Apps programmieren

Sofern Sie dieses Buch sequenziell lesen (was nicht unbedingt erforderlich ist), sind Sie aus den vorangegangenen Kapiteln so weit mit den internen Zusammenhängen von Android und der Android-Programmierung vertraut, dass Sie jetzt eigentlich loslegen können. Bevor wir aber so richtig anfangen, wollen wir auf eine Reihe grundlegender Details der Programmierpraxis eingehen.

App-Komponenten

Die wichtigsten Bestandteile jeder Android-Anwendung sind Java- oder Kotlin-Klassen, die man als *App-Komponenten* bezeichnet und die gewissermaßen die Grundbausteine der Anwendung bilden. Im Folgenden wollen wir uns zu jedem dieser Komponententypen ansehen, wie er innerhalb einer Anwendung zu verwenden ist.

App-Komponenten sind bestimmte Arten von Java-Klassen, die als Grundbausteine die wesentliche Funktionalität jeder Anwendung definieren und die Schnittstelle zum System darstellen. Die Kommunikation mit ihnen muss in einer bestimmten Art über Framework-Funktionen erfolgen. Auf diese Weise werden die Hauptbestandteile einer Anwendung nur lose aneinandergebunden und können leicht mit App-Komponenten anderer Anwendungen kommunizieren oder sogar durch sie ersetzt werden; außerdem behält das Android-Framework jederzeit die volle Kontrolle über das Geschehen.

Es gibt folgende Arten von Komponenten:

- Eine *Activity* stellt die visuelle Programmoberfläche auf dem Bildschirm dar und reagiert auf die Eingaben der Anwender. Activities repräsentieren den sichtbaren Teil der Anwendung.

- Ein *Service* hat keine eigene Oberfläche, sondern stellt entweder Funktionalität für andere Komponenten zur Verfügung oder läuft parallel im Hintergrund.
- Ein *Broadcast-Receiver* reagiert auf Ereignisse, die vom System oder anderen Komponenten ausgelöst werden.
- Ein *Content-Provider* bietet über eine standardisierte Schnittstelle anderen Komponenten die Möglichkeit zum Lesen und Schreiben von Informationen.
- Die *Application* ist eine Komponente, die den Gesamtzustand der Anwendung repräsentiert.

Die Komponenten sind im Manifest zu deklarieren und werden (mit Ausnahme der Broadcast-Receiver) nur von den zur Plattform gehörenden spezifischen Manager-Klassen instanziiert, nicht von der Anwendung selbst. Die drei Komponententypen Activity, Service und Broadcast-Receiver werden über sogenannte *Intents* aktiviert. Ein Intent ist eine Instanz der Klasse `android.content.Intent` und kann als eine Art Mitteilung angesehen werden, die eine bestimmte zu startende Komponente, die Art einer auszuführenden Funktionalität oder ein aufgetretenes Ereignis benennt und dazu begleitende Informationen enthält. Das Betriebssystem stellt fest, welche Komponente am ehesten als Empfänger infrage kommt, startet sie und gibt bei Bedarf nach ihrer Beendigung einen Rückgabewert an den Aufrufer zurück. Wenn ein Intent von einer Android-Anwendung zu einer anderen – und damit zwischen zwei virtuellen Maschinen – übertragen werden muss, sorgt das Framework automatisch für die erforderliche Serialisierung und Deserialisierung der von einem Prozess zum anderen zu übertragenden Daten.

Komponenten des Typs Content-Provider werden nicht über einen Intent aktiviert, sondern automatisch, sobald eine andere Komponente über einen *Content-Resolver* auf die von ihm angebotenen Informationen zugreifen will.

Wenn eine Komponente ausgeführt werden soll, stellt das Android-Framework zuerst fest, ob die zugehörige Anwendung bereits läuft. Ist das nicht der Fall, erzeugt es aus einem vorbereiteten Template (»Zygote«) einen neuen Prozess mit einer bereits geladenen virtuel-

len Maschine und lädt darin die Anwendung. Dann instanziiert es die Komponente und ruft verschiedene ihrer Funktionen auf, die ihren Lebenszyklus markieren und es den Programmierern erlauben, durch Überschreiben ihre eigene Funktionalität zur Anwendung zu bringen.

