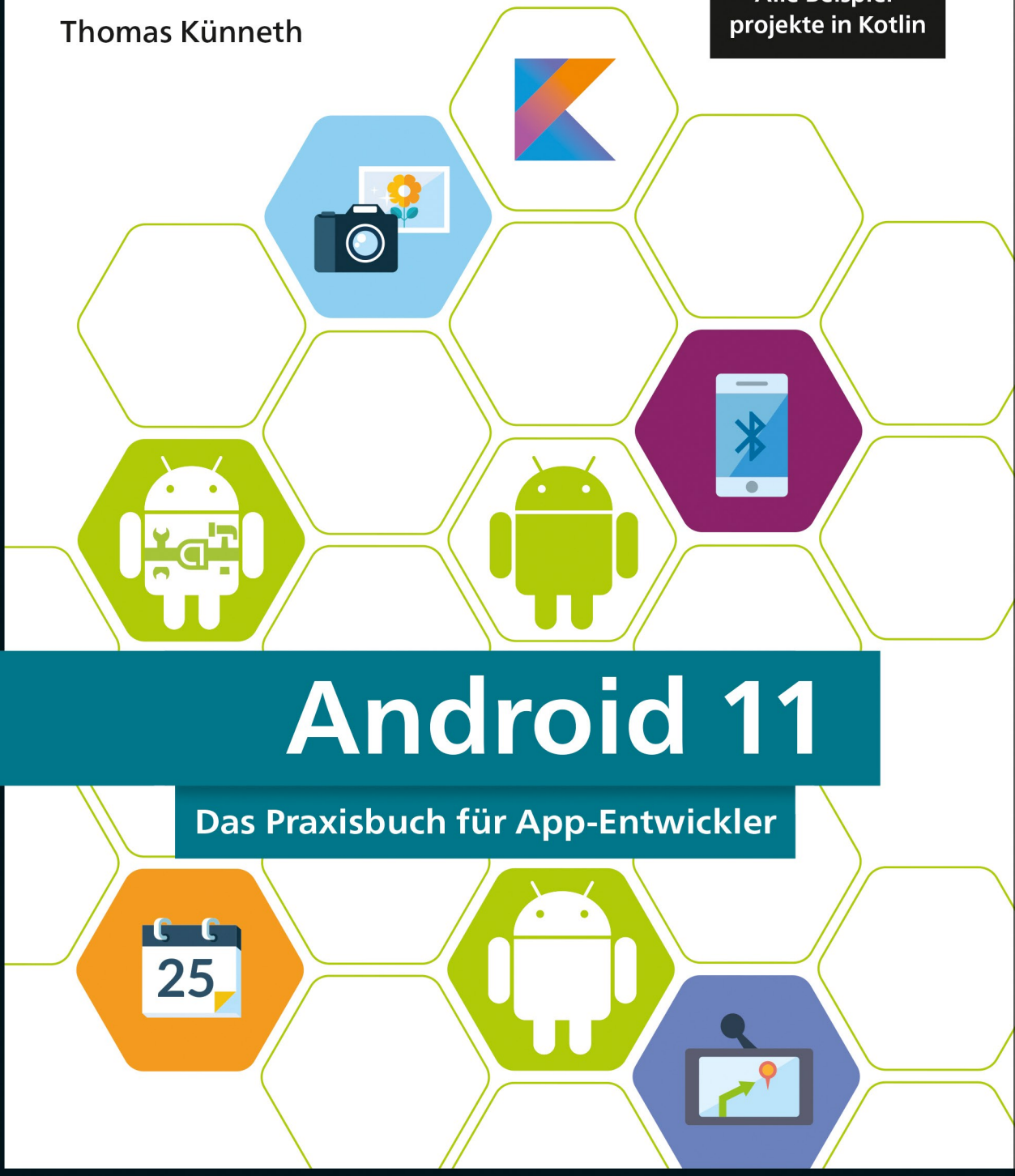
Thomas Künneth

Alle Beispiel-projekte in Kotlin



Android 11

Das Praxisbuch für App-Entwickler

- Professionelle Apps für Smartphone und Tablet
- Von der Idee bis zur Veröffentlichung in Google Play
- Multimedia, Bluetooth, Sensoren, GPS, Kalender, GUIs, Multitasking u. v. m.

Alle Beispielprojekte zum Download

Rheinwerk Computing

Kapitel 2
Hallo Android!

Die erste eigene App ist schneller fertig, als Sie vielleicht glauben. Dieses Kapitel führt Sie in leicht nachvollziehbaren Schritten zum Ziel.

Seit vielen Jahrzehnten ist es schöne Tradition, anhand des Beispiels »Hello World!« in eine neue Programmiersprache oder Technologie einzuführen. Dahinter steht die Idee, erste Konzepte und Vorgehensweisen in einem kleinen, überschaubaren Rahmen zu demonstrieren. Google bleibt dieser Tradition treu: Wenn Sie in Android Studio ein neues Projekt anlegen, entsteht eine minimale, aber lauffähige Anwendung, die den Text »Hello World« ausgibt. Im Verlauf dieses Kapitels erweitern Sie diese Anwendung um die Möglichkeit, einen Nutzer namentlich zu begrüßen. Ein Klick auf FERTIG schließt die App.

Hinweis

Sie finden die vollständige Version des Projekts sowie alle weiteren Beispiele auf der Seite *https://www.rheinwerk-verlag.de/android-11_4891/*. Alternativ können Sie auch das Repository unter *https://github.com/tkuenneth/begleitmaterialien-zu-android-11* klonen. Um sich mit den Entwicklungswerkzeugen vertraut zu machen, rate ich Ihnen aber, sich die fertige Fassung erst nach der Lektüre dieses Kapitels anzusehen.

2.1 Android-Projekte

Projekte fassen alle Artefakte einer Android-Anwendung zusammen. Dazu gehören unter anderem Quelltexte, Konfigurationsdateien, Testfälle, aber auch Grafiken, Sounds und Animationen. Natürlich sind Projekte keine Erfindung von Android Studio, sondern bilden eines der Kernkonzepte praktisch aller modernen Entwicklungsumgebungen. Grundsätzlich können Sie mit beliebig vielen Projekten gleichzeitig arbeiten. Projekte werden über die Menüleiste angelegt, (erneut) geöffnet und geschlossen. Ein Android-Studio-Hauptfenster bezieht sich aber stets auf ein Projekt. Wenn Sie ein vorhandenes Projekt öffnen, fragt die IDE normalerweise nach, ob Sie es in einem neuen oder im aktuellen Fenster bearbeiten möchten. Im letzteren Fall wird das aktuelle Projekt geschlossen. Sie können dieses Verhalten übrigens im SETTINGS-

Dialog auf der Seite Appearance & Behavior • System Settings unter Project Opening ändern.

Projekte können aus einem oder mehreren *Modulen* bestehen. Google nutzt dieses Konzept unter anderem, um Projekte für Smartwatches zu strukturieren. Diese bestehen oft aus einem Teil für das Smartphone oder Tablet sowie einem für das Wearable. »Klassische« Android-Apps kommen üblicherweise mit einem Modul aus. In diesem Fall nennt der Assistent das Modul *app*. Beispiel-Apps von Google verwenden als Modulnamen gelegentlich *Application*.

2.1.1 Projekte anlegen

Um ein neues Projekt anzulegen, wählen Sie in der Menüleiste des Hauptfensters File • New • New Project. Alternativ können Sie im Willkommensbildschirm auf Start a new Android Studio project klicken. In beiden Fällen öffnet sich der Assistent Create New Project, der Sie in wenigen Schritten zu einem neuen Android-Projekt führt. Auf der ersten Seite, Select a Project Template, legen Sie dessen grundlegenden Inhalt fest. Selektieren Sie, wie in Abbildung 2.1 dargestellt, auf der Registerkarte Phone and Tablet bitte Empty Activity, und klicken Sie danach auf Next.

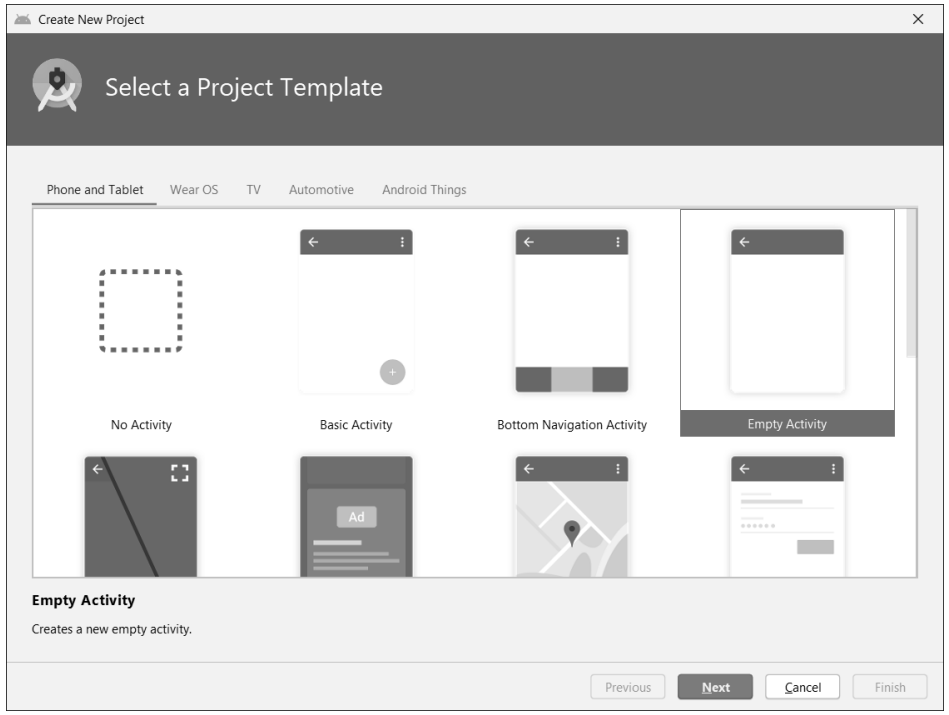


Abbildung 2.1 Projektvorlage auswählen

Auf der nun angezeigten Seite Configure Your Project machen Sie wichtige Angaben zu Ihrer App. Der Name wird später auf dem Gerät bzw. im Emulator angezeigt. Bitte geben Sie dort »Hallo Android« ein. Bei der Vergabe des Package name müssen Sie sorgfältig vorgehen, vor allem, wenn Sie eine Anwendung in *Google Play* veröffentlichen möchten. Denn der Paketname, den Sie hier eintragen, referenziert *genau eine* App, muss also eindeutig sein. Gelegentlich wird der Package Name deshalb auch *Application ID* genannt. Idealerweise folgen Sie den Namenskonventionen für Kotlin- oder Java-Pakete und tragen in umgekehrter Reihenfolge den Namen einer Ihnen gehörenden Internetdomain ein, gefolgt von einem Punkt und dem Namen der App. Verwenden Sie nur Kleinbuchstaben, und vermeiden Sie Sonderzeichen, insbesondere das Leerzeichen.

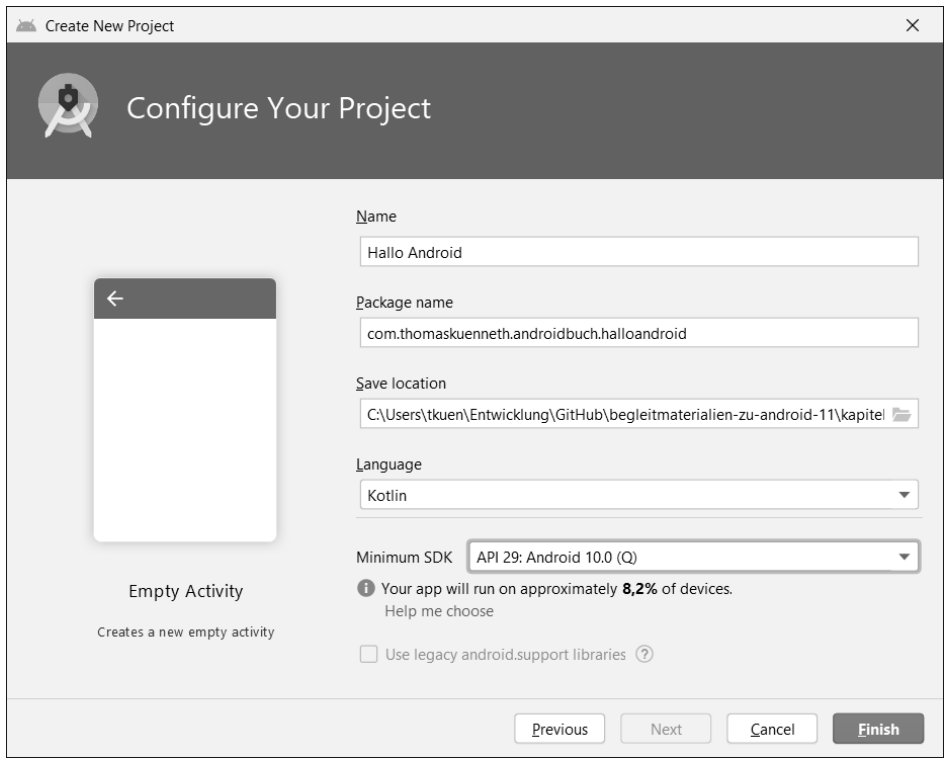


Abbildung 2.2 Das Projekt konfigurieren

Bitte tragen Sie »com.thomaskuenneth.androidbuch.halloandroid« als Package name ein. Unter Save location legen Sie den Speicherort des Projekts fest. Als Programmiersprache verwenden wir Kotlin. Die übrigen Einstellungen können Sie unverändert lassen. Der Dialog sollte in etwa Abbildung 2.2 entsprechen. Mit Finish schließen Sie den Assistenten. Android Studio wird nun eine Reihe von Dateien anlegen und das neue Projekt einrichten.

Kurzer Rundgang durch Android Studio

Danach sollte das Hauptfenster der IDE in etwa Abbildung 2.3 entsprechen. Es enthält unter anderem eine Menüleiste (unter macOS sowie manchen Linux-Distributionen erscheint diese stattdessen am oberen Bildschirmrand), eine Toolbar, mehrere Editorfenster für die Eingabe von Java- bzw. Kotlin-Quelltexten und anderen Dateiformaten, einen Designer für die Gestaltung der Benutzeroberfläche, eine Statuszeile sowie mehrere Werkzeugfenster.

Beginnt der Name eines solchen Fensters mit einer Ziffer, können Sie es über die Tastatur anzeigen und verbergen. Drücken Sie hierzu die angegebene Zahl zusammen mit der Alt-Taste. Auf dem Mac verwenden Sie cmd.

