HANSER

Android-Ergänzungen

zu

„Programmieren in Java“

von Fritz Jobst

ISBN 978-3-446-41771-7

Weitere Informationen und Bestellungen unter
http://www.hanser-fachbuch.de/978-3-446-41771-7
sowie im Buchhandel

Vorwort zu den Android-Ergänzungen

Liebe Leserin, lieber Leser,

ist es nicht einfach schön, das eigene Smartphone programmieren zu können? Es gibt zwar viele Apps, denn Android gewinnt immer höhere Anteile am Markt für Smartphones. Dennoch fasziniert es viele Studentinnen und Studenten, Anwendungen für das eigene Gerät selbst zu schreiben. Mit dem Buch „Programmieren in Java“ erhalten Sie eine solide Grundlage in der Programmiersprache Java. Damit könnten Sie Apps für Android selbst entwickeln. Aber aller Anfang ist schwer, so auch der Start in Android. Deswegen bieten diese Ergänzungen eine Hilfe zum Einstieg an.

Kapitel 1 dieser Ergänzungen begleitet Sie auf dem Weg der Installation des Entwicklungssystems für Android. Google stellt das System zur Verfügung. Kapitel 1 erklärt die Erstellung von Apps bis zum Ablauf am eigenen Smartphone. Damit schaffen wir die erste App für Android.

Im Gegensatz zum PC benötigen wir am Smartphone bereits für einfachste Anwendungen Grundkenntnisse im Umgang mit grafischen Benutzungsoberflächen. *Kapitel 7* dieser Ergänzungen führt Sie in die Grundproblematik ein. Dabei müssen wir auch einige Besonderheiten bei Android-Smartphones beachten.

Meine Empfehlung: Arbeiten Sie die Kapitel 1, 2, 3 und 4 des Buches „Programmieren in Java“ durch. Danach können Sie über Kapitel 1 dieser Ergänzungen in die Entwicklung für Android einsteigen.

Zum Schluss ein Wort dazu, was diese Ergänzungen nicht sind: Apps für Android müssen nicht nur den Bildschirm adressieren, sondern auch die Sensoren, die Netzanbindung, Datenbanken usw. Diese Ergänzungen dienen also nur dem Einstieg in die Programmierung grafischer Benutzungsoberflächen für Android.

Viel Erfolg mit diesen Ergänzungen zum Buch!

Regensburg, im Juni 2013

Fritz Jobst

Inhalt der Android-Ergänzungen

1 Der Einstieg in Java für Android 1

1.1 Wie entwickelt man Apps für Android? 1
1.2 Installation der Entwicklungsumgebung für Android 3
1.3 Vorbereitungen der Entwicklungsumgebung 5
1.4 Erstellung und Ablauf des ersten Programms 8
1.5 Aufbau einer Applikation für Android 13
1.6 Aufgaben 16

7 Graphik-Anwendungen für Android 247

7.1 GUI-Anwendungen für Android 247
7.1.1 Prinzip der ereignisgesteuerten Programmierung 249
7.1.2 Statische Hierarchie von View-Klassen 253
7.1.3 Elementare Steuerelemente 254
7.1.4 Das Model-View-Controller-Paradigma und Android 258
7.1.5 Anordnung der Komponenten 260
7.2 Verwaltung von Activities durch Android 263
7.2.1 Lebenszyklus von Activities 263
7.2.2 Drehen des Smartphones 265
7.2.3 Wie sichern wir den Zustand einer Activity? 267
7.3 Kurs: GUI-Anwendungen 268
7.3.1 Erstellung einer Zeichnung 269
7.3.2 Reaktion auf Berührungen mit dem Finger 271
7.3.3 Reaktion auf Fingerbewegungen 273
7.3.4 Turtle-Graphik 275
7.3.5 Dialoge in Android 279
7.3.6 Die Türme von Hanoi 283
7.3.7 Auswahl aus einer Liste von Alternativen 288
7.4 Aufgaben 294

Aufstellung der Software-Projekte

Alle Projekte beginnen mit *PiJ* (Programmieren in Java). Die Projekte können Sie von der Begleitseite zum Buch „Programmieren in Java“ (ISBN 978-3-446-41771-7) beim Carl Hanser Verlag herunterladen:

http://downloads.hanser.de

Projekte der Begleitsoftware und Zuordnung zum Text:

Projekt	Kapitel	Inhalt
PiJHelloWorld	1	Einstieg in die Entwicklung mit Android.
PiJSchalter, PiJSchalter1	7	Reaktion auf Drücken einer Schaltfläche.
PiJSteuerelemente	7	Steuerelemente und die Reaktion auf ihre Betätigung.
PiJLayout	7	Sammlung von Layouts für Eingabe von z.B. Adresse.
PiJUebungenStart	7	Eine App mit leeren Activities für die Übungsaufgaben. Hier können Sie die Lösungen zu den Aufgaben eintragen.
PiJUebungenLoesungen	7	Lösungsvorschläge für einzelne Übungsaufgaben.
PiJKurs	7	Eine App mit allen Activities für den Kurs.
PiJListen	7	App mit Activities für die Auswahl aus Listen: ArrayAdapter.
PiJLifecycle	7	Eine App zur Demonstration des Lebenszyklus, Sichern des Zustands.

1

1 Der Einstieg in Java für Android

Diese Ergänzung zum Buch „Programmieren in Java“ will Ihnen dabei helfen, erste Programme für Smartphones der Android-Plattform von Google zu erstellen und diese Programme ablaufen zu lassen. Diese Ergänzungen folgen dem Aufbau des o. a. Buches: In einzelnen Kapiteln zeigen wir, wie Sie Apps für Android entwickeln können.

Hinweise

Diese Ergänzung zum Buch „Programmieren in Java“ erläutert nur die Besonderheiten zum Einstieg in Android. Sie benötigen das o. a. Buch, welches hier kurz als „das Buch“ bezeichnet wird. Smartphones für die Android-Plattform sind kurz als Smartphones bezeichnet.

Smartphones sind batteriebetriebene Geräte. Der Stromverbrauch ist eine kritische Größe, die man nie aus dem Auge verlieren darf. Außerdem sind die Größen des RAM und des Hintergrundspeichers sowie die Rechenleistung im Vergleich zu PCs eng beschränkt. Diese Restriktionen erfordern in Teilen eine gegenüber dem PC erheblich aufwändigere Programmierung. Meine Empfehlung: Starten Sie bei der Java-Programmierung mit dem Buch „Programmieren in Java“. Wenn Sie eine gewisse Sicherheit gewonnen haben, kann es viel Spaß machen, Apps für Android zu entwickeln. Dann können Sie diese Ergänzungen für Ihren persönlichen Start in die Welt der Androiden benützen.

1.1 Wie entwickelt man Apps für Android?

Eine App für Android ist eine Applikation für das System Android. Eine solche App läuft im Prinzip auf einem Smartphone für Android. Wir entwickeln die App aber nicht auf dem Smartphone, sondern genauso wie die anderen Java-Programme auf dem PC mit der integrierten Entwicklungsumgebung Eclipse. Dazu benötigen wir das Android SDK mit einem

Zusatzmodul für Eclipse, dem Plugin für die Entwicklung von Android[1]-Apps. Das Android SDK enthält mit dem AVD[2] einen Simulator, mit dessen Hilfe wir Apps auch auf dem PC laufen lassen können. Abbildung 1.1 zeigt die prinzipiellen Abläufe bei der Erstellung und dem Ablauf einer App unter Android. Eine App besteht nicht nur aus einem Programm, sondern enthält noch sog. Ressourcen wie Bilder, Texte oder Beschreibungen im `.xml`-Format. Außerdem muss die App alle verlangten Zugriffe, z. B. auf die Kontakte oder auf die SD-Karte des Smartphones benennen.

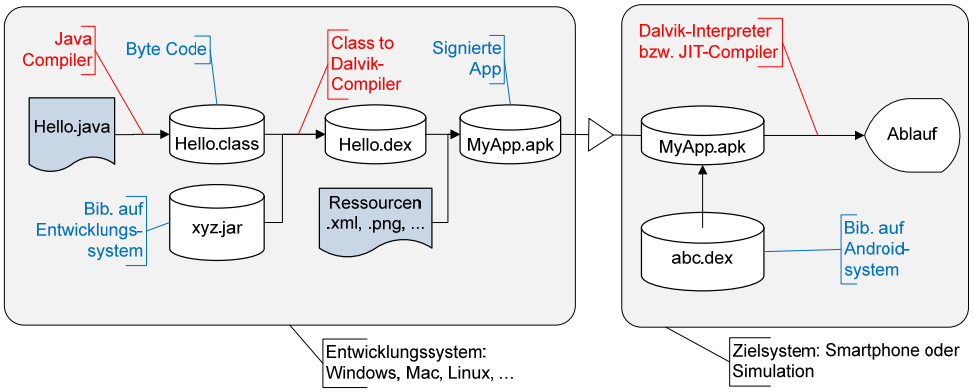


Abbildung 1.1: Entwicklung von Apps für Android

Der Java-Compiler übersetzt das Java-Programm für die App wie alle anderen Java-Programme in eine `.class`-Datei. Diese `.class`-Datei ist nicht direkt auf dem Smartphone ablauffähig, sondern wird von einem Compiler in das Dalvík Executable Format `.dex` umgewandelt. Dan Bornstein hat dieses Format erfunden, um besonders kurze und effiziente Programme zu erzeugen, damit das Smartphone Strom sparen kann. (Dalvík ist eine kleine Stadt in Island.) Die Entwicklungsumgebung schnürt nun aus dem Programm sowie seinen Ressourcen ein `.apk`-Paket und versieht es mit einer Prüfsumme, der sog. digitalen Signatur. Diese Signatur kann nicht verfälscht werden. Damit kann das Smartphone als Zielsystem die Prüfsumme neu berechnen und mit dem in der App angegebenen Wert vergleichen und so überprüfen, ob die App verändert wurde.

Von diesen Abläufen merkt der Benutzer der Eclipse-Entwicklungsumgebung nichts, man bringt seine App wie ein übliches Java-Programm zum Ablauf. Dazu kann man das Smartphone über ein USB-Kabel an den PC anschließen. Dann besorgt das Entwicklungssystem alle Schritte: Übersetzen, Packen, Signieren, auf das Smartphone kopieren, dort installieren und auch gleich zum Ablauf zu bringen. Wenn man kein Smartphone für Android hat, benützt man den Simulator.

[1] Die Schritte zur Installation finden Sie im nächsten Abschnitt.
[2] Android Virtual Device

1.2 Installation der Entwicklungsumgebung für Android

Zusätzlich zum Java Development Kit JDK benötigen Sie das SDK für Android. Dies enthält die Entwicklungsumgebung Eclipse, die Sie auch für die Entwicklung von Anwendungen für Java einsetzen können. Unter

http://developer.android.com/sdk/installing.html

sind alle Schritte beschrieben. Zur Installation empfiehlt es sich, in der unter

http://developer.android.com/sdk/index.html

beschriebenen Reihenfolge vorzugehen.

Tabelle 1.1: Werkzeuge zur Entwicklung von Apps für Android

Komponente	Download bzw. Beschreibung
JDK	Download. Empfehlung: 64-Bit-Version einsetzen http://www.oracle.com/technetwork/java/javase/downloads/index.html Java Development Kit für Java-Programmierung. In jedem Fall: Zuerst installieren.
Android SDK	Download. http://developer.android.com/sdk/index.html Android Software Development Kit für Android-Programmierung Neu: Android Studio als Entwicklungssystem. Erfordert 64-Bit-Version des JDK.
Android SDK	Konfiguration in Eclipse unter *Window/Android SDK Manager.* Wählen Sie die Version(en) von Android aus, für die Sie programmieren möchten.
AVD	Konfiguration in Eclipse unter *Window/Android Virtual Device Manager.* Verwalten Sie die virtuellen Devices für den Simulator.

Die Qual der Wahl der Versionen und Varianten der Komponenten

Bei Java empfiehlt sich schon zum Schutz Ihres Computers vor Viren und Trojanern die jeweils aktuelle Version. Bei Android-Smartphones gibt es diverse Versionen. Besonders weit ist die Version 2.3 verbreitet, die wesentlich kompletter als die Version 2.2 ausgestattet ist. Achten Sie auf die Android-Version auf Ihrem Smartphone und wählen Sie bei der Konfiguration des SDK für Android eine Version des API, die auf Ihrem Smartphone lauffähig ist. Abbildung 1.2 zeigt eine mögliche Installation des Android SDK für Android 4.2.2. Neben der SDK für die Plattform sollten Sie die Dokumentation sowie das Systemabbild „ARM EABI v7a System Image“ installieren. Der Simulator benötigt ein Systemabbild, um ein AVD zu erzeugen. Die Beispiele zum SDK sind hilfreich, wenn man einzelne Details studieren möchte. Die Online-Dokumentation in Android nimmt häufig Bezug auf diese Beispiele.

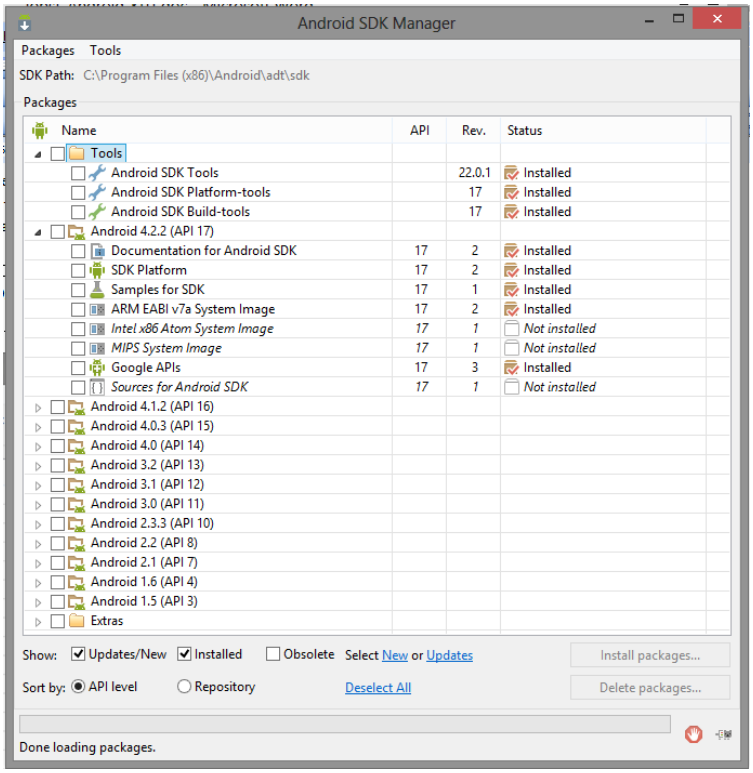


Abbildung 1.2: Der Android SDK Manager

Smartphone über USB anschließen und Apps laufen lassen

Wir erstellen die Programme mit Hilfe der Entwicklungsumgebung am PC und lassen sie auf dem Simulator oder einem Smartphone laufen. Für einen Einstieg in die Entwicklung für Android können wir unsere Programme auf einem AVD laufen lassen.

Zur Verbindung zwischen PC und Smartphone benützen wir ein Micro-USB-Kabel, wie es für Smartphones üblich ist. Unter Windows benötigen Sie einen für das Smartphone geeigneten USB-Treiber. Bei Smartphones der Firma Samsung ist u. U. die Installation des Dienstprogramms „Kies“ der Firma Samsung erforderlich. Bei der Installation von „Kies“ werden die benötigten Treiber für den USB mit installiert, siehe

http://developer.android.com/sdk/installing.html

Die Installation von Apps auf Smartphones ist ein sicherheitskritischer Schritt und deswegen nicht ohne eine ausdrückliche Erlaubnis möglich. Unter Android 4.1 muss man sie unter Einstellungen/Entwickleroptionen/USB-Debugging explizit erlauben, bei älteren Android-Versionen unter Einstellungen/Anwendungen/Entwicklung. Besonders gut versteckt ist der Entwicklermodus unter Android 4.2. Hier muss man unter „Einstellungen/Über das Telefon“ insgesamt 7 mal auf die Kernel-Version tippen, um die Entwickleroptionen freizuschalten.

1.3 Vorbereitungen der Entwicklungsumgebung

Nach dem Start fragt Eclipse nach dem Ordner für den Arbeitsbereich zum Ablegen aller erstellten Programme. Sie können den Ordner bei den eigenen Daten oder in einem beliebigen Verzeichnis anlegen. Wenn man bei zukünftigen Starts von Eclipse Fragen nach der Lage des Arbeitsbereichs vermeiden will, kann man das Häkchen bei der *Use this as default…*-Option anklicken.

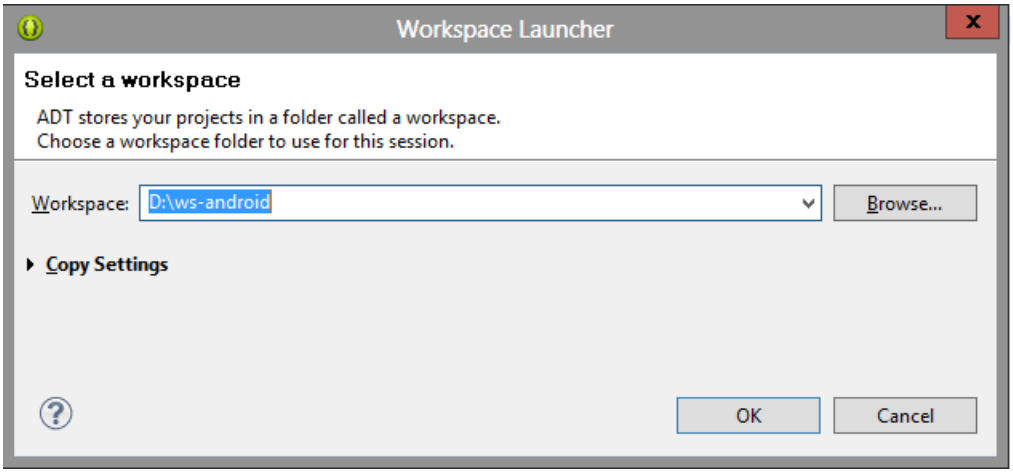


Abbildung 1.3: Auswahl eines Arbeitsbereichs

Danach meldet sich Eclipse mit seinem Willkommensbildschirm in Abbildung 1.4. Er enthält u. a. Hinweise zum Erstellen der ersten App. Sie können jetzt zum Arbeitsbereich gehen, indem Sie auf das mit J[3] gekennzeichnete Symbol am linken Rand des Bildschirms klicken.

Danach sehen Sie die Arbeitsoberfläche von Eclipse in Abbildung 1.5. Am linken Rand finden Sie den „Package Explorer“, mit dem Sie die später die einzelnen Projekte erkunden können.

[3] Java Perspektive

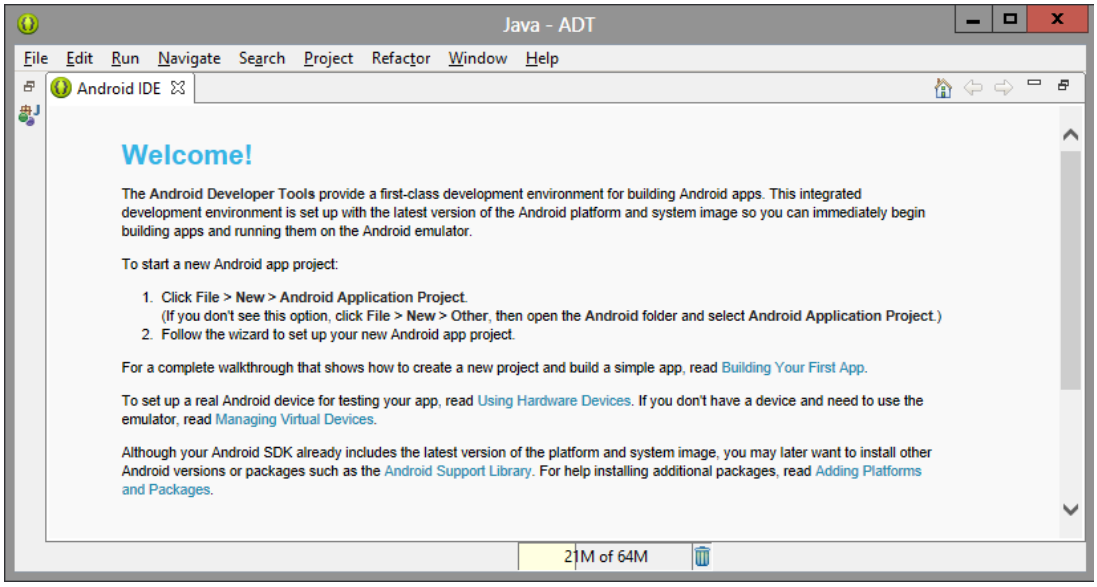


Abbildung 1.4: Begrüßungsbildschirm von Eclipse für Android

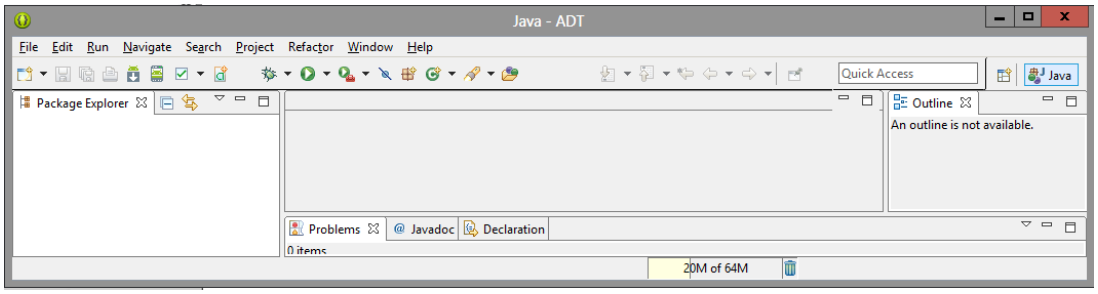


Abbildung 1.5: Arbeitsoberfläche von Eclipse

Auswahl für Android wichtiger Hilfsmittel in Eclipse

Abbildung 1.6 zeigt für die Entwicklung unter Android mögliche Views, die man aus der Eclipse-Entwicklungsumgebung unter *Window/Show View/Other...* aktivieren kann. Die View *Devices* zeigt die angeschlossenen Geräte. Drückt man in dieser View das Symbol für Kamera, so erhält man Bildschirmabdrucke vom über USB angeschlossenen Smartphone. Dies ist auch unabhängig vom Programmieren nützlich. Die View *LogCat* liefert eine Ansicht auf alle Systemmeldungen. Damit kann man das Protokoll des Programms auf dem Smartphone lesen. Hier finden wir auch die Ausgaben, die wir mit `System.out.println(…)` erzeugen. Dieses sog. *Logging* kann bei der Suche nach Fehlern sehr hilfreich sein. Mit der View *File Explorer* haben Sie Zugriff auf einen Teil des Dateisystems des Smartphones. Sie können auf das Dateisystem der SD-Karte zugreifen und Dateien vom PC auf das Smartphone und umgekehrt übertragen.