Eigene Komponenten implementieren

Es gibt einige Aspekte, die unabhängig vom Komponententyp beim Bau eigener Komponenten zu beachten sind.

Ereignisfunktionen und Lebenszyklus

Alle Komponenten-Basisklassen sind nach dem gleichen Muster entworfen: Sie bieten eine Reihe von Funktionen, deren Namen in der Regel mit `on` beginnen und die dazu dienen, von der konkreten Implementierung überschrieben zu werden. Mit diesen Funktionen werden immer bestimmte Ereignisse gemeldet, auf die eine Komponente irgendwie reagieren kann. Ein Teil dieser Ereignisfunktionen bezieht sich auf die verschiedenen Phasen im Lebenszyklus, z.B. ihre Erzeugung (`onCreate()`) oder ihre Beendigung (`onDestroy()`).

Die Lebenszyklusfunktionen sollten, wie in diesem Beispiel, in der Regel als Erstes ihre Parent-Implementierung aufrufen:

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

Neben den Ereignisfunktionen stellen sie den implementierenden Klassen diverse Funktionen zur Verfügung, mit deren Hilfe sie systembezogene Aktionen auslösen können, wie etwa auf Dateien zugreifen oder andere Komponenten starten. Bei Activities und Services sowie dem `Application`-Objekt sind viele von ihnen von der Parent-Klasse `Context` geerbt.

Komponenten und Manifest

Jede Komponente muss beim Android-System angemeldet sein, damit sie funktionieren kann. Sie instanziieren nicht selbst Komponen-

ten und greifen nicht von außen auf sie zu, daher wird das Framework benötigt, um Komponenten bei Bedarf hochzufahren und Zugriffe auf sie zu vermitteln, und daher muss jede Komponente dem System im Manifest bekannt gemacht werden (die einzige Ausnahme ist der Komponententyp Broadcast-Receiver, der aus dem Programm heraus angemeldet werden kann und auf den man direkt zugreifen kann, wenn er im selben Prozess läuft).

Tags für Komponenten

Die Anmeldung einer Komponente im Manifest erfolgt mit einem der Tags `<activity>`, `<service>`, `<receiver>` oder `<provider>` innerhalb des Tags `<application>`, z. B. so:

```
<application . . .>
   <activity
     android:name=".MeineKomponente"
     android:enabled="true"
     android:exported="true"
     android:icon="@drawable/meine_komponente_icon"
     android:label="@string/meine_komponente_label"
     android:name="@string/meine_komponente"
     android:permission="com.example.hallo.MEINE_ERLAUBNIS"
     android:process="com.example.hallo.MEIN_PROZESS" >
     ...
   </activity>
</application>
```

Gemeinsame Attribute

Die vier Komponententags haben jeweils eigene Attribute; viele gelten jedoch für alle in gleicher Weise. Die wichtigsten von ihnen wollen wir in Tabelle 4-1 kurz benennen, um sie nicht bei jedem Komponententyp wiederholen zu müssen (allen Attributnamen muss das Präfix `android:` vorangestellt werden).

Tabelle 4-1: Standardattribute für die Anmeldung von Komponenten

Attributname	Bedeutung	Standardwert
`name`	Voll qualifizierter Name der implementierenden Klasse; wenn am Anfang ein Punkt steht, wird das Wurzel-Package der Anwendung vorangestellt. Muss explizit angegeben werden.	Kein Standardwert; der Name der Klasse muss immer angegeben werden.

Tabelle 4-1: Standardattribute für die Anmeldung von Komponenten (Forts.)