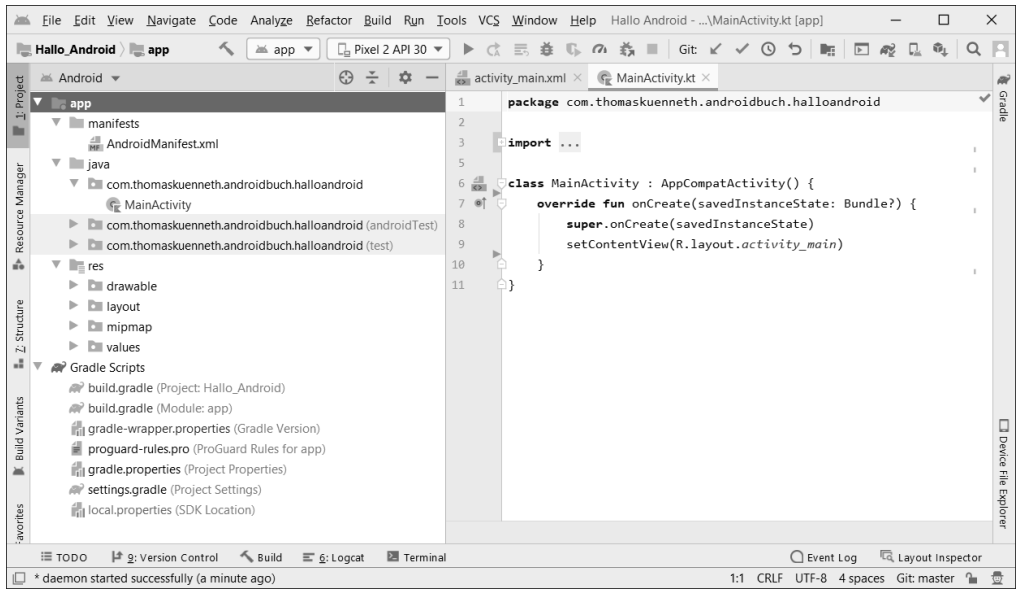


Abbildung 2.3 Das Hauptfenster nach dem Anlegen eines Projekts

Werkzeugfenster erscheinen im unteren, linken oder rechten Bereich des Hauptfensters. Ihre Position lässt sich über den Eintrag Move To des Kontextmenüs steuern, das Sie durch Anklicken des Fenstertitels mit der rechten Maustaste aufrufen. Ein Beispiel ist in Abbildung 2.4 zu sehen. Situationsabhängig kann eine ganze Reihe von zusätzlichen Menüpunkten enthalten sein.

Die Aufteilung des Android-Studio-Hauptfensters lässt sich praktisch nach Belieben den eigenen Bedürfnissen anpassen. Beispielsweise können Sie Werkzeugfenster als schwebende Panels anzeigen lassen (View Mode • Float) oder bei Nichtgebrauch automatisch ausblenden (View Mode • Dock Unpinned). Über das Window-Menü übernehmen Sie Ihre Anpassungen als Standard. Restore Default Layout kehrt zu den zuletzt gespeicherten Standardeinstellungen zurück.

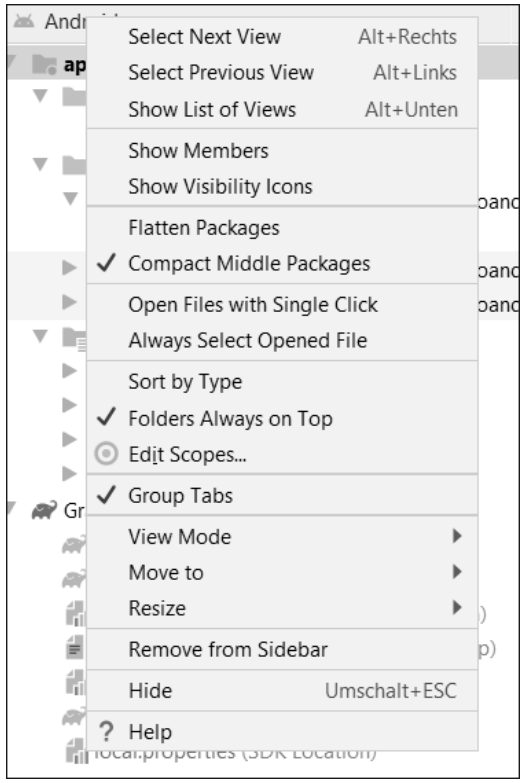


Abbildung 2.4 Kontextmenü eines Werkzeugfensters

Eine Statuszeile am unteren Rand des IDE-Hauptfensters zeigt situationsabhängige Informationen an, beispielsweise die aktuelle Cursorposition oder den Fortschritt eines Build-Vorgangs. Ganz links befindet sich ein Symbol, mit dem Sie drei *Werkzeugfensterbereiche* ein- und ausblenden können. Klicken Sie es mehrere Male an, und achten Sie darauf, wie sich das Android-Studio-Fenster verändert. Lassen Sie sich dabei nicht irritieren, denn sobald Sie mit der Maus über das Symbol fahren, erscheint ein Pop-up-Menü mit allen verfügbaren Werkzeugfenstern. Das ist praktisch, wenn die Werkzeugfensterbereiche nicht sichtbar sind. Innerhalb eines Bereichs können Sie die Reihenfolge der Fenster übrigens per Drag & Drop nach Belieben ändern. Auch das Verschieben in einen anderen Werkzeugfensterbereich ist möglich.

Der Dialog Settings enthält zahlreiche Optionen, um das Aussehen und Verhalten der IDE Ihren Vorstellungen anzupassen. Sie erreichen ihn über File • Settings. Unter macOS finden Sie den Menüpunkt unter Android Studio. Öffnen Sie den Knoten Appearance & Behavior, und klicken Sie dann auf Appearance. Wie in Abbildung 2.5 zu sehen, können Sie beispielsweise ein Theme einstellen. Darcula färbt Android Studio – das Wortspiel lässt es bereits vermuten – dunkel ein. Falls Sie möchten, können Sie die Standardschriften gegen von Ihnen gewählte Fonts austauschen. Setzen

Sie hierzu ein Häkchen vor USE CUSTOM FONT, und wählen Sie in der Klappliste daneben die gewünschte Schrift und Größe aus.

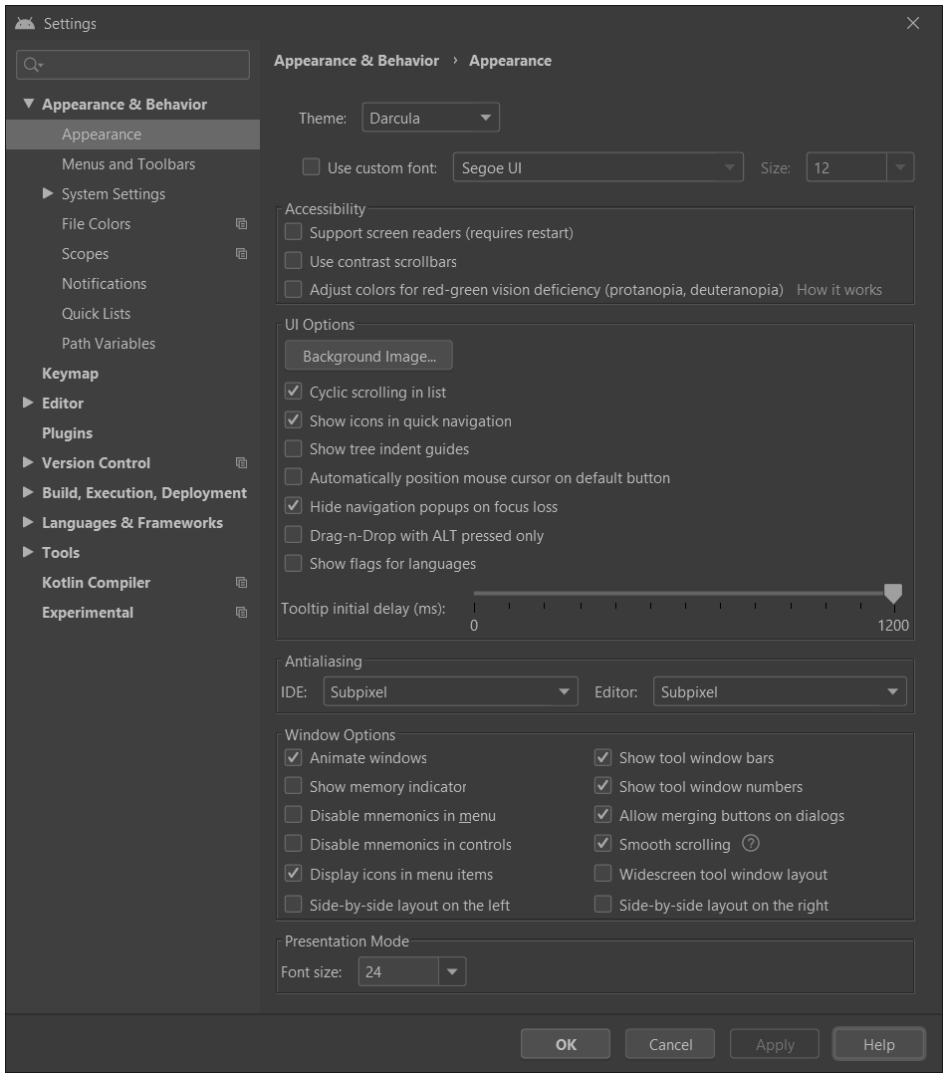


Abbildung 2.5 Der Dialog »Settings«

Klicken Sie im Abschnitt APPEARANCE & BEHAVIOR bitte auf SYSTEM SETTINGS. Unter STARTUP/SHUTDOWN können Sie einstellen, ob beim Start das zuletzt bearbeitete Projekt automatisch geöffnet werden soll. Ist das Häkchen bei REOPEN LAST PROJECT ON STARTUP nicht gesetzt, erscheint der Willkommensbildschirm. Er enthält eine Liste der kürzlich verwendeten Projekte. Das ist praktisch, wenn Sie mit mehreren Projekten im Wechsel arbeiten. Klicken Sie das gewünschte Projekt einfach im Willkommensbildschirm an.

Tipp

Sie können im Willkommensbildschirm angezeigte Projekte zu Gruppen zusammenfassen. Klicken Sie hierzu im linken Bereich des Fensters mit der rechten Maustaste auf eine freie Stelle, und wählen Sie dann NEW PROJECT GROUP. Danach klicken Sie das Projekt, das Sie einer Gruppe hinzufügen möchten, ebenfalls mit der rechten Maustaste an, und wählen dann die gewünschte Gruppe aus dem Untermenü MOVE TO GROUP aus.

CONFIRM APPLICATION EXIT legt fest, ob eine Rückfrage erscheint, wenn Sie Android Studio durch Anklicken des Fensterschließsymbols oder über die Menüleiste verlassen. Unter PROJECT OPENING können Sie konfigurieren, ob Projekte in einem neuen Android-Studio-Hauptfenster geöffnet werden. Wenn Sie OPEN PROJECT IN THE SAME WINDOW auswählen, schließt die IDE das aktuelle Projekt und öffnet danach das neue. CONFIRM WINDOW TO OPEN PROJECT IN lässt Ihnen in einem entsprechenden Dialog die Wahl. Mit DEFAULT DIRECTORY können Sie das Verzeichnis festlegen, das beim Anlegen von neuen Projekten und dem DATEI ÖFFNEN-Dialog angezeigt wird.

Da Android Studio kontinuierlich weiterentwickelt und von Fehlern befreit wird, empfehle ich Ihnen die gelegentliche Suche nach Aktualisierungen. Sie können dies mit HELP • CHECK FOR UPDATE jederzeit selbst auslösen. Es ist allerdings bequemer, dies der IDE zu überlassen. Öffnen Sie in den Settings den Knoten APPEARANCE & BEHAVIOR • SYSTEM SETTINGS, und klicken Sie dann auf UPDATES. Sofern dies nicht bereits der Fall ist, aktivieren Sie die Option AUTOMATICALLY CHECK UPDATES FOR. In der Klappliste rechts daneben sollten Sie STABLE CHANNEL auswählen. Kanäle legen fest, welche Aktualisierungen eingespielt werden. Der *Stable Channel* enthält nur ausreichend erprobte Änderungen. Die anderen Kanäle liefern Updates schneller aus, allerdings sind diese oftmals noch fehlerbehaftet oder experimentell.

Damit möchte ich unseren kleinen Rundgang durch Android Studio beenden. Im folgenden Abschnitt stelle ich Ihnen die Struktur von Android-Projekten vor.

2.1.2 Projektstruktur

Android-Apps bestehen aus einer ganzen Reihe von Artefakten, die als baumartige Struktur dargestellt werden können. Das Android-Studio-Werkzeugfenster PROJECT bietet hierfür mehrere Sichten an, unter anderem PROJECT, PACKAGES und ANDROID. Sichten wirken als Filter, d. h., nicht jedes Artefakt (eine Datei oder ein Verzeichnis) ist unbedingt in allen Sichten zu sehen.

Die Sicht PROJECT entspricht weitestgehend der Repräsentation auf Ebene des Dateisystems. Sie visualisiert die hierarchische Struktur eines Projekts. PACKAGES grup-

piert Dateien analog zu Java-Paketen, soweit dies sinnvoll ist. Diese Sicht werden Sie möglicherweise eher selten verwenden. Am praktischsten für die Entwicklung ist wahrscheinlich die Sicht ANDROID, die in Abbildung 2.6 zu sehen ist. Sie zeigt eine vereinfachte, in Teilen flach geklopfte Projekt-Struktur und gestattet dadurch den schnellen Zugriff auf wichtige Dateien und Verzeichnisse. Thematisch zusammengehörende Artefakte werden nämlich auch dann gemeinsam dargestellt, wenn sie physikalisch in unterschiedlichen Verzeichnissen liegen.

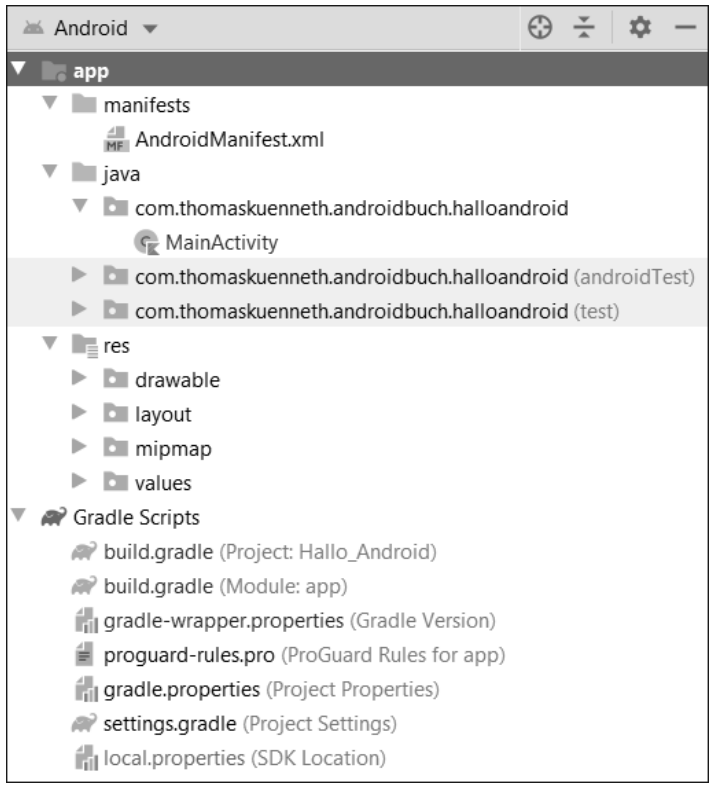


Abbildung 2.6 Die Struktur einer Android-App

Das Werkzeugfenster PROJECT stellt Sichten entweder als Registerkarten oder als Klappliste dar, was Sie mit dem Kommando GROUP TABS im Kontextmenü des Fensters einstellen können. Um es zu öffnen, klicken Sie den Fenstertitel mit der rechten Maustaste an.

Lassen Sie uns nun einen ersten Blick auf wichtige Dateien und Verzeichnisse werfen. Aktivieren Sie hierzu die Sicht ANDROID. Sie sehen zwei Knoten, APP und GRADLE SCRIPTS, von denen Sie bitte den letzteren aufklappen. Die Datei *build.gradle* kommt zweimal vor, die Dateien *gradle.properties*, *settings.gradle* und *local.properties* jeweils einmal. Unter Umständen sehen Sie noch weitere Dateien, zum Beispiel *proguard-*

rules.pro und *gradle-wrapper.properties*. Diese Dateien berühren fortgeschrittene Themen und können fürs Erste außen vor bleiben.

local.properties wird automatisch von Android Studio generiert und sollte nicht von Hand bearbeitet werden. Sie enthält einen Eintrag, der auf das für das Projekt verwendete *Android SDK* verweist. *settings.gradle* listet alle *Module* eines Projekts sowie den Projektnamen auf. Unser Hallo-Android-Projekt besteht aus einem Modul: *app*. Die Datei *settings.gradle* wird aktualisiert, sobald ein Modul hinzugefügt oder gelöscht wird. Viele Apps benötigen nur ein Modul. Mit *gradle.properties* können Sie Einfluss auf den Build-Vorgang nehmen, zum Beispiel indem Sie Variablen setzen.