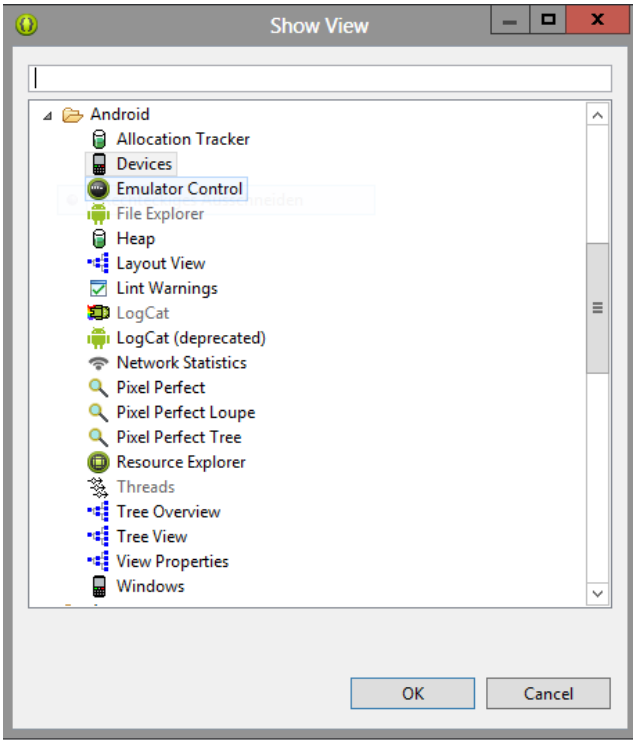


Abbildung 1.6: Neue Ansichten für Android auswählen: Devices, File Explorer, LogCat, …

Entwicklungssystem für Android über Eclipse verwalten

Wenn das ADT Plugin installiert ist, können Sie das Entwicklungssystem für Android unter *Window/Android SDK Manager* verwalten. Sie können Teile entfernen oder aktualisieren. Achten Sie darauf, dass Sie zumindest diejenigen Teile installieren, die Sie für einen Ablauf auf Ihrem Smartphone benötigen. (Siehe auch Abbildung 1.2.)

Unter *Window/Android Virtual Device Manager* verwalten Sie die sog. *Android Virtual Devices*. Hier legt man Simulatoren für die eigenen Apps an, wenn man sie nicht auf dem Smartphone laufen lassen will bzw. kann. Der Start eines sog. AVD kann relativ lange dauern, man braucht etwas Geduld, bis sich der Simulator zeigt. Deswegen sollte man den Simulator nicht nach jedem Programmlauf einer App beenden. Der Simulator bildet das reale Smartphone nicht vollständig ab, es fehlen diverse Sensoren, die Kamera usw. Für den Einstieg in die Programmierung mit Android reicht der Simulator aus.

1.4 Erstellung und Ablauf des ersten Programms

Zunächst müssen wir unter dem Menüpunkt *File/New/Android Project* ein neues Projekt für Android erzeugen. Das Projekt enthält die Java-Programme der App sowie Bilder und sonstige benötigte Ressourcen. Bei der Erstellung des Projekts hilft uns ein Assistent. Mit seiner Hilfe können wir in mehreren Schritten die erste App konfigurieren. Wir geben in Abbildung 1.7 einen Namen für das Projekt sowie das Package ein. Der Name des Packages kennzeichnet Ihre App für Android und muss eindeutig sein. Außerdem müssen wir die möglichen Plattformen für den Ablauf auswählen. Abbildung 1.8 zeigt den Bereich von Android 2.2 bis Android 4.2. Wenn Sie die Programme auf einem Smartphone ablaufen lassen wollen, muss die Version von Android auf dem Smartphone in diesem Bereich liegen.

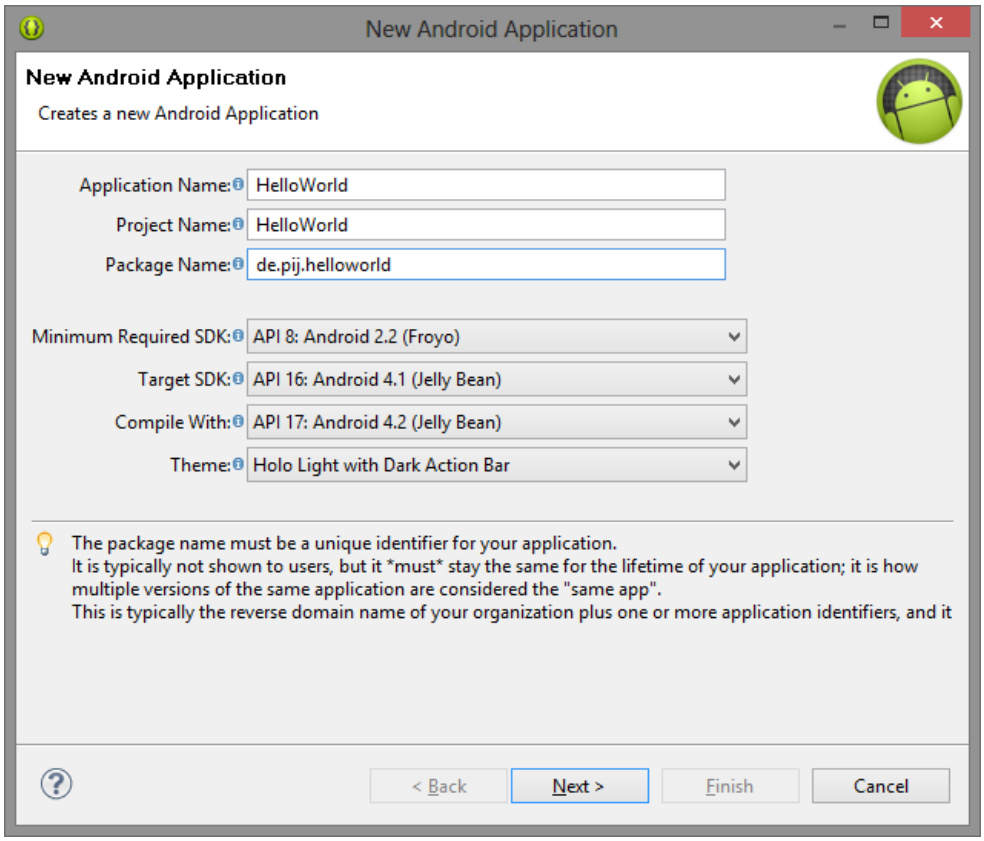


Abbildung 1.7: Der 1. Schritt beim Anlegen eines Projekts für Android

Mit *Next* erhalten wir den in Abbildung 1.8 angegebenen Dialog, bei dem wir auch einfach auf *Next* klicken können.

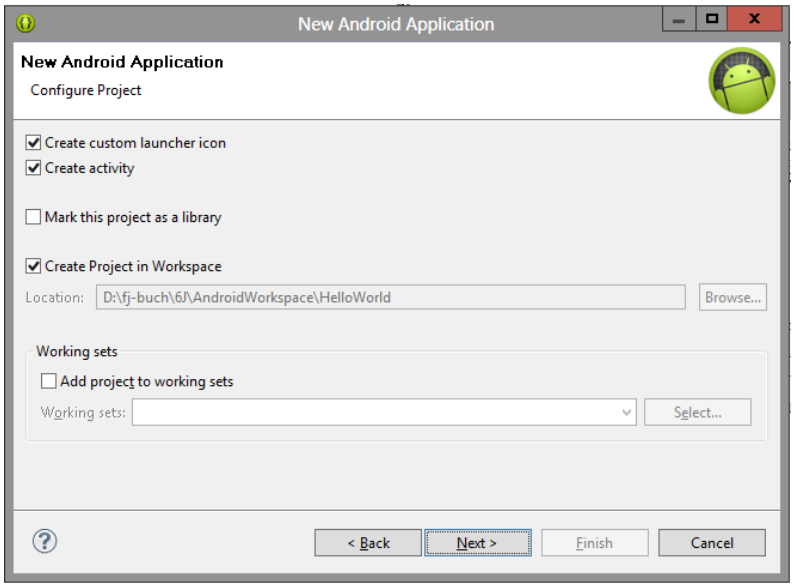


Abbildung 1.8: Der 2. Schritt beim Anlegen eines Projekts für Android: Auswahl einer Plattform

Nach Drücken von *Next* in Abbildung 1.8 kann man das Icon für die erste App auswählen (Abbildung 1.9).

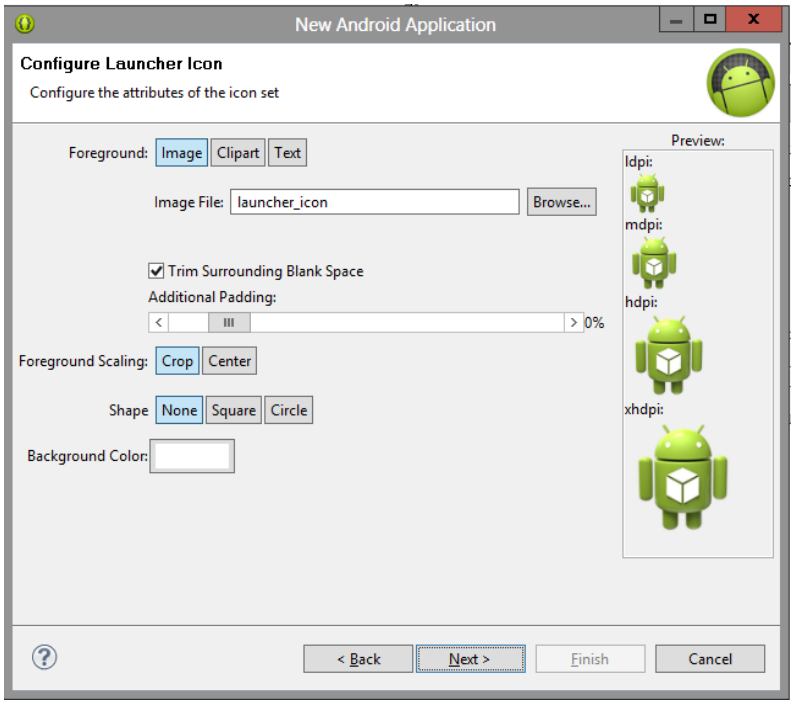


Abbildung 1.9: Der 3. Schritt beim Anlegen eines Projekts für Android

Nach Drücken von *Next* in Abbildung 1.9 können wir eine sog. Activity anlegen. Für den Start wählen wir eine möglichst einfache Anwendung, die sog. *BlankActivity*.

Nach Drücken von *Next* in Abbildung 1.10 erhalten wir einen weiteren Auswahldialog (Abbildung 1.11). Hier können wir den Namen der Java-Klasse der Activity angeben und auch den Namen für das Layout wählen.

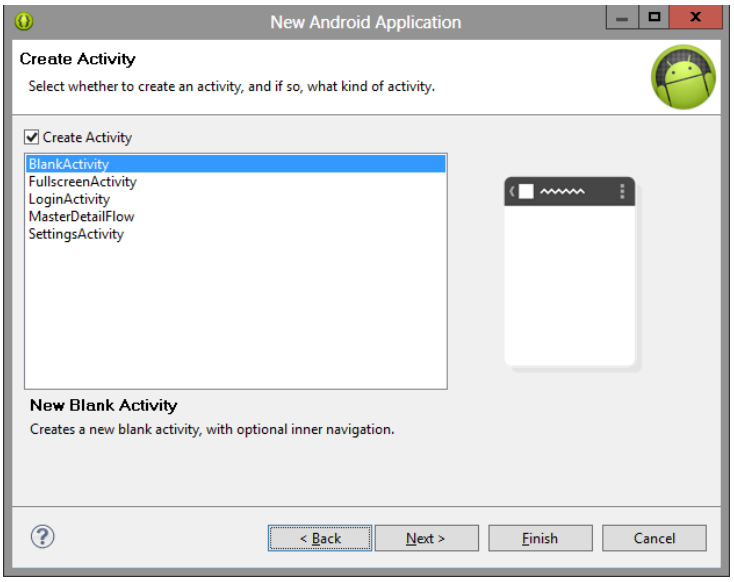


Abbildung 1.10: Der 4. Schritt beim Anlegen eines Projekts für Android

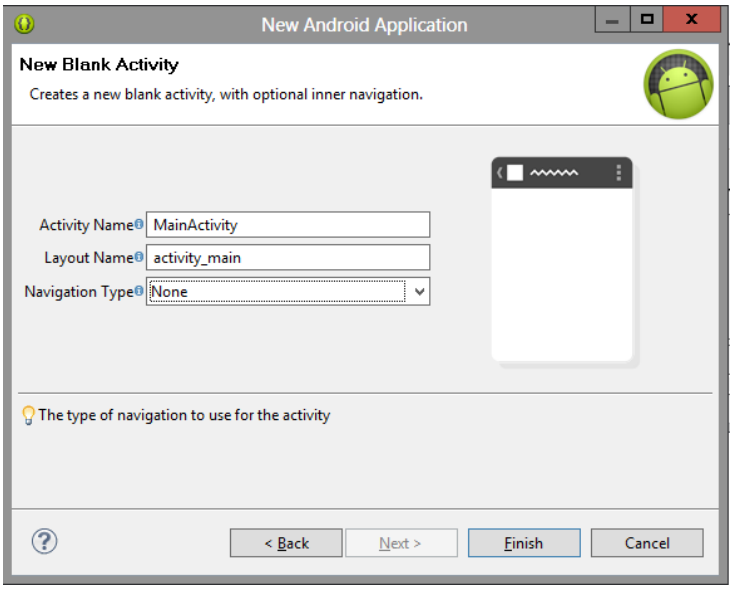


Abbildung 1.11: Der 5. Schritt beim Anlegen eines Projekts für Android

Nach Drücken von *Finish* erstellt der Assistent eine lauffähige App mit all ihren Bestandteilen (Abbildung 1.12).

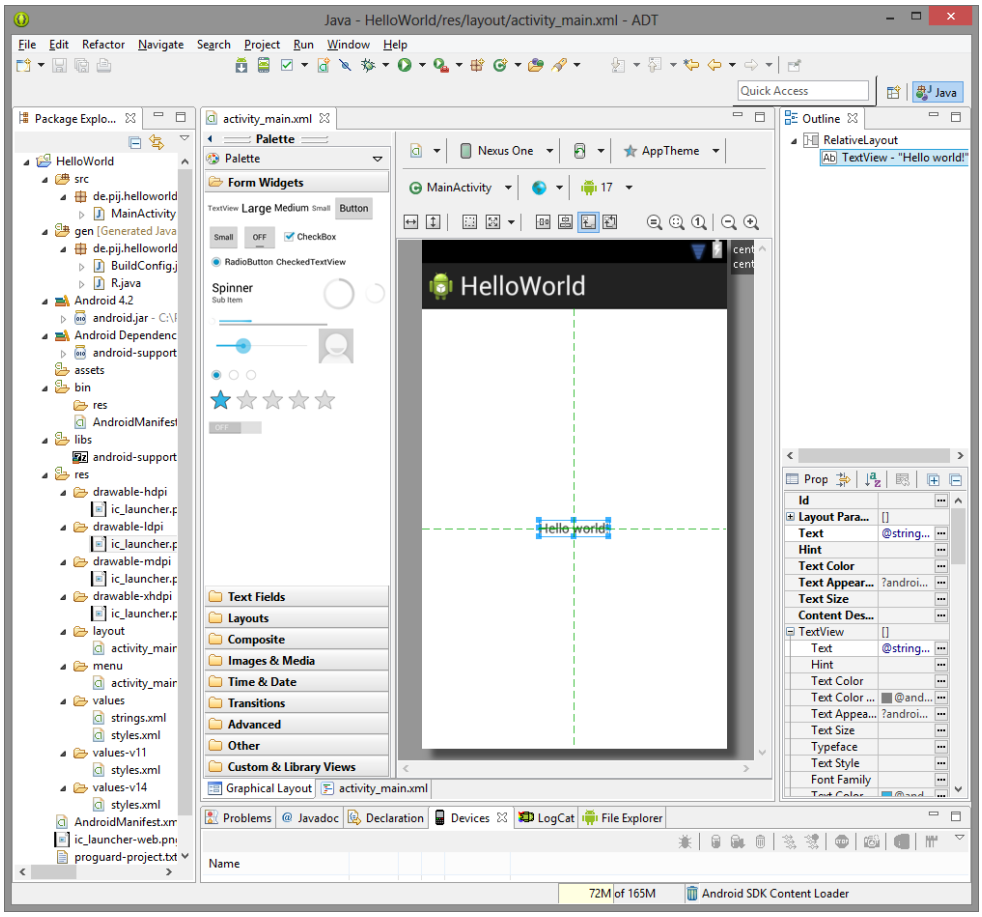


Abbildung 1.12: Die erste App unter Android

Unsere erste App enthält eine Activity `de.pij.helloworld.MainActivity` im Verzeichnis *src*. Die erste Activity spricht mit der Konstanten `R.layout.activity_main` das Layout für die App an. Im Verzeichnis *gen* finden wir die Datei zur Klasse R `de.pij.helloworld.R.java`. Diese Datei wird vom Entwicklungssystem für Android generiert und bei jeder Änderung der sog. Ressourcen neu erzeugt. Das Verzeichnis *res* enthält die Ressourcen:

- Bilder in verschiedener Auflösung in den Verzeichnissen `drawable-?.dpi` (?=l, m, h, xh, usw.),
- das Layout der Activity in der Datei `activity_main.xml`
- sowie die Datei `strings.xml` mit Definitionen von Zeichenketten.

Listing 1.1: Der Programmcode für die erste App unter Android

```
public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.activity_main, menu);
        return true;
    }

}
```

Eclipse übersetzt das Programm ohne besondere Aufforderung durch den Anwender. Nach einer Sicherung des Programms im Arbeitsbereich über das File-Menü oder mit der Tastenkombination *Strg+S* können Sie das Programm mit *Strg+F11* als „Android-Application“ laufen lassen und erhalten wie in Abbildung 1.13 die Ausgabe des ersten Programms.

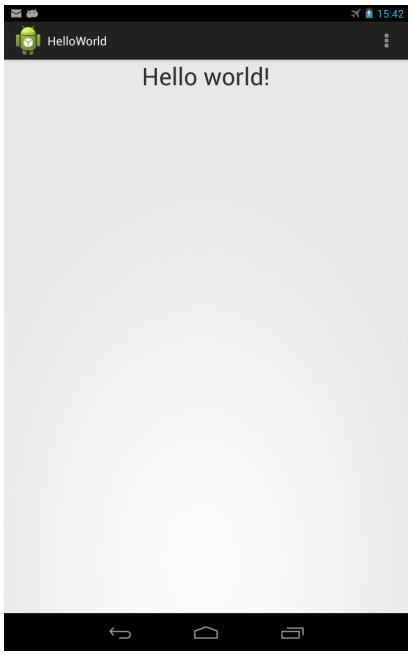


Abbildung 1.13: Unsere erste Android-App läuft!

1.5 Aufbau einer Applikation für Android

Listing 1.2: Vom Android-Assistenten erzeugte Verzeichnisse (vereinfacht)

```
+---assets
+---bin …
+---gen
|   \---de/pij/helloworld:        R.java
+---libs
+---res
|   +---drawable-hdpi             Bilder
|   +---drawable-ldpi
|   +---drawable-mdpi
|   +---drawable-xhdpi
|   +---layout:                   activity_main.xml
|   +---menu
|   +---values                    strings.xml
|   +---values-v11
|   \---values-v14
\---src
    \---de/pij/helloworld  MainActivity.java
AndroidManifest.xml
```

Die Applikation in Listing 1.1 für Android ist wie in Listing 1.2 beschrieben aufgebaut und besteht im Wesentlichen aus folgenden Teilen:

- Eine Activity: `MainActivity.java` (Listing 1.2)
- Die Datei mit den generierten Konstanten: `R.java` (Listing 1.3)
- Eine Datei mit der Beschreibung des Layouts: `activity_main.xml` (Listing 1.4)
- Eine Datei mit Werten: `strings.xml` (Listing 1.5)
- Die Datei mit den Einstellungen für die App: `AndroidManifest.xml` (Listing 1.6)

Unsere Activity ist von der Basisklasse `android.app.Activity` abgeleitet. Dabei überschreiben wir die Methode `onCreate (…)`. Diese Methode wird von Android aufgerufen, wenn das Exemplar der Klasse `MainActivity` erzeugt ist, um die Anzeige der App aufzubauen.

Wir rufen zunächst die entsprechende Methode der Basisklasse auf, die dann evtl. vorhandene Komponenten der Benutzungsoberfläche wiederherstellt. Danach setzen wir die `View` für das Fenster der Applikation. Die `View` ist die Basisklasse für alle Komponenten grafischer Benutzungsoberflächen unter Android, wobei wir an die Klassen `Component` bzw. `JComponent` von Swing (vgl. Kapitel 7) denken könnten. In Listing 1.1 wird die `View` aus einer Beschreibung in einer Datei im Ressourcen-Teil der Applikation aufgebaut.

Man bezeichnet `onCreate(…)` als sog. *Life Cycle Callback*. Jede Activity unter Android hat einen Lebenszyklus (siehe Kapitel 7 der Ergänzungen). `onCreate(…)` ist die Methode, die als Erste aufgerufen wird. Hier initialisieren wir unsere Variablen, die Benutzungsoberfläche usw.