Attributname	Bedeutung	Standardwert
`exported`	Wenn `"true"`, darf die Komponente außerhalb der eigenen Anwendung genutzt werden.	`"true"`, wenn ein Intent-Filter angegeben ist, sonst `"false"`.
`enabled`	Wenn `"false"`, ist die Komponente gesperrt und kann nicht vom System instanziiert werden.	`"true"`, d. h., die Komponente ist nicht gesperrt.
`icon`	Symbolbild für die Komponente als Verweis auf eine Bildressource (`"@drawable/..."`).	Symbol der Anwendung.
`label`	Titel der Komponente als Verweis auf eine Textressource (`"@text/..."`).	Titel der Anwendung.
`permission`	Name eines Zugriffsrechts, über das aufrufende Komponenten verfügen müssen, wenn sie diese Komponente nutzen wollen.	Zugriffsrechte der Anwendung.

Untergeordnete Tags

Das untergeordnete Tag `<Intentfilter>` beschreibt bei Komponenten der Typen Activity, Service und Broadcast-Receiver die Eigenschaften, die ein Intent haben muss, damit er die jeweilige Komponente aufrufen kann (siehe den Abschnitt »Intent-Filter« auf Seite 117). Wenn kein Intent-Filter angegeben ist, kann die Komponente nur über ihre Klasse aufgerufen werden.

Ein weiteres untergeordnetes Tag namens `<meta-data>` dient der Angabe beliebiger Name/Wert-Paare, die von der Klasse in Form eines `Bundle` in `this.applicationInfo.metaData` in Empfang genommen werden können. Als Wert kann ein String oder eine Ressource angegeben werden, wobei im zweiten Fall nicht der Inhalt, sondern die ID-Nummer der Ressource übergeben wird:

```
<meta-data android:name="city"
               android:value="Berlin" />
<meta-data android:name="city"
               android:resource="@string/city_b" />
```

Der Kontext

Der *Kontext* ist ein Objekt, mit dessen Hilfe man den allgemeinen Zustand der Anwendung und des unterliegenden Systems abfragen und verschiedene Systemaktionen auslösen kann. Es implementiert die abstrakte Klasse `android.content.Context`.

`Context` enthält eine Fülle von Funktionen, die unter anderem folgenden Zwecken dienen:

- Kommunikation mit anderen Komponenten in derselben und anderen Anwendungen
- Prüfung von Zugriffsrechten
- Zugriff auf die interne Datenbank
- Zugriff auf das interne und externe Dateisystem
- Zugriff auf Metadaten der Anwendung
- Zugriff auf Ressourcen
- Zugriff auf Benutzerpräferenzen
- Zugriff auf Themes
- Zugriff auf aktive Hintergründe (Wallpaper)

Auf viele dieser Funktionen werden wir in den weiteren Kapiteln dieses Buchs eingehen.

Die Komponententypen `Activity`, `Service` und `Application` sind von der Klasse `Context` abgeleitet und bieten daher die Möglichkeit, deren Funktionen direkt aufzurufen. Dabei ist allerdings zu beachten, dass es sich um unterschiedliche Instanzen handelt, deren Lebenszeit jeweils an die der Komponente gekoppelt ist. Zwar ist es bei vielen Aktionen egal, von welcher Instanz sie aufgerufen werden, sobald Sie aber irgendwelche Systemressourcen an einen Kontext binden, müssen Sie berücksichtigen, dass eine Activity bzw. ein Service eine begrenzte Laufzeit hat, während das `Application`-Objekt vorhanden ist, solange die Anwendung als Ganzes aktiv ist. Andernfalls kann es zum einen passieren, dass eine benötigte Ressource nicht mehr vorhanden ist, sobald die entsprechende Komponente vom Garbage Collector abgeräumt worden ist, und zum anderen kann es leicht zu einem Speicherleck kommen.

Den dauerhaften Kontext des `Application`-Objekts, auch *Anwendungskontext* genannt, erhalten Sie in jedem spezifischen Kontext – also auch in jeder Activity und jedem Service – durch die Property `applicationContext`.

Es ist möglich, ein eigenes Anwendungsobjekt in Form einer von `android.app.Application` abgeleiteten Klasse zu definieren und im Manifest anzumelden (Attribut `android:name` im `<application>`-Tag). Damit soll eine Möglichkeit geboten werden, anwendungsweite Zustandswerte zu speichern. Davon wird jedoch selten Gebrauch gemacht, da man für den gleichen Zweck auch eine beliebige mit dem Anwendungskontext initialisierte Singleton-Klasse verwenden kann.