Wieso ist die Datei *build.gradle* eigentlich mehrfach vorhanden? Eine Version bezieht sich auf das Projekt als Ganzes, und zu jedem Modul gehört eine weitere Ausprägung. Da *Hallo Android* aus einem Modul (*app*) besteht, gibt es *build.gradle* also zweimal. Lassen Sie uns einen Blick auf die Version für das Modul *app* werfen: Ein Doppelklick auf BUILD.GRADLE (MODULE: APP) öffnet die Datei in einem Texteditor. Wie das aussehen kann, sehen Sie in Abbildung 2.7. Bitte nehmen Sie zunächst keine Änderungen vor. Sie können das Editorfenster jederzeit durch Anklicken des Kreuzes auf der Registerkarte oder durch Drücken der Tastenkombination Strg+F4 schließen. Auf dem Mac ist es cmd+W.

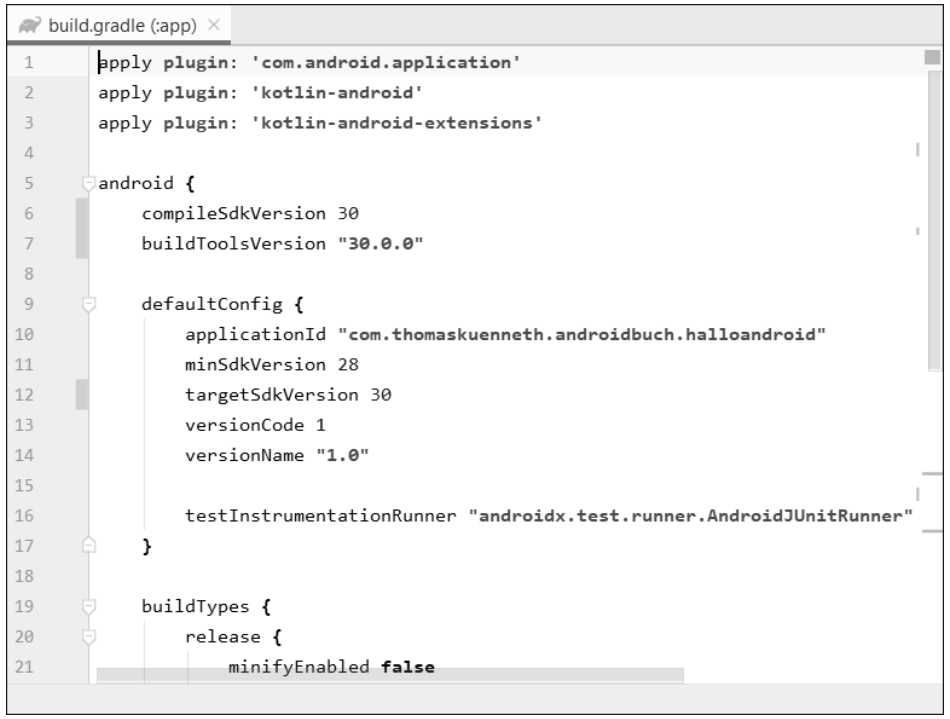


Abbildung 2.7 Die Datei »build.gradle« im Editor von Android Studio

Der Block `android { … }` enthält Informationen, die Sie beim Anlegen des Projekts eingegeben haben. Beispielsweise entspricht `applicationId` dem PACKAGE NAME. `minSdkVersion` gibt an, welche Android-Version auf einem Gerät mindestens vorhanden sein muss, damit man die App nutzen kann. Ist diese Voraussetzung nicht erfüllt, wird die Installation abgebrochen, und *Google Play* zeigt das Programm in so einem Fall gar nicht erst an. Beispielsweise ist erst ab Android 4.x ein Zugriff auf Kalenderdaten über offizielle Schnittstellen möglich. Eine App, die diese nutzt, ist auf sehr alten Geräten mit *Gingerbread* oder gar *Cupcake* nicht lauffähig. Falls Sie keinen Wert setzen, geht Android davon aus, dass die App ab der ersten Android-Version lauffähig ist. Um potenzielle Probleme zu vermeiden, sollten Sie `minSdkVersion` deshalb auf jeden Fall angeben.

Die `targetSdkVersion` legt fest, gegen welche Android-Version eine App entwickelt, optimiert und getestet wurde. Man könnte auch sagen, unter der sich die App am wohlsten fühlt. Im Laufe der Zeit hat Google immer wieder das Aussehen oder Verhalten von Systembausteinen in einer Weise geändert, die Auswirkungen auf Apps hat. Um Inkompatibilitäten vorzubeugen, werden ältere Apps in einem Kompatibilitätsmodus gefahren. Das Attribut `targetSdkVersion` gibt also an, bis zu welcher Plattformversion dies aus Sicht der App nicht nötig ist.

`minSdkVersion` und `targetSdkVersion` erwarten den sogenannten *API-Level*. Für Android 1.5 (*Cupcake*) war dieser beispielsweise 3, Android 2.x (*Froyo*) hatte API-Level 8, *Lollipop* und *Nougat* entsprechen den API-Levels 21 respektive 24. Android 10 hat den API-Level 29.

[+]

Tipp

Unter *https://developer.android.com/guide/topics/manifest/uses-sdk-element.html* finden Sie eine vollständige Aufstellung aller API-Levels. In der Klasse `android.os.Build.VERSION_CODES` sind entsprechende Konstanten definiert.

Plattformen können mit dem *SDK Manager* installiert und gelöscht werden. Dieses Buch beschreibt die Anwendungsentwicklung mit Android 11. Aus diesem Grund basieren die meisten Beispiele auf *API-Level* 30.

`versionCode` und `versionName` repräsentieren die Versionsnummer Ihrer App. Während `versionCode` eine Zahl ist, die zur Auswertung durch Programmcode dient, enthält das Attribut `versionName` die Versionsnummer in einer für den Anwender verständlichen Form, zum Beispiel 1.2 oder 1.2.3. Google schlägt vor, für die erste veröffentlichte Version einer App `versionCode` auf 1 zu setzen und mit jedem Update beispielsweise um 1 zu erhöhen. Sie sollten stets beide Werte angeben. `compileSdkVersion` und `buildToolsVersion` geben Aufschluss darüber, welche Android-Plattform für die Entwicklung verwendet wurde und welche *Build Tools* eingesetzt wurden. Wenn nach dem Öffnen eines Projekts unerklärlich viele Fehler moniert werden, wurden

entweder die Plattform oder die *Build Tools* noch nicht in der »gewünschten« Version heruntergeladen.

Übrigens müssen Sie die Datei *build.gradle* nicht unbedingt in einem Texteditor bearbeiten, um beispielsweise `buildToolsVersion` oder `compileSdkVersion` zu ändern. Das geht viel bequemer über den Dialog PROJECT STRUCTURE. Sie öffnen ihn mit FILE • PROJECT STRUCTURE.

Lassen Sie uns nun einen Blick auf das Modul APP werfen; es enthält die Zweige MANIFESTS, JAVA und RES. Quelltexte werden unter JAVA abgelegt. Das gilt auch dann, wenn Sie Ihre Apps in Kotlin schreiben. `com.thomaskuenneth.androidbuch.halloandroid` erscheint dreimal. Das mag irritieren, vor allem, wenn Sie schon mit anderen Entwicklungsumgebungen gearbeitet haben. Bitte denken Sie daran, dass die Sicht ANDROID eine optimierte und, wenn Sie so möchten, künstliche Sicht auf ein Projekt darstellt. Ein Paket enthält die Klasse `ExampleInstrumentedTest`, ein zweites `ExampleUnitTest` und das dritte schließlich `MainActivity`. Um die Testklassen müssen Sie sich zunächst nicht kümmern. Übrigens können Sie bequem neuen Quelltext hinzufügen, indem Sie ein Paket mit der rechten Maustaste anklicken und NEW • KOTLIN FILE/CLASS wählen.

Der Zweig RES besteht aus mehreren Unterknoten. Beispielsweise enthält VALUES die Datei *strings.xml*. Sie nimmt Texte auf, die später im Quelltext oder in Beschreibungsdateien für die Benutzeroberfläche referenziert werden. Hierzu wird von den Werkzeugen des Android SDK eine Klasse mit Namen `R` generiert, die Sie allerdings nicht von Hand bearbeiten dürfen. Deshalb ist sie in der Sicht ANDROID auch nicht vorhanden. Im Unterknoten LAYOUT wird die Benutzeroberfläche einer App definiert. Haben Sie noch ein klein wenig Geduld, wir kommen in diesem Kapitel noch dazu. DRAWABLE und MIPMAP enthalten die Grafiken einer App. Das Programm-Icon liegt in MIPMAP, alle anderen in DRAWABLE. Bitmaps können in unterschiedlichen Auflösungen (Pixeldichten) abgelegt werden. Sie landen in Unterverzeichnissen, die einem bestimmten Namensmuster folgen. Mehr dazu etwas später. Für Vektorgrafiken ist das Bereitstellen in unterschiedlichen Größen natürlich nicht nötig. Sie liegen in DRAWABLE.

Der Unterknoten MANIFESTS enthält die Datei *AndroidManifest.xml*. Sie ist die zentrale Beschreibungsdatei einer Anwendung. In ihr werden unter anderem die Bestandteile des Programms aufgeführt. Wie Sie später noch sehen werden, sind dies sogenannte *Activities*, *Services*, *Broadcast Receiver* und *Content Provider*. Die Datei enthält aber auch Informationen darüber, welche Rechte eine App benötigt und welche Hardware sie erwartet.

Bitte öffnen Sie mit einem Doppelklick die Datei *AndroidManifest.xml*, um sich einen ersten Eindruck von ihrer Struktur zu verschaffen. Es gibt ein Wurzelelement `<manifest>` mit einem Kind `<application>`. Android-Apps bestehen neben den weiter oben

bereits genannten anderen Bausteinen aus mindestens einer *Activity*. Hierbei handelt es sich stark vereinfacht ausgedrückt um eine Bildschirmseite. Verschiedene Aspekte einer Anwendung, wie Listen, Übersichten, Such- und Eingabemasken, werden als eigene Activities realisiert und als Unterelemente von `<application>` in *AndroidManifest.xml* eingetragen.

Hinweis

Wenn Sie einen Blick auf Googles Entwicklerdokumentation zur Manifestdatei werfen, stellen Sie fest, dass es neben `<application>` eine ganze Reihe Kinder von `<manifest>` gibt. Das Tag `<uses-sdk>` gibt beispielsweise die Zielplattform an. Schon seit dem Wechsel von Eclipse auf Android Studio werden diese Angaben aber nicht mehr direkt in das Manifest eingetragen, sondern in *build.gradle* gepflegt. Beim Bauen der Anwendung werden sie dann automatisch in das Manifest übernommen.

2.1.3 Bibliotheken

Bevor Sie im nächsten Abschnitt erste Erweiterungen an *Hallo Android* vornehmen, möchte ich Ihr Augenmerk noch einmal auf die Datei *build.gradle* für das Modul *app* richten. Sie enthält den Abschnitt `dependencies { … }` mit Verweisen auf externen Code. Wie das aussehen kann, ist beispielhaft in Listing 2.1 dargestellt.

```
dependencies {
  implementation fileTree(dir: "libs", include: ["*.jar"])
  implementation "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
  …
  implementation 'androidx.appcompat:appcompat:1.1.0'
  implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
  testImplementation 'junit:junit:4.12'
  androidTestImplementation 'androidx.test.ext:junit:1.1.1'
  …
}
```

Listing 2.1 Projekt-Abhängigkeiten

`fileTree()` bindet *.jar*-Dateien im Unterverzeichnis *libs* ein und macht deren Inhalt im Projekt verwendbar. Wenn Sie also eine vorhandene und kompatible Java-Bibliothek in Ihrer App nutzen möchten, müssen Sie diese nur in den Ordner kopieren.

Die Schlüsselwörter `implementation`, `testImplementation` und `androidTestImplementation` legen fest, wie bzw. wann die referenzierten Bibliotheken verwendet werden können. Beispielsweise würden die Bestandteile des Testframeworks JUnit in der fertigen App nur unnötig Platz verbrauchen. Deshalb wird mit `testImplementation` festgelegt, dass sie nur beim Ausführen von Tests verfügbar sind. Gleiches gilt für sogenannte Instrumentation Tests. Solche Frameworks binden Sie mit `androidTestImplementation` ein.

Bibliotheken, die immer zur Verfügung stehen sollen, referenzieren Sie mit `implementation`. Der Projektassistent hat unter anderem `org.jetbrains.kotlin:kotlin-stdlib`, `androidx.appcompat:appcompat` und `androidx.constraintlayout:constraintlayout` eingebunden. `kotlin-stdlib` ist die Standardklassenbibliothek von Kotlin. Sie ist nötig, um Ihre App in Kotlin schreiben zu können. Die anderen beiden, `constraintlayout` und `appcompat`, gehören zu Googles *Jetpack*.

Hinter dem Bibliotheksnamen wird die zu verwendende Versionsnummer angegeben. Dabei sind auch Bereiche möglich. Man könnte beispielsweise durch Hinzufügen des Plus-Zeichens konfigurieren, Version 1.2.3 oder neuer zu verwenden. Das gilt allgemein aber als schlechter Stil, weil sich das Verhalten der App beim Bauen und zur Laufzeit nicht mehr ohne Weiteres reproduzieren lässt. Beides kann ja direkt oder indirekt durch die Version der Bibliothek beeinflusst werden. Besser ist deshalb, stets vollständige Versionsnummern zu verwenden.

Hinweis

Wo Android Studio bzw. das Build System Gradle nach Bibliotheken sucht, ist in der zweiten, projektweit gültigen Datei *build.gradle* hinterlegt. In den Blöcken `buildscript { … }` und `allprojects { … }` befindet sich jeweils ein Unterelement `repositories { … }`, das üblicherweise die Einträge `jcenter()` und `google()` enthält.

In den folgenden Kapiteln werden Sie viele Aspekte von Jetpack und AndroidX kennenlernen. Zunächst kümmern wir uns aber um eine Reihe von Grundlagen. Als Erstes werde ich Ihnen zeigen, wie in Android Grafiken und Texte gespeichert werden und wie man in einer App auf diese zugreift.

2.2 Benutzeroberfläche

Die Benutzeroberfläche ist das Aushängeschild einer Anwendung. Gerade auf mobilen Geräten sollte jede Funktion leicht zugänglich und intuitiv erfassbar sein. Android unterstützt Sie bei der Gestaltung durch eine große Auswahl an Bedienelementen.

2.2.1 Grafiken

Neben Smartphones, deren Bildschirmdiagonalen üblicherweise zwischen 5 und 7 Zoll groß sind, gibt es Tablets, deren Displays bis zu 13 Zoll betragen. Auch die Zahl der horizontal und vertikal darstellbaren Pixel variiert drastisch. Um den durch diese

Vielfalt entstehenden Aufwand für Entwickler in Grenzen zu halten, definiert Android einige Klassen für Bildschirmgrößen und Pixeldichten. Fordert eine App zur Laufzeit eine Grafik an, sucht das System die am besten zur Hardware passende aus. Wie das funktioniert, erkläre ich Ihnen in diesem Abschnitt. Zuvor möchte ich aber noch ein paar Begriffe erklären:

- Die *Bildschirmgröße* wird üblicherweise in Zoll angegeben. Sie beschreibt den Abstand von der linken unteren zur rechten oberen Ecke der Anzeige.
- Das *Seitenverhältnis* (engl. *Aspect Ratio*) entspricht dem Quotienten aus physikalischer Breite und Höhe. Android kennt in diesem Zusammenhang die beiden Resource-Bezeichner `long` und `notLong`. Ob ein Bildschirm lang oder nicht lang ist, hat übrigens nichts mit dessen Ausrichtung zu tun. WVGA (800 × 480 Pixel) und FWVGA (854 × 480 Pixel) sind lang, VGA (640 × 480 Pixel) hingegen nicht. Deshalb bleibt das Seitenverhältnis zur Laufzeit auch stets gleich.
- Die *Auflösung* gibt die Zahl der horizontal und vertikal ansprechbaren physikalischen Pixel an.
- Die *Pixeldichte* schließlich wird in Punkten pro Zoll angegeben und ist letztlich ein Maß für die Größe eines Pixels. Sie errechnet sich aus der Bildschirmgröße und der physikalischen Auflösung.