Listing 1.3: Teile der generierten Datei `R.java`

```
/* AUTO-GENERATED FILE.  DO NOT MODIFY… */

package de.pij.helloworld;
public final class R {

  public static final class drawable {
    public static final int ic_launcher = 0x7f020000;
  }

  public static final class layout {
    public static final int activity_main = 0x7f030000;
  }

  public static final class string {
    public static final int app_name = 0x7f040000;
    public static final int hello_world = 0x7f040001;
  }

}
```

Diese Datei enthält Konstanten, die wir im Programm verwenden können. Diese Konstanten werden von einem Programm der Entwicklungsumgebung automatisch aus den Ressourcen erzeugt. In der ersten Activity sprechen wir das Layout über die Konstante `R.layout.activity_main` an. Sie sehen auch die Konstanten `R.string.hello_world` bzw. `R.string.app_name` für Texte. Wenn wir alle Texte in der Datei `strings.xml` ablegen, können wir die Internationalisierung von Android erhalten, wenn wir zusätzlich die Texte in den jeweiligen Sprachen angeben. Sind die Texte dagegen im Programm „hart“ eincodiert, müssen alle entsprechenden Programmteile im Quellcode geändert werden.

Listing 1.4: Das Layout: activity_main.xml

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity" >

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_centerVertical="true"
        android:text="@string/hello_world" />
</RelativeLayout
```

Das `RelativeLayout` kann alle seine Komponenten relativ zueinander anordnen, siehe auch Kapitel 7. Die `TextView` dient der Darstellung von Texten. Sie ist eine `View` und belegt im Bereich der Eltern-Komponente (hier das relative Layout) nur so wenig Platz wie nötig, um den Inhalt darzustellen (`layout_width="wrap_content"`). Die Komponente ist horizontal und vertikal zentriert angeordnet(siehe auch Abbildung 1.13). Den Text in diesem Textelement sprechen wir über den Bezeichner `@string/hello_world` an.

Listing 1.5: Die Datei mit den Zeichenketten: strings.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="hello_world">Hello World, HelloWorldActivity!</string>
  <string name="app_name">HelloWorld</string>
  <string name="menu_settings">Settings</string>
</resources>
```

In dieser Datei fassen wir die Texte im Programm zusammen, wenn wir uns eine einfache Internationalisierung ermöglichen wollen. Im Programm könnten wir einen Text wie folgt ausgeben. *LogCat* zeigt uns diesen Text für die Applikation `de.pij.helloworld` an.

```
System.out.println(getResources().getString(R.string.app_name));
```

Listing 1.6: Die Datei mit den Einstellungen für die App: AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android=
       "http://schemas.android.com/apk/res/android"
  package="de.pij.helloworld"
  android:versionCode="1"
  android:versionName="1.0" >

  <uses-sdk
  android:minSdkVersion="8"
  android:targetSdkVersion="16" />

  <application
  android:allowBackup="true"
  android:icon="@drawable/ic_launcher"
  android:label="@string/app_name"
  android:theme="@style/AppTheme" >
  <activity
    android:name="de.pij.helloworld.MainActivity"
    android:label="@string/app_name" >
    <intent-filter>
    <action android:name="android.intent.action.MAIN"/>
    <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
  </activity>
  </application>

</manifest>
```

Die Datei `AndroidManifest.xml` enthält die zentralen Einstellungen für die App. Dort legen wir das Package, die Version sowie die benötigte Version von Android fest. Als einzige Änderung gegenüber der von Android generierten Variante sehen Sie das Android-Thema „Light“: Der Hintergrund ist standardmäßig dunkel bei hellem Vordergrund. Mit der „Light“-Einstellung erscheint der Hintergrund hell und der Vordergrund dunkel. Sie könnten auch ein eigenes Icon erstellen, das Sie dann für die App festlegen.

In dieser Datei führen wir alle von der App angeforderten Rechte auf. Wenn der Anwender bei der Installation der App die angeforderten Rechte wie Zugriff auf das Netzwerk usw. einräumt, dann – und nur dann – darf die App auf das Netzwerk zugreifen.

Jede Activity der App müssen wir in dieser Datei aufführen. Mehr dazu in Kapitel7.

1.6 Aufgaben

Aufgabe 1.1

Legen Sie ein Projekt für Android an und lassen Sie die App unverändert laufen. Dann ändern Sie den Text für die Ausgabe ab und lassen die App wieder laufen. Ändern Sie die Datei `AndroidManifest.xml` wie in Listing 1.6 beschrieben ab: Setzen Sie das Light-Thema.

Aufgabe 1.2

Fügen Sie den nach Listing 1.5 angegebenen Befehl zur Ausgabe in die Methode `onCreate(…)` ein. Was beobachten Sie in der Ausgabe von `LogCat`, wenn Sie die App auf Ihrem Smartphone laufen lassen und dieses um 90° drehen?

7

7 Graphik-Anwendungen für Android

In diesem Kapitel lernen Sie, grafische Benutzungsoberflächen von Apps für Android zu erstellen: Steuerelemente wie z. B. Schalter, Textfelder und deren Anordnung, Bedienung durch Berühren und Programmierung elementarer Grafiken. Man spricht auch von *Graphical User Interfaces* und verwendet die Abkürzung *GUI*, die wir in diesem Kapitel durchgängig benützen. In Kapitel 1 haben wir eine erste App für Android erstellt. Hier wollen wir tiefer in die Programmierung von Benutzungsoberflächen für Apps einsteigen. Deswegen müssen wir uns mit Grundlagen beschäftigen:

- Wie ist eine grafische Benutzungsoberfläche prinzipiell aufgebaut?
- Welche Steuerelemente (sog. *Views*) gibt es für die Programmierung des Bildschirms?
- Wie kann man diese Bestandteile auf dem Bildschirm anordnen?
- Wie reagiert ein Programm auf Aktionen des Benutzers?
- Wie kann man Daten im Programm über die GUI bearbeiten?
- Was passiert, wenn wir das Smartphone drehen?

Wir eignen uns diese Grundlagen mit beispielhaften Programmen an. Dabei lernen wir Schritt für Schritt die jeweils benötigten Grundlagen zu Android bzw. zur Theorie kennen. Wenn wir die Grundlagen kennen, vertiefen wir unsere Fertigkeiten durch Arbeit an Projekten. Schließlich erproben wir unsere neu gewonnenen Fertigkeiten an Übungsaufgaben.

Empfehlung
Starten Sie mit dem Kapitel 1 der Ergänzungen zu „Programmieren in Java“.
Erstellen Sie die erste App, dann fahren Sie mit diesem Kapitel 7 fort.

7.1 GUI-Anwendungen für Android

Bei Android unterscheidet man zwischen sog. *Activities* und *Services*. Services laufen im Hintergrund, Activities haben Zugriff auf den Bildschirm. Eine Activity ist im Sinne der Klassenhierarchie in Android ein Ausführungskontext (siehe Abbildung 7.1) mit einer sog.

View zur Interaktion mit dem Anwender. Eine solche View kann wie in Abbildung 7.2 mehrere Steuerelemente enthalten. Wir besprechen hier nur Activities.

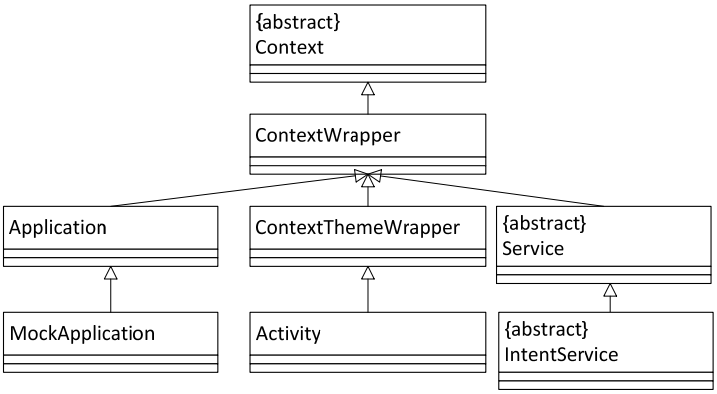


Abbildung 7.1: Android: Activities, Services und Applications

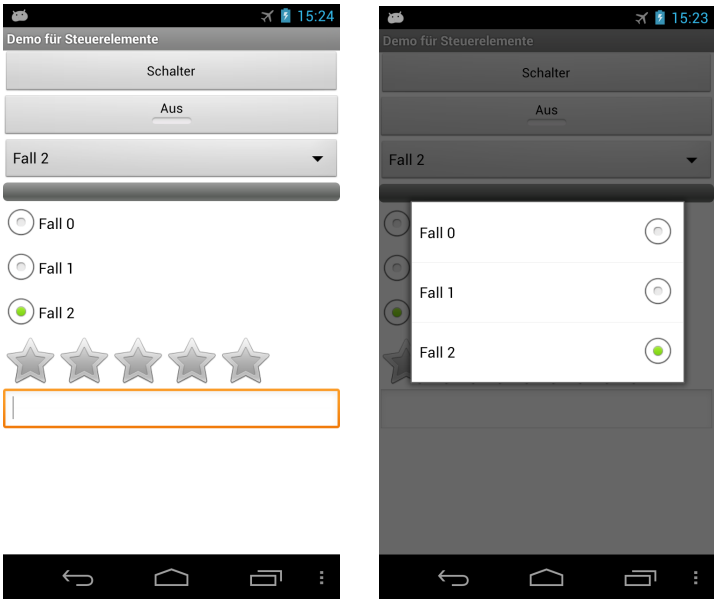


Abbildung 7.2: Eine App zeigt einige Steuerelemente unter Android

Abbildung 7.3 stellt die Hierarchie der Views aus Abbildung 7.2 dar. Die „oberste“ View ist eine `ViewGroup`, sie enthält alle Views, wie den Schalter, den Kippschalter usw. Die einzelnen Views sind am Bildschirm nacheinander angeordnet. Die Regeln zur Platzierung bezeichnet man auch als *Layout*. In Listing 7.6 finden Sie die zugehörige Definition des Layouts. Die `RadioGroup` fasst die `RadioButtons` zusammen. Das zugehörige Programm finden Sie unter Listing 7.7.

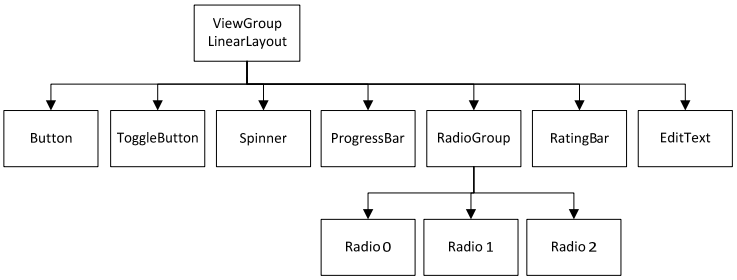


Abbildung 7.3: Dynamische Hierarchie der Views für den Fall der App aus Abbildung 7.2

7.1.1 Prinzip der ereignisgesteuerten Programmierung

Programme für Konsolen enthalten die `main()`-Methode, bei der der Ablauf des Programms beginnt. Die Abläufe folgen dann im Prinzip einem „EVA"-Schema (*Eingabe-Verarbeitung-Ausgabe)* und stellen häufig einen Zyklus aus Schreib- bzw. Lesevorgängen dar. Bei der Eingabe wartet das Programm an einer bestimmten Stelle.

Wie soll man mit dieser Strategie eine Situation beherrschen, bei der es wie in Abbildung 7.4 mehrere Quellen für Ereignisse gibt? Schon die Integration von zwei Schaltern würde nicht in dieses einfache Konzept passen: Wir warten an dem einen Schalter, während der andere für uns unbeobachtbar gedrückt wird. Das EVA-Schema der Verarbeitung lässt sich nicht auf Anwendungen für graphische Benutzungsoberflächen übertragen, denn dort gibt es viele Quellen für Ereignisse, auf die das Programm reagieren soll. Diese Ereignisse kommen von sog. Steuerelementen wie dem Schalter im Programm von Listing 7.3, aber auch aus diversen anderen Quellen wie z. B. Wischbewegungen oder von externen Nachrichten.

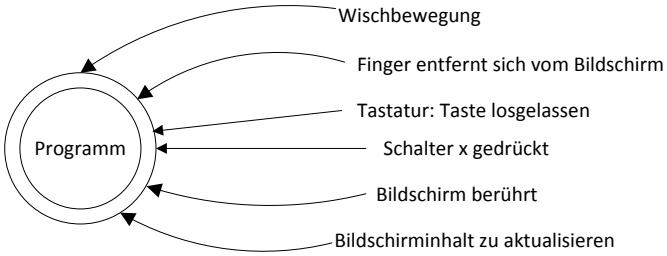


Abbildung 7.4: Programm und Ereignisse

Die ereignisgesteuerte Programmierung verwaltet alle Ereignisse in einer Warteschlange von Nachrichten wie in Abbildung 7.5. Android fügt die Nachrichten in diese Warteschlangen ein. Der für die Benutzungsoberfläche zuständige Thread, der bei Android sog. `main`-Thread, zieht eine Nachricht heraus und sorgt für den Aufruf einer bestimmten Methode. Damit sind wir mit Hilfe der Warteschlange der Nachrichten bei einem Schema *Ereignis-Reaktion* gelandet.

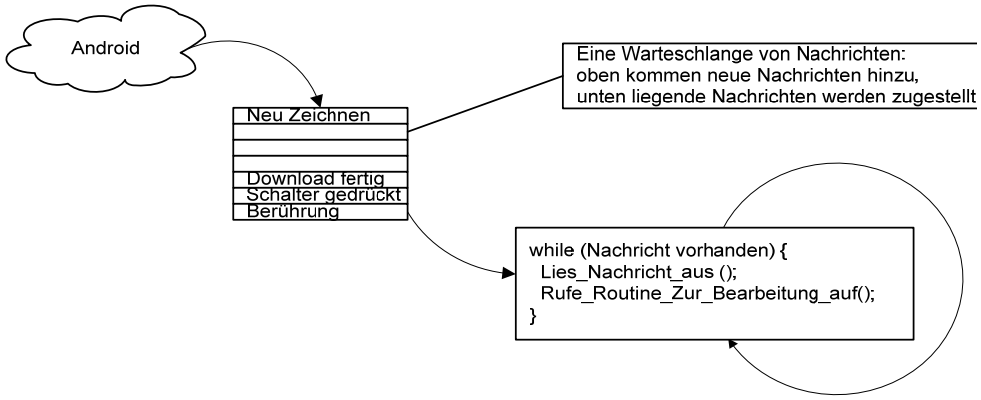


Abbildung 7.5: Prinzip der ereignisgesteuerten Programmierung

Wie können wir Android dazu veranlassen, die Methode unserer Wahl bei einem bestimmten Ereignis aufzurufen? Im folgenden Beispiel betrachten wir ein besonders einfaches Verfahren.

Eine App reagiert auf das Drücken eines Schalters

Beim Drücken eines Schalters zeigt die App den Namen des Schalters sowie das aktuelle Datum in Form eines sog. Toast[1] auf dem Bildschirm an. Wir könnten auch einen Text mit `System.out.println()` ausgeben, aber dieser Text würde nur im Protokoll der Testhilfe `LogCat` erscheinen. Der Toast dagegen erscheint am Bildschirm und erlischt nach einiger Zeit wieder. Außerdem erhält er keinen Fokus und stört somit keine der Eingaben. Die App `PiJSchalter` besteht aus mehreren Teilen. Listing 7.1 definiert die verwendeten Texte. Listing 7.2 beschreibt das Layout der `View` unserer Activity und Listing 7.3 enthält den Programmcode.

In Listing 7.2 definieren wir ein `LinearLayout`. Ein solches Layout ist eine `View` bzw. sogar eine `ViewGroup` (siehe Abbildung 7.3 und Abbildung 7.7) und reiht die darin enthaltenen Views nacheinander auf, wie es in Abbildung 7.3 für das Beispiel der Steuerelemente aus Abbildung 7.2 gezeigt ist. Das Layout in Listing 7.2 ist ein `LinearLayout`. Es enthält nur eine einzige `View`, nämlich einen `Button` mit der symbolischen Bezeichnung `id = @id/button1`. Das Präfix `@id/` vor dem Namen des Schalters sorgt für eine eindeutige Nummerierung der definierten Elemente. Im Java-Programm kann man über den Namen `R.id.button1` auf diesen Schalter zugreifen.

Der Schalter ist so breit wie seine „elterliche“ View (`layout_width="match_parent"`) und so hoch, dass sein Inhalt hineinpasst (`layout_height="wrap_content"`). Achten Sie in Listing 7.2 besonders auf die umrahmte Definition `onClick= "onClickButton01"`. Wenn man auf die Schaltfläche drückt, ruft das Laufzeitsystem

[1] Trinkspruch

von Android eine Methode dieses Namens mit einer `View` als Parameter in derjenigen Activity auf, die als Host fungiert. Diese Methode ist hier `onClickButton01(View v)` der `MainActivity` in Listing 7.3[2].

Beachten Sie, dass der Text `answer_text` in Listing 7.1 zwei Stellen zum Einfügen von Text enthält: `%1$...` sowie `%2$...` In diese Stellen fügt Android den Namen des Schalters sowie das Datum als Text ein, der in der in Listing 7.3 mit (*) gekennzeichneten Stelle übergeben wird. Die auf diese Weise formatierte Ausgabe in Java finden Sie in Kap. 4.

Listing 7.1: Die Datei res/values/strings.xml

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

  <string name="app_name">PiJSchalter</string>
  <string name="button_text">Hier drücken!</string>
  <string name="menu_settings">Settings</string>
  <string name="answer_text">Der Schalter %1$s wurde gedrückt am:
                %2$tY-%2$tm-%2$td</string>

</resources>
```

Listing 7.2: Die Datei res/layout/activity_main.xml

```
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  tools:context=".MainActivity" >

  <Button
    android:id="@+id/button1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:onClick="onClickButton01"
    android:text="@string/button_text" />

</LinearLayout>
```

Ressourcen

Die App beinhaltet Texte, Beschreibungen für Layouts, Bilder usw. Diese sog. Ressourcen definiert man im Unterverzeichnis `res`, das wir in den Ergänzungen zu Kapitel 1 näher beschreiben.

Listing 7.3: Die Datei src/de.pij.schalter/MainActivity

```
public class MainActivity extends Activity {

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    // Setze die View der Activity: aus der Layout-Datei
    setContentView(R.layout.activity_main);
```

[2] Dazu benützt Android die Reflection-Technik, siehe Kap. 4

```
    // So könnte man den Schalter finden:
    Button b = (Button)findViewById (R.id.button1); (**)
  }

  public void onClickButton01 (View v) {
    final CharSequence text = ((TextView) v).getText();
    String antwort = getString(R.string.answer_text,
       text, Calendar.getInstance()); // (*)
    final Toast toast = Toast.makeText(this, antwort, Toast.LENGTH_LONG);
    toast.show();
  }
}
```

Wie reagiert man objektorientiert auf Ereignisse?

In Listing 7.3 haben wir eine Methode zur Reaktion benützt, deren Name in der Layoutdatei beim entsprechenden Schalter angegeben war. Dieser bequeme Weg führt leider nicht immer zum Ziel, wir benötigen ein allgemeines Schema. Quellen (= *Source*) erzeugen Ereignisse (= *Events*), Beobachter (= *Listener*) registrieren sich bei den Quellen, um sich über Ereignisse informieren zu lassen. Man kann dies mit dem Abonnieren einer Zeitung vergleichen. Die Produzenten liefern Zeitungen (= Ereignisse) nur an die Abonnenten. Damit folgt die Ereignissteuerung in Java dem in [GHJV96] beschriebenen Beobachter-Muster.

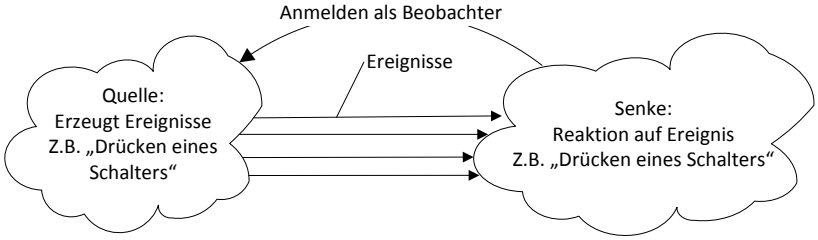


Abbildung 7.6: Beobachter-Muster

In Listing 7.4 reagieren wir auf der Ebene der Activity auf das Drücken des Schalters. Dazu muss die Klasse `MainActivity` die `OnClickListener`-Schnittstelle implementieren, indem die Methode `onClick` überschrieben wird.

Bei Apps mit mehreren Schaltern müsste man die Quelle des Ereignisses ermitteln. In Listing 7.5 zeigt eine Lösung, die diesen Aufwand vermeidet und in vielen Fällen einsetzbar ist. Dazu verwenden wir eine anonyme innere Klasse zur Reaktion. Wir erstellen ein Objekt einer Klasse, welche die `OnClickListener`-Schnittstelle implementiert. Diese Klasse hat nur die einzige Aufgabe, auf das Klick-Ereignis zu reagieren. Würde man eine externe Klasse schreiben, so könnte diese nicht auf die internen Daten der `MainActivity` zugreifen. Deswegen wählen wir eine innere Klasse. Da es genau einen Einsatzfall für diese Klasse gibt, bietet sich eine anonyme innere Klasse an.

Beachten Sie bitte, dass wir als Ausführungskontext an den Toast als Programmkontext nicht die **`this`**-Referenz der inneren anonymen Klasse übergeben können, sondern die **`this`**-Referenz der umschließenden Klasse `MainActivity` übergeben müssen, die indi-

rekt von `Context` abgeleitet ist. Zum Thema „innere Klassen“ siehe Abschnitt 3.2.5, zum Thema Kontext vgl. Abbildung 7.1.

Listing 7.4: Wir reagieren in der MainActivity auf das Drücken eines Schalters

```
public class MainActivity extends Activity implements OnClickListener {

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Button b = (Button) findViewById(R.id.button1);
    b.setOnClickListener(this);
  }

  @Override
  public void onClick(View v) {
    final CharSequence text = ((TextView) v).getText();
    String antwort = getString(R.string.answer_text, text,
          Calendar.getInstance());
    final Toast toast = Toast.makeText(this, antwort, Toast.LENGTH_LONG);
    toast.show();
  }
}
```

Listing 7.5: Variante mit inneren Klassen zur Reaktion auf das Drücken eines Schalters

```
public class MainActivity extends Activity {

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Button b = (Button) findViewById(R.id.button1);

    b.setOnClickListener(new OnClickListener() {
      @Override
      public void onClick(View v) {
        final CharSequence text = ((TextView) v).getText();
        String antwort = getString(R.string.answer_text, text,
             Calendar.getInstance());
        final Toast toast = Toast.makeText(MainActivity.this,
             antwort, Toast.LENGTH_LONG);
        toast.show();
      }

    });
  }

}
```

7.1.2 Statische Hierarchie von View-Klassen

Alle Komponenten graphischer Anwendungen in Android sind von `View` abgeleitete Klassen. Sie verfügen damit über Attribute und Methoden dieser Klasse: wichtige Methoden der Ereignissteuerung, die Attribute einer GUI-Welt, wie etwa Zeichensatz, Farben für Vordergrund und Hintergrund usw. Die Klasse `ViewGroup` ist von `View` abgeleitet und dient als Basisklasse für Views, die ihrerseits Views enthalten. Dies sind die …`Layout`-Klassen sowie die Klasse `AdapterView` zur Anbindung von Daten (=Model) an Views. Abbildung 7.7 zeigt diese Vererbungsbeziehung für ausgewählte Klassen.