Intents und Parcels

Die Komponenten einer Android-Anwendung laufen weitgehend unabhängig voneinander, dürfen gegenseitig keine Referenzen haben und nicht gegenseitig auf Funktionen oder Felder zugreifen. Trotzdem können sie miteinander kommunizieren, und das auch dann, wenn sie in unterschiedlichen Anwendungen – und damit unterschiedlichen Prozessen – laufen. Für drei der vier Komponententypen (und zwar Activities, Services und Broadcast-Receiver) geschieht dies mithilfe von Intents; für die vierte Art (Content-Provider) gibt es einen anderen Mechanismus.

Die Rolle der Intents

Ein *Intent* ist eine passive Instanz der Klasse `android.content.Intent`, die gewissermaßen einen Auftrag oder eine Meldung an eine andere Komponente in derselben oder einer anderen Anwendung verkörpert. Dabei gibt es zwei Varianten:

- Ein *konkreter Intent* benennt genau die Java-Klasse, die den Auftrag ausführen oder die Meldung erhalten soll.
- Ein *abstrakter Intent* benennt nur die Art der Aufgabe, die es auszuführen gilt, beispielsweise eine Webseite anzuzeigen oder eine Telefonnummer zu wählen; oder die Art der Meldung, die

verarbeitet werden soll, z.B. dass die Batterie schwach ist. Die Auswahl der konkreten Implementierung bleibt aber dem Framework überlassen.

Neben den Angaben zur Adressierung der richtigen Komponente zur Ausführung enthält der Intent noch weitere Informationen, also beispielsweise die Adresse der anzuzeigenden Webseite oder die zu wählende Telefonnummer. Das Framework sucht dann anhand der jeweiligen Manifest-Informationen nach bestimmten Regeln die Komponente heraus, die die im Intent formulierte Aufgabe am besten erledigen kann. Bei einem konkreten Intent ist das die Klasse mit dem angegebenen Namen, bei einem abstrakten Intent ist es diejenige Komponente, die mit einem zum Intent passenden Intent-Filter angemeldet ist und die höchste Priorität hat.

Ist die passende Komponente gefunden, wird sie gegebenenfalls erst einmal gestartet; wenn die Anwendung, zu der sie gehört, noch nicht läuft, muss diese zuvor auch noch hochgefahren werden. Nachdem das geschehen ist, wird der Intent der Komponente in einem Funktionsaufruf zur Weiterverarbeitung übergeben.

Serialisierung mit Parcels

Da die Absenderkomponente und die Empfängerkomponente zu verschiedenen Anwendungen gehören können, die in verschiedenen virtuellen Maschinen laufen, muss der Intent mitsamt Inhalt zur Übertragung erst serialisiert und dann wieder deserialisiert werden können. Dafür steht in Android ein spezieller Mechanismus namens *Parcels* zur Verfügung.

Ein Parcel (`android.os.Parcel`) ist im Prinzip nichts weiter als ein interner binärer Speicher mit einer Unzahl von Funktionen, mit denen Werte unterschiedlichster Art in diesen Speicher geschrieben bzw. aus ihm ausgelesen werden können. Allerdings können in einem Parcel nur Werte verpackt werden, die *parcelable* sind, von denen also bekannt ist, wie sie in eine Bytefolge verwandelt werden können. Im Wesentlichen sind das diese hier:

- Elementare Typen: `Boolean`, `Byte`, `Int`, `Long`, `Double`, `Float`, `Char` und `String`.

- Objekte von Typen, die das Interface `android.os.Parcelable` implementieren und einige weitere Regeln einhalten, sodass sie in Parcels geschrieben und aus ihnen ausgelesen werden können. Dazu gehört auch die Klasse `android.os.Bundle`.
- Arrays, Listen und Maps, die ihrerseits nur Parcelable-Objekte enthalten.

Wie man eigene Parcelable-Klassen baut, ist im Detail beschrieben in der Dokumentation zur Parcel-Klasse unter *https://developer.android.com/reference/android/os/Parcel.html*.