Wie Sie bereits wissen, legt Android Studio beim Erzeugen eines Projekts das App-Icon in Verzeichnissen ab, die alle mit *mipmap* beginnen. Konkret heißen sie *mipmap-mdpi*, *mipmap-hdpi*, *mipmap-xhdpi*, *mipmap-xxhdpi* und *mipmap-xxxhdpi* und enthalten das Programm-Symbol als Rastergrafik in unterschiedlichen Pixeldichten. Zur Laufzeit der App lädt die Plattform je nach Bildschirmkonfiguration die am besten geeignete Datei aus dem Verzeichnis für *mittlere* (*-mdpi*), *hohe* (*-hdpi*), *sehr hohe* (*-xhdpi*), *sehr, sehr hohe* (*-xxhdpi*) oder *sehr, sehr, sehr hohe* (*-xxxhdpi*) Dichte. Das x steht übrigens für das englische »extra«. Hat die Plattform einen API-Level von 26 oder höher, wird (sofern vorhanden) stattdessen die Vektorgrafik im Verzeichnis *mipmap-anydpi-v26* verwendet. Die Dateinamen der Grafiken ohne Erweiterung sind stets gleich, zum Beispiel *ic_launcher.png* und *ic_launcher_round.png* für die Bitmap-App-Icons, die der Projektassistent erstellt hat.

Wenn Sie an anderer Stelle Rastergrafiken anzeigen möchten, sollten auch diese in unterschiedlichen Pixeldichten vorliegen. Damit Android sie findet, müssen die Dateien in Verzeichnissen liegen, die mit *drawable* beginnen und mit einem der genannten Suffixe (*-mdpi*, *-xxhdpi* ...) enden. Bitte achten Sie darauf, dass der Dateiname stets gleich ist.

Bildschirme mit niedriger Dichte stellen etwa 120 Punkte pro Zoll (engl. *dots per inch* – dpi) dar. Bei mittlerer Dichte sind dies ungefähr 160 dpi, was übrigens den beiden ersten Android-Geräten G1 und Magic entspricht (und deshalb als Basislinie für die Umrechnung gilt). Smartphones oder Tablets mit hoher Pixeldichte lösen ca. 240 dpi

auf, bei sehr hoher Dichte sind es 320 dpi. *-xxhdpi* und *-xxxhdpi* entsprechen etwa 480 bzw. 640 dpi.

Vielleicht fragen Sie sich, warum Android diesen Aufwand treibt. Aus Sicht des Anwenders soll die Pixeldichte keine Auswirkung auf die Größe der Benutzeroberfläche haben. Genau das ist bei einer Bitmap aber der Fall. Mit zunehmender Pixeldichte wirkt sie immer kleiner. Um das zu kompensieren, muss sie entweder zur Laufzeit skaliert werden (was zu Qualitätseinbußen führen kann) oder schon in der richtigen Größe vorliegen. Die folgende Tabelle zeigt, welchen Einfluss die Pixeldichte auf Breite und Höhe einer Bitmap hat.

Größe in Pixel	Umrechnungsfaktor	Pixeldichte
36 × 36	0,75	ldpi
48 × 48	**1,0**	**mdpi**
72 × 72	1,5	hdpi
96 × 96	2,0	xhdpi
144 × 144	3,0	xxhdpi
192 × 192	4,0	xxxhdpi

Tabelle 2.1 Umrechnungstabelle für Pixeldichten

Bitmaps, die im Verzeichnis *drawable* (also ohne Postfix) abgelegt werden, skaliert das System zur Laufzeit. Android geht in diesem Fall davon aus, dass solche Grafiken für eine mittlere Dichte vorgesehen sind. Eine solche Konvertierung unterbleibt für Dateien im Verzeichnis *drawable-nodpi*.

Um das Erstellen von Layouts zu vereinfachen, kennt Android sogenannte *density-independent pixels*. Diese abstrakte Einheit basiert auf der Pixeldichte des Bildschirms in Relation zu 160 dpi. `160dp` entsprechen also immer einem Zoll. Die Formel zur Umrechnung ist sehr einfach:

pixels = dps × (density ÷ 160)

Normalerweise müssen Sie solche Berechnungen in Ihrer App aber gar nicht durchführen. Wichtig ist eigentlich nur, in allen Layouts diese Einheit zu verwenden.

Adaptive Icons

Bevor ich Ihnen im folgenden Abschnitt den Umgang mit Text erkläre, möchte ich noch auf eine Spezialität von Android eingehen, die sogenannten adaptiven Symbole. Ab Android 7.1 konnten Apps kreisrunde Icon-Ressourcen für ihre Programmstarter-Symbole bereitstellen. Auf welchen Geräten diese dann angezeigt wurden, hing von

der sogenannten *Device Build Configuration* ab. Tatsächlich waren nur Smartphones der Pixel-Baureihe entsprechend konfiguriert. Wenn ein Programmstarter Icons beim System erfragte, lieferte das Android-Framework entweder ein Icon aus dem Manifestattribut (mehr dazu etwas später) `android:icon` oder `android:roundIcon`.

Mit Oreo hat Google diese Idee zu adaptiven Icons weiterentwickelt. Gerätehersteller können eine Maske definieren, die mit dem eigentlichen Programmsymbol verknüpft wird. Stellen Sie sich das am besten wie eine Form zum Ausstechen von Plätzchen vor – alles außerhalb der Maske fällt weg. Der Vorteil für den Entwickler: Das eigene Icon erscheint stets in der richtigen Form. Damit adaptive Icons funktionieren, müssen Sie Ihr Symbol in zwei Ebenen aufteilen, jeweils eine für Vorder- und Hintergrund. Effekte wie Schlagschatten oder Maskierungen sind leider tabu. Denn diese können unter Umständen durch das System hinzugefügt werden. Dazu gleich mehr.

Traditionell waren Programmstarter-Icons 48 × 48 dp groß. Die beiden Ebenen adaptiver Icons hingegen messen 108 × 108 geräteunabhängige Pixel. Die inneren 72 dp erscheinen innerhalb der vom Gerätehersteller gelieferten Maske. Der sichtbare Bereich kann an bestimmten Punkten auf einen Radius von 33 dp begrenzt werden. Wie viel vom eigentlichen App-Icon zu sehen ist, lässt sich deshalb nicht so ohne Weiteres vorhersagen. Darüber hinaus können das System bzw. der Programmstarter für visuelle Effekte oder Animationen bis zu 18 geräteunabhängige Pixel an allen vier äußeren Rändern verwenden.

```
<?xml version="1.0" encoding="utf-8"?>
<adaptive-icon xmlns:android="http://schemas.android.com/apk/res/android">
  <background android:drawable="@drawable/ic_launcher_background" />
  <foreground android:drawable="@drawable/ic_launcher_foreground" />
</adaptive-icon>
```

Listing 2.2 Beschreibungsdatei eines adaptiven Icons

Android Studio unterstützt Sie beim Erstellen von adaptiven Icons mit dem sogenannten Asset Studio. Es generiert eine XML-Datei (siehe Listing 2.2) mit den zwei Tags `<foreground />` und `<background />`. Diese verweisen auf Drawables, die mit dem Attribut `android:drawable` referenziert werden. Wie Sie mit dem Asset Studio ein App-Icon erstellen, zeige ich Ihnen in Abschnitt 3.3.1, »Die App vorbereiten«.

2.2.2 Texte

Bilder und Symbole sind ein wichtiges Gestaltungsmittel. Sinnvoll eingesetzt, helfen sie dem Anwender nicht nur beim Bedienen des Programms, sondern sorgen zudem für ein angenehmes und schönes Äußeres. Dennoch spielen auch Texte eine sehr wichtige Rolle. Sie werden in den unterschiedlichsten Bereichen einer Anwendung eingesetzt:

- als Beschriftungen von Bedienelementen
- für erläuternde Texte, die durch einen Screenreader vorgelesen werden
- für Hinweis- und Statusmeldungen

Die fertige Version von *Hallo Android* soll den Benutzer zunächst begrüßen und ihn nach seinem Namen fragen. Im Anschluss wird ein persönlicher Gruß angezeigt. Nach dem Anklicken einer Schaltfläche beendet sich die App.

Aus dieser Beschreibung ergeben sich die folgenden Texte. Die Bezeichner vor dem jeweiligen Text werden Sie später im Programm wiederfinden:

- `welcome` – *Guten Tag. Schön, dass Sie mich gestartet haben. Bitte verraten Sie mir Ihren Namen.*
- `next` – *Weiter*
- `hello` – *Hallo <Platzhalter>. Ich freue mich, Sie kennenzulernen*.
- `finish` – *Fertig*

Ein Großteil der Texte wird zur Laufzeit so ausgegeben, wie sie schon während der Programmierung erfasst wurden. Eine kleine Ausnahme bildet die Grußformel, denn sie besteht aus einem konstanten und einem variablen Teil. Letzterer ergibt sich erst, nachdem der Anwender seinen Namen eingetippt hat. Wie Sie gleich sehen werden, ist es in Android sehr einfach, dies zu realisieren. Da Sie Apps in Kotlin schreiben, könnten Sie die auszugebenden Meldungen einfach als Raw String im Quelltext ablegen. Das ist praktisch, weil Sie auf diese Weise die Zeilenumbrüche an der gewünschten Stelle setzen können. Das sähe folgendermaßen aus:

```
message.text = """
    Guten Tag. Schön, dass Sie mich gestartet haben.
    Bitte verraten Sie mir Ihren Namen.
""".trimIndent()
```

Das hat allerdings mehrere Nachteile: Zum einen müssen Sie die aus Gründen der Lesbarkeit eingefügten Leerzeichen am Zeilenanfang mit `trimIndent()` wieder löschen. Sonst werden sie ebenfalls ausgegeben, was nicht gewünscht ist. Wenn man seinen App-Code auf mehrere Quelltextdateien aufteilt (und das ist bei umfangreicheren Programmen auf jeden Fall eine gute Idee), merkt man oft nicht, wenn man gleiche Texte mehrfach definiert. Das kann die Installationsdatei der App vergrößern und unnötig Speicher kosten. Außerdem wird es auf diese Weise sehr schwer, mehrsprachige Anwendungen zu bauen. Wenn Sie aber eine App über Google Play vertreiben möchten, sollten Sie neben den deutschsprachigen Texten mindestens eine englische Lokalisierung ausliefern. Unter Android werden Texte daher zentral in der Datei *strings.xml* abgelegt. Sie befindet sich im Verzeichnis *values*. Ändern Sie die durch den Projektassistenten angelegte Fassung folgendermaßen ab:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- Name der App -->
    <string name="app_name">Hallo Android!</string>
    <!-- Willkommensmeldung -->
    <string name="welcome">
Guten Tag. Schön, dass Sie mich gestartet haben.
Bitte verraten Sie mir Ihren Namen.
    </string>
    <!-- Persönlicher Gruß -->
    <string name="hello">
Hallo %1$s. Ich freue mich, Sie kennenzulernen.
    </string>
    <!-- Beschriftungen für Schaltflächen -->
    <string name="next">Weiter</string>
    <string name="finish">Fertig</string>
</resources>
```

Listing 2.3 »strings.xml«

Das Attribut `name` des Elements `<string>` wird später im Quelltext als Bezeichner verwendet. Der Name muss deshalb innerhalb des Projekts eindeutig sein. Ich betone das, weil das Ablegen von Zeichenketten in *strings.xml nur* eine Konvention ist (der Sie unbedingt folgen sollten). Zeichenketten können auch in anders genannten XML-Dateien definiert werden, die unter *res/values* abgelegt wird. Ist Ihnen im Listing die fett gesetzte Zeichenfolge `%1$s` aufgefallen? Android wird an dieser Stelle den vom Benutzer eingegebenen Namen einfügen. Wie dies funktioniert, zeige ich Ihnen später.

Hinweis

Die Zeilen `Hallo %1$s…` und `Guten Tag.` sind nicht eingerückt, weil die führenden Leerzeichen sonst in die App übernommen werden, was in der Regel nicht gewünscht ist.

Vielleicht fragen Sie sich, wie Sie Ihr Programm mehrsprachig ausliefern können, wenn es genau eine zentrale Datei *strings.xml* gibt. Neben dem Verzeichnis *values* kann es lokalisierte Ausprägungen geben, die auf das Minuszeichen und auf ein Sprachkürzel aus zwei Buchstaben enden, zum Beispiel *values-en* oder *values-fr*. Die Datei *strings.xml* in diesen Ordnern enthält Texte in den entsprechenden Sprachen, also auf Englisch oder Französisch. Muss Android auf eine Zeichenkette zugreifen, geht das System vom Speziellen zum Allgemeinen. Ist die Standardsprache also beispielsweise Englisch, wird zuerst versucht, den Text in *values-en/strings.xml* zu finden. Gelingt dies nicht, wird *values/strings.xml* verwendet. In dieser Datei müssen

also alle Strings definiert werden, Lokalisierungen hingegen können unvollständig sein. Bei der Erstellung mehrsprachiger Texte unterstützt Sie Android Studio mit dem Translations Editor. Ich werde in einem späteren Kapitel darauf zurückkommen.

Im folgenden Abschnitt stelle ich Ihnen sogenannte *Views* vor. Bei ihnen handelt es sich um die Grundbausteine, aus denen die Benutzeroberfläche einer App zusammengesetzt wird.

2.2.3 Views

Hallo Android besteht auch nach vollständiger Realisierung aus sehr wenigen Bedienelementen, und zwar aus

- einem nicht editierbaren Textfeld, das den Gruß unmittelbar nach dem Programmstart sowie nach Eingabe des Namens darstellt,
- einer Schaltfläche, die je nach Situation mit WEITER oder FERTIG beschriftet ist, und aus
- einem Eingabefeld, das nach dem Anklicken der Schaltfläche WEITER ausgeblendet wird.

Wie die Komponenten auf dem Bildschirm platziert werden sollen, zeigt ein sogenannter *Wireframe*, den Sie in Abbildung 2.8 sehen. Man verwendet solche abstrakten Darstellungen gern, um die logische Struktur einer Bedienoberfläche in das Zentrum des Interesses zu rücken.

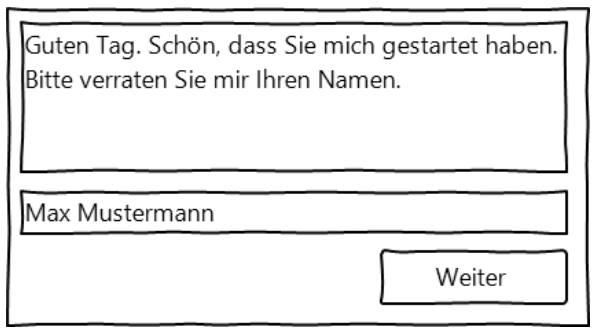


Abbildung 2.8 Prototyp der Benutzeroberfläche von »Hallo Android«

Unter Android sind alle Bedienelemente direkte oder indirekte Unterklassen der Klasse `android.view.View`. Jede View belegt einen rechteckigen Bereich des Bildschirms. Seine Position und Größe wird durch Layouts bestimmt, die wiederum von `android.view.ViewGroup` erben, die ebenfalls ein Kind von `View` ist. Sie haben üblicherweise keine eigene grafische Repräsentation, sondern sind Container für weitere Views und ViewGroups.

Die Text- und Eingabefelder sowie die Schaltflächen, die in *Hallo Android* verwendet werden, sind also Views. Konkret verwenden wir die Klassen `Button`, `TextView` und `EditText`. Wo sie auf dem Bildschirm positioniert werden und wie groß sie sind, wird hingegen durch die ViewGroup `LinearLayout` festgelegt.

Zur Laufzeit einer Anwendung manifestiert sich ihre Benutzeroberfläche demnach als Objektbaum. Aber nach welcher Regel wird er erzeugt? Wie definieren Sie als Entwickler den Zusammenhang zwischen einem Layout, einem Textfeld und einer Schaltfläche? Sie könnten die Bedienelemente im Code zusammenfügen:

```
val v = ScrollView(context)
val layout = LinearLayout(context)
layout.orientation = LinearLayout.VERTICAL
v.addView(layout)
layout.addView(getCheckBox(context, Locale.GERMANY))
layout.addView(getCheckBox(context, Locale.US))
layout.addView(getCheckBox(context, Locale.FRANCE))
```

Listing 2.4 Beispiel für den programmgesteuerten Bau einer Oberfläche

Allerdings ist dies nicht die typische Vorgehensweise. Die lernen Sie im folgenden Abschnitt kennen.

2.2.4 Oberflächenbeschreibungen

Eine Android-App beschreibt ihre Benutzeroberflächen mittels XML-basierter Layoutdateien, die zur Laufzeit zu Objektbäumen »aufgeblasen« werden. Alle Bedienelemente von *Hallo Android* werden in einen Container des Typs `LinearLayout` gepackt. Seine Kinder erscheinen entweder neben- oder untereinander auf dem Bildschirm. Wie Sie gleich sehen werden, steuert das Attribut `android:orientation` die Laufrichtung. Für die Größe der Views und der ViewGroups gibt es die beiden Attribute der `android:layout_width` und `android:layout_height`.

Oberflächenbeschreibungen werden in *layout*, einem Unterverzeichnis von *res*, gespeichert. Beim Anlegen des Projekts hat der Android-Studio-Projektassistent dort die Datei *activity_main.xml* abgelegt. Öffnen Sie diese mit Doppelklick, und ändern Sie sie entsprechend Listing 2.5 ab. Damit Sie den Quelltext eingeben können, klicken Sie in der oberen rechten Ecke des Editorfensters auf Code.

```
<?xml version="1.0" encoding="utf-8"?>

<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
```



```
    android:layout_height="match_parent"
    android:orientation="vertical">

    <TextView
        android:id="@+id/message"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

    <EditText
        android:id="@+id/input"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

    <Button
        android:id="@+id/next_finish"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="end" />

</LinearLayout>
```

Listing 2.5 »activity_main.xml«

Die XML-Datei bildet die Hierarchie der Benutzeroberfläche ab. Demzufolge ist `<LinearLayout>` das Wurzelelement. Mein Beispiel enthält die drei Kinder `<TextView>`, `<EditText>` und `<Button>`. Jedes Element hat die bereits kurz angesprochenen Attribute `android:layout_width` und `android:layout_height`. Deren Wert `match_parent` besagt, dass die Komponente die Breite oder Höhe des Elternobjekts erben soll. Der Wert `wrap_content` hingegen bedeutet, dass sich die Größe aus dem Inhalt der View ergibt, beispielsweise aus der Beschriftung einer Schaltfläche. Die Zeile `android:layout_gravity="end"` sorgt dafür, dass die Schaltfläche rechtsbündig angeordnet wird.

Tipp

Anstelle von `match_parent` finden Sie im Internet oft noch die ältere Notation `fill_parent`. Diese wurde schon in Android 2.2 (API-Level 8) von `match_parent` abgelöst. Für welche Variante Sie sich entscheiden, ist nur von Belang, wenn Sie für sehr alte Plattformversionen entwickeln. Denn abgesehen vom Namen sind beide identisch. Ich rate Ihnen trotzdem, `match_parent` zu verwenden.

Ist Ihnen aufgefallen, dass keinem Bedienelement ein Text oder eine Beschriftung zugewiesen wird? Und was bedeuten Zeilen, die mit `android:id="@+id/` beginnen? Wie Sie bereits wissen, erzeugt Android zur Laufzeit einer Anwendung aus den Oberflä-

chenbeschreibungen entsprechende Objektbäume. Zu der in der XML-Datei spezifizierten Schaltfläche gibt es also eine Instanz der Klasse `Button`. Um auf diese Instanz eine Referenz ermitteln zu können, wird ein Name definiert, beispielsweise `next_finish`. Wie auch bei *strings.xml* sorgen die Android-Entwicklungswerkzeuge dafür, dass nach Änderungen an Layoutdateien korrespondierende Einträge in der generierten Klasse `R` vorgenommen werden. Wie Sie diese nutzen, sehen Sie gleich.

Speichern Sie Ihre Eingaben, und wechseln Sie zurück zum grafischen Editor, indem Sie auf die Registerkarte DESIGN klicken. Er sollte in etwa so wie in Abbildung 2.9 aussehen. Machen Sie sich über die angezeigte Warnung keine Gedanken, wir kümmern uns etwas später darum.

Hinweis

In XML-Dateien nutzt Google gern den Underscore als verbindendes Element, zum Beispiel in `layout_width`, `layout_height` oder `match_parent`. Sie sollten zumindest in Layoutdateien diesem Stil folgen. Aus diesem Grund habe ich die ID der Schaltfläche zum Weiterklicken und Beenden der App `next_finish` genannt. In Kotlin-Quelltexten ist aber die sogenannte *CamelCase*-Schreibweise gebräuchlich, deshalb heißt die Variable der Schaltfläche `nextFinish`.

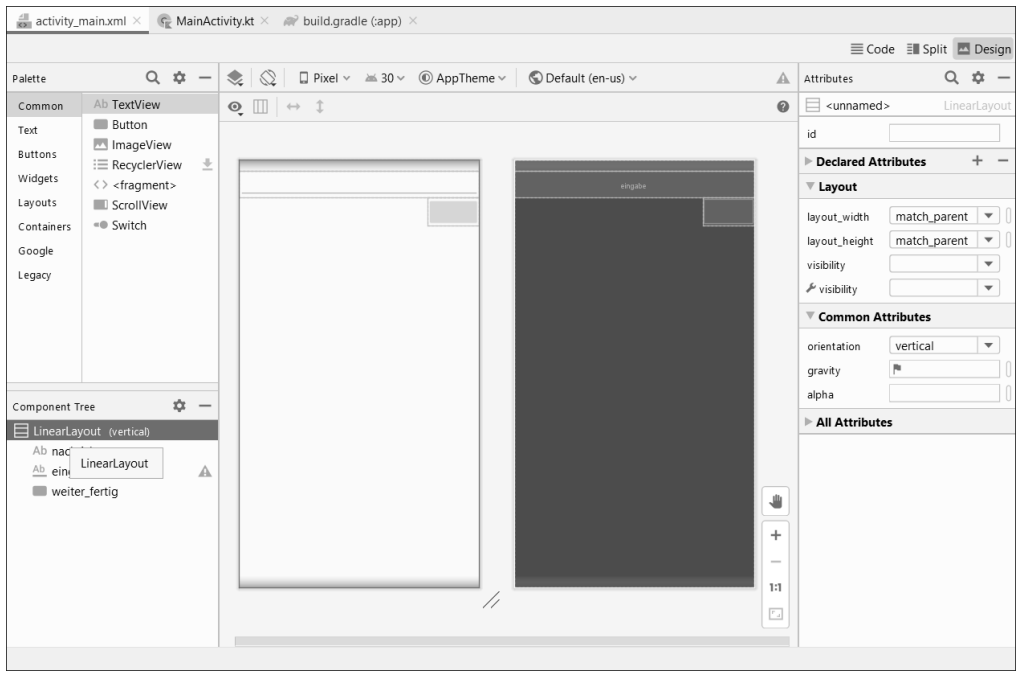


Abbildung 2.9 Erstellen der Benutzeroberfläche im Design-Modus

2.3 Programmlogik und -ablauf

Viele Desktop-Anwendungen sind datei- oder dokumentenzentriert. Egal, ob Textverarbeitung, Tabellenkalkulation oder Layoutprogramm – ihr Aufbau ist stets gleich. Den überwiegenden Teil des Bildschirms oder Fensters belegt ein Arbeitsbereich, der ein Dokument oder einen Teil davon darstellt. Um diesen Bereich herum gruppieren sich Symbolleisten und Paletten, mit deren Werkzeugen die Elemente des Dokuments bearbeitet werden. Das gleichzeitige Darstellen von Werkzeugen und Inhalt ist auf den im Vergleich zu PC- oder Laptop-Monitoren kleinen Bildschirmen mobiler Geräte nur bedingt sinnvoll. Der Benutzer würde kaum etwas erkennen. Als Entwickler sollten Sie Ihre App deshalb in Funktionsblöcke oder Bereiche unterteilen, die genau einen Aspekt Ihres Programms abbilden.

Ein anderes Beispiel: E-Mail-Clients zeigen die wichtigsten Informationen zu eingegangenen Nachrichten häufig in einer Liste an. Neben oder unter der Liste befindet sich ein Lesebereich, der das aktuell ausgewählte Element vollständig anzeigt. Auch dies lässt sich aufgrund des geringe(re)n Platzes auf Smartphones nicht sinnvoll realisieren. Stattdessen zeigen entsprechende Anwendungen dem Nutzer zunächst eine Übersicht, nämlich die Liste der eingegangenen Nachrichten, und verzweigen erst in eine Detailansicht, wenn eine Zeile der Liste angeklickt wird.

Tablet-Bildschirme bieten im Vergleich zu einem Smartphone deutlich mehr Platz für Informationen. Um Benutzeroberflächen für beide Welten entwickeln zu können, hat Google mit Android 3 sogenannte Fragmente eingeführt. Bevor ich Ihnen im nächsten Kapitel zeige, wie Sie damit Benutzeroberflächen für unterschiedliche Bildschirmgrößen anbieten, wollen wir uns den wahrscheinlich wichtigsten Anwendungsbaustein ansehen.

2.3.1 Activities

Unter Android ist das Zerlegen einer App in aufgabenorientierte Teile bzw. Funktionsblöcke ein grundlegendes Architekturmuster. Die gerade eben skizzierten Aufgaben bzw. »Aktivitäten« *E-Mail auswählen* und *E-Mail anzeigen* werden dann zu Bausteinen, die die Plattform *Activities* nennt. Eine Anwendung besteht aus mindestens einer solchen Activity, je nach Funktionsumfang können es aber auch viele mehr sein. Normalerweise ist jeder Activity eine Benutzeroberfläche, also ein Baum bestehend aus Views und ViewGroups, zugeordnet.

Activities bilden demnach die vom Anwender wahrgenommenen Bereiche einer App. Sie können sich gegenseitig aufrufen. Die Vorwärtsnavigation innerhalb einer Anwendung wird auf diese Weise realisiert. Da das System Activities auf einem Stapel ablegt, müssen Sie sich als Entwickler nicht darum kümmern, von wem Ihre Activity aufgerufen wird. Drückt der Benutzer die reale oder eine virtuelle ZURÜCK-Schaltflä-

che (oder führt die korrespondierende Wischgeste aus), wird automatisch die zuvor angezeigte Activity reaktiviert. Vielleicht fragen Sie sich, aus wie vielen Activities *Hallo Android* besteht. Theoretisch könnten Sie die App in drei Activities unterteilen, die Sie unabhängig voneinander anlegen müssten:

1. Begrüßung anzeigen
2. Namen eingeben
3. personalisierten Gruß anzeigen

Das wäre sinnvoll, wenn die entsprechenden Aufgaben umfangreiche Benutzereingaben oder aufwendige Netzwerkkommunikation erforderten. Dies ist hier nicht der Fall. Da die gesamte Anwendung aus sehr wenigen Bedienelementen besteht, ist es hier zielführender, alle Funktionen in einer Activity abzubilden. Bitte übernehmen Sie die Klasse `MainActivity` aus der im Folgenden dargestellten ersten Version.

In der Methode `onCreate()` wird mit `setContentView()` die Benutzeroberfläche geladen und angezeigt. Danach werden durch den Aufruf der Methode `findViewById()` zwei Referenzen auf Bedienelemente ermittelt und den Variablen `message` und `nextFinish` zugewiesen. `setText()` setzt die Beschriftung der Schaltfläche sowie des Textfeldes. Hierzu erfahren Sie gleich mehr. Bitte achten Sie darauf, in Ihren Apps `findViewById()` erst nach `setContentView()` aufzurufen. Andernfalls drohen Abstürze.

Hinweis

Etwas später zeige ich Ihnen, wie Sie auf Views zugreifen können, ohne Referenzen auf diese in Instanzvariablen zu halten. Allerdings ist dazu *Magie* nötig. Da ich glaube, dass für ein solides Verständnis der Plattform die Kenntnis möglichst vieler Zusammenhänge wichtig ist, sehen Sie zunächst den klassischen Weg, wie er seit der ersten Android-Version verwendet wird.

```
package com.thomaskuenneth.androidbuch.halloandroid

import android.os.Bundle
import android.widget.Button
import android.widget.TextView
import androidx.appcompat.app.AppCompatActivity

class MainActivity : AppCompatActivity() {

  private lateinit var message: TextView
  private lateinit var nextFinish: Button

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
```



```
    setContentView(R.layout.activity_main)

    message = findViewById(R.id.message)
    nextFinish = findViewById(R.id.next_finish)

    message.setText(R.string.welcome)
    nextFinish.setText(R.string.next)
  }
}
```

Listing 2.6 Erste Version der Klasse »MainActivity«

Abbildung 2.10 Toolbar-Symbole zum Starten von Apps

Um die Anwendung zu starten, wählen Sie wie in Abbildung 2.10 dargestellt das gewünschte echte oder virtuelle Gerät aus und klicken danach auf den grünen Play-Button.

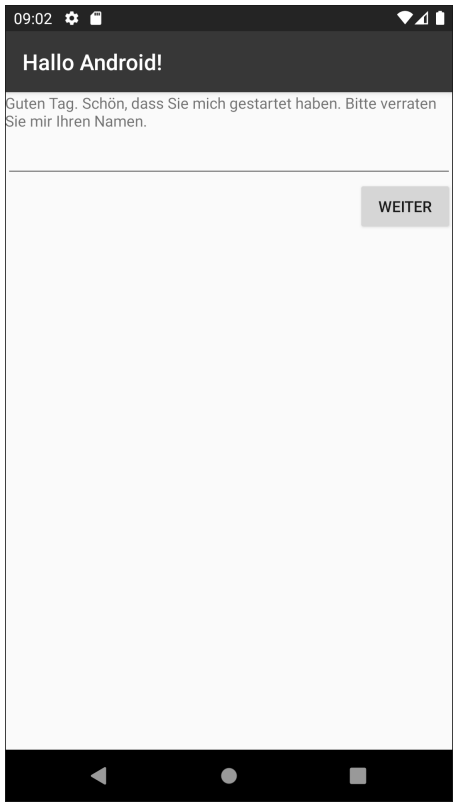


Abbildung 2.11 Die erste Version von »Hallo Android«

Nach der Installation sollte das Emulator-Fenster bzw. der Bildschirm des echten Geräts in etwa Abbildung 2.11 entsprechen. »In etwa«, weil es ein paar Faktoren gibt, die die Darstellung einer App beeinflussen, zum Beispiel:

- die Plattformversion des Emulators bzw. echten Geräts
- der API-Level in der Datei *build.gradle*
- Bildschirmeinstellungen und Schriftgröße im (simulierten) Gerät

Lassen Sie uns zunächst weiter auf Ihre erste eigene App konzentrieren. Das Textfeld nimmt zwar Eingaben entgegen, das Anklicken der Schaltfläche WEITER löst aber selbstverständlich noch keine Aktion aus. Diese Aktion werden wir im nächsten Abschnitt implementieren. Zuvor möchte ich Sie aber mit einigen Schlüsselstellen des Quelltextes vertraut machen. Ganz wichtig: Jede Activity erbt von der Klasse `android.app.Activity` oder von spezialisierten Kindklassen. Mein Beispiel verwendet `androidx.appcompat.app.AppCompatActivity`. Sie stellt eine Reihe von Funktionen zur Verfügung, die dem Original fehlen. Wir werden im weiteren Verlauf des Buches noch ausführlich darauf zu sprechen kommen.

Haben Sie bemerkt, dass die gesamte Programmlogik in der Methode `onCreate()` liegt? Activities haben einen ausgeklügelten Lebenszyklus, den ich Ihnen in Kapitel 4, »Wichtige Grundbausteine von Apps«, ausführlicher vorstelle. Seine einzelnen Stationen werden durch bestimmte Methoden der Klasse `Activity` realisiert, die Sie bei Bedarf überschreiben können. Beispielsweise informiert die Plattform eine Activity, kurz bevor sie beendet, unterbrochen oder zerstört wird. Die Methode `onCreate()` wird immer überschrieben. Sie ist der ideale Ort, um die Benutzeroberfläche aufzubauen und Variablen zu initialisieren. Ganz wichtig ist, mit `super.onCreate()` die Implementierung der Elternklasse aufzurufen. Sonst wird zur Laufzeit die Ausnahme `SuperNotCalledException` ausgelöst. Das Laden und Anzeigen der Bedienelemente reduziert sich auf eine Zeile Quelltext:

```
setContentView(R.layout.activity_main)
```

Sie sorgt dafür, dass alle Views und ViewGroups, die in der Datei *activity_main.xml* definiert wurden, zu einem Objektbaum entfaltet werden und dieser als Inhaltsbereich der Activity gesetzt wird. Warum ich den Begriff »entfalten« verwende, erkläre ich Ihnen in Kapitel 5, »Benutzeroberflächen«.

Möglicherweise fragen Sie sich, woher die Klasse `R` stammt. Sie wird von den *Build Tools* automatisch generiert und auf dem aktuellen Stand gehalten. Ihr Zweck ist es, Elemente aus Layout- und anderen XML-Dateien im Java- oder Kotlin-Quelltext verfügbar zu machen. `R.layout.activity_main` referenziert also die XML-Datei mit Namen *activity_main*.

Der Inhalt des Textfeldes `message` und die Beschriftung der Schaltfläche `nextFinish` werden auf sehr ähnliche Weise festgelegt: Zunächst ermitteln wir durch Aufruf der Methode `findViewById()` eine Referenz auf das gewünschte Objekt. `R.id.message` und `R.id.next_finish` verweisen hierbei auf Elemente, die wir ebenfalls in *activity_main.xml* definiert haben. Sehen Sie sich zur Verdeutlichung folgendes Dateifragment an:

```
<TextView
  android:id="@+id/message"
  android:layout_width="match_parent"
  android:layout_height="wrap_content" />
...
<Button
  android:id="@+id/next_finish"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:layout_gravity="end" />
```

Listing 2.7 Auszug aus der Datei »activity_main.xml«

Durch den Ausdruck `android:id="@+id/xyz"` entsteht ein Bezeichner, auf den Sie mit `R.id.xyz` zugreifen können. `xyz` ist der Name des Bezeichners. `@+id/` definiert ihn. Dies funktioniert nicht nur in Layoutdateien, sondern auch für die Definition von Texten, die in der Datei *strings.xml* abgelegt werden. Auch hierzu ein kurzer Auszug:

```
<!-- Beschriftungen für Schaltflächen -->
<string name="next">Weiter</string>
<string name="finish">Fertig</string>
```

Listing 2.8 Auszug aus der Datei »strings.xml«

Die Anweisung `nextFinish.setText(R.string.next)` legt den Text der einzigen Schaltfläche unserer App fest.

2.3.2 Benutzereingaben

Um *Hallo Android* zu komplettieren, müssen wir auf das Anklicken der Schaltfläche `nextFinish` reagieren. Beim ersten Mal wird das Textfeld `input` ausgelesen und als persönlicher Gruß in `message` eingetragen. Anschließend wird das Textfeld ausgeblendet und die Beschriftung der Schaltfläche geändert. Wird diese ein zweites Mal angeklickt, beendet sich die App. Im Folgenden sehen Sie die entsprechend erweiterte Fassung der Klasse `MainActivity`:

```
package com.thomaskuenneth.androidbuch.halloandroid

import android.os.Bundle
import android.view.View
import android.widget.Button
import android.widget.EditText
import android.widget.TextView
import androidx.appcompat.app.AppCompatActivity

class MainActivity : AppCompatActivity() {

  private lateinit var message: TextView
  private lateinit var nextFinish: Button
  private lateinit var input: EditText

  private var firstClick = true

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    message = findViewById(R.id.message)
    nextFinish = findViewById(R.id.next_finish)
    input = findViewById(R.id.input)

    message.setText(R.string.welcome)
    nextFinish.setText(R.string.next)
    nextFinish.setOnClickListener(fun(_: View) {
      if (firstClick) {
        message.text = getString(
          R.string.hello,
          input.text
        )
        input.visibility = View.INVISIBLE
        nextFinish.setText(R.string.finish)
        firstClick = false
      } else {
        finish()
      }
    })
  }
}
```

Listing 2.9 Zweite Fassung der Klasse »MainActivity«

Um auf das Anklicken der Schaltfläche reagieren zu können, wird ein sogenannter *OnClickListener* registriert. Dieses Interface besteht aus der Methode `onClick()`. Ihr wird nur ein Wert übergeben, nämlich die angeklickte View. Wenn Sie in Ihrem Code nicht weiter damit arbeiten, bietet es sich an, den Unterstrich _ anstelle eines Variablennamens zu verwenden. Die hier vorgestellte Implementierung unter Verwendung einer anonymen Funktion nutzt die `Boolean`-Variable `firstClick`, um die durchzuführenden Aktionen zu bestimmen. `input.visibility = View.INVISIBLE` blendet das Eingabefeld aus. `getString(R.string.hello, input.getText())` liefert den in *strings.xml* definierten persönlichen Gruß und fügt an der Stelle `%1$s` den vom Benutzer eingetippten Namen ein. Um die App zu beenden, wird die Methode `finish()` der Klasse `Activity` aufgerufen.

Hinweis

Kotlin bietet vielfach eine schlankere Syntax als Java. Beispielsweise ist es oft möglich, Klammern wegzulassen und Lambda-Ausdrücke unmittelbar anzuschließen. Im weiteren Verlauf des Buches werden Sie mehr und mehr diese kompakten Formen sehen. Für den Einstieg halte ich die konservative Darstellung aber für zielführender.

2.3.3 Der letzte Schliff

In diesem Abschnitt möchte ich Ihnen zeigen, wie Sie *Hallo Android* den letzten Schliff geben. Zum Beispiel kann das System in leeren Eingabefeldern einen Hinweis anzeigen, was der Benutzer eingeben soll. Hierzu fügen Sie in der Datei *strings.xml* die folgende Zeile ein:

```
<string name="firstname_surname">Vorname Nachname</string>
```

Anschließend erweitern Sie in *activity_main.xml* das Element `<EditText>` um das Attribut `android:hint="@string/firstname_surname"`. Damit verschwindet übrigens auch die Warnung, die Sie etwas weiter vorne gesehen haben. Starten Sie die App, um sich das Ergebnis anzusehen. Abbildung 2.12 zeigt das entsprechend abgeänderte Programm.

Schon besser, aber noch nicht perfekt. Drücken Sie während der Eingabe eines Namens nämlich auf ↵, wandert der Cursor in die nächste Zeile, und auch die Höhe des Eingabefeldes nimmt zu. Dieses Verhalten lässt sich zum Glück leicht korrigieren. Erweitern Sie hierzu `<EditText>` um die folgenden vier Attribute:

```
android:lines="1"
android:inputType="textCapWords"
android:autofillHints="personName"
android:imeOptions="actionNext"
```

Abbildung 2.12 Leeres Eingabefeld mit Hinweis

Damit begrenzen wir die Eingabe auf eine Zeile, und der erste Buchstabe eines Wortes wird automatisch in einen Großbuchstaben umgewandelt. Ferner teilen wir dem *Autofill framework* von Android mit, dass hier Personennamen (Vorname und Nachname) eingegeben werden. Schließlich löst das Drücken von ↵ bzw. das Anklicken des korrespondierenden Symbols auf der virtuellen Gerätetastatur eine Aktion aus. Um auf diese reagieren zu können, müssen wir in der Activity ebenfalls eine Kleinigkeit hinzufügen. Unter die Zeile `input = findViewById(R.id.input)` gehören die folgenden Zeilen:

```
input.setOnEditorActionListener(fun(_, _, _): Boolean {
  if (nextFinish.isEnabled) {
    nextFinish.performClick()
  }
  return true
})
```

Listing 2.10 Einen Button im Code auslösen

Das Interface `TextView.OnEditorActionListener` definiert eine Methode, `onEditorAction()`. Sie erhält als Argumente die betroffene `TextView`, eine ID, die die ausgelöste

Aktion repräsentiert, sowie ein `KeyEvent` oder `null`. Der Aufruf der Methode `performClick()` simuliert das Antippen der Schaltfläche WEITER. Dadurch wird der Code ausgeführt, den wir in der Methode `onClick()` der Klasse `OnClickListener` implementiert haben. Alternativ hätten wir diesen Code auch in eine eigene Methode auslagern und diese an beiden Stellen aufrufen können. Aber Sie wissen nun, wie Sie das Antippen einer Komponente simulieren können. Übrigens prüft `isEnabled`, ob die Schaltfläche aktiv oder inaktiv ist. Das werden wir gleich noch brauchen.

Schließlich wollen wir noch dafür sorgen, dass die Bedienelemente nicht mehr an den Rändern der Anzeige kleben. Eine kurze Anweisung schiebt zwischen ihnen und dem Rand einen kleinen leeren Bereich ein. Fügen Sie dem XML-Tag `<LinearLayout>` einfach das Attribut `android:padding="10dp"` hinzu. *Padding* wirkt nach innen. Das `LinearLayout` ist eine Komponente, die weitere Elemente enthält. Diese werden in horizontaler oder vertikaler Richtung angeordnet. Mit `android:padding` legen Sie fest, wie nahe die Schaltfläche, das Textfeld und die Eingabezeile der oberen, unteren, linken und rechten Begrenzung kommen können.

Im Gegensatz dazu wirkt *Margin* nach außen. Hiermit können Sie einen Bereich um die Begrenzung einer Komponente herum definieren. Auch hierzu ein Beispiel: Fügen Sie dem XML-Tag `<Button>` das Attribut `android:layout_marginTop="16dp"` hinzu, wird die Schaltfläche deutlich nach unten abgesetzt. Sie haben einen oberen Rand definiert, der gegen die untere Begrenzung der Eingabezeile wirkt. Werte, die auf `dp` enden, geben übrigens geräteunabhängige Pixelgrößen an. Sie beziehen die Auflösung der Anzeige eines Geräts mit ein.

> **Tipp**
>
> Wenn nach dem Einfügen von Kotlin-Codeschnipseln Teile des Quelltextes mit roten Schlangenlinien unterkringelt werden, liegt das sehr häufig an fehlenden `import`-Anweisungen. Um diese einzufügen, klicken Sie die angemeckerte Klasse an und drücken danach Alt+↵. Wiederholen Sie dies für alle nicht erkannten Klassen.

Fällt Ihnen noch ein Defizit der gegenwärtigen Version auf? Solange der Benutzer keinen Namen eingetippt hat, sollte die Schaltfläche WEITER nicht anwählbar sein. Das lässt sich mithilfe eines sogenannten *TextWatchers* leicht realisieren. Dazu fügen Sie in der Methode `onCreate()` vor dem Ende des Methodenrumpfes, also vor ihrer schließenden geschweiften Klammer, das folgende Quelltextfragment ein:

```
input.doAfterTextChanged {
    nextFinish.isEnabled = it?.isNotEmpty() ?: false
}
nextFinish.isEnabled = false
```

Listing 2.11 Auf Texteingaben reagieren

Jedes Mal, wenn ein Zeichen eingegeben oder gelöscht wird, ruft Android unsere Implementierung von `doAfterTextChanged` auf. Diese ist sehr einfach gehalten: Nur wenn der Name mindestens ein Zeichen lang ist, kann die Schaltfläche WEITER angeklickt werden. `doAfterTextChanged` ist eine Erweiterungsfunktion für `android.widget.TextView`. Sie gehört zu der Jetpack-Komponente *Android KTX*. Sie wird der modulspezifischen *build.gradle*-Datei in `dependencies { ... }` hinzugefügt:

```
implementation 'androidx.core:core-ktx:1.3.1'
```

Ferner müssen Sie in `android { ... }` die Zeile

```
kotlinOptions { jvmTarget = "1.8" }
```

eintragen. Als kleine Übung können Sie versuchen, die Prüfroutine so zu erweitern, dass Vor- und Nachname vorhanden sein müssen. Prüfen Sie der Einfachheit halber, ob der eingegebene Text ein Leerzeichen enthält, das nicht am Anfang und nicht am Ende steht.

Abbildung 2.13 Die fertige App »Hallo Android«

Damit haben Sie Ihre erste eigene Anwendung fast fertiggestellt. Es gibt nur noch eine kleine Unvollkommenheit: Die Schaltfläche FERTIG befindet sich gegenüber der Schaltfläche WEITER etwas näher am oberen Bildschirmrand. Der Grund dafür ist, dass die Grußfloskel meistens in eine Zeile passt, der Begrüßungstext aber zwei Zeilen benötigt. Beheben Sie dieses Malheur, indem Sie in der Layoutdatei innerhalb des `<TextView />`-Tags, zum Beispiel unterhalb von `android:id="@+id/message"`, die Zeile `android:lines="2"` einfügen. Abbildung 2.13 zeigt die fertige App *Hallo Android*.

2.4 Zusammenfassung

Sie haben in diesem Kapitel mithilfe des Projektassistenten ein neues Projekt angelegt und zu einer vollständigen App mit Benutzerinteraktion erweitert. Dabei haben Sie Layoutdateien kennengelernt und erste Erfahrungen mit dem quelltextunabhängigen Speichern von Texten gesammelt. In den folgenden Kapiteln vertiefen Sie dieses Wissen.

Kapitel 8

Sensoren, GPS und Bluetooth

Android-Geräte enthalten zahlreiche Sensoren und Schnittstellen, die sich mit geringem Aufwand in eigenen Apps nutzen lassen. Wie das funktioniert, zeige ich Ihnen in diesem Kapitel.

Geräte schalten ihre Anzeige ab, sobald man sie in Richtung des Kopfes bewegt, und die Darstellung auf dem Bildschirm passt sich der Ausrichtung an. Spiele reagieren auf Bewegungsänderungen. Karten-Apps erkennen automatisch den gegenwärtigen Standort. Restaurant- oder Kneipenführer beschreiben nicht nur den kürzesten Weg zur angesagten Döner-Bude, sondern präsentieren die Meinungen anderer Kunden und bieten Alternativen an. Und mit der Funktechnologie Bluetooth lassen sich im Handumdrehen Geräte in Reichweite ansprechen und vernetzen. Dies und noch viel mehr ist möglich, weil die Android-Plattform eine beeindruckende Sensoren- und Schnittstellenphalanx beinhaltet, die von allen Apps genutzt werden kann.

8.1 Sensoren

Android stellt seine Sensoren über eine Instanz der Klasse `SensorManager` zur Verfügung. Wie Sie diese verwenden, zeige ich Ihnen anhand des Projekts *SensorDemo1*. Die gleichnamige App (sie ist in Abbildung 8.1 zu sehen) ermittelt alle zur Verfügung stehenden Sensoren und gibt unter anderem deren Namen, Hersteller und Version aus. Außerdem verbindet sich das Programm mit dem Helligkeitssensor des Geräts und zeigt die gemessenen Werte an. Sensoren lassen sich grob in drei Kategorien unterteilen:

- *Bewegungssensoren* messen Beschleunigungs- und Drehkräfte entlang dreier Achsen. Zu dieser Kategorie gehören Accelerometer, Gyroskop und Gravitationsmesser.
- *Umweltsensoren* erfassen verschiedene Parameter der Umwelt, beispielsweise die Umgebungstemperatur, den Luftdruck, Feuchtigkeit und Helligkeit. Zu dieser Kategorie gehören Barometer, Photometer und Thermometer.
- *Positionssensoren* ermitteln die Position bzw. die Lage eines Geräts im Raum. Diese Kategorie beinhaltet Orientierungssensoren sowie Magnetometer.

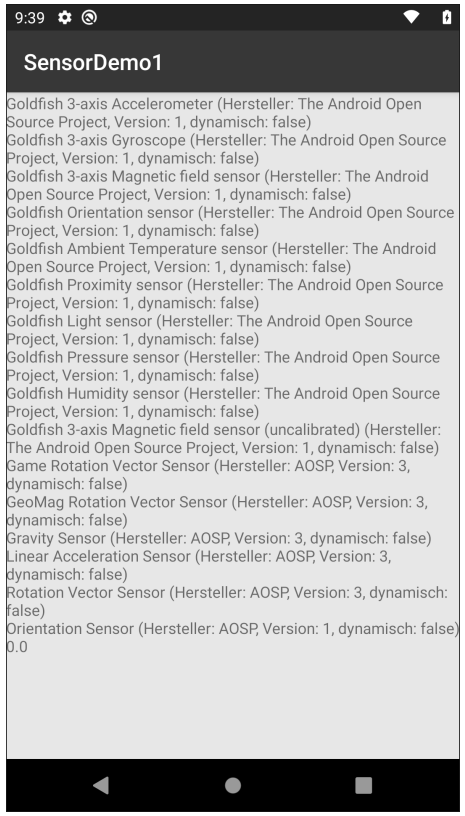


Abbildung 8.1 Die App »SensorDemo1«

Welche Messfühler einer App zur Verfügung stehen, hängt sowohl von der Plattformversion als auch von der Hardware ab, auf der die App ausgeführt wird. Android hat im Laufe der Zeit nämlich kontinuierlich neue Sensoren »gelernt«. Sensoren können durch Hard- oder Software realisiert werden. Je nach Typ verbrauchen sie viel oder wenig Strom. Einige liefern kontinuierlich Daten, andere nur, wenn sich seit der letzten Messung etwas geändert hat. Die Nutzung der Sensoren erfolgt primär über die Klasse `android.hardware.SensorManager`, die ich nun ausführlich vorstellen werde.

8.1.1 Die Klasse »SensorManager«

Die Methode `onCreate()` meiner Beispielklasse `SensorDemo1Activity` kümmert sich nur um das Laden und Anzeigen der Benutzeroberfläche. Alle sensorbezogenen Aktivitäten finden in `onResume()` und `onPause()` statt. Beim Fortsetzen der Activity wird mit `getSystemService(SensorManager::class.java)` die Referenz auf ein Objekt des Typs `android.hardware.SensorManager` ermittelt. Diese Methode ist in allen von `android.content.Context` abgeleiteten Klassen vorhanden, beispielsweise `android.app.`

`Activity` und `android.app.Service`. Anschließend können Sie mit `getSensorList()` herausfinden, welche Sensoren in Ihrer App zur Verfügung stehen. `name`, `vendor` und `version` liefern den Namen, den Hersteller und die Version des Sensors. Mit `isDynamicSensor` können Sie ermitteln, ob ein Sensor dynamisch ist. Was es damit auf sich hat, erkläre ich Ihnen im folgenden Abschnitt.

Beim Aufruf von `getSensorList()` können Sie anstelle von `TYPE_ALL` die übrigen mit `TYPE_` beginnenden Konstanten der Klasse `Sensor` nutzen, um nach einer bestimmten Art von Sensor »Ausschau zu halten«. Beispielsweise begrenzt `TYPE_LIGHT` die Trefferliste auf Helligkeitssensoren. `TYPE_STEP_DETECTOR` liefert Schrittdetektoren; solche Sensoren melden sich, wenn der Nutzer einen Fuß mit genügend »Schwung« auf den Boden stellt, also einen Schritt macht. In Abschnitt 8.1.3, »Ein Schrittzähler«, erfahren Sie mehr darüber.

Wenn Sie die Art des gewünschten Sensors schon kennen, ist es meist einfacher, anstelle von `getSensorList()` die Methode `getDefaultSensor()` aufzurufen. Allerdings weist die Android-Dokumentation darauf hin, dass diese Methode unter Umständen einen Sensor liefert, der gefilterte oder gemittelte Werte produziert. Möchten Sie dies – zum Beispiel aus Genauigkeitsgründen – nicht, dann verwenden Sie `getSensorList()`. Neben ihren Namen und Herstellern liefern Sensoren eine ganze Menge an Informationen, beispielsweise zu ihrem Stromverbrauch (`power`), ihrem Wertebereich (`maximumRange`) und ihrer Genauigkeit (`resolution`).

```
package com.thomaskuenneth.androidbuch.sensordemo1

import android.hardware.Sensor
import android.hardware.SensorEvent
import android.hardware.SensorEventListener
import android.hardware.SensorManager
import android.hardware.SensorManager.DynamicSensorCallback
import android.os.*
import android.util.Log
import androidx.appcompat.app.AppCompatActivity
import kotlinx.android.synthetic.main.activity_main.*

private val TAG = SensorDemo1Activity::class.simpleName
class SensorDemo1Activity : AppCompatActivity() {
  private lateinit var manager: SensorManager
  private val map = HashMap<String, Boolean>()
  private val listener = object : SensorEventListener {
    override fun onAccuracyChanged(sensor: Sensor,
                                   accuracy: Int) {
      Log.d(TAG, "onAccuracyChanged(): $accuracy")
    }
```

```
  override fun onSensorChanged(event: SensorEvent) {
    if (event.values.isNotEmpty()) {
      val light = event.values[0]
      var text = light.toString()
      if (SensorManager.LIGHT_SUNLIGHT <= light &&
          light <= SensorManager.LIGHT_SUNLIGHT_MAX) {
        text = getString(R.string.sunny)
      }
      // jeden Wert nur einmal ausgeben
      if (!map.containsKey(text)) {
        map[text] = true
        text += "\n"
        textview.append(text)
      }
    }
  }
}

private val callback = object : DynamicSensorCallback() {
  override fun onDynamicSensorConnected(sensor: Sensor) {
    textview.append(getString(R.string.connected,
        sensor.name))
  }

  override fun onDynamicSensorDisconnected(sensor: Sensor) {
    textview.append(getString(R.string.disconnected,
        sensor.name))
  }
}

private var listenerWasRegistered = false
private var callbackWasRegistered = false

override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_main)
}

override fun onResume() {
  super.onResume()
  map.clear()
  textview.text = ""
```



```
  manager = getSystemService(SensorManager::class.java)
  // Liste der vorhandenen Sensoren ausgeben
  manager.getSensorList(Sensor.TYPE_ALL).forEach {
    textview.append(getString(R.string.template, it.name,
        it.vendor, it.version, it.isDynamicSensor.toString()))
  }
  // Helligkeitssensor ermitteln
  manager.getDefaultSensor(Sensor.TYPE_LIGHT)?.also { sensor ->
    manager.registerListener(listener, sensor,
        SensorManager.SENSOR_DELAY_NORMAL)
    listenerWasRegistered = true
  } ?: textview.append(getString(R.string.no_seonsor))
  // Callback für dynamische Sensoren
  if (manager.isDynamicSensorDiscoverySupported) {
    manager.registerDynamicSensorCallback(callback,
        Handler(Looper.getMainLooper()))
    callbackWasRegistered = true
  }
}

override fun onPause() {
  super.onPause()
  if (listenerWasRegistered) {
    manager.unregisterListener(listener)
  }
  if (callbackWasRegistered) {
    manager.unregisterDynamicSensorCallback(callback)
  }
 }
}
```

Listing 8.1 Die Klasse »SensorDemo1Activity«

Seit *API-Level 21* kennt die Methode `getDefaultSensor()` einen optionalen zweiten Parameter. Er steuert, ob das System sogenannte *Wake-up-* oder *Non-Wake-up*-Sensoren liefert. Der Unterschied besteht darin, ob Sensoren für das Melden von Daten den SoC (*System on a Chip*) aus dem Ruhezustand aufwecken und das Wechseln in diesen Modus verhindern (wake-up) oder nicht (non-wake-up). Sofern Sensordaten nur während der Ausführung einer Activity erhoben und angezeigt werden, ist die Unterscheidung irrelevant. Für eine möglichst unterbrechungsfreie Aufzeichnung im Hintergrund sind Wake-up-Sensoren die bessere Wahl. Andernfalls muss die App selbstständig dafür sorgen, dass der SoC nicht in den Ruhezustand wechselt. Weitere Informationen finden Sie unter *https://source.android.com/devices/sensors/suspend-mode.html*.

SensorEventListener

Mit den Methoden `registerListener()` und `unregisterListener()` der Klasse `SensorManager` können Sie sich über Sensorereignisse informieren lassen sowie entsprechende Benachrichtigungen wieder deaktivieren. `registerListener()` erwartet ein Objekt des Typs `android.hardware.SensorEventListener`, den Sensor sowie eine Angabe zur Häufigkeit, mit der Wertänderungen übermittelt werden sollen. Sie können einen vordefinierten Wert, zum Beispiel `SensorManager.SENSOR_DELAY_NORMAL`, oder eine Zeitspanne in Mikrosekunden übergeben. Android garantiert die Einhaltung dieses Wertes allerdings nicht. Sensorereignisse können also häufiger oder seltener zugestellt werden.

Das Activity-Methodenpaar `onResume()` und `onPause()` bietet sich an, um `SensorEventListener` zu registrieren bzw. zu entfernen. Prüfen Sie genau, ob das Sammeln von Sensordaten auch dann erforderlich ist, wenn Ihre Activity nicht ausgeführt wird. Je nach Sensor kann das Messen nämlich in erheblichem Maße Strom verbrauchen.

`SensorEventListener`-Objekte implementieren die Methoden `onAccuracyChanged()` und `onSensorChanged()`. Erstere wird aufgerufen, wenn sich die Genauigkeit eines Sensors geändert hat. Wie wichtig diese Information ist, hängt von der Art des verwendeten Messfühlers ab. Sollte es beispielsweise Probleme beim Ermitteln der Herzfrequenz geben, weil der Sensor kalibriert werden muss (`SENSOR_STATUS_UNRELIABLE`) oder weil er keinen Körperkontakt hat (`SENSOR_STATUS_NO_CONTACT`), dann sollte Ihre App auf jeden Fall einen entsprechenden Hinweis anzeigen. Ist hingegen die Genauigkeit des Barometers nicht mehr hoch (`SENSOR_STATUS_ACCURACY_HIGH`), sondern nur noch durchschnittlich (`SENSOR_STATUS_ACCURACY_MEDIUM`), ist vielleicht keine diesbezügliche Aktion erforderlich.

Die Methode `onSensorChanged()` wird aufgerufen, wenn neue Sensordaten vorliegen. Die App *SensorDemo1* nutzt den Helligkeitssensor eines Geräts und gibt je nach Helligkeit den gemessenen Wert oder den Text »sonnig« aus. Die Klasse `SensorManager` enthält zahlreiche Konstanten, die sich auf die vorhandenen Ereignistypen beziehen. Auf diese Weise können Sie, wie im Beispiel zu sehen ist, das Ergebnis der Helligkeitsmessung auswerten, ohne selbst in entsprechenden Tabellen nachschlagen zu müssen.

Tipp

Liefert die `Sensor`-Methode `isAdditionalInfoSupported()` den Wert `true`, kann ein Sensor über einen neuen Mechanismus weitere, zusätzliche Informationen preisgeben. Sie sind in der Klasse `SensorAdditionalInfo` enthalten. Um solche Objekte zu empfangen, registrieren Sie mit `registerListener()` anstelle von `SensorEventListener` ein Objekt des Typs `SensorEventCallback` und überschreiben zusätzlich die Methode `onSensorAdditionalInfo()`.

Welche Werte in dem `SensorEvent`-Objekt übermittelt werden und wie Sie diese interpretieren, hängt vom verwendeten Sensor ab. Beispielsweise liefert der Umgebungstemperatursensor (`TYPE_AMBIENT_TEMPERATURE`) in `values[0]` die Raumtemperatur in Grad Celsius. Luftdruckmesser (`Sensor.TYPE_PRESSURE`) tragen dort hingegen den atmosphärischen Druck in Millibar ein.

Die von einem Android-Gerät oder dem Emulator zur Verfügung gestellten Sensoren können Sie erst zur Laufzeit Ihrer App ermitteln. Selbstverständlich sollten Sie nicht einfach Ihre Activity beenden, wenn ein benötigter Sensor nicht zur Verfügung steht, sondern einen entsprechenden Hinweis ausgeben.

Mithilfe des Elements `<uses-feature>` der Manifestdatei können Sie die Sichtbarkeit in *Google Play* auf geeignete Geräte einschränken. Hierzu ein Beispiel:

```
<uses-feature android:name="android.hardware.sensor.barometer"
   android:required="true" />
```

Apps, deren Manifest ein solches Element enthält, werden in Google Play nur auf Geräten angezeigt, in die ein Barometer eingebaut ist. Beachten Sie hierbei aber, dass diese Filterung eine Installation nicht verhindert, falls die App auf anderem Wege auf das Gerät gelangt ist. Deshalb ist es wichtig, vor der Nutzung eines Sensors seine Verfügbarkeit wie weiter oben gezeigt zu prüfen.

8.1.2 Dynamische Sensoren und Trigger

Mit Android 7 hat Google sogenannte *dynamische Sensoren* eingeführt. Bisher war es so, dass ein Sensor entweder immer »da ist« (weil er in ein Smartphone oder Tablet eingebaut wurde) oder eben nicht. Was aber wäre, wenn man ein Gerät durch Module erweitern und je nach Bedarf Sensoren andocken oder abklemmen könnte? Google hatte mit dem Projekt *Ara* die Vision eines voll modularen Smartphones. Unglücklicherweise wurde es eingestellt, aber es ist möglich, dass andere Hersteller die Idee irgendwann wieder aufgreifen.

Apps können über `isDynamicSensorDiscoverySupported` abfragen, ob das System das Erkennen von dynamischen Sensoren unterstützt. In diesem Fall lässt sich mit `registerDynamicSensorCallback()` ein Objekt des Typs `DynamicSensorCallback` registrieren. Seine Methoden `onDynamicSensorConnected()` und `onDynamicSensorDisconnected()` werden nach dem Verbinden bzw. Trennen eines dynamischen Sensors aufgerufen. Dies ist in Listing 8.1 zu sehen.

Analog zu `SensorManager.getSensorList()` können Sie übrigens mit `getDynamicSensorList()` die Liste aller aktuell bekannten dynamischen Sensoren eines Typs abfragen.

Trigger-Sensoren

Viele Daten (zum Beispiel Temperatur, Luftdruck und Helligkeit) können bei Bedarf kontinuierlich erfasst werden, denn sie liegen immer vor. Deshalb ist es bewährte Praxis, Listener nur bei Bedarf zu registrieren und nach Gebrauch wieder zu entfernen. Je nach Sensor ist der Akku des Geräts sonst möglicherweise schnell leer. Eine Ausnahme von dieser Regel stelle ich Ihnen übrigens im nächsten Abschnitt vor. Es gibt aber auch Ereignisse, die unvorhersehbar irgendwann eintreten; dann ist eine kontinuierliche Messung sinnlos.

Für solche Fälle kennt Android *Trigger*-Sensoren. Der *Significant-Motion*-Sensor ist ein Trigger. Er meldet sich, wenn das System eine Bewegung erkennt, die wahrscheinlich zu einer Positionsänderung führt. Dies ist beim Laufen, Fahrrad- oder Autofahren der Fall. Trigger-Sensoren liefern beim Zugriff auf `reportingMode` den Wert `REPORTING_MODE_ONE_SHOT`. Um einen solchen Sensor zu aktivieren, registrieren Sie nicht mit `SensorManager.register()` einen `SensorEventListener`, sondern rufen `requestTriggerSensor()` auf und übergeben der Methode ein `TriggerEventListener`-Objekt. Dessen Methode `onTrigger()` wird vom System aufgerufen, wenn der Trigger aktiviert wurde. Danach wird der Trigger automatisch deaktiviert. Um erneut informiert zu werden, müssen Sie deshalb wieder `requestTriggerSensor()` aufrufen.

Mein Beispiel *SensorDemo2* fasst dies in einer kompakten App zusammen. Sie funktioniert folgendermaßen: Nach dem Start wartet die App auf eine plötzliche Bewegung und gibt dann die aktuelle Uhrzeit aus. Nach einem Klick auf WEITER beginnt der Vorgang von vorne. Bei Gerätedrehungen merkt sie sich den aktuellen Zustand. Und so sieht die Hauptklasse `SensorDemo2Activity` aus:

```
package com.thomaskuenneth.androidbuch.sensordemo2

import android.hardware.Sensor
import android.hardware.SensorManager
import android.hardware.TriggerEvent
import android.hardware.TriggerEventListener
import android.os.Bundle
import android.view.View
import androidx.appcompat.app.AppCompatActivity
import kotlinx.android.synthetic.main.activity_main.*
import java.text.DateFormat
import java.util.*

private const val KEY1 = "shouldCallWaitForTriggerInOnResume"
private const val KEY2 = "tv"
class SensorDemo2Activity : AppCompatActivity() {
  private val dateFormat = DateFormat.getTimeInstance()
```



```
  private val listener = object : TriggerEventListener() {
    override fun onTrigger(event: TriggerEvent) {
      shouldCallWaitForTriggerInOnResume = false
      button.visibility = View.VISIBLE
      textview.text = dateFormat.format(Date())
    }
  }
  private var shouldCallWaitForTriggerInOnResume = false
  private var sensor: Sensor? = null
  private lateinit var manager: SensorManager

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    button.setOnClickListener {
      shouldCallWaitForTriggerInOnResume = true
      waitForTrigger()
    }
    manager = getSystemService(SensorManager::class.java)
    sensor = manager.getDefaultSensor(Sensor.TYPE_SIGNIFICANT_MOTION)
    sensor?.also {
      shouldCallWaitForTriggerInOnResume = true
      savedInstanceState?.run {
        shouldCallWaitForTriggerInOnResume = getBoolean(KEY1)
        textview.text = getString(KEY2)
      }
    } ?: run {
      shouldCallWaitForTriggerInOnResume = false
      button.visibility = View.GONE
      textview.setText(R.string.no_sensors)
    }
  }

  override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)
    outState.putBoolean(KEY1, shouldCallWaitForTriggerInOnResume)
    outState.putString(KEY2, textview.text.toString())
  }

  override fun onResume() {
    super.onResume()
    sensor?.let {
      if (shouldCallWaitForTriggerInOnResume) {
```

```
            waitForTrigger()
          }
        }
      }

      override fun onPause() {
        super.onPause()
        sensor?.let {
          manager.cancelTriggerSensor(listener, sensor)
        }
      }

      private fun waitForTrigger() {
        button.visibility = View.GONE
        textview.setText(R.string.wait)
        manager.requestTriggerSensor(listener, sensor)
      }
    }
```

Listing 8.2 Die Klasse »SensorDemo2Activity«

In `onCreate()` wird als Erstes die Benutzeroberfläche angezeigt und danach mit `getDefaultSensor(Sensor.TYPE_SIGNIFICANT_MOTION)` der Standard-Trigger-Sensor ermittelt. Aktiviert wird er durch den Aufruf von `requestTriggerSensor()`. Das geschieht allerdings in `onResume()`. Hierzu rufe ich die private Methode `waitForTrigger()` auf. Gleiches passiert beim Anklicken des Buttons `button`. Beim Pausieren (`onPause()`) wird der Sensor mit `cancelTriggerSensor()` deaktiviert.

Um beim Drehen des Geräts den aktuellen Zustand speichern und wiederherstellen zu können, habe ich `onSaveInstanceState()` überschrieben. Meine Implementierung schreibt zwei Werte, die `Boolean`-Variable `shouldCallWaitForTriggerInOnResume` sowie den Inhalt des Textfeldes `textview`. Beide werden gegebenenfalls in `onCreate()` wieder gesetzt, wenn `savedInstanceState` nicht `null` ist. Möchten Sie, dass Ihre App auch dann informiert wird, wenn keine Activity abgearbeitet wird, müssen Sie die beiden Methodenaufrufe in einen Service auslagern. Geht das Gerät aber in den Ruhezustand, während `SensorDemo2Activity` aktiv ist, wird die Aktivität nach dem Aufwachen des Geräts aktualisiert. Der *Significant-Motion*-Sensor arbeitet nämlich weiter, während das Gerät schläft.

Vielleicht ist Ihnen beim Stöbern in der Dokumentation aufgefallen, dass die Klasse `TriggerEvent` einen Zeitstempel enthält, der den Zeitpunkt des Auftretens in Nanosekunden angibt. Dieser Wert ist nicht dafür gedacht, Uhrzeiten oder Datumsangaben abzuleiten. Er sollte nur verwendet werden, um Abstände zwischen Aufrufen eines Sensors zu ermitteln.

8.1.3 Ein Schrittzähler

In diesem Abschnitt sehen wir uns einen weiteren Sensor an, den Schrittzähler. Er meldet die Anzahl der Schritte seit dem letzten Start des Geräts, allerdings nur, solange er aktiviert ist. Google empfiehlt in der Dokumentation deshalb, **nicht** die Methode `unregisterListener()` aufzurufen, wenn Langzeitmessungen erfolgen sollen. Diese sind unproblematisch, weil der Sensor in Hardware implementiert ist und wenig Strom verbraucht. Befindet sich das Gerät im Ruhemodus, werden bei aktiviertem Sensor weiterhin Schritte gezählt und nach dem Wiederaufwachen gemeldet. Alles in allem eine wirklich praktische Angelegenheit.

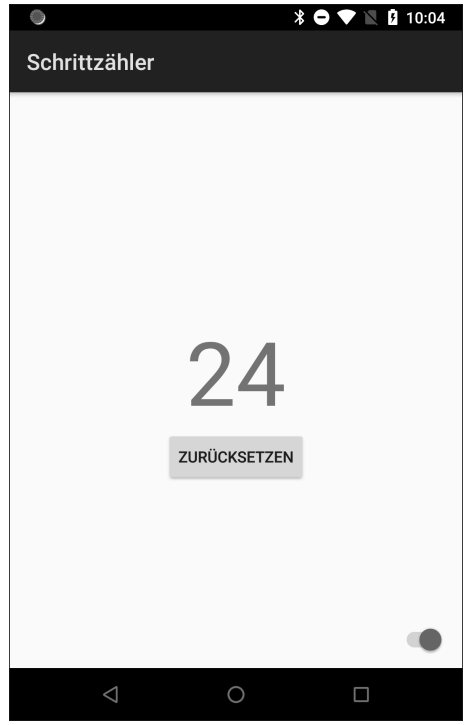


Abbildung 8.2 Die App »SensorDemo3«

Vielleicht fragen Sie sich, was passiert, wenn Sie die Anzahl der Schritte ganz bewusst zurücksetzen möchten. Da der Zähler die Schritte seit dem letzten Systemstart zählt, müssten Sie das Smartphone oder Tablet neu starten. Das klingt nicht sehr elegant. In meiner App *SensorDemo3* (sie ist in Abbildung 8.2 zu sehen) zeige ich Ihnen, wie Sie dieses Problem mit SharedPreferences und einem Broadcast Receiver lösen. Bitte sehen Sie sich Listing 8.3 an. Als Erstes fällt auf, dass es neben der Klasse `SensorDemo3Activity` die Toplevel-Funktionen `updateSharedPrefs()` und `getSharedPreferences()` (sie ist privat) enthält. Wir brauchen sie für das Zurücksetzen des Zählers. Ich komme etwas später darauf zurück.

In der Methode `onCreate()` wird die Benutzeroberfläche geladen und angezeigt. Da ab Android 10 Schrittzähler-Ereignisse nur dann an Apps gemeldet werden, wenn diese die Berechtigung `ACTIVITY_RECOGNITION` angefordert haben und der Nutzer sie auch gewährt hat, findet eine Fallunterscheidung statt: Wurde die Berechtigung schon erteilt oder ist das gar nicht nötig, weil die Android-Version älter ist, wird durch Aufruf der Methode `showStepCounterUi()` die Schrittzähler-Oberfläche dargestellt. Andernfalls erscheint ein Hinwies, dass die Berechtigung noch fehlt, sowie ein Button. Ein Klick auf Anfordern führt `requestPermissions()` aus. In `onRequestPermissionsResult()` wird ebenfalls `showStepCounterUi()` aufgerufen.

`getSystemService()` und `getDefaultSensor()` liefern wie gehabt Referenzen auf Objekte des Typs `SensorManager` bzw. `Sensor`. In `showStepCounterUi()` wird je nach Zustand des Schalters `on_off` entweder `registerListener()` oder `unregisterListener()` aufgerufen. Das sonst übliche Deaktivieren des Sensors in `onPause()` entfällt. Sie erinnern sich: Google rät zu diesem Vorgehen, um Langzeitmessungen vornehmen zu können. Sie sollten in Ihrer App auf jeden Fall eine Möglichkeit vorsehen, den Sensor zu deaktivieren.

```
package com.thomaskuenneth.androidbuch.sensordemo3

import android.Manifest
import android.content.Context
import android.content.pm.PackageManager
import android.hardware.Sensor
import android.hardware.SensorEvent
import android.hardware.SensorEventListener
import android.hardware.SensorManager
import android.os.*
import androidx.appcompat.app.AppCompatActivity
import kotlinx.android.synthetic.main.activity_main.*
import kotlinx.android.synthetic.main.no_permission.*

private const val PREFERENCES_KEY = "last"
private const val REQUEST_ACTIVITY_RECOGNITION = 123
fun updateSharedPrefs(
  context: Context?,
  last: Int
) {
  val edit = getSharedPreferences(context)?.edit()
  edit?.putInt(PREFERENCES_KEY, last)
  edit?.apply()
}
```



```
private fun getSharedPreferences(context: Context?) = context?
.getSharedPreferences(
  SensorDemo3Activity::class.simpleName,
  Context.MODE_PRIVATE
)

class SensorDemo3Activity : AppCompatActivity(), SensorEventListener {
  private lateinit var manager: SensorManager
  private var sensor: Sensor? = null
  private var hasSensor = false
  private var last = 0

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    manager = getSystemService(SensorManager::class.java)
    sensor = manager.getDefaultSensor(Sensor.TYPE_STEP_COUNTER)
    hasSensor = sensor != null
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.Q &&
      checkSelfPermission(Manifest.permission.ACTIVITY_RECOGNITION)
        != PackageManager.PERMISSION_GRANTED
    ) {
      setContentView(R.layout.no_permission)
      button_permission.setOnClickListener {
        requestPermissions(
          arrayOf(Manifest.permission.ACTIVITY_RECOGNITION),
          REQUEST_ACTIVITY_RECOGNITION
        )
      }
    } else {
      showStepCounterUi()
    }
  }

  override fun onRequestPermissionsResult(
    requestCode: Int,
    permissions: Array<out String>,
    grantResults: IntArray
  ) {
    super.onRequestPermissionsResult(requestCode, permissions,
                                     grantResults)
    if (requestCode == REQUEST_ACTIVITY_RECOGNITION &&
      grantResults.isNotEmpty() &&
```

```
      grantResults[0] == PackageManager.PERMISSION_GRANTED
    ) {
      showStepCounterUi()
    }
  }

  override fun onSensorChanged(sensorEvent: SensorEvent) {
    val values = sensorEvent.values
    updateSteps(values[0].toInt())
  }

  override fun onAccuracyChanged(sensor: Sensor?, i: Int) {
  }

  private fun updateSteps(currentSteps: Int) {
    last = currentSteps
    steps.text = "${currentSteps - getLastStoredStepCount()}"
  }

  private fun getLastStoredStepCount() = getSharedPreferences(this)
    ?.getInt(PREFERENCES_KEY, 0) ?: 0

  private fun showStepCounterUi() {
    setContentView(R.layout.activity_main)
    reset.setOnClickListener {
      updateSharedPrefs(this, last)
      updateSteps(last)
    }
    on_off.setOnCheckedChangeListener { _, isChecked ->
      if (isChecked) {
        manager.registerListener(
          this, sensor,
          SensorManager.SENSOR_DELAY_UI
        )
      } else {
        manager.unregisterListener(this)
      }
    }
    on_off.isEnabled = hasSensor
    on_off.isChecked = hasSensor
    reset.isEnabled = hasSensor
    steps.text = getString(
      if (!hasSensor) {
        R.string.no_sensor
```



```
      } else {
        R.string.waiting
      }
    )
  }
}
```

Listing 8.3 Die Datei »SensorDemo3Activity.kt«

In `onSensorChanged()` wird aus dem Feld `sensorEvent.values` die Anzahl der Schritte seit dem letzten Systemstart ausgelesen und der privaten Methode `updateSteps()` übergeben. Diese sieht in den anwendungsspezifischen Voreinstellungen nach, ob dort ein Schlüssel mit dem Namen »last« (in der `String`-Konstante `PREFERENCES_KEY` hinterlegt) gespeichert wurde. Falls ja, wird der gespeicherte Wert von der Anzahl der Schritte seit dem letzten Systemstart abgezogen. Wenn der Anwender die Schaltfläche ZURÜCKSETZEN anklickt, wird in den Voreinstellungen der aktuelle Zählerstand gespeichert. Damit lässt sich ohne großen Aufwand das Zurücksetzen des Schrittzählers nachbauen. Eine Kleinigkeit gibt es aber zu beachten: Wenn das Gerät neu gestartet wurde, muss der gespeicherte Wert selbst auf 0 gesetzt werden, damit die App-Anzeige nicht verfälscht wird. Deshalb registriere ich in der Manifestdatei einen Receiver, der auf `android.intent.action.BOOT_COMPLETED` reagiert. Die Implementierung ist einfach. Sie ist in Listing 8.4 zu sehen:

```
package com.thomaskuenneth.androidbuch.sensordemo3

import android.content.BroadcastReceiver
import android.content.Context
import android.content.Intent

class BootCompletedReceiver : BroadcastReceiver() {
  override fun onReceive(context: Context?, intent: Intent) {
    if (Intent.ACTION_BOOT_COMPLETED == intent.action) {
      updateSharedPrefs(context, 0)
    }
  }
}
```

Listing 8.4 Die Klasse »BootCompletedReceiver«

Durch Aufruf der Toplevel-Funktion `updateSharedPrefs()` wird der beim letzten Anklicken der Schaltfläche ZURÜCKSETZEN gespeicherte Wert mit 0 überschrieben, und damit stimmt die Berechnung der Schritte wieder. Damit verlassen wir den spannenden Bereich der klassischen Sensoren. Im Folgenden kümmern wir uns um Standortbestimmungen und um die Integration in Google Maps.

Auf einen Blick

Inhalt

10 Datenbanken 451

TEIL V Multimedia und Produktivität

11 Multimedia 483

12 Kontakte und Organizer 535

Anhang 559