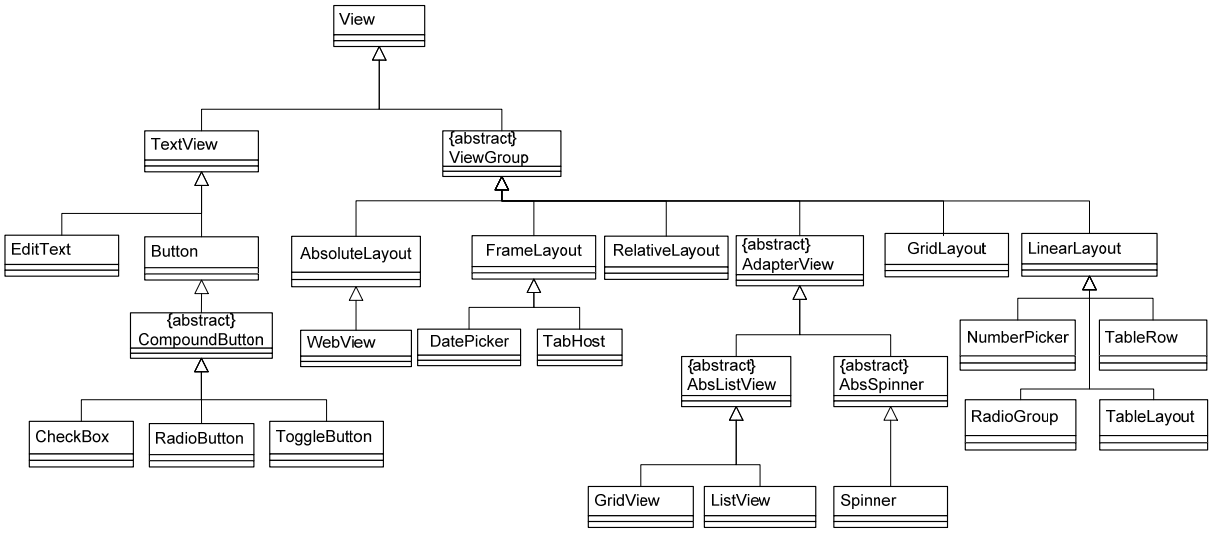


Abbildung 7.7: Klassenhierarchie einiger Views unter Android

Bei näherer Betrachtung dieser Klassen erkennen wir verschiedene Gruppen.

- Elementare Steuerelemente
  - `TextView`, `EditText`
  - `Button`, `CheckBox`, `RadioButton`, `ToggleButton`
  - `DatePicker`, `NumberPicker`…
- Layout-Klassen zur Anordnung ihrer Mitglieder (*children*)
  - `FrameLayout`, `RelativeLayout`, `GridLayout`
  - `LinearLayout`, `TableLayout`, `RadioGroup`, `AbsoluteLayout`
- Views zur Anzeige von Daten „AdapterViews“: siehe Abschnitt 7.3.7
  - `GridView`, `ListView`
  - `Spinner`
- Spezielle Views
  - `WebView`, `TableRow`
  - `TabHost`

7.1.3 Elementare Steuerelemente

Tabelle 7.1 enthält Steuerelemente aus dem Package `android.widget`. Abbildung 7.2 zeigt einige dieser Elemente in einer Anwendung. Mehr zu diesem Thema finden Sie unter

http://developer.android.com/guide/topics/ui/controls.html.

Tabelle 7.1: **Android: elementare Steuerelemente für grafische Benutzungsoberflächen**

Steuerelement	Beschreibung
`View`	Zeichenfläche für Programme. Basisklasse aller speziellen „Ansichten“
`ViewGroup`	Spezielle View als abstrakte Basisklasse, die andere Views enthalten kann: sog. *children*
`CheckBox`	Checkbox
`Button`	Schalter löst bei Drücken eine Aktion aus
`RadioButton`	Mehrere Schalter zur Auswahl einer Option im Stile der alten Analogradios
`ToggleButton`	Kippschalter
`RatingBar`	Schalterleiste zur Abgabe einer Bewertung
`TextView`	Textfeld, zur Anzeige von Text konfiguriert
`EditText`	Textfeld, zum Editieren von Text konfiguriert
`ProgressBar`	Fortschrittsanzeige: linear oder kreisförmig

Beispiel: Einsatz diverser Steuerelemente

Das Beispielprogramm aus Abbildung 7.2 soll die Interaktion eines Programms unter Android mit den elementaren Steuerelementen zeigen. Abbildung 7.3 zeigt die hierarchische Abhängigkeit der Views, Listing 7.6 die statischen Deklarationen für die Steuerelemente. In Listing 7.7 ermitteln wir in der Methode `findUIComponents()` die Bezüge zu den Steuerelementen der GUI. In der Methode `registerListeners()` registrieren wir Beobachter für Ereignisse der Steuerelemente. Wir reagieren auf Drücken der Schalter, Änderungen der Bewertung oder die Auswahl von Optionen.

Listing 7.6: Layout-Datei für die Steuerelemente

```
<?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" >

    <Button
        android:id="@+id/button"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/button_text" />

    <ToggleButton
        android:id="@+id/toggleButton"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textOff="Aus"
        android:textOn="An" />

    <Spinner
        android:id="@+id/spinner"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
```

```
            android:entries="@array/optionen" />

        <ProgressBar
            android:id="@+id/progressBar"
            style="?android:attr/progressBarStyleHorizontal"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />

        <RadioGroup
            android:id="@+id/radioGroup"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" >

            <RadioButton
                android:id="@+id/radio0"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:checked="true"
                android:text="@string/radio0" />

            <RadioButton
                android:id="@+id/radio1"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="@string/radio1" />

            <RadioButton
                android:id="@+id/radio2"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="@string/radio2" />
        </RadioGroup>

        <RatingBar
            android:id="@+id/ratingBar"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />

        <EditText
            android:id="@+id/editText"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:inputType="text" >

            <requestFocus />
        </EditText>

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

    </LinearLayout>
```

Listing 7.7: Reaktion eines Programms auf Eingaben von Benutzern

```
public class MainActivity extends Activity {
  private static final String TAG = MainActivity.class.getSimpleName();

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    findUIComponents();
    registerListeners();
  }

  private Button button = null;
```

```
private ToggleButton toggleButton = null;
private Spinner spinner = null;
private ProgressBar progressBar = null;
private RadioGroup radioGroup = null;
private RatingBar ratingBar = null;
private EditText editText = null;
private TextView textView = null;

// UI Komponenten nur einmal suchen und notieren
private void findUIComponents() {
  button = (Button) findViewById(R.id.button);
  toggleButton = (ToggleButton) findViewById(R.id.toggleButton);
  spinner = (Spinner) findViewById(R.id.spinner);
  progressBar = (ProgressBar) findViewById(R.id.progressBar);
  radioGroup = (RadioGroup) findViewById(R.id.radioGroup);
  ratingBar = (RatingBar) findViewById(R.id.ratingBar);
  editText = (EditText)findViewById (R.id.editText);
  textView = (TextView)findViewById (R.id.textView);
}

// Alle Listener registrieren
private void registerListeners() {
  // Reaktion auf Drücken des Schalters
  // Auslesen der aktuellen Stände anderer Steuerelemente
  button.setOnClickListener(new OnClickListener() {

    public void onClick(View v) {
      Log.i(TAG, "onClick ");
      // Radiobuttons 0, 1, 2 usw.
      // Umrechnen der id für UI
      int checkedRadioButtonId = radioGroup.getCheckedRadioButtonId();
      checkedRadioButtonId -= R.id.radio0;
      Log.i(TAG, "getCheckedRadioButtonId = " + checkedRadioButtonId);

      // Spinner auslesen
      Log.i(TAG, "spinner: " + spinner.getSelectedItemPosition());

      Log.i(TAG, "toggleButton: " + toggleButton.isChecked());

      Log.i(TAG, "ratingBar: " + ratingBar.getRating());

      Log.i(TAG, "text=" + editText.getText().toString());
      textView.setText(textView.getText()+
         "\n"+editText.getText().toString());
    }
  });


  // Rating Bar beobachten und Fortschrittsanzeige schalten
  ratingBar.setOnRatingBarChangeListener(
    new OnRatingBarChangeListener() {
      public void onRatingChanged(RatingBar ratingBar, float rating,
          boolean fromUser) {
      Toast.makeText(MainActivity.this, "Neues Rating: " + rating,
          Toast.LENGTH_SHORT).show();
      progressBar.setProgress((int)(rating*20.0f));
    }
  });

  // Reaktion auf den Kippschalter
  toggleButton.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
      if (toggleButton.isChecked())
        Toast.makeText(MainActivity.this,
        "Kippschalter ist aktiviert", Toast.LENGTH_SHORT).show();
      else
        Toast.makeText(MainActivity.this,
        "Kippschalter ist nicht aktiviert", Toast.LENGTH_SHORT).show();
    }
```

```
    });

    // Reaktion auf Änderungen des Auswahlfelds
    spinner.setOnItemSelectedListener(
        new AdapterView.OnItemSelectedListener() {
      @Override
      public void onItemSelected(AdapterView<?> parent, View view,
          int position, long id) {
        Toast.makeText(MainActivity.this, "Spinner = " + position,
            Toast.LENGTH_SHORT).show();
      }

      @Override
      public void onNothingSelected(AdapterView<?> parent) {
        Toast.makeText(MainActivity.this, "Spinner: nichts gewählt",
          Toast.LENGTH_SHORT).show();
      }
    });
  }
}
```

7.1.4 Das Model-View-Controller-Paradigma und Android

Das klassische *Model-View-Controller* (MVC) Paradigma beschreibt eine Architektur für interaktive Anwendungen zur Bearbeitung von Daten. Dabei ist das interne Modell zuständig für die Datenhaltung (*Model*) und von der Darstellung der Daten am Bildschirm (*View*) entkoppelt. Der Controller steuert die Interaktion mit dem Anwender. Wenn sich das *Model* ändert, wird die *View* informiert und kann die geänderten Daten anzeigen. Damit kann man ein und dieselben Daten verschieden präsentieren. So lässt sich eine Tabelle als Matrix anzeigen oder die Daten durch ein entsprechendes Diagramm visualisieren.

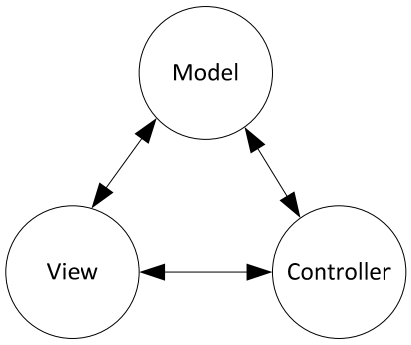


Abbildung 7.8: Model-View-Controller

Elementare Steuerelemente wie Schalter lassen sich auch in Android ohne Kenntnis der MVC-Architektur programmieren, nicht aber aufwendige Anwendungen für Tabellen.

Beispiel: Auswahl von Alternativen mittels Arrayadapter

Ein Anwender unserer App soll eine Alternative aus einer Liste von Vorschlägen auswählen. Das *Model* besteht im Programm aus einem Array aus Zeichenketten. Die *View* zeigt diese Liste an; wir verwenden in diesem Spezialfall eine `ListActivity`. Der *Controller* sorgt für die Interaktion mit dem Anwender.

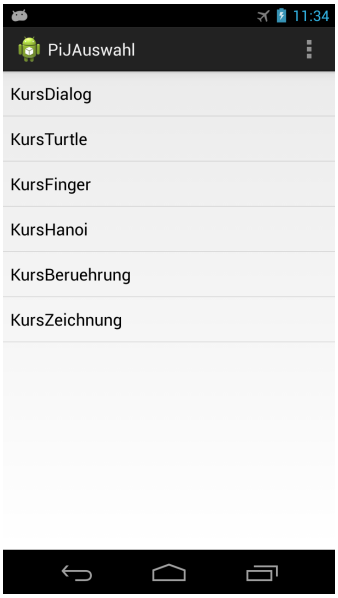


Abbildung 7.9:
Auswahl aus einer Folge von Alternativen mit ArrayAdapter

Für diese technisch anspruchsvolle Aufgabenstellung bietet sich in Android ein `ArrayAdapter` an, den wir mit dem Aufruf `createFromResource` aus Listing 7.8 erstellen können. Als Programmierer versorgen wir einen solchen Adapter mit dem Kontext der Activity, einem Array aus Texten über eine `id`-Nummer eines `string-array` in der Datei `values.xml` sowie einer `id`-Nummer eines Layouts als `TextView`. Der Controller muss unser Programm informieren, wenn der Anwender eine der Alternativen auswählt: Wir überschreiben die Methode `onListItemClick` aus Listing 7.8 von `ListActivity`. Android übergibt beim Aufruf dieser Methode die ausgewählte `View` im Parameter `v`. Dies ist eine `TextView`, deren Inhalt wir ermitteln können. So erhalten wir den ausgewählten Text. Alternativ könnten wir auch den Index `position` benützen, um über den Array den ausgewählten Text zu ermitteln.

Listing 7.8: Benützte Methoden: ArrayAdapter erzeugen, auf Klick reagieren

```
// Klasse ArrayAdapter:
ArrayAdapter<CharSequence> createFromResource (Context context,
    int textArrayResId, int textViewResId);

// Klasse ListActivity:
void onListItemClick (ListView l, View v, int position, long id)
```

Als konkrete Anwendung benützen wir in Listing 7.11 eine `ListActivity`, um aus einer Liste von Namen den Namen einer Activity auszuwählen. Wir könnten dann die Activity mit dem ausgewählten Namen innerhalb unserer App starten. Wir benützen diese Konstruktion, um in den Abschnitten 7.1.5 und im Kurs 7.3 diverse Activities zeigen zu können, ohne jeweils gleich eine komplette App für Android schreiben zu müssen.

Listing 7.9: Die Datei strings.xml mit den Zeichenfolgen

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">PiJAuswahl</string>
    <string name="activity_zeichnung">KursZeichnung</string>
…
</resources>
```

Listing 7.10: Die Datei values.xml mit dem Array aus Zeichenfolgen

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string-array name="kursthemen">
    <item>@string/activity_zeichnung</item>
…
    </string-array>
</resources>
```

Listing 7.11: Eine ListActivity zur Auswahl aus einer Reihe von Alternativen

```
public class MainActivity extends ListActivity {

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    ArrayAdapter<CharSequence> adapter = ArrayAdapter.createFromResource(
        this, R.array.kursthemen, android.R.layout.simple_list_item_1);

    setListAdapter(adapter);
  }

  @Override
  protected void onListItemClick(ListView l, View v,
      int position, long id) {
    super.onListItemClick(l, v, position, id);
    final CharSequence text = ((TextView) v).getText();
    final Toast toast = Toast.makeText(this, "Kurs: " + text,
          Toast.LENGTH_LONG);
    toast.show();
//  So könnten wir Activities starten:
//  Jede Activity muss in AndroidManifest.xml aufgeführt sein!
//    Intent intent = new Intent();
//    intent.setComponent(new ComponentName(getPackageName(),
//      getPackageName() + "." + text.toString()));
//    startActivity(intent);

  }

}
```

7.1.5 Anordnung der Komponenten

Mit Hilfe einer `ViewGroup` können wir in Android einzelne Elemente einer GUI (d.h. Views) zu einer Gruppe zusammenfassen. `ViewGroup` ist eine abstrakte Klasse, d.h. konkret können wir nur davon abgeleitete nicht abstrakte Klassen wie `LinearLayout`, `FrameLayout` usw. benützen. Diese Klassen dienen nicht nur dem Zusammenfassen einzelner Views, sondern implementieren bestimmte Strategien zur Anordnung ihrer Elemente. Da eine `ViewGroup` ihrerseits eine `View` ist, kann sie auch wiederum eine `ViewGroup` enthalten. Als Konsequenz kann man Layouts schachteln. Dadurch entstehen aber laut der Doku-

mentation zu Android erhöhte Aufwendungen zur Laufzeit. Sie sollten diesen höheren Verbrauch bei tragbaren Geräten gegenüber eventuellen Vorteilen abwägen. Die Dokumentation zu Android empfiehlt anstelle geschachtelter Layouts den Einsatz von `RelativeLayout` bzw. `GridLayout`: [*http://developer.android.com/guide/topics/ui/declaring-layout.html*]. In der Praxis verursacht die Entwicklung eines optisch ansprechenden und leicht bedienbaren Layouts einen erheblichen Aufwand. Am leichtesten zu programmieren ist das `LinearLayout`, welches für viele Beispiele zum Einstieg ausreicht.

Layout und der grafische Editor für Benutzungsoberflächen

Die Entwicklungsumgebung enthält einen grafischen Editor, der uns bei der Platzierung der Komponenten tatkräftig unterstützt. Außerdem zeigt der grafische Editor alle Optionen für die einzelnen Layouts sowie die Steuerelemente an.

Tabelle 7.2: Einige …Layout-Klassen in Android

`LinearLayout`	Komponenten der Reihe nach horizontal bzw. vertikal anordnen.
`FrameLayout`	Spezielles Layout für nur eine View.
`RelativeLayout`	Komponenten in Abhängigkeit von anderen positionieren.
`TableLayout`	Komponenten in Zeilen mit spaltengerechter Ausrichtung anordnen.
`GridLayout`	Komponenten in einem flexiblen Matrix-Raster anordnen.
`RadioGroup`	Zusammenfassen von RadioButtons zu einer Gruppe.
`AbsoluteLayout`	Überholt. Aber: WebView aus dem WebKit benützt AbsoluteLayout.

AbsoluteLayout

Damit legt man die Koordinaten für die einzelnen Elemente fest. Dies führt zu wenig flexiblen Anordnungen. Deswegen gilt dieses Layout seit Version 3 als veraltet.

FrameLayout

Ein `FrameLayout` ist dafür vorgesehen, eine bestimmte Fläche am Bildschirm für eine einzige `View` freizuhalten. Er sollte nur eine `View` enthalten. Die enthaltenen `Views` werden wie bei einem Stack der Reihe nach übereinander angezeigt, so dass die zuletzt hinzugefügte `View` oben erscheint. Android benützt dieses Layout intern z. B. zur Darstellung von per Tabulator auswählbaren Views.

RelativeLayout

Ein relatives Layout erlaubt eine Positionierung in Bezug auf die Eltern-`View` oder in Bezug auf durch eine Ressouren-ID angegebene `View`.

Tabelle 7.3: Werte für die Positionierung aus RelativeLayout.LayoutParams

`android:layout_alignParentTop`	Falls `"true"`: die obere Kante der `View` wird an der oberen Kante der elterlichen View ausgerichtet.
`android:layout_centerVertical`	Falls `"true"`: die `View` wird vertikal in der elterlichen View zentriert.
`android:layout_below="…"`	Positioniert die obere Kante der `View` unterhalb der durch eine Ressourcen-Id spezifizierte `View`.
`android:layout_toRightOf="…"`	Positioniert die linke Kante der `View` rechts von der durch eine Ressourcen-Id spezifizierte `View`.

TableLayout

Ein `TableLayout` ordnet ähnlich wie eine Tabelle in HTML seine Elemente in einzelnen Zeilen und Spalten an. Das jeweils breiteste Element bestimmt die Breite einer Spalte. Ein `TableLayout` enthält Zeilen in Form von einzelnen `TableRow`-Elementen. Die Zeile mit den meisten Spalten definiert die Anzahl der Spalten. Die einzelnen Zeilen können kein `layout_width`-Attribut angeben, die Breite ist immer `match_parent`. Einzelne Zellen können sich über mehrere Spalten erstrecken.

GridLayout

Ein `GridLayout` platziert seine Komponenten in einzelnen Zellen in einem Raster. Im Gegensatz zum TableLayout können sich einzelne Zellen über mehrere Zeilen und Spalten erstrecken: `rowSpec` bzw. `columnSpec` legt deren Anzahl fest.

LinearLayout

Ein `LinearLayout` wie in Listing 7.2 oder in Listing 7.6 ordnet seine Elemente horizontal (`android:orientation="horizontal"`) bzw. vertikal (`android:orientation="vertical"`) an. Die Angabe von `android:orientation="horizontal"` ist überflüssig.

`LinearLayout` positioniert seine Komponenten linksbündig. Bei den einzelnen Komponenten kann man über das Attribut `android:gravity="…"` mit den Werten `"top"`, `"bottom"`, `"left"`, `"right"`, `"center"`, `"center_horizontal"` bzw. `"center_vertical"` noch die relative Position innerhalb ihres Platzes ohne Änderung der Größe angeben. Die Werte `"fill_horizontal"`, `"fill_vertical"`, `"fill"` würden dagegen die Komponenten horizontal, vertikal bzw. in alle Richtungen dehnen, um sie in die entsprechenden Plätze einzupassen.

Mit dem Attribut `android:layout_weight="0"` bei einer Komponente beeinflusst man die Größe des zugewiesenen Platzes. Der Wert `"0"` ist die Vorbelegung. Haben alle Komponenten dasselbe „Gewicht“ bei der Platzvergabe, dann erhalten sie einen Platz entsprechend ihrer Größe. Bei ungleichen Gewichten verteilt Android den Platz entsprechend der Relation der Gewichtung. Den restlichen Platz würde die Komponente mit dem höchsten Gewicht erhalten.

RadioGroup

Mit einer `RadioGroup` fasst man eine Menge von `RadioButtons` zu einer Gruppe zusammen. Drückt man dann einen Schalter der Gruppe, entfernt Android die Markierung von den anderen Schaltern.

Beispiele zu Layouts

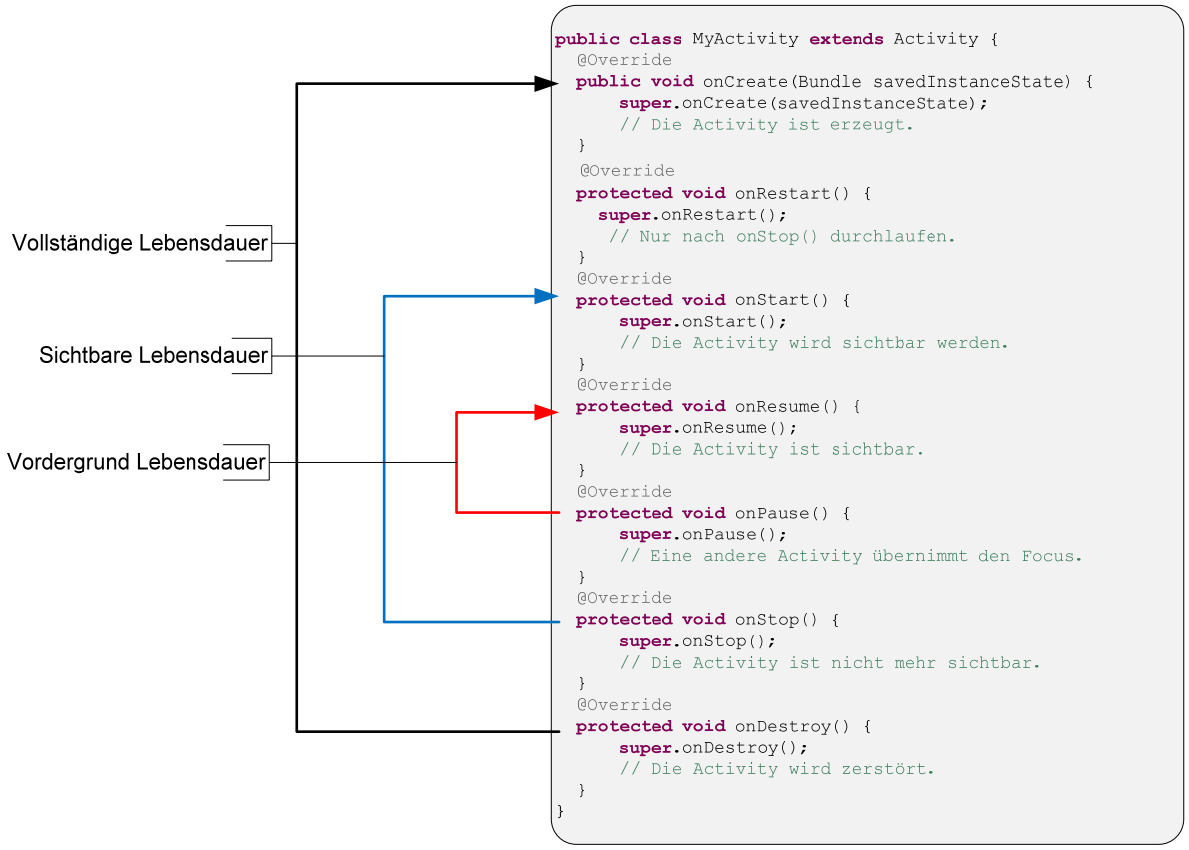
Eine App soll die Anwendung einiger Layouts demonstrieren. Eine Activity erlaubt die Auswahl einer speziellen Activity, die dann die gewählte Strategie für das Layout demonstriert. Sie finden diese App bei den Quellprogrammen.

7.2 Verwaltung von Activities durch Android

Android verwaltet die einzelnen Activities. Android erzeugt die konkreten Exemplare bzw. die Objekte unserer von `Activity` abgeleiteten Klassen. Dazu benützt es den Default-Konstruktor, der von Java automatisch angelegt wird, sofern man keinen eigenen Konstruktor definiert (siehe auch Kapitel 3). Die Activity „lebt“ dann bis zu ihrem von Android veranlassten Ende und durchläuft einen sog. *Lebenszyklus* bzw. *Lifecycle*.

7.2.1 Lebenszyklus von Activities

In der Klasse `Activity` gibt es spezielle Methoden, die man in den eigenen Activities überschreiben kann, die sog. *Lifecycle-Callbacks*. Android ruft die einzelnen Methoden auf, um die Aktionen in der Activity zu veranlassen, die entsprechend dem aktuellen Zustand der App erforderlich sind. `onCreate` ist eine dieser Methoden, die von Android gerufen wird. Im Programm können wir davon ausgehen, dass die Activity nicht nur vorhanden ist, sondern auch Aufrufe an ihre Umgebung benützen kann. In unseren Beispielen erzeugen wir in dieser Methode eine View, z. B. aus einer Layout-Datei. Diese View kann von Android am Bildschirm angezeigt werden, wenn es soweit ist. In Abbildung 7.10 sehen wir einige *Callbacks*, die den Lebenszyklus einer Activity definieren. Die einzelnen Aufrufe können nicht in beliebiger Reihenfolge auftreten. Android garantiert, dass die Abfolge der Aufrufe `onCreate` – `onStart` – `onResume` – `onPause` stets in dieser Reihenfolge abläuft.




```
public class MyActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      // Die Activity ist erzeugt.
  }
  @Override
  protected void onRestart() {
    super.onRestart();
     // Nur nach onStop() durchlaufen.
  }
  @Override
  protected void onStart() {
      super.onStart();
      // Die Activity wird sichtbar werden.
  }
  @Override
  protected void onResume() {
      super.onResume();
      // Die Activity ist sichtbar.
  }
  @Override
  protected void onPause() {
      super.onPause();
      // Eine andere Activity übernimmt den Focus.
  }
  @Override
  protected void onStop() {
      super.onStop();
      // Die Activity ist nicht mehr sichtbar.
  }
  @Override
  protected void onDestroy() {
      super.onDestroy();
      // Die Activity wird zerstört.
  }
}
```

Abbildung 7.10: Lebenszyklus einer Activity mit Lifecycle-Callbacks

In der Zeitspanne zwischen `onCreate` und `onDestroy` „lebt“ die Activity. Zwischen `onStart` und `onStop` ist sie sichtbar und zwischen `onResume` und `onPause` ist sie im Vordergrund des Bildschirms. Zum „Herunterfahren“ einer App dienen die Aufrufe `onStop` und `onDestroy`. Nach `onStop` kann eine App wieder sichtbar werden. Falls genügend Speicher vorhanden ist, lässt Android die App im Hintergrund weiterlaufen und wir können sie per Multitasking-Taste schnell wieder in den Vordergrund holen. Wir sollten spätestens bei `onStop` aufwendige Animationen beenden, da der Anwender ohnehin nichts davon sehen kann. Mit `onDestroy` wird eine App definitiv beendet.

Beim Programmieren müssen wir beachten, dass es nach Ablauf der `onPause`-Methode verschiedene Möglichkeiten gibt:

- Der Anwender holt die App wieder in den Vordergrund. Dann folgt `onResume`.
- Der Anwender stellt die App in den Hintergrund. Dann folgt `onStop`. Falls die Activity danach wieder in den Vordergrund kommt, folgt `onRestart`.
- Android bricht die App z. B. wegen Speichermangel ab.

Vorsicht: Abbruch der Activity möglich!

Eine Activity unter Android vor der Version 3.0 kann nach dem Ablauf von `onPause` von Android abgebrochen werden, z. B. um Speicher für andere Activities zu gewinnen! Wir können uns nicht darauf verlassen, dass einer der Aufrufe `onStop` bzw. `onDestroy` folgt, bevor die Activity abgebrochen wird. Ab der Version 3.0 folgt auf den Aufruf von `onPause` in jedem Fall noch der Aufruf von `onStop`.

7.2.2 Drehen des Smartphones

Viele Apps für Smartphones zeigen uns verschiedene Ansichten, je nachdem, wie wir das Gerät halten:

- Hochformat: „Portrait“
- Querformat: „Landscape“

Wenn wir das Gerät drehen, wechselt die Anzeige zwischen Hoch- und Querformat. Die einzelnen Apps reagieren unterschiedlich darauf.

Android startet eine Activity nach dem Drehen neu. Dazu fährt Android die App herunter. Danach wird ein neues Exemplar der Activity erzeugt und der Lebenszyklus der Activity beginnt von vorne. Listing 7.12 illustriert die Konsequenzen für uns Programmierer: Die Attribute einer Klasse werden bei diesem Neustart alle wieder auf ihre Anfangswerte zurückgesetzt! Listing 7.12 zeigt eine Activity mit einem Schalter. Bei jedem Drücken des Schalters zählen wir eine Nummerierung hoch und zeigen sie auf der Schaltfläche an. Außerdem protokollieren wir alle in Abbildung 7.10 genannten Callbacks.

Listing 7.12: Programm zur Protokollierung der Zustände einer Activity

```
public class MainActivity extends Activity {

  private final static String tag = MainActivi-
ty.class.getPackage().getName();

  private void log(String text) {
    Log.i (tag, text + ": Variable = " + variable);
  }

  public MainActivity() {
    log("new MainActivity");
  }

  private Button b;

  @Override
  public void onCreate(Bundle sicherung) {
    super.onCreate(sicherung); // ***
    setContentView(R.layout.activity_main);
    b = (Button) findViewById(R.id.button);
    // Die Activity ist erzeugt.
    log("onCreate");
  }

  public void onClickButton(View v) {
    variable++;
    b.setText(String.valueOf(variable));
  }
```

```
  @Override
  protected void onRestart() {
    super.onRestart();
    log("onRestart");
    // Nur nach onStop() durchlaufen.
  }

  @Override
  protected void onStart() {
    super.onStart();
    log("onStart");
    // Die Activity wird sichtbar werden.
  }

  @Override
  protected void onResume() {
    super.onResume();
    log("onResume");
    // Die Activity ist sichtbar.
  }

  @Override
  protected void onPause() {
    super.onPause();
    log("onPause");
    // Eine andere Activity übernimmt den Focus.
  }

  @Override
  protected void onStop() {
    super.onStop();
    log("onStop");
    // Die Activity ist nicht mehr sichtbar.
  }

  @Override
  protected void onDestroy() {
    super.onDestroy();
    log("onDestroy");
    // Die Activity wird zerstört.
  }

  // Wird bei Erzeugung eines Objekts der Klasse mit Wert belegt
  private int variable = 0;

}
```

Wir halten das Smartphone im Hochformat und lassen die App laufen. Wir drücken den Schalter und drehen dann das Smartphone, wie in Abbildung 7.11 gezeigt.

Nach dem Drehen des Smartphones ist der Zähler wieder auf 0 zurückgefallen. Das Ablaufprotokoll Listing 7.13 zeigt uns zunächst die erwarteten Werte. Die Activity wird erzeugt. Darauf folgen die Callback-Aufrufe `onCreate` – `onStart` – `onResume`. Wir drücken die Schaltfläche und der Wert der Variablen wird erhöht. Dann drehen wir das Smartphone. Jetzt folgen weitere Aufrufe: `onPause` – `onStop` – `onDestroy`. Android zerstört also die Activity. Danach sehen wir, dass der Konstruktor der Activity wieder aufgerufen wurde: Android erzeugt wieder eine Activity. Damit erhalten alle Attribute im Objekt die in der Initialisierung vorgesehenen Werte.

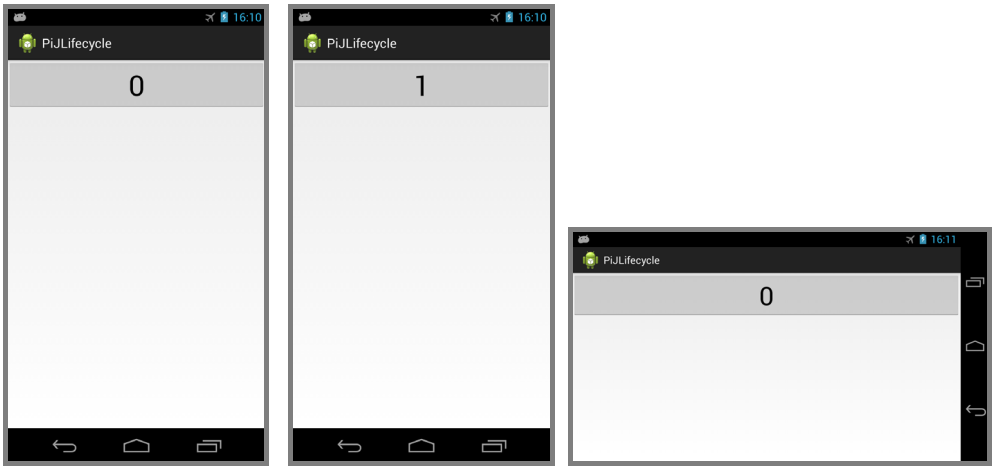


Abbildung 7.11: App nach Start, dann drücken wir, dann drehen wir

Listing 7.13: Ablaufprotokoll in LogCat

```
de.pij.lifecycle: new MainActivity: Variable = 0
de.pij.lifecycle: onCreate: Variable = 0
de.pij.lifecycle: onStart: Variable = 0
de.pij.lifecycle: onResume: Variable = 0
de.pij.lifecycle: onPause: Variable = 1
de.pij.lifecycle: onStop: Variable = 1
de.pij.lifecycle: onDestroy: Variable = 1
de.pij.lifecycle: new MainActivity: Variable = 0
de.pij.lifecycle: onCreate: Variable = 0
de.pij.lifecycle: onStart: Variable = 0
de.pij.lifecycle: onResume: Variable = 0
```

7.2.3 Wie sichern wir den Zustand einer Activity?

Wie wir in Listing 7.13 sehen, müssen wir auch bei einfachen Apps an die Sicherung des Zustands der Activity denken: Für die Wiederherstellung der Attribute unserer Activities sind wir als Programmierer selbst zuständig. Wir können in Listing 7.12 den Programmcode zur Sicherung ergänzen, indem wir die Methode `onSaveInstanceState` überschreiben, siehe Listing 7.14. Zur Wiederherstellung müssen wir in der `onCreate`-Methode auf das übergebene Bundle `sicherung` zugreifen, und die dort gesicherten Daten auslesen und wieder herstellen.

Listing 7.14: Sichern und Wiederherstellen von Attributen

```
@Override
public void onCreate(Bundle sicherung) {
  super.onCreate(sicherung);
  setContentView(R.layout.activity_main);
  b = (Button) findViewById(R.id.button);
  // Die Activity ist erzeugt.
  log("onCreate");

  // Wiederherstellen der Attribute
```

```
    if (sicherung != null) {
      variable = sicherung.getInt(getString(R.string.variable_i));
    }
    b.setText(String.valueOf(variable));
  }

  // Sichern der Attribute in Bundle
  @Override
  protected void onSaveInstanceState(Bundle sicherung) {
    // Nicht vergessen: Sichern der UI. Aufruf der Methode von Activity
    super.onSaveInstanceState(sicherung);
    sicherung.putInt(getString(R.string.variable_i), variable);
  }
```

Für die Sicherung der Daten der App im Verzeichnis der App bzw. auf der SD-Karte des Smartphones verwendet man die `onPause`-Methode. Sie wird in jedem Fall und bei jeder Version von Android aufgerufen, bevor die App beendet wird. Die `onSaveInstanceState`-Methode wird nur dann aufgerufen, wenn Android davon ausgeht, dass die Benutzungsoberfläche wiederhergestellt werden muss. Deswegen sollte man in dieser Methode die für den Weiterlauf der Activity nötigen Daten sichern.

7.3 Kurs: GUI-Anwendungen

Die bisherigen Abschnitte in diesem Kapitel stellen einzelne Elemente des Android-Systems vor. In diesem Abschnitt wählen wir eine problemorientierte Sicht auf die Dinge und versuchen, Grundprobleme bei der Entwicklung von Anwendungen für grafische Benutzeroberflächen anhand ausgewählter Beispiele aus der Sicht des Entwicklers zu lösen:

- Wir erstellen einer grafischen Komponente.
- Wir steuern Apps über Berührung.
- Wir folgen der Bewegung des Fingers.
- Wir zeichnen mit der Turtle-Grafik.
- Wir implementieren Dialoge unter Android.
- Wir animieren die Türme von Hanoi.
- Wir bearbeiten eine Liste mit Alternativen.

Zunächst stellen wir jeweils das Problem dar. Danach entwickeln wir die gedankliche Lösung. Schließlich implementieren wir den gewählten Weg: *Ziel-Vorgehen-Umsetzung*.

Wir könnten für jede Kurseinheit eine eigene App erstellen. Die einzelnen Apps wären relativ ähnlich. Um den Gesamtumfang zu reduzieren, erstellen wir eine Rahmenapplikation. Hier kann man die jeweilige Activity auswählen. Im Buchtext zeigen wir aus Platzgründen nur die spezifischen Teile der jeweiligen Activity; das vollständige Programm finden Sie auf der begleitenden Web-Seite.

7.3.1 Erstellung einer Zeichnung

Eine Anwendung soll eine Fläche mit einem Rautenmuster füllen und ausgeben.

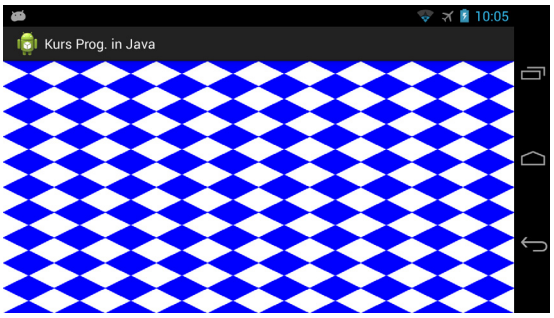


Abbildung 7.12:
Bildschirmabdruck für das Rautenmuster

Vorgehen

Ein Programm benötigt zum Zeichnen eine Zeichenfläche. Dazu setzen wir unter Android eine `View` ein: Wir leiten unsere Klasse `Zeichenflaeche` von der Klasse `View` ab. Grafische Ausgaben eines Programms auf eine Fläche sind aufwendig: Android entscheidet, wann welche Fläche neu dargestellt werden muss. Wenn der Neuaufbau einer Fläche oder einer Teilfläche unvermeidlich ist, ruft Android die Methode `onDraw(Canvas canvas)` unser Klasse auf. Diese Zeichenfläche setzt man als View in der `Activity`:

```
setContentView(new Zeichenflaeche (this));
```

Das Graphik-Koordinatensystem beschreiben wir **in** Abschnitt 1.6.2 in Kapitel 1. dort zeigt Abbildung 1.11, dass die linke obere Ecke des Bildschirms die Koordinaten (0, 0) hat. Die Koordinaten der rechten unteren Ecke kann man in der Methode `onDraw(Canvas canvas)` mit `canvas.getWidth()-1` bzw. `canvas.getHeight()-1` ermitteln. Auf dem `Canvas` können wir mit Methoden `canvas.drawXyz...` zeichnen.

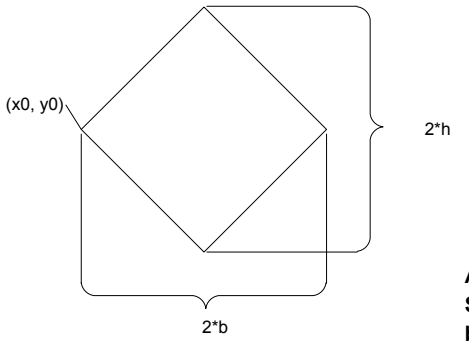


Abbildung 7.13:
Skizze zum Zeichnen einer Raute bzw. eines Parallelogramms

Wir zeichnen das Rautenmuster, indem wir zunächst die komplette Fläche mit einer Farbe des Musters füllen. Danach zeichnen wir gefüllte Rauten mit der anderen Farbe. Wenn wir eine einzelne Raute zeichnen wollen, können wir die Raute als Pfad betrachten. Der Aus-

gangspunkt ist die linke Ecke. Es folgt die untere Ecke, danach die rechte und schließlich die obere Ecke. Die Koordinaten dieser Punkte fügen wir zu einen `Path`-Objekt hinzu. Abbildung 7.13 zeigt die Überlegungen, um die x- bzw. y-Koordinaten der Punkte des Polygons zu ermitteln, wie in Listing 7.15 in der Methode `zeichneRaute(…)` benützt. Wir teilen dann die gesamte Breite der Fläche durch die doppelte Anzahl der gewünschten Rauten `anzahlRauten` und erhalten die Größe b als halbe Breite einer Raute. Analog ermitteln wir die Größe h als halbe Höhe einer Raute. Wir füllen die Fläche in einzelnen Zeilen. Jede Zeile füllen wir der Reihe nach mit Rauten. Wir benötigen für jede der insgesamt `anzahlRauten` Zeilen zu jeweils `anzahlRauten` Rauten.

Listing 7.15: Füllen einer Fläche mit einem Rautenmuster

```
public class KursZeichnung extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    // Wir setzen die View direkt ohne Umweg über eine Layout-Datei
    setContentView(new Zeichenflaeche (this));
  }

  public static class Zeichenflaeche extends View {

  private final static int anzahlRauten = 10;

    // Konstruktor für unsere View-Klasse
    public Zeichenflaeche (Context context) {
      super (context, null);
    }

    // Der „Malkasten": Einstellungen für die Farbe usw.
    private Paint paint = new Paint();


    private void zeichneRaute(Canvas canvas,
         float x0, float y0, float b, float h, int color) {
      float[] xPunkte = { x0, x0 + b, x0 + 2 * b, x0 + b, x0 };
      float[] yPunkte = { y0, y0 - h, y0,         y0 + h, y0 };
      Path path = new Path ();
      path.moveTo(xPunkte[0], yPunkte[0]);   // Erster Punkt
      for (int i = 1; i < xPunkte.length; i++) {
        path.lineTo(xPunkte[i], yPunkte[i]); // Folgende Punkte
      }
      paint.setColor(color);
      canvas.drawPath(path, paint); // Pfad wird gefüllt gezeichnet
    }

    @Override
    protected void onDraw(Canvas canvas) {
      canvas.drawColor(Color.WHITE);
      float xSchritt = (float) canvas.getWidth() / anzahlRauten / 2.0f;
      float ySchritt = (float) canvas.getHeight() / anzahlRauten / 2.0f;
      for (int i = 0; i < anzahlRauten; i++) {
        for (int j = 0; j <= anzahlRauten; j++) {
          zeichneRaute (canvas, xSchritt*i*2.0f, ySchritt*j*2.0f,
                                xSchritt,        ySchritt, Color.BLUE);
        }
      }
    }
  }

}
```

7.3.2 Reaktion auf Berührungen mit dem Finger

Ein Programm soll in einer Fläche ein Schachbrett mit 8x8 Feldern als Gitter anzeigen. Wenn ein Mausklick innerhalb eines Rechtecks erfolgt, dann soll das betreffende Rechteck ab diesem Zeitpunkt gefüllt dargestellt werden.

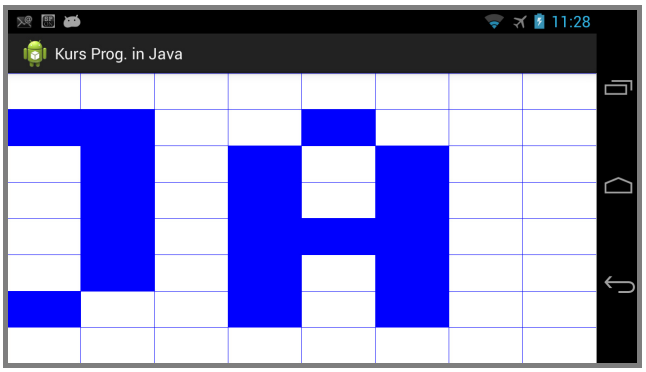


Abbildung 7.14:
Beispiel für eine Bildschirmanzeige nach einigen Berührungen

Problemanalyse: Struktur der Anwendung

Die Anwendung schreibt man als eine von `Activity` abgeleitete Klasse. Für die Arbeitsfläche verwendet man eine von `View` abgeleitete Klasse `Zeichenflaeche`. Die Zeichenfläche enthält eine Matrix namens `klicks`, in der wir die Nummern der Zeilen und Spalten im Raster notieren, die berührt wurden.

Problemanalyse: Größe der Rechtecke im Raster

Die Größe des Gitters kann man nur aus der aktuellen Größe des Feldes berechnen. Hierzu liest man die Breite bzw. die Höhe der Arbeitsfläche mit `getWidth()` bzw. `getHeight()` aus und teilt diese Werte durch 8. Wir zeichnen das Gitter in Form von 7 waagrechten bzw. 7 senkrechten Linien, den Trennlinien zwischen den 8 Teilen.

Problemanalyse: Ereignissteuerung

Der Anwender in Abbildung 7.15 berührt den Bildschirm. Danach stellt Android eine Nachricht *„Berührung (xKoordinate, y-Koordinate, ...)“* in die Schlange der Ereignisse. Wenn diese Nachricht zur Verarbeitung ansteht, ruft Android eine Methode auf.

Hierzu überschreiben wir in unserer `Zeichenflaeche` die Methode `onTouchEvent`. Wir lesen die Mauskoordinaten aus dem zugestellten Ereignis `event` vom Typ `MotionEvent` mit den Aufrufen `event.getX()` und `event.getY()` aus. Jetzt gibt es ein Problem: Wir können die Methode `onDraw(…)` nicht direkt aufrufen, das fällt in die Zuständigkeit von Android(siehe Abbildung 7.15). Deswegen müssen wir alle Informationen für die irgendwann von Android aufzurufende `onDraw(…)` – Methode z. B. auf der Ebene der Klasse in der Variablen `klick` hinterlegen. Beim Aufruf der Methode `onDraw(..)` kann man aus den

Pixel-Koordinaten des Mausklicks die Zeile und Spalte in dem Raster in unserer Zeichenfläche berechnen.

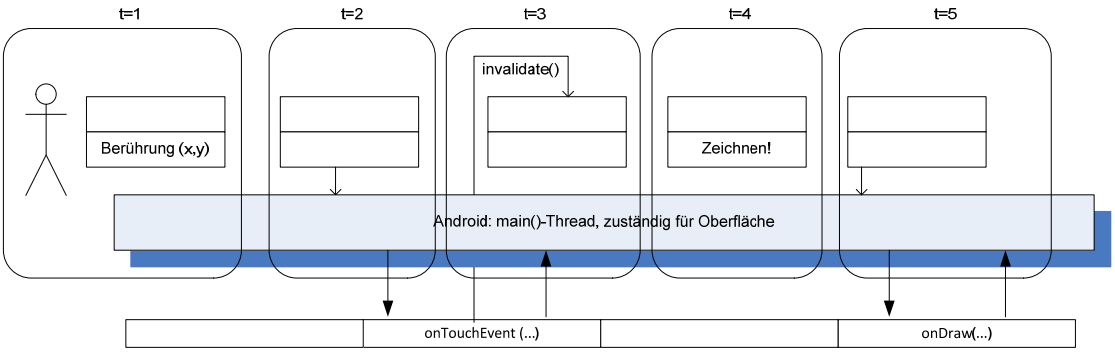


Abbildung 7.15: Abläufe von der Berührung bis zum Zeichnen

In der Methode `onTouchEvent` rufen wir die `invalidate()`-Methode der Zeichenfläche. Danach sorgt das Android-Laufzeitsystem irgendwann für einen Aufruf der `onDraw(…)`-Methode. Wir können den Zeitpunkt, zu dem der Aufruf tatsächlich erfolgt, nicht beeinflussen. `invalidate` signalisiert lediglich den Aktualisierungsbedarf. Tatsächlich generiert der `invalidate()`-Aufruf eine Nachricht, die in die Ereigniswarteschlange wie in Abbildung 7.5 bzw. Abbildung 7.15 eingestellt wird. Android entscheidet, wann die Nachricht „zugestellt", d. h. wann die `onDraw()`-Methode aufgerufen wird.

Listing 7.16: Reaktion auf Berühren des Bildschirms

```
public class KursBeruehrung extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    // Wir setzen die View direkt ohne Umweg über eine Layout-Datei
    setContentView(new Zeichenflaeche (this));
  }

  public static class Zeichenflaeche extends View {

    // Notiere alle bisherigen Klicks in einem Feld
    private boolean klicks[][];
    // Der Punkt, auf den zuletzt geklickt wurde
    private PointF klick;

    public Zeichenflaeche(Context context) {
      super(context, null);
      // bisher keine Klicks
      klicks = new boolean[8][8];
      for (int i = 0; i < klicks.length; i++)
        for (int j = 0; j < klicks[i].length; j++)
          klicks[i][j] = false;
      klick = new PointF();
      klick.x = -1.0f; // ungültige Koordinaten
    }

    private Paint paint = new Paint();
```

```
    @Override
    protected void onDraw(Canvas canvas) {
      canvas.drawColor(Color.WHITE);
      paint.setColor(Color.BLUE);

      float b = getWidth() - 1;
      float h = getHeight() - 1;
      float y = h / klicks.length;
      float x = b / klicks[0].length;

      // Waagrechte Linien
      for (int i = 1; i < klicks.length; i++) {
        canvas.drawLine(0, y * i, b, y * i, paint);
      }

      // Senkrechte Linien
      for (int i = 1; i < klicks[0].length; i++) {
        canvas.drawLine(x * i, 0, x * i, h, paint);
      }

      // Ermittle das berührte Feld
      if (klick.x != -1.0f) {
        int zeile = (int) (klick.y / y);
        int spalte = (int) (klick.x / x);
        klick.x = -1.0f;
        klicks[zeile][spalte] = true;
      }

      // Füllen aller bisher angeklickten Rechtecke
      for (int i = 0; i < klicks.length; i++)
        for (int j = 0; j < klicks[i].length; j++)
          if (klicks[i][j])
            canvas.drawRect(x*j, y*i, x*(j+ ), y*(i+1), paint);
    }

    @Override
    public boolean onTouchEvent(MotionEvent event) {
      switch (event.getAction()) {
      case MotionEvent.ACTION_DOWN:
        klick.x = event.getX(); // Notiere die Koordinaten
        klick.y = event.getY(); // in Attributen des Objekts
        invalidate();
        break;
      case MotionEvent.ACTION_MOVE:
        break;
      case MotionEvent.ACTION_UP:
        break;
      }
      return true;
    }
  }

}
```

7.3.3 Reaktion auf Fingerbewegungen

Ein Programm soll Fingerbewegungen als durchlaufenden Linienzug aufzeichnen und am Fenster wiedergeben. Damit kann der Anwender einfache Freihandzeichnungen erstellen. Wenn das Fenster neu gezeichnet wird, soll die Zeichnung nicht verschwinden, sondern erneut am Bildschirm angezeigt werden.

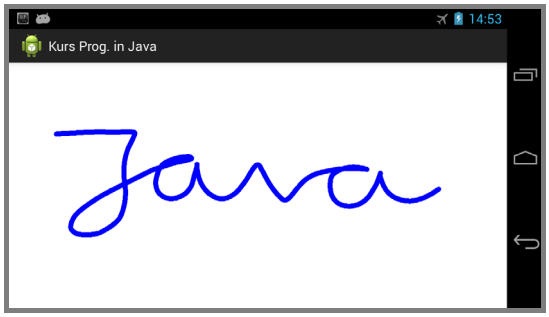


Abbildung 7.16:
Mit dem Finger malen

Vorgehen

Unsere Unterklasse `Zeichenflaeche` von `View` muss die `onTouchEvent(MotionEvent event)`-Methode überschreiben (siehe Kurseinheit 7.3.2). Die Klasse `Path` kann Pfade für das Zeichnen speichern. Wir werden die vom Finger zurückgelegte Strecke hier als Folge einzelner Linien aufzeichnen. Dazu lesen wir in der Methode `onTouchEvent` die Koordinaten der Berührung mit dem Finger aus dem zugestellten Ereignis `event` mit den Aufrufen `event.getX()` bzw. `event.getY()` aus. Wenn der Finger die Oberfläche berührt, setzen wir den Pfad zurück und speichern die Koordinaten des Punktes ab. Wenn sich der Finger bewegt hat, fügen wir eine Linie zur neuen Position hinzu.

Die `onDraw(…)`-Methode muss den Kantenzug vollständig ausgeben. Dazu genügt die Anweisung `canvas.drawPath (path, paint)`. Die Einstellung für den Stil des Zeichnens muss `Paint.Style.STROKE` sein, sonst würde der Kantenzug als gefüllte Fläche dargestellt.

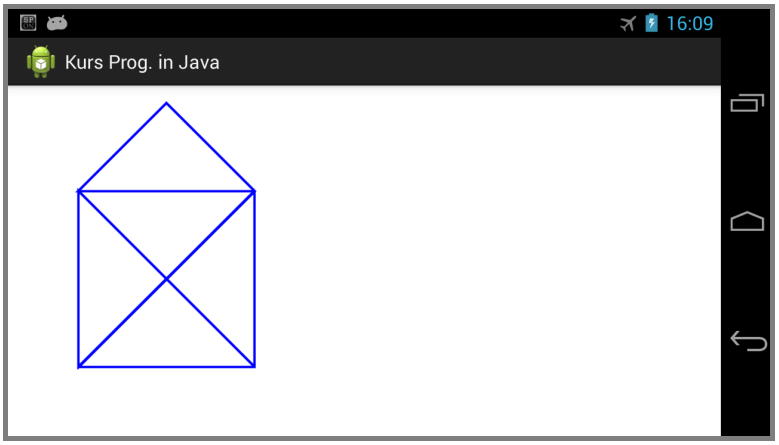
Listing 7.17: Malen mit dem Finger auf dem Smartphone

```
public class KursFinger extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    // Wir setzen die View direkt ohne Umweg über eine Layout-Datei
    setContentView(new Zeichenflaeche (this));
  }

  public static class Zeichenflaeche extends View {

    // Notiere alle bisherigen Bewegungen
    private Path path;

    // Der "Malkasten" paint
    private Paint paint;

    public Zeichenflaeche(Context context) {
      super(context, null);
      paint = new Paint();
      // Paint.Style.FILL würde eine Fläche füllen!!
      paint.setStyle(Paint.Style.STROKE);
      paint.setStrokeWidth(12);
      path = new Path();
    }
    @Override
    protected void onDraw(Canvas canvas) {
      canvas.drawColor(Color.WHITE);
```

```
        paint.setColor(Color.BLUE);
        canvas.drawPath(path, paint);
      }

      @Override
      public boolean onTouchEvent(MotionEvent event) {
        float x = event.getX();
        float y = event.getY();

        switch (event.getAction()) {
        case MotionEvent.ACTION_DOWN:
          path.reset();
          path.moveTo(x, y); // Starte mit diesem Punkt
          break;
        case MotionEvent.ACTION_MOVE:
          path.lineTo(x, y); // Linie zum neuen Punkt
          break;
        case MotionEvent.ACTION_UP:
          break;
        }
        invalidate();// Bildschirm neu aufbauen!!
        return true;
      }
    }

  }
```

7.3.4 Turtle-Graphik

Eine Turtle-Graphik besteht aus einer „Schildkröte“ mit einem aktivierbaren Zeichenstift, die sich über eine Zeichenfläche bewegt. Die „Schildkröte“ befindet sich auf einem durch die Koordinaten `x`, `y` bestimmten Punkt und zeigt in eine durch einen Winkel `angle` bestimmte Richtung. Sie dreht sich bei einem `turn(Winkel)`-Befehl um den angegebenen Winkel und fährt bei einem `move(Länge)`-Befehl in der momentan eingeschlagenen Richtung eine Strecke der angegebenen Länge. Dabei hinterlässt die „Schildkröte“ eine Spur auf der imaginären Zeichenfläche. Mit einer solchen Schildkröte kann man Zeichnungen aus aufeinanderfolgenden Strichen programmieren, wie das berühmte „Haus des Nikolaus“, bei dem man ein Haus in einem Zug als Folge von 8 Linien zeichnen muss. Wir starten unseren Streckenzug stets in der linken unteren Ecke des Hauses.



Vorgehen: Schildkröte

Für die Turtle-Graphik schreiben wir eine eigene Klasse `Turtle`. Sie enthält als Attribute die Position `x`, `y` und den Richtungswinkel `angle` der „Schildkröte“. Die `turn(angle)`-Methode dreht die Schildkröte um den angegebenen Winkel. Abbildung 7.17 zeigt eine Problemanalyse für die `move(Länge)`-Methode. Wenn der Startpunkt der Schildkröte die Koordinaten $(x0, y0)$ hat und die Schildkröte den angegebenen Winkel eingenommen hat, ergibt sich der Endpunkt (xN, yN) der Bewegung zu $xN = x0 + l * \cos(angle)$ und $yN = y0 + l * \sin(angle)$. Dabei ist l die Länge der zurückgelegten Strecke und *angle* der Winkel. Wenn wir den Winkel in der Turtle im Gradmaß verwalten, müssen wir diesen Winkel noch ins Bogenmaß umrechnen: $180^o \equiv \pi$. Zur Anzeige der Schildkröte zeichnen wir eine Line vom Anfangspunkt zum Endpunkt der Strecke. Danach versetzen wir die Schildkröte an den Endpunkt der Linie.

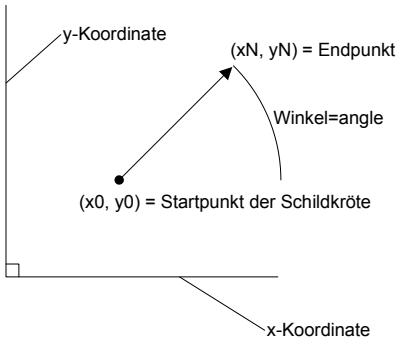


Abbildung 7.17: Problemanalyse für die move – Methode der Schildkröte

Bei Android können wir die Bewegungen der Turtle in einem sog. `Path` aufzeichnen. Zur Anzeige am Bildschirm lesen wir den Pfad aus.

Listing 7.18: Die Turtle-Grafik für Android

```
public class Turtle {
  // Die Turtle hat einen Punkt sowie eine Richtung in Form eines Winkels
  private double x, y, angle;

  private Path path = new Path();

  public Turtle(double x, double y) {
    reset(x, y, 0.0);
  }

  // Setze die Turtle neu
  final public void reset(double x, double y, double angle) {
    this.x = x;
    this.y = y;
    this.angle = angle;
    path.reset();
    path.moveTo((float) x, (float) y);
  }

  // Drehen um einen Winkel: Reduktion auf 0..360 Grad
  final public void turn(double angle) {
    this.angle += angle;
    this.angle = Math.IEEEremainder(this.angle, 360.0);
```

```
  }

  // Die Turtle zeichnet die Bewegung in path auf
  final public void move(double length) {
    double xN, yN;
    xN = x + length * Math.cos(angle / 180.0 * Math.PI);
    yN = y + length * Math.sin(angle / 180.0 * Math.PI);
    path.lineTo((float) xN, (float) yN);
    x = xN;
    y = yN;
  }

  // Auslesen des Pfades als Folge aller aufgezeichneten Bewegungen
  public Path getPath() {
    return path;
  }

}
```

Vorgehen: Das Haus des Nikolaus

Wir zeichnen das Haus des Nikolaus aus einzelnen Linien und beginnen bei der linken unteren Ecke. Die erste Linie ist der „Boden“ des Hauses. Die Länge dieser Strecke ist die Seitenlänge des Quadrats des Hauses (ohne das Dach). Nun zeigt die Turtle in die Richtung dieser Linie. Bevor wir die nächste Linie zeichnen, müssen wir die Turtle um -90° drehen. Im Sinne des Programms besteht eine Bewegung aus dem Fahren einer bestimmten Strecke gefolgt von einer Drehung:

```
Fahre Seitenlänge, danach: Drehe um -90° (Linksdrehung)
Fahre Seitenlänge, danach: Drehe um -90° (Linksdrehung)
Fahre Seitenlänge, danach: Drehe um +90°+45° = 135° (Rechtsdrehung)
Fahre eine halbe Diagonale, danach: Drehe um 90° (Rechtsdrehung)
Fahre eine halbe Diagonale, danach: Drehe um 90° (Rechtsdrehung)
Fahre eine Diagonale, danach: Drehe um +90°+45° = 135° (Rechtsdrehung)
Fahre Seitenlänge, danach: Drehe +90°+45° = 135° (Rechtsdrehung)
Fahre eine Diagonale, fertig.
```

In Listing 7.19 fassen wir in der Klasse Bewegung jeweils einen `move(l)` und einen darauffolgenden `turn(angle)` zusammen. Die absolute Länge der Strecke, die die Schildkröte zurücklegen soll, kennen wir leider erst in der `onDraw(…)`-Methode. Deswegen müssen wir diese Strecken dort berechnen. Die relativen Längen dagegen können wir vorausberechnen, wenn wir annehmen, dass die Seitenlänge des Quadrats gleich 1.0 ist. Wir speichern alle „Fahrbefehle“ in dem Feld `hausDesNikolaus` ab und führen jeweils genau `statusDesHauses` Befehle mit der Schildkröte durch. Diesen Zustand schalten wir bei jedem Mausklick höher und beginnen bei 8 wieder von 0 an zu zählen.

Listing 7.19: Programm zum Zeichnen für das Haus des Nikolaus.

```
public class KursTurtle extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(new Zeichenflaeche(this));
  }

  public static class Zeichenflaeche extends View {

    // Der "Malkasten" paint
```

```
private Paint paint;

// Zum Hausbau berechnen wir die nötigen Aktionen
private static class Haus {
  // Eine Bewegung besteht aus Fahren und anschl. Drehung
  static class Bewegung {
    public double length, angle;
    public Bewegung (double length, double angle) {
      this.length = length;
      this.angle = angle;
    }
  };

  // Relative Seitenlängen bezogen auf Länge = 1.0
  private static final double seite = 1.0;
  private static final double diagonale = seite * Math.sqrt (2.0);
  private static final double diagonaleHalbe = diagonale / 2.0;

  // Alle Bewegungen sind vor Ablauf bekannt. Z.B.:
  private static final Bewegung[] hausDesNikolaus = new Bewegung[]{
    new Bewegung (seite,           -90.0),
    new Bewegung (seite,           -90.0),
    new Bewegung (seite,           135.0),
    new Bewegung (diagonaleHalbe,   90.0),
    new Bewegung (diagonaleHalbe,   90.0),
    new Bewegung (diagonale,       135.0),
    new Bewegung (seite,           135.0),
    new Bewegung (diagonale,         0.0), // fertig
 };
  // Wir zählen die Anzahl der Striche mit:
  private int statusDesHauses = 0;
  public void weiterSchalten () {// Bei Berührung
    statusDesHauses++;
    if (statusDesHauses > hausDesNikolaus.length)
      statusDesHauses = 0;
  }

  // Auslesen des nächsten Schritts, sofern vorhanden
  public Bewegung schritt (int i) {
    if (i >= statusDesHauses)
      return null;
    return hausDesNikolaus[i];
  }
}

// Hier bauen wir das Haus des Nikolaus:
private Haus haus;
private Turtle turtle;

public Zeichenflaeche(Context context) {
  super(context, null);
  paint = new Paint();
  // Paint.Style.FILL würde eine Fläche füllen
  paint.setColor(Color.BLUE);
  paint.setStyle(Paint.Style.STROKE);
  paint.setStrokeWidth(4);
  turtle = new Turtle (0,0);
  haus  = new Haus ();
}

@Override
protected void onDraw(Canvas canvas) {
  canvas.drawColor(Color.WHITE);
  double breite = Math.min (getWidth(), getHeight())/2.0;
  turtle.reset(getWidth()*0.1, getHeight()*0.8, 0.0);
  Haus.Bewegung schritt = null;
  int i = 0;
  while ((schritt = haus.schritt(i)) != null) {
    // Die tatsächliche Breite des Hauses kennen wir erst jetzt:
    turtle.move (schritt.length*breite);
```

```
            turtle.turn(schritt.angle);
            i++;
          }
          canvas.drawPath(turtle.getPath(), paint);
          haus.weiterSchalten();
        }

        @Override
        public boolean onTouchEvent(MotionEvent event) {
          if (event.getAction() == MotionEvent.ACTION_DOWN)
            invalidate();
          return true;
        }
      }

    }
```

7.3.5 Dialoge in Android

Manche Aktionen eines Programms benötigen mehrere Parameter, wie z. B. Name, E-Mail-Adresse und Anschrift. Man kann diese Daten in einem *Dialog* bearbeiten. Wir kennen solche Dialoge, die ein eigenes Fenster über eine vorhandene Anzeige am Bildschirm legen. Der Dialog hat in der Regel zwei Schalter: *Ok* bzw. *Einverstanden* und *Abbrechen* bzw. *Cancel*. Wenn wir *Ok* drücken, sind wir mit den eingegebenen Daten bzw. der angezeigten Aktion einverstanden (*Wollen Sie wirklich alle Daten löschen?*). Wenn wir *Abbrechen* drücken, so dürfen die angezeigten Daten nicht übernommen bzw. die angedrohte Aktion nicht ausgeführt werden. In dieser Kurseinheit wollen wir einen Datensatz in einer Activity für Android mit Hilfe eines Dialogs editieren.

Grundsätzlich sollten wir bei Apps für Smartphones Dialoge mit Bedacht einsetzen. Für den Anwender sind in vielen Fällen Folgen von Activities leichter zu bedienen, bei denen jede einen Teil der Eingabe abwickelt. Bei einem Dialog hingegen kehrt das Programm wieder zur aufrufenden Activity zurück.

Programmierung von Dialogen unter Android

Unter Android sind Dialoge mit einer Activity verbunden. Im Programm können wir Dialoge nicht direkt starten, sondern nur auf Umwegen: Wir können diesen Vorgang mit `showDialog (id)` initiieren. Der GUI-Thread von Android ruft danach irgendwann eine Methode `protected Dialog onCreateDialog(int id)` auf, die wir in unserer Activity überschreiben können. In dieser Methode können wir anhand der `id` den Dialog bestimmen und erzeugen. Dabei können wir die Schalter *Ok* mit `setPositiveButton` bzw. *Abbrechen* mit `setNegativeButton` setzen. Dabei geben wir auch die Callback-Methoden für die Reaktion auf die Eingabe des Anwenders an.

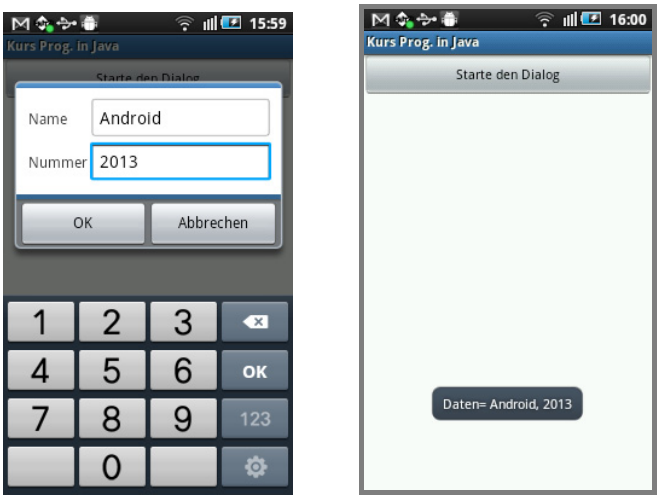


Abbildung 7.18: Ein Dialog unter Android und die Anzeige der Ergebnisse als Toast

Vorgehen: Aufbau der Benutzungsoberfläche

Die Activity soll ein `LinearLayout` mit einem Schalter enthalten. Listing 7.20 zeigt das Layout. Beim Schalter ist der Name einer Methode angegeben, die beim Drücken aktiviert wird. Diese Methode sorgt für das Starten des Dialogs.

Listing 7.20: Die Datei kursmain.xml mit dem LinearLayout der Activity

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

    <Button
        android:id="@+id/startdialog"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:onClick="starteDialog"
        android:text="@string/start_dialog" />

</LinearLayout>
```

Für den Dialog wählen wir ein `TableLayout`, mit dem wir die Eingabefelder für den Namen bzw. die Zahl matrixförmig anordnen können. Wenn wir den Typ eines Eingabefeldes kennen, können wir dies gleich im `EditText`-Steuerelement angeben. Dann erzeugt Android z. B. bei Zahlen gleich eine Tastatur für die Eingabe von Ziffern.

Listing 7.21: Die Datei kursdialog.xml mit dem TableLayout für den Dialog

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/layout_root"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="#ffffff"
```

```
        android:orientation="horizontal"
        android:padding="10dp" >

        <TableRow>

            <TextView
                android:layout_column="1"
                android:padding="3dip"
                android:text="Name"
                android:textSize="16sp" />

            <EditText
                android:id="@+id/editName"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_weight="1"
                android:ems="10"
                android:inputType="text|textPersonName" >

                <requestFocus />
            </EditText>
        </TableRow>

        <TableRow>

            <TextView
                android:layout_column="1"
                android:padding="3dip"
                android:text="Nummer"
                android:textSize="16sp" />

            <EditText
                android:id="@+id/editNummer"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_weight="1"
                android:ems="10"
                android:inputType="number" />
        </TableRow>

    </TableLayout>
```

In der Activity `KursDialog` erzeugen wir einen Dialog zur Editierung von Daten der Klasse. Mit einem `LayoutInflater`, den wir von Android erhalten, erstellen wir eine `View` namens `layout`. Aus dieser `View` können wir mit `findViewbyId` die beiden `EditText`-Steuerelemente ermitteln, mit welchen der Anwender unseres Programms die beiden Zeichenketten `name` und `nummer` editieren kann. Zum Aufbau eines Dialoges nutzen wir den `AlertDialog.Builder`. Einige seiner Aufrufe liefern wieder das `Builder`-Objekt zurück. Deswegen können wir hier mit Aufrufketten der Art `builder().a().b().c()` arbeiten. Als Reaktion auf das Drücken des OK-Schalters geben wir den geänderten Stand der Daten in der Activity `KursDialog` aus. Falls der Anwender auf den Abbrechen-Schalter gedrückt hat, verändern wir die Daten nicht, entfernen aber den Dialog. Denn der Cache-Mechanismus in Android würde sonst bei einer neuen Aktivierung des Dialogs nicht die alten Daten aus der Activity `KursDialog` holen, sondern den Dialog in seinem letzten Zustand wiederherstellen. Damit würden die geänderten, aber nicht abgespeicherten Daten von Neuem angezeigt.

Listing 7.22: Die Activity zur Anzeige des Dialogs

```
// Dialoge für Android ab V2.x
public class KursDialog extends Activity {
  private final static int MYDIALOG = 1;
  private EditText editName = null;
  private EditText editNummer = null;
  private String name = "name";
  private String nummer = "0";

  private void zeigeDaten() {
    final CharSequence text = name + ", " + KursDialog.this.nummer;
    final Toast toast = Toast.makeText(this,
         "Daten= " + text, Toast.LENGTH_LONG);
    toast.show();
  }

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.kursmain);
  }

  public void starteDialog(View v) {
    showDialog(MYDIALOG);
  }

  private Dialog createDialog() {
    final LayoutInflater inflater =
        (LayoutInflater) getSystemService(LAYOUT_INFLATER_SERVICE);
    final View layout = inflater.inflate(R.layout.kursdialog,
        (ViewGroup) findViewById(R.id.layout_root));

    editName = (EditText) layout.findViewById(R.id.editName);
    editNummer = (EditText) layout.findViewById(R.id.editNummer);

    editName.setText(name);
    editNummer.setText(nummer);
    AlertDialog.Builder builder = new AlertDialog.Builder(this);
    builder.setView(layout);
    builder.setPositiveButton("OK",
     new DialogInterface.OnClickListener(){
      public void onClick(DialogInterface dialog, int whichButton) {
        name = editName.getText().toString();
        nummer = editNummer.getText().toString();
        zeigeDaten();
      }
    }).setNegativeButton("Abbrechen",
     new DialogInterface.OnClickListener() {
      public void onClick(DialogInterface dialog, int whichButton) {
        // Warnung vor dem Cache-Mechanismus in Android:
        // Auch bei Cancel würde der alte Zustand wieder auftauchen!
        // Deswegen empfiehlt es sich, den alten Dialog zu schließen.
        removeDialog(MYDIALOG);
      }
    });
    return builder.create();
  }

  @Override
  protected Dialog onCreateDialog(int id) {
    switch (id) {
    case MYDIALOG:
      return createDialog();
    default:
      return super.onCreateDialog(id);
    }
  }
}
```

7.3.6 Die Türme von Hanoi

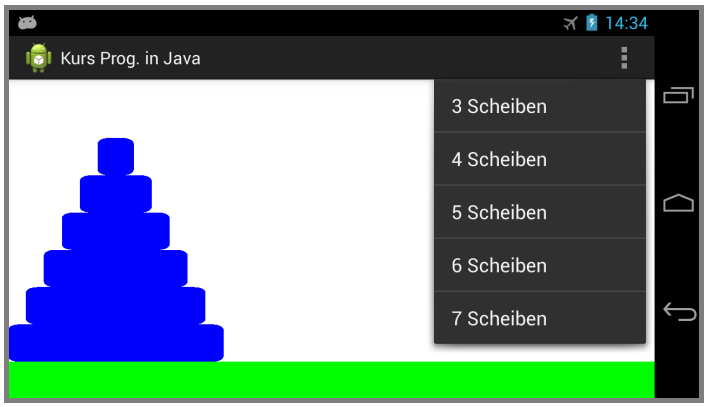


Abbildung 7.19: Türme von Hanoi

Wir wollen die Folge der Züge zur Lösung der Türme von Hanoi bedienbar machen. Das Programm soll wie in Abbildung 7.19 zunächst den Ausgangszustand darstellen, bei dem alle Scheiben auf dem linken Stapel liegen. Mit Hilfe des Programms aus Listing 2.33 können wir den nächsten Zug ermitteln, um den Stapel von der linken Seite auf die rechte Seite zu transferieren. Unser neues Programm soll die Folge dieser Spielzüge visualisieren. Bei Berührung des Bildschirms zeigen wir den jeweils nächsten Zustand an. Der Anwender kann die Anzahl der Scheiben über ein Menu bzw. die sog. *Action-Bar* auswählen.

Vorgehen: Model-View-Controller

Wir unterscheiden zwischen der Logik des Programmablaufs (dem sog. *Model*) auf der einen Seite und der Ansicht (*View*) bzw. der Ablaufsteuerung auf der anderen Seite. Das *Model* stellt den Zustand des Programms dar. Es beschreibt die Türme in ihren momentanen Zuständen und die Übergänge beim Bewegen einer Scheibe von einem Turm zu einem anderen. Die *View* visualisiert diese Zustände am Bildschirm. Die *Controller*-Anteile sorgen für die Reaktion auf den Benutzer.

7.3.6.1 Zum Model

Zunächst müssen wir das Problem in die Sprache der Objektorientierung übersetzen. Zur Analyse muss man sich die Fragen nach dem Aufbau der beteiligten Objekte stellen:

- Was ist eine Scheibe?
- Was ist ein Turm?
- Wie ist das Spielbrett aufgebaut?
- Wie stellen wir eine Bewegung bzw. eine Folge von Bewegungen dar?

Was ist eine Scheibe?

Eine Scheibe ist in unserem vereinfachten Modell nur eine ganze Zahl. Diese ganze Zahl gibt die Breite in Längeneinheiten: 1, 2, 3, usw. an. Beim Start liegen auf einem Turm von der Spitze her gesehen n Scheiben: 1, 2, 3, …, n. Die anderen beiden Türme sind leer.

Was ist ein Turm?

Ein `Turm` hat einen Stapel aus Scheiben und erlaubt nur die Operationen *Wegnehmen* und *Anfügen* einer Scheibe. Nach den Regeln für die Türme von Hanoi ist dies nur am oberen Ende erlaubt. Ein `Turm` mit n Türmen entsteht aus einem leeren Stapel durch Anfügen der n Scheiben, wobei wir entsprechend den Regeln des Spiels die Scheibe mit der größten Breite zuerst auf den Stapel legen, dann die nächst kleinere usw. Ein `Turm` verhält sich damit im Sinne der Informatik wie ein Stapel (andere Bezeichnung: `Stack`), der die folgenden Operationen erlaubt:

```
Stack<Integer> scheiben = new Stack<Integer>()
scheiben.push (scheibe);
scheibe = scheiben.pop ();
```

Ein Turm bietet diese Operationen, die er an sein Attribut `scheiben` delegieren kann.

Delegation und Eclipse

Eclipse unterstützt den Prozess der Delegation von Aufträgen an ein Mitglied der Klasse unter dem Menüpunkt *Source/Generate Delegate Methods*. Dazu klickt man auf das Attribut `scheiben` und wählt den angegebenen Menüpunkt. Dann entscheidet man, für welche Methoden die Delegationsmethoden zu generieren sind.

Listing 7.23: Die Klasse Turm verwaltet die Scheiben

```
public class Turm {
  // Ein Turm enthält Scheiben
  // Die Breite der Scheiben ist ganzzahlig
  private Stack<Integer> scheiben;

  // Liefere die Anzahl der Scheiben
  public int size() {
    return scheiben.size();
  }

  // Ein neuer Turm entsteht durch Auflegen der Scheiben.
  // Die breiteste Scheibe liegt unten (Zuerst gelegt)
  public Turm(int anzahl) {
    scheiben = new Stack<Integer>();
    for (int scheibe = anzahl; scheibe > 0; scheibe--)
      scheiben.push(scheibe);
  }

  // Abnehmen der Scheiben oben
  public Integer pop() {
    return scheiben.pop();
  }

  // Auflegen von Scheiben oben
  public Integer push(Integer scheibe) {
    return scheiben.push(scheibe);
  }
```

```
  public Stack<Integer> getScheiben() {
    return scheiben;
  }

}
```

Wie funktioniert ein Spielbrett?

Das Spielbrett `Tuerme` besteht aus drei Türmen. Wir realisieren einen Zug durch Wegnehmen einer Scheibe von oben vom Turm `von` und durch Ablegen eben dieser Scheibe oben auf einem anderen Turm `nach`. Die Methode `reset` setzt das Spielbrett auf den Anfangszustand mit n Scheiben zurück.

Listing 7.24: Turme modelliert das Spielbrett

```
public class Tuerme {
  private Turm[] tuerme = null;
  private static int maxScheiben = 0;

  public Tuerme(int anzahlTuerme) {
    reset(anzahlTuerme);
  }

  // Ein Zug entfernt eine Scheibe oben von einem Stapel (=von)
  // und legt die Scheibe auf einen Stapel (=nach)
  public synchronized void zug(int von, int nach) {
    tuerme[nach].push(tuerme[von].pop());
  }

  public Turm[] getTuerme() {
    return tuerme;
  }

  public static int getMaxScheiben() {
    return maxScheiben;
  }

  // Zurück auf Anfang mit der Anzahl der Scheiben
  public synchronized void reset(int anzahlScheiben) {
    tuerme = new Turm[3];
    tuerme[0] = new Turm(anzahlScheiben);
    tuerme[1] = new Turm(0);
    tuerme[2] = new Turm(0);
    maxScheiben = anzahlScheiben;
  }

}
```

Darstellung einer Bewegung: ein Zug

Ein Zug auf dem Spielbrett besteht aus einer Bewegung von einem Stapel zu einem Stapel.

Listing 7.25: Darstellung des Transports einer Scheibe: ein Zug im Sinne des Spiels

```
public class Zug {
  public int von, nach;

  public Zug(int von, int nach) {
    this.von = von;
    this.nach = nach;
  }

}
```

Abspeichern der Zugfolge

Der Anwender kann durch Drücken auf den Bildschirm eine Bewegung auf dem Spielbrett auslösen. Wir berechnen die Folge dieser Bewegungen bzw. Züge mit dem Algorithmus für die Türme von Hanoi aus Listing 2.33 und speichern sie in einer Liste von Zügen ab. Dann kann man auf Anfrage den nächsten Zug liefern.

Listing 7.26: Die Zugverwaltung rechnet die Bewegungen vorher aus

```
public class Zugverwaltung {
  private List<Zug> loesung = null;

  private Zugverwaltung () {
  }

  public static Zugverwaltung neueZugFolge(int anzahlScheiben) {
    Zugverwaltung zv = new Zugverwaltung();
    zv.index = 0;
    zv.loesung = new ArrayList<Zug>(2 << anzahlScheiben - 1);
    zv.zugFolge(0, 1, 2, anzahlScheiben);
    return zv;
  }

  // Siehe Listing 2.33
  private void zugFolge(int von, int hilf, int nach, int n) {
    if (n <= 1) {
      loesung.add(new Zug(von, nach));
    } else {
      zugFolge(von, nach, hilf, n - 1);
      zugFolge(von, 0, nach, 0);
      zugFolge(hilf, von, nach, n - 1);
    }
  }

  private int index = 0;

  // Liefere den nächsten Zug zur Anzeige
  public Zug naechsterZug() {
    if (index < loesung.size())
      return loesung.get(index++);
    else
      return null;
  }

  public void reset() {
    index = 0;
  }
}
```

7.3.6.2 Türme von Hanoi: die View

Die App enthält eine `Activity`, die in der `HanoiView` die Türme anzeigt. Diese Klasse enthält das Model sowie die zur Anzeige der Daten der Logik erforderlichen Klassen. Jedem der Türme haben wir eine eigene `TurmView` zugeordnet. In der `onDraw`-Methode berechnen wir die Größe des Rechtecks für die Darstellung eines Turms. Wir zeichnen die 3 Türme nebeneinander. Damit erhält jeder Turm ein Drittel der zur Verfügung stehenden Breite.

Listing 7.27: Die Türme von Hanoi

```
public class HanoiView extends View {

  // Attribute für das Model
  private Tuerme tuerme;
  private Zugverwaltung zv;

  // Attribute für die View
  private TurmView[] turmView;
  private Paint paint = new Paint();;
  private RectF r = new RectF();

  public void initHanoi() {
    // Initialisiere das Model (=Logik)
    tuerme = new Tuerme(anzahlScheiben);
    zv = Zugverwaltung.neueZugFolge(anzahlScheiben);
    // Initialisiere die View
    turmView = new TurmView[3];
    for (int i = 0; i < tuerme.getTuerme().length; i++)
      turmView[i] = new TurmView(tuerme.getTuerme()[i]);
  }

  public HanoiView(Context context) {
    super(context, null);
    initHanoi();
  }

  protected void onDraw(Canvas canvas) {
    canvas.drawColor(Color.WHITE);
    r.set(canvas.getClipBounds()); // Clonen des umrandenden Rechtecks
    // Breite für jeden Turm = Gesamtbreite / 3
    float breite = r.width() / 3;
    float startX = r.left; // X-Koordinate zum Start
    for (int turm = 0; turm < turmView.length; turm++) {
      r.set(startX, 0, startX + breite, r.height());
      startX += breite; // Nächster Turm weiter rechts
      turmView[turm].zeichne(canvas, r, paint);
    }
  }

  private int anzahlScheiben = 4;

  @Override
  public boolean onTouchEvent(MotionEvent event) {
    if (event.getAction() == MotionEvent.ACTION_DOWN) {
      Zug z = zv.naechsterZug();
      if (z != null)
        tuerme.zug(z.von, z.nach);
      else
        initHanoi(); // Wieder von vorne: zurück auf Anfang
      invalidate();
    }
    return true;
  }

  public void setAnzahlScheiben(int anzahlScheiben) {
    this.anzahlScheiben = anzahlScheiben;
    initHanoi();
    invalidate();
  }

}
```

Listing 7.28: TurmView übernimmt die Selbstdarstellung eines Turmes

```
public class TurmView {
  private Turm turm;

  public TurmView(Turm turm) {
    this.turm = turm;
  }

  private RectF r = new RectF();
  private RectF scheibe = new RectF();

  public void zeichne(Canvas canvas, RectF umrandung, Paint paint) {
    this.r.set(umrandung); // Clonen des umrandenden Rechtecks
    // Höhe für jede Scheibe = Gesamthöhe / max. Anzahl Scheiben
    float hoehe = r.height() / Tuerme.getMaxScheiben() * 0.7f;
    // Wir zeichnen zuerst den Sockel, dann die breiteste Scheibe usw.
    float startY = r.top + r.height() - hoehe;
    // Zeichne den Sockel
    paint.setColor(Color.GREEN);
    canvas.drawRect(r.left, startY, r.right, r.bottom, paint);
    // Zeichne der Reihe nach die Scheiben des Turmes
    Stack<Integer> scheiben = turm.getScheiben();
    for (int i = 0; i < scheiben.size(); i++) {
      int breiteLogisch = scheiben.elementAt(i);
      float breitePixel = umrandung.width() *
        breiteLogisch / Tuerme.getMaxScheiben();
      startY -= hoehe; // Nächste Scheibe obenauf legen
      float einruecken = (umrandung.width() - breitePixel) / 2;
      paint.setColor(Color.BLUE);
      scheibe.set(r.left + einruecken, startY,
                  r.right - einruecken, startY + hoehe);
      canvas.drawRoundRect(scheibe, 20.0f, 10.0f, paint);
    }
  }
}
```

Effizienz in onDraw

Diese Methode wird bei jedem Neuaufbau des Bildschirms aufgerufen. Deswegen empfiehlt die Dokumentation von Android, das Erzeugen neuer Objekte während dieser Routine zu vermeiden und stattdessen vorhandene Objekte zu benützen. Deswegen benützen beide o. a. Programme Rechtecke, die auf Klassenebene definiert sind.

7.3.7 Auswahl aus einer Liste von Alternativen

Bei der Eingabe am Smartphone muss man häufig aus einfachen Listen von Alternativen auswählen: Welche Farbe bevorzugen wir, in welchem Bundesland leben wir usw. Unsere App muss alle möglichen Alternativen anzeigen und über Berührung einer Alternative eine Auswahl gestatten. Falls ein Bildschirm nicht zur Anzeige aller Möglichkeiten ausreicht, muss automatisch ein Rollbalken auf einer Seite angezeigt und ausgewertet werden.

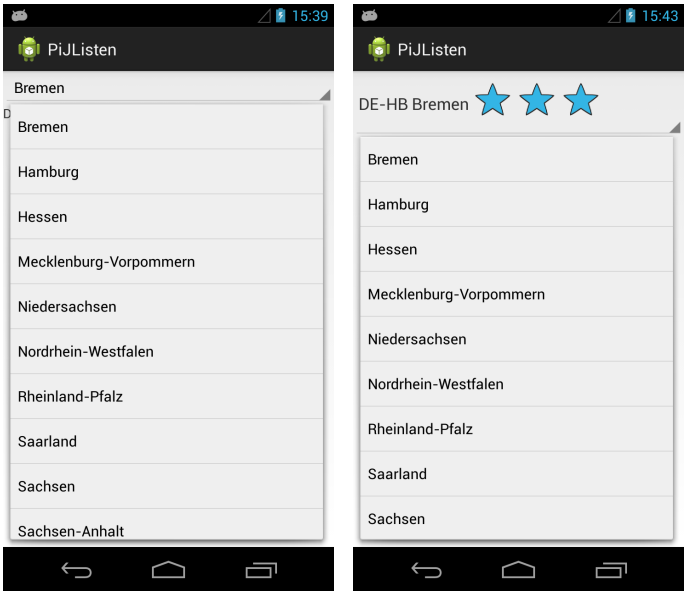


Abbildung 7.20: Links Fall 1: einfache Texte, rechts Fall 2: mehrere Steuerelemente pro Zeile

Vorgehen

Android trennt im Rahmen der MVC-Architektur klar zwischen den Daten im *Model* sowie der *View* zur Anzeige und dem *Controller* zur Bearbeitung. Wir stellen im Model die Daten für die Anzeige in der View bereit. Außerdem müssen wir im Model Eingaben des Anwenders übernehmen.

Zur Anpassung der Daten aus dem Model benützt man in Android einen der vorgegebenen Adapter. Liefert man die Alternativen zur Auswahl im Model als Array an, bietet sich ein `ArrayAdapter` an. Wir unterscheiden zwei Fälle:

1. Einfache Daten: Jede Zeile der Auswahl enthält nur einen Text, z. B. einen Namen.
2. Zusammengesetzte Daten: Jede Zeile enthält einen Datensatz: Name, Kürzel, Rating.

Lösung zu Fall 1: Jede Zeile enthält nur einen Text

Die Alternativen liegen häufig als Array aus Texten vor. Diese Texte kann man, wie in Java üblich, als Array angeben. In Android kann man den Array für die Texte wie Listing 7.29 in einem `<string-array…` bei den Ressourcen in `res/values/values.xml` definieren. Die Texte selbst schreibt man in die Datei `res/values/strings.xml` in Listing 7.30.

Dann könnte man die Texte leicht internationalisieren, indem man die übersetzten Texte in Unterverzeichnissen mit dem Kürzel für die Nationen mitliefert. Für eine automatische Auswahl von Texten in englischer Sprache müssten die Texte in einer eigenen Datei `res/values-en/strings.xml` vorhanden sein.

Listing 7.29: Arrays aus Texten als Ressourcen

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

    <string-array name="laendernamen">
        <item>@string/baden_wuerttemberg</item>
        <item>@string/bayern</item>
… usw.
        <item>@string/thueringen</item>
    </string-array>

    <string-array name="laenderkuerzel">
        <item>@string/de_bw</item>
        <item>@string/de_by</item>
…usw
        <item>@string/de_th</item>
    </string-array>

</resources>
```

Listing 7.30: Die zugehörigen Texte

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="baden_wuerttemberg">Baden-Württemberg</string>
    <string name="bayern">Bayern</string>
… usw.
    <string name="thueringen">Thüringen</string>

    <string name="de_bw">DE-BW</string>
    <string name="de_by">DE-BY</string>
… usw.
    <string name="de_th">DE-TH</string>

</resources>
```

Mit der Methode `ArrayAdapter.createFromResource` erzeugen wir einen Adapter aus der Ressource `laendernamen`. Als Layout für einen Eintrag verwenden wir das von Android vordefinierte Layout `android.R.layout.simple_spinner_item`. Der Adapter muss noch bei dem Steuerelement für die Auswahl gesetzt werden.

Zur Reaktion auf die Auswahl gehen wir nach dem Beobachter-Muster vor. Die Quelle für die Ereignisse in Listing 7.31 ist unser Steuerelement für die Auswahl, der sog. `Spinner`. Dort registrieren wir einen `OnItemSelectedListener` für die Beobachter. Wenn ein Eintrag aus der Liste der Bundesländer ausgewählt wird, zeigen wir hierfür das Kürzel aus zwei Buchstaben aus dem Array der Länderkürzel an.

Listing 7.31: Auswahl aus einer Liste der Bundesländer

```
public class ActivityArrayAdapter1 extends Activity {

  private TextView ergebnis; // Ergebnis der Auswahl anzeigen

  private CharSequence[] laenderkuerzel;

  private void showToast(CharSequence msg) {
    Toast.makeText(this, msg, Toast.LENGTH_SHORT).show();
  }

  @Override
  protected void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);

        // Länderkürzel laden
        Resources res = getResources();
        laenderkuerzel = res.getStringArray(R.array.laenderkuerzel);

        // Layout laden und Steuerelemente suchen
        setContentView(R.layout.activity_array1);
        Spinner auswahl = (Spinner) findViewById(R.id.auswahl);
        ergebnis = (TextView) findViewById(R.id.ergebnis);

        // ArrayAdapter mit einem Element pro Zeile
        // Ländernamen verwenden
        ArrayAdapter<CharSequence> adapter = ArrayAdapter.createFromResource
          (this, R.array.laendernamen, android.R.layout.simple_spinner_item);
        adapter.setDropDownViewResource
           (android.R.layout.simple_spinner_dropdown_item);
        auswahl.setAdapter(adapter);

        // Wir treffen eine Vorauswahl: Land Nr. 4
        auswahl.setSelection(4);
        CharSequence text = laenderkuerzel[4];
        ergebnis.setText(text);
        auswahl.setOnItemSelectedListener(new OnItemSelectedListener() {
          public void onItemSelected(
           AdapterView<?> parent, View view, int position, long id) {
            showToast("Auswahl: position=" + position + " id=" + id);
            CharSequence text = laenderkuerzel[position];
            ergebnis.setText(text);
          }

          public void onNothingSelected(AdapterView<?> parent) {
            showToast("Keine Auswahl");
          }
        });

      }

    }
```

Lösung zu Fall 2: Jede Zeile enthält mehr als eine Komponente

In dieser Kurseinheit wollen wir eine Liste bearbeiten, bei der jede Zeile aus 3 Teilen besteht: Ländername, Kürzel sowie eine editierbare *RatingBar* für das jeweilige Land. Listing 7.32 zeigt den Aufbau. Wie in Fall 1 benützen wir einen Spinner zur Auswahl. Der Spinner zeigt nur einen kompletten Datensatz aus den genannten 3 Teilen an und erinnert an die sog. Combobox bei der GUI für den PC. Diese *View* erlaubt eine Interaktion mit den darin enthaltenen Steuerelementen. In unserem Beispiel können wir die *RatingBar* für das ausgewählte Bundesland editieren. Bei der Auswahl zeigt der Spinner wie in Fall 1 nur einen Text für den jeweiligen Eintrag an. Der Text muss den Eintrag eindeutig identifizieren. Dazu wählen wir den Namen des jeweiligen Bundeslandes.

Bei dieser Aufgabe können wir für die einzelnen Zeilen kein vorgefertigtes Layout aus der Palette von Android verwenden, sondern müssen ein eigenes Layout erstellen. Wenn Android einen der Datensätze für die Länder aus unserem Array präsentiert, dann ruft es die Methode `getView(int position, View convertView, ViewGroup parent)` auf. Diese Methode muss dann eine fertige *View* liefern, deren Layout wir in Listing 7.33 definieren. Außerdem müssen wir die Steuerelemente der *View* für eine Zeile mit den Daten

für das entsprechende Land versorgen. Dazu ermittelt man aus der `id` des Layouts die entsprechenden Steuerelemente für das Kürzel, den Namen und die RatingBar.

Problem: Wie erhalten wir zu einer View gehörende Daten?

Android erlaubt die Markierung einer View mit einem sog. *tag*. Als *tag* verwenden wir die Daten für das entsprechende Land. Wenn wir in einer Methode, z.B. bei Reaktion auf die RatingBar, diese View erhalten, können wir die geänderten Daten dem entsprechenden Objekt zuordnen und das neue Rating für das Land setzen.

Listing 7.32: Daten für ein Bundesland

```
public class Land {
  CharSequence name;
  CharSequence kuerzel;
  float rating;

  public Land(CharSequence name, CharSequence kuerzel, float rating) {
    this.name = name;
    this.kuerzel = kuerzel;
    this.rating = rating;
  }

  @Override
  public String toString() {
    return name.toString();// Identifikation des Landes
  }

}
```

Listing 7.33: Layout für eine Zeile: zeile.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal" >


    <TextView
        android:id="@+id/kuerzel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="left|center_vertical"
        android:padding="2dip"
        android:textSize="18sp" />

    <TextView
        android:id="@+id/name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="left|center_vertical"
        android:padding="2dip"
        android:textSize="18sp" />

    <RatingBar
        android:id="@+id/rating"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:numStars="3"
        android:rating="1"
        android:stepSize="1" />

</LinearLayout>
```

Listing 7.34: Fall 2: mehrere Steuerelemente in einer Zeile anzeigen

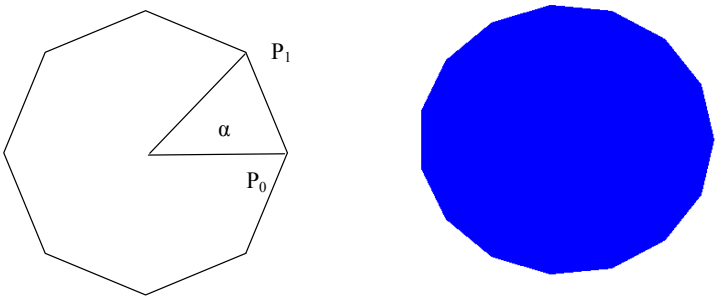
```
public class ActivityArrayAdapter2 extends Activity {

  private Land[] laender;

  // Aufbau der Länder anhand der Namen und der Kürzel
  void initLaender(Resources res) {
    CharSequence[] laenderkuerzel =
         res.getStringArray(R.array.laenderkuerzel);
    CharSequence[] laendernamen =
         res.getStringArray(R.array.laendernamen);
    laender = new Land[laenderkuerzel.length];
    for (int i = 0; i < laendernamen.length; i++) {
      Land land = new Land(laendernamen[i], laenderkuerzel[i], 1.0f);
      laender[i] = land;
    }
  }

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    initLaender(getResources());
    setContentView(R.layout.activity_array2);
    Spinner auswahl = (Spinner) findViewById(R.id.auswahl);
    ArrayAdapter<Land> adapter =
       new LandArrayAdapter(this, R.layout.zeile, laender);
    auswahl.setAdapter(adapter);
    adapter.setDropDownViewResource
       (android.R.layout.simple_spinner_dropdown_item);
  }

  // Ein ArrayAdapter speziell für unsere Daten: Land
  public class LandArrayAdapter extends ArrayAdapter<Land> {
    private LayoutInflater inflator;

    public LandArrayAdapter(Context ctx, int res, Land[] obj) {
      super(ctx, res, obj);
      inflator = LayoutInflater.from(ctx);
    }

    // Die View muss für jede Zeile separat aufgebaut werden.
    // Dazu müssen wir die Steuerelemente finden und die
    // Werte für das jeweilige Land eintragen.
    // In der View notieren wir die Referenz auf ein Land.
    @Override
    public View getView(int position, View convertView,ViewGroup parent){
      if (convertView == null)
        convertView = inflator.inflate(R.layout.zeile, null);
      // Kennzeichen an die View anhängen
      convertView.setTag(laender[position]);
      // Suche die einzelnen Steuerelemente: name, kuerzel, rating
      LinearLayout p = (LinearLayout) convertView;
      TextView name = (TextView) p.findViewById(R.id.name);
      name.setText(laender[position].name);
      TextView kuerzel = (TextView) p.findViewById(R.id.kuerzel);
      kuerzel.setText(laender[position].kuerzel);
      RatingBar rb = (RatingBar) p.findViewById(R.id.rating);
      rb.setRating(laender[position].rating);

      // Beobachter für Änderungen am Rating eines Landes registrieren
      RatingBar.OnRatingBarChangeListener listener =
     new RatingBar.OnRatingBarChangeListener() {
        public void onRatingChanged
          (RatingBar ratingBar, float rating, boolean fromTouch) {
          LinearLayout parent = (LinearLayout) ratingBar.getParent();
          // Anhand des oben angehängten "tags": Wir finden das Land
          Land tag = (Land) parent.getTag();
          tag.rating = rating; // Wir setzen das Rating für das land neu
```

```
            }
          };
          rb.setOnRatingBarChangeListener(listener);

          return convertView;
        }

      }

    }
```

7.4 Aufgaben

Aufgabe 7.1: Ein reguläres n-Eck

Eine App für Android soll ein reguläres n-Eck zeichnen. Der Mittelpunkt des n-Ecks soll der Mittelpunkt der Zeichenfläche sein. In der Skizze sehen Sie die Problemanalyse für n = 8 sowie einen Probelauf für n = 15.



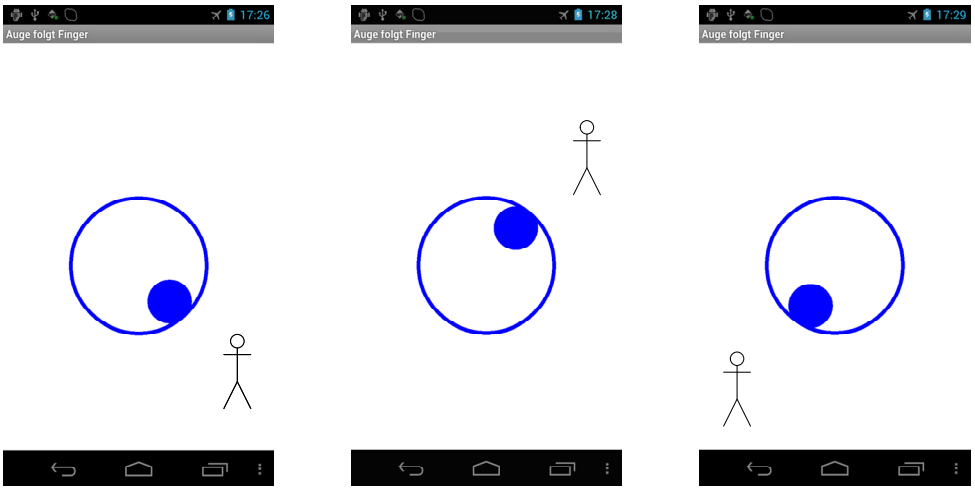
Das reguläre n-Eck ist ein Polygon mit n Punkten P_0, P_1, P_{n-1}. Die Punkte P_i liegen auf einem Kreis um den Mittelpunkt (mx, my) mit Radius r. Der Winkel α ist dann α = 360 / n (im Gradmaß). Damit ergeben sich für den Punkt P_i = (pxi, pyi) die Koordinaten

pxi = mx + 1 * cos (α*i) pyi = my + r* sin (α*i)

Tipp: Fügen Sie die so erhaltenen Punkte P_0, P_1, P_{n-1} zu einem `android.graphics.Path` namens `p` hinzu: den ersten Punkt mit `p.moveTo (…)`, weitere Punkte mit `p.lineTo (…)`. Zeichnen Sie den Pfad `canvas.drawPath(p, paint)`. Android füllt den geschlossenen Pfad.

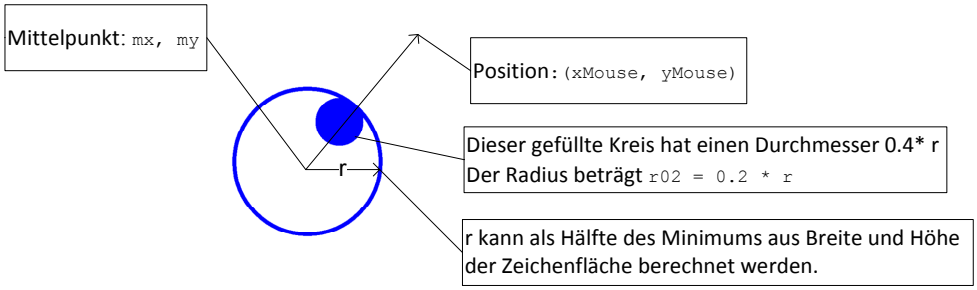
Aufgabe 7.2: Das magische Auge folgt dem Finger

Die Activity `XEye` besteht aus einem „Auge“, das scheinbar der Bewegung des Fingers folgt, wie die folgenden drei Momentaufnahmen zeigen. Die Strichzeichnung soll die Position (des Fingers) des Betrachters markieren.



Scheinbar folgen wir dem Finger, indem wir nach jeder Bewegung des Fingers auf dem Bildschirm einen gefüllten Kreis (als Augapfel) in Richtung der Position des Fingers innerhalb eines Kreises (das Auge) zeichnen.

Zur Berechnung:



`(mx, my)` bezeichnet den Mittelpunkt der Zeichenfläche: `mx = Breite/2`, `my = Höhe/2`. Für den Mittelpunkt `(mx1, my1)` eines gefüllten Kreises vom Radius `r02 = 0.2 * r` gilt:

```
mx1 = xMouse / Abstand * r * 0.8
my1 = yMouse / Abstand * r * 0.8
```

„Abstand“ ist der Abstand des durch Position gegebenen Punktes vom Mittelpunkt der Zeichenfläche. Alle Koordinaten sind hier in Bezug auf den Mittelpunkt gegeben.

Aufgabe 7.3: Ein reguläres n-Eck mit Farbe

Bei einem Klick *innerhalb des n-Eck* (vgl. Beschreibung der Klasse `Path`!!!!) soll das n-Eck die Farbe wechseln. Setzen Sie hierzu die Farbe im Graphikkontext `Paint paint` auf die jeweils nächste Farbe (wie unten angegeben, Nachfolger von rot wird wieder schwarz).

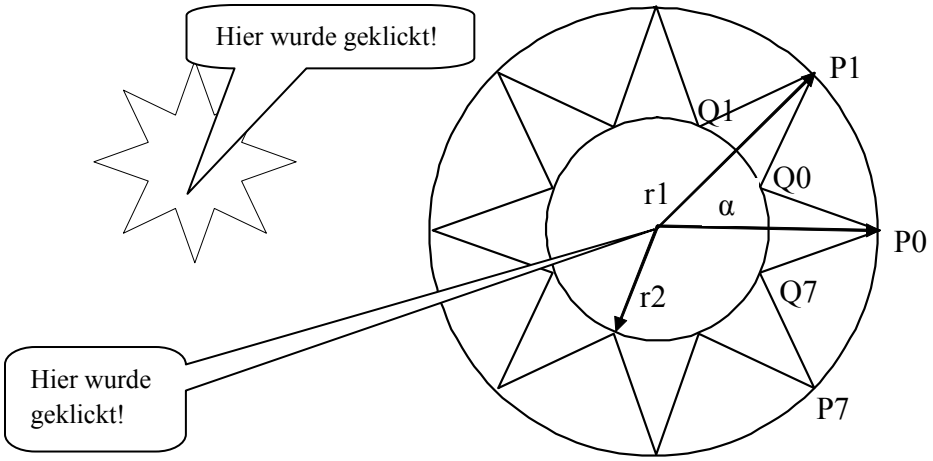
```
private int[] colors = {
    Color.BLACK,
    Color.BLUE,
    Color.CYAN,
    Color.DKGRAY,
    Color.GRAY,
    Color.GREEN,
    Color.LTGRAY,
    Color.MAGENTA,
    Color.YELLOW,
    Color.CYAN,
    Color.RED,
  };
```

Tipp zum Testen, ob ein Klick innerhalb eines Bereichs vorliegt: Benützen Sie die Klasse `android.graphics.Region`. Zur Konstruktion von Objekten dieser Klasse benötigt man ein Rechteck. Dazu nimmt man einfach den ganzen Bildschirm.

Aufgabe 7.4: Sterne zeichnen

Das unten angegebene Programm soll auf Klick an der Stelle, auf die geklickt wurde, einen Stern mit n Ecken zeichnen. Dabei sollen die Ecken (wie der Punkt `P0`) auf einem Kreis mit Radius `r1` um den Punkt der Zeichenfläche liegen, auf den geklickt wurde. Die Zwischenpunkte (wie der Punkt `Q0`) sollen auf einem Kreis mit Radius `r2` < `r1` liegen. Wählen Sie `r1 = 10`, `r2 = 20` Pixel.

Skizze:



Beispiel: der Stern mit 8 Zacken Zur Anleitung

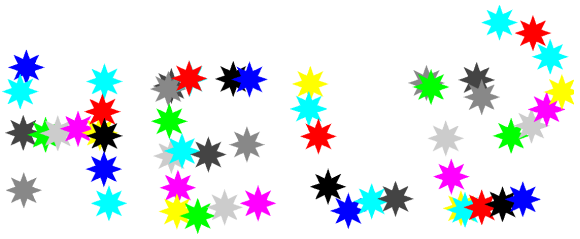
Anleitung

Der Stern mit n Ecken ist ein Polygon mit 2*n Punkten P_0, Q_0, P_1, Q_1, P_{n-1}, Q_{n-1}. Die Punkte P_i liegen auf einem Kreis um den Mittelpunkt (mx, my) mit Radius r1. Der Winkel beträgt dann α = 360 / n (im *Gradmaß*). Damit ergeben sich für den Punkt P_i = (pxi, pyi) bzw. Q_i = (qxi, qyi) die Koordinaten

```
pxi = mx + r1 * cos (•*i)          pyi = my + r1 * sin (•*i)
qxi = mx + r2 * cos (•*i + •/2)    qyi = my + r2 * sin (•*i + •/2)
```

Die so erhaltenen Punkte P_0, Q_0, P_1, Q_1, P_{n-1}, Q_{n-1} sollen zu einem Pfad `Path` p der Klasse `android.graphics.Path` hinzugefügt werden. Dieser Pfad p kann dann mit `canvas.drawPath(p, paint)` gezeichnet werden.

Zur Umsetzung in Java: Benutzen Sie `double Math.sin(double)`, `double Math.cos(double)` und hierfür zur Umrechnung vom *Gradmaß* ins *Bogenmaß* `double Math.toRadians(double)`.



Aufgabe 7.5: Game of Life nach Conway (Projektcharakter, aufwendige Lösung)

„Schauplatz des Spiels ist ein unendliches zweidimensionales Gitter aus quadratischen Zellen, die je nach dem Zustand der acht Nachbarfelder tot oder lebendig sind. Die Zeit verstreicht in diskreten Schritten. Von einem Schlag der kosmischen Uhr bis zum nächsten verharrt die Zelle im zuvor eingenommenen Zustand, beim Gong aber wird nach den folgenden einfachen Regeln erneut über Leben und Tod entschieden:

- Eine Zelle, die zur Zeit t tot war, wird dann und nur dann zur Zeit t+1 lebendig, wenn genau drei ihrer acht Nachbarn zur Zeit t gelebt haben.
- Eine Zelle, die zur Zeit t gelebt hat, stirbt zur Zeit t+1 dann und nur dann, wenn zur Zeit t weniger als zwei oder mehr als drei Nachbarn am Leben waren.“

(Nach [Dew88], Seite 187). Realisieren Sie Conways Game of Life in Java.

Aufgabe 7.6: Acht Damen (Projektcharakter)

Acht Damen sind so auf ein Schachbrett zu setzen, dass sie sich gegenseitig nicht bedrohen. Geben Sie für dieses klassische Problem eine Lösung in Java an. Auf Knopfdruck sollte die jeweils nächste Lösung angezeigt werden.