Inhalt eines Intents

Im Einzelnen kann ein Intent die folgenden Informationen aufnehmen.

Expliziter Intent: die Zielkomponente

Die Bezeichnung der Komponente, die die gewünschte Aktion ausführen soll, wird nur bei expliziten Intents benötigt und ist dann alleiniges Kriterium für deren Adressierung.

Wenn die Zielkomponente zur selben Anwendung gehört, gibt man sie durch den aktuellen Kontext und eine Referenz auf das Klassenobjekt an:

```
val intent = Intent(this, AndereActivity::class.java)
```

Komponenten in fremden Anwendungen kann man auf diese Weise nicht ansprechen, da man dann in der Regel keinen Zugriff auf die entsprechenden Klassen hat. In solchen Fällen kann man die Zielkomponente mit einer Setter-Funktion setzen, die Zeichenketten für die App-ID und den Activity-Namen anstelle des Kontexts und der direkten Klassenreferenz akzeptiert:

```
val intent = Intent()
intent.setClassName("com.example.andereapp",
    "com.example.andereapp.AndereActivity")
```

Impliziter Intent: Aktion und Daten

Für einen abstrakten Intent muss man zumindest die Aktion benennen, die es auszuführen gilt, und meist auch einen Hinweis auf die Daten, auf die sich die Aktion bezieht.

Die auszuführende Aktion wird in Form eines Strings benannt. Einem selbst definierten Aktionsnamen sollte immer die Package-ID vorangestellt werden, z. B. so:

```
val intent = Intent("com.example.hallo.MEINE_AKTION")
```

Häufig kann man sich aber einer der zahlreichen Aktionsbezeichnungen bedienen, die in der Intent-Klasse vordefiniert sind, z. B.

- `Intent.ACTION_VIEW` – Daten anzeigen,
- `Intent.ACTION_EDIT` – Daten bearbeiten,
- `Intent.ACTION_DIAL` – Telefonnummer wählen oder
- `Intent.ACTION_SEND` – Daten versenden.

Inhalt und Bedeutung der zugehörigen Daten sind weitgehend von der definierten Aktion abhängig. Meistens werden sie als URI angegeben:

```
val intent = Intent(Intent.ACTION_VIEW,
    Uri.parse("http://www.oreilly.de"))
```

In diesem Fall der URI-Typ bekannt: Offenbar soll eine Webseite angezeigt werden. Es gibt aber für Android verschiedene spezielle URI-Typen, die meist auch einen engen Bezug zur ebenfalls angegebenen Aktion haben. Eine Liste von URI-Typen, die in Android verwendet werden, ist unter *http://www.openintents.org/intentsregistry* zu finden.

Anstelle der Daten oder zusätzlich zu den Daten muss in manchen Fällen auch ein MIME-Typ als String angegeben werden. Dafür gibt es die Intent-Funktionen `setType()` und `setDataAndType()`.

Extra-Daten

Neben dem Datenfeld und dem MIME-Typ bieten Intents noch die Möglichkeit, beliebige zusätzliche Informationen in Form von Schlüssel/Wert-Paaren zu speichern. Die Art dieser Informationen wird allein durch die Zielkomponente bestimmt.

Auch für die Schlüssel der Extra-Daten gibt es diverse vordefinierte Konstanten in der Klasse `Intent`.

Folgendes Beispiel zeigt, wie einem Mail-Intent Adressaten hinzugefügt werden:

```
intent.putExtra(Intent.EXTRA_EMAIL,
    arrayOf("meier@beispiel.com","schulz@beispiel.com"))
```

Da ein Intent zusammen mit allen seinen Inhalten serialisierbar sein muss, sind als Werte für die Schlüssel/Wert-Paare im Prinzip nur Datentypen erlaubt, die *parcelable* sind (siehe oben) oder `Serializable` implementieren.

Kategorien und Flags

Weiterhin kann man dem Intent noch eine oder mehrere Kategorien zuweisen. Das sind Strings, mit denen der Intent in irgendeiner Weise klassifiziert wird. Das Framework benutzt einige Kategorien, um Intents in besonderer Weise zu behandeln. So weist etwa die Kategorie `Intent.CATEGORY_HOME` darauf hin, dass die durch den Intent zu startende Activity als Startbildschirm des Geräts dient.

```
intent.addCategory(Intent.CATEGORY_HOME)
```

Schließlich gibt es noch die Möglichkeit, in einem Intent durch `setFlags()` verschiedene Flags zu setzen, um eine besondere Behandlung der Zielkomponente zu bewirken. Sie wird aber in der Regel nur Framework-intern genutzt.

Intent-Filter

Wie findet das Android-Framework nun heraus, welche Komponente aufgrund eines Intents aufgerufen werden soll, in dem nicht die Komponente selbst, sondern nur die auszuführende Aktion nebst begleitenden Informationen genannt ist? Android löst dieses Problem mithilfe des *Intent-Filters*, eines XML-Tags innerhalb der Definition einer Komponente im Manifest. Hier das Beispiel einer Haupt-Activity, wie wir sie schon oben gesehen haben:

```
<activity android:name=".MainActivity" . . . >
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER"
/>
  </intent-filter>
</activity>
```

Diese Activity müsste demnach durch einen Intent mit der Aktion `"android.intent.action.MAIN"` und der Kategorie `"android.intent.category.LAUNCHER"` gestartet werden. (Tatsächlich ist die Logik allerdings etwas anders: Der Launcher fragt einfach ab, welche Activities in einer Anwendung einen Filter wie oben definieren, und ruft diese dann über ihre App-ID und den Klassennamen auf.)

Grundsätzlich gibt es in einem Intent-Filter folgende als *Test* bezeichnete Filtermöglichkeiten, die in beliebiger Kombination – auch mehrfach – verwendet werden können.

Nach der Aktion:

```
<action android:name="android.intent.action.SEND" />
```

Nach der Kategorie:

```
<category android:name="android.intent.category.DEFAULT" />
```

Nach der Art der referenzierten Daten:

```
<data android:mimeType="video/mpeg" />
<data android:scheme="http" />
<data android:host="com.example.hallo.Beispiel" />
<data android:port="8080" />
<data android:path="/daten" />
```

Im `<data>`-Tag können die Attribute, die jeweils einen URI-Teil spezifizieren, auch kombiniert werden.

Ein Intent wird vom Intent-Filter akzeptiert, wenn *alle* folgenden Bedingungen zutreffen:

- Wenn im Intent eine Aktion angegeben ist, muss sie auch im Filter vorkommen – der Filter kann aber darüber hinaus auch noch weitere Aktionen bezeichnen.
- Alle im Intent genannten Kategorien müssen auch im Filter aufgeführt sein – im Filter können aber darüber hinaus auch noch weitere Kategorien genannt sein.
- Wenn im Intent-Filter `<data>`-Angaben enthalten sind, muss im Intent das Datenfeld gesetzt sein und einer dieser Angaben entsprechen.

Pending-Intents

Ein *Pending-Intent* (`android.app.PendingIntent`) ist ein Objekt, das einen kompletten Komponentenaufruf (z.B. den Start einer Activity) repräsentiert, der erst später ausgeführt werden soll. Dieses Objekt kann man einer anderen Anwendung übergeben, sodass diese den Komponentenaufruf zu einem späteren Zeitpunkt ausführt. Der Aufruf erfolgt dann im Namen des Absenders, es werden also die Rechte derjenigen Anwendung zugrunde gelegt, in der der Pending-Intent ursprünglich erzeugt worden ist.

Typische Anwendungsfälle sind die Übergabe eines Komponentenaufrufs an die Notifications (wird ausgeführt, wenn der Benutzer auf eine Meldung tippt) oder an den Alarm-Manager (wird zu einem festgelegten Zeitpunkt ausgeführt).

Eine Instanz der Klasse `PendingIntent` erhält man durch den Aufruf einer ihrer statischen Funktionen, die sich jeweils auf den aufzurufenden Komponententyp beziehen. So erzeugt folgende Zeile einen Pending-Intent zum Aufruf einer Activity aus der aktuellen Anwendung:

```
val intent = Intent(this,MeineActivity::class.java)
val pendingIntent =
    PendingIntent.getActivity(this,0,intent,0)
```

Als erster Parameter wird der Kontext der aufrufenden Komponente und als dritter der Intent für die zu startende Komponente angegeben. Der zweite und der vierte Parameter sind hier ohne Belang.

Den Pending-Intent können Sie, da er *parcelable* ist, beispielsweise in den Extra-Daten eines weiteren Intents an eine andere Anwendung übergeben. Diese ruft ihn dann irgendwann auf:

```
pendingIntent.send()
```

Damit wird der enthaltene Intent genau so und mit denselben Rechten ausgeführt, als würde er von der Komponente, die ihn ursprünglich angelegt hat, selbst mit `startActivity()` aufgerufen.

Berechtigungen deklarieren und prüfen

Zugriffe auf Funktionen und Daten des Android-Systems sind in vielen Fällen an Berechtigungen (*Permissions*) geknüpft, die eine App deklariert haben muss, um sie ausführen zu können. Liegen die Berechtigungen nicht vor, führen die Zugriffe zu keinem Ergebnis oder sogar zu einem Programmabsturz.

Im Folgenden behandeln wir den Umgang mit den von Android vordefinierten Systemberechtigungen. Eine App kann aber auch eigene Berechtigungen definieren.

Berechtigungen deklarieren

Wenn Ihre App eine bestimmte Berechtigung benötigt, beispielsweise um auf das Internet zuzugreifen, muss diese im Manifest deklariert sein. Das geschieht in einem `<uses-permission>`-Element unterhalb des Wurzelelements.

```
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.hallo">
    <uses-permission
        android:name="android.permission.INTERNET "/>
    . . .
</manifest>
```

Es gibt Varianten des Elements und Attribute, mit denen Sie die Berechtigung auf bestimmte Android-Versionen einschränken können, da die Berechtigungen nicht in allen Android-Versionen gleich behandelt werden.

Die Entwicklungsumgebung weist Sie bereits beim Programmieren darauf hin, wenn Sie eine Android-Funktion aufrufen, ohne die dafür notwendige Berechtigung deklariert zu haben.

Alle in Android definierten Systemberechtigungen sind in der Dokumentation zum Manifest unter *https://developer.android.com/reference/android/Manifest.permission.html* aufgelistet.

Beachten Sie, dass Sie mit der Deklaration einer Berechtigung möglicherweise implizit angeben, dass Ihre App ein bestimmtes Geräte-

feature voraussetzt. Dies gilt derzeit für Bluetooth, Kamera, Ortsbestimmung, Mikrofon, Telefonie und WLAN. Wenn Sie also beispielsweise die Berechtigung für den Zugriff auf das Mikrofon deklarieren, lässt sich Ihre App nicht auf Geräten installieren, die kein Mikrofon besitzen.

Normale und gefährliche Berechtigungen

Android unterscheidet zwischen »normalen« und »gefährlichen« Berechtigungen. Die normalen Berechtigungen werden beim Start automatisch ohne Rückfrage zugeordnet. Wenn Sie sie im Manifest korrekt deklariert haben, brauchen Sie sich im Programm nicht weiter darum zu kümmern.

Bei gefährlichen Berechtigungen ist die Zustimmung des Benutzers erforderlich, die seit Android 6.0 (API-Level 23) bei der Verwendung der betreffenden Funktion einzeln abgefragt werden muss. Bei älteren Android-Versionen muss der Benutzer bei der Installation pauschal zustimmen.

Die Berechtigungen sind in Gruppen aufgeteilt, und die Benutzerzustimmung gilt immer für eine ganze Berechtigungsgruppe. Trotzdem müssen Sie die Berechtigungen einzeln abfragen, da sich die Gruppenzuordnung der Berechtigungen in Zukunft ändern kann.

Alle gefährlichen Berechtigungen sind mit ihren Gruppen im Referenzteil unter »Manifest« aufgelistet.

Berechtigung abfragen

Damit die Berechtigungsabfrage so realisiert werden kann, muss die Activity, in der sie stattfindet, das Interface `ActivityCompat.OnRequestPermissionsResultCallback` ableiten.

```
class MainActivity : AppCompatActivity(),
        ActivityCompat.OnRequestPermissionsResultCallback {
    . . .
}
```

Sie stellen fest, ob eine bestimmte Berechtigung für die App bereits gegeben ist, indem Sie die statische Funktion `checkSelfPermission`

der Klasse ActivityCompat aus der Unterstützungsbibliothek V4 aufrufen (this ist hier die aktuelle Activity).

```
val erlaubnis = ActivityCompat.checkSelfPermission(this,
                Manifest.permission.INTERNET)
```

Wenn die Berechtigung bereits vorliegt, ist das Ergebnis die in PackageManager definierte Int-Konstante PERMISSION_GRANTED, und Sie können mit dem Programm fortfahren. Bei älteren Android-Versionen ist dies immer der Fall, da die Berechtigung dort schon bei der Installation abgefragt wird.

Ist das Ergebnis nicht PERMISSION_GRANTED, können Sie jetzt den Benutzer um Erlaubnis fragen. Dazu rufen Sie die Funktion requestPermission() auf und übergeben ihr neben einer Referenz auf die Activity ein Array aus den Namen der gewünschten Berechtigungen sowie eine selbst definierte Int-Konstante (hier INTERNET_ERLAUBNIS), die zur Identifikation Ihrer Anfrage dient.

```
ActivityCompat.requestPermissions(this,
    arrayOf(Manifest.permission.INTERNET),
    INTERNET_ERLAUBNIS)
```

Die Funktion öffnet einen kleinen Dialog, in dem der Benutzer der ganzen Berechtigungsgruppe zustimmen kann oder nicht. Das Ergebnis erhalten Sie in einer Callback-Funktion namens onRequestPermissionsResult(), die Ihre Activity implementieren muss. Diese bekommt die selbst definierte Konstante, das Array mit den Berechtigungsnamen sowie ein Array mit den resultierenden Berechtigungen übergeben. Wenn Letzteres an der Array-Position der abgefragten Berechtigung den Wert PackageManager.PERMISSION_GRANTED enthält, wurde der Berechtigung zugstimmt, und Sie können die gewünschte Funktion ausführen.

```
override fun onRequestPermissionsResult(
    code:Int,
    permissions:Array<String>,
    results:IntArray) {
  if (code == INTERNET_ERLAUBNIS &&
      results.size==1 &&
      results[0]==PackageManager.PERMISSION_GRANTED) {
    // Erlaubnis erteilt, Funktion ausführen
      } else {
```

```
        // Funktion nicht ausführen
    }
}
```

Berechtigung erläutern

Vor dem Einholen der Berechtigung sollten Sie gegebenenfalls dem Benutzer erklären können, wozu Sie sie benötigen.

Ob das gewünscht ist, erfahren Sie durch die Funktion `shouldShowRequestPermissionRationale()`. Wenn sie `true` zurückgibt, zeigen Sie einen Dialog an, der den Sinn der Berechtigung erläutert.

```
if (ActivityCompat.shouldShowRequestPermissionRationale(this,
                    Manifest.permission.CAMERA)) {
    // Erläuterung asynchron anzeigen
}
```

Der schematische Ablauf der Berechtigungsabfrage sieht dann also etwa so aus:

```
val erlaubnis = checkSelfPermission(...)
if (erlaubnis != PERMISSION_GRANTED) {
    if (ActivityCompat.shouldShowRequestPermissionRationale(...) {
        // Erlaubnis erläutern, neu starten
    } else {
        // um Erlaubnis bitten
        ActivityCompat.requestPermissions(...)
        // Ergebnis in onRequestPermissionsResult()
    }
}
```

Multithreading

Beim Entwickeln von interaktiven Android-Programmen müssen grundsätzlich bestimmte Zusammenhänge beachtet werden, die den Umgang mit Parallelverarbeitung betreffen.

Android handelt die gesamte Benutzerinteraktion in einem einzigen Thread ab, dem sogