. This button must be centered horizontally between the vertical guideline and the right of the screen.
  - Add a Horizontal Divider that we will use to separate the upper part where the hints are shown from the bottom part with the two buttons. The divider should have a height of 3dp, and left/right margins of 16dp. Align the divider to the parent (on all contraints).
  - Align the top of the first button to the divider. And align the top of the second button to the top of the first button. **Question: why don't we align the top of the button to the divider, like we did for the first button?**

    Solution: Because if we want to change something, we only have to fiddle with the constraints of the first button and the second one will automatically shift.
  - Set the vertical bias attribute of the Horizontal Divider to get a pleasant layout.
  - Add a `FrameLayout` with a single child element `TextView`. Center this `FrameLayout` horizontally and vertically. It should have 16 dp margin left, right, above and below (until the divider). Increase the font size of the `TextView` to the same value you confi-

gured for the `TextView` in the main activity. Using a `FrameLayout` allows us to use the same constraints for additional views grouped in this layout.

- You can test this Activity by moving the intent-filter in the `Manifest` from MainActivity to HintsActivity and run the app. If you forgot to add a horizontal and/or vertical constraint for a `View` element, it will be floating in an unexpected place. Rather annoyingly, you will only discover this when running the app and **not** in the preview pane!
- Make sure that all widths and heights of the direct child elements of `ConstraintLayout` are set to `match_constraint` (in design view) or 0dp (in text view) and not to a hardcoded size. The only exception is the height of the divider.

5.2 Communication between activities

We will now set up the communication between the two activities.

- When the user presses the Hint button in the main activity, the `HintsActivity` must be started. This can be done with an `Intent`.
  - The `HintsActivity` must know for which question it should show the hints and it can use the shared model `QuizMaster`. When no shared model is used you can store additional information in the intent with the `putExtra()` and `getXXXExtra()` methods.
  - Implement a factory method in the `HintsActivity` that creates an `Intent` with the appropriate extras. **Question: why don't we create and set the extras in the calling class without using factory method?**

    Solution: It is not required to use a factory method but is a good guideline to follow. A factory method places the responsibility to the developer of that class.

  - When the user presses the Hints button in the main activity, call this factory method and use the `Intent` to start the `HintsActivity`. Remember that we want a result back from the `HintsActivity`, so use the correct method to launch this activity!
- Load the correct hints into the View using databinding. You can bind the `QuizMaster` to the hints layout and get the hints text through the current question.
- When the user presses the back button, we must return to the main activity.

5.3 Showing and hiding the hints

- Add an attribute to the `HintsActivity` that indicates whether the textual hint, the image hint or no hint is visible. The `TextView` is only visible if the user has pressed the left hint button.
- Add a method `onShowTextHint` that will be called when the user presses the left button. In the body of this method, you make this `TextView` visible and set the results for the `MainActivity` (use the setResult methode).
- Create the necessary bindings in XML
  - The `onShowTextHint` method must be called when the left button is pressed.
  - Bind the visibility of the `TextView` to the corresponding variable in your `Activity`, using a binding expression. You can find an example here: https://developer.android.com/topic/libraries/data-binding/expressions. To use the `View.GONE` and `View.VISIBLE`

constants in your XML, you need to import the `View` class. You can find the documentation on how to do this on the same page.

5.4 Switching between the TextView and the ImageView

We will now implement the functionality to show a visual hint. The image will be downloaded from a webserver.

- Add an `ImageView` to the `FrameLayout` of the layout of `HintsActivity`.
- Add the necessary code to set the visibility of the `ImageView`.
  - Introduce a second attribute in your `Activity` to show the image hint. **Why can't we just use the inverse of the attribute we used to track the visibility of the `TextView`?**

    Solution: Because the `TextView` is also invisible by default. Using the inverse would thus mean that the `ImageView` is visible by default!.
  - Implement all necessary functionality to appropriately toggle the visibility: handle button presses, the `TextView` must be set to invisible when showing the visual hint.
- Set the result for `MainActivity` when the button of the image hint is pressed.
- The image to show must be set in the `app:imageUrl` attribute of the `ImageView`. However, this only works for locale files while our image must be downloaded from the internet and this cannot be coded in a binding expression. Instead, we will use a binding adapter: this is a method that will be called when the attribute must be set.
  - All relevant documentation about binding adapters can be found here: https://developer.android.com/topic/libraries/data-binding/binding-adapters
  - Create a new Kotlin object `ImageViewBindingAdapter` with a binding for a custom attribute called `imageRemoteUri` and use this attribute in your XML layout file. The Android SDK will automatically look for annotated methods (`@BindingAdapter`). We should also add `@JvmStatic` because the binding code generator expects a static method when the code is translated to Java. You can find more information here: https://medium.com/@thinkpanda_75045/defining-android-binding-adapter-in-kotlin-b08e82116704.
  - Inside this method we will use the Glide library to download an image. This library has built-in support for caching, resuming interrupted downloads, etc. To set up Glide, you need to add the dependencies your Gradle file (of your app module, not of your project):

    ```
    implementation 'com.github.bumptech.glide:glide:4.11.0'
    kapt 'com.github.bumptech.glide:compiler:4.11.0'
    ```

  - You can easily download an image in your bindingadapter with the following one-liner: `Glide.with(imageView.context).load(imageUri).into(imageView)`. `imageView` and `imageUri` are arguments of your binding method.
  - In the code above, `imageView.context` is the `Context` of the `View` element into which the image must be shown; `imageUri` is the URI of the image, and `imageView` is a reference to the `ImageView` object.
- You will need to configure your manifest file to use the internet permission. You can do this by adding the following lines to above the application tag:

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
```

5.5 Back to the MainActivity

In the `MainActivity` activity, we will show a `Toast` message that indicates if we viewed the hint or not.

- ▸ Override the `onActivityResult` method in `MainActivity`, check the request code, check the result code (RESULT_OK or RESULT_CANCELLED) and parse the intent showing the feedback message to the user. This feedback depends on which hint(s) have been viewed.

6 Optional - i18n

Make the text of the toasts (the feedback message) dependent of the language that the user has selected on his device. Use `plural resources` (nl/fr/en) to achieve this.
Changing the language of the quiz questions (and answers) is not part of this task.

7 Optional - Up vs back navigation

- Read the up vs back guidelines here: https://developer.android.com/design/patterns/navigation.html
- In our case it is not strictly necessary to provide an up button but it might improve the user experience. Add an up button to the second activity (https://developer.android.com/training/appbar/up-action
- Override the method `onOptionsItemSelected` so it will finish the current Activity. Without doing this the result for this Activity will not be reported to the `MainActivity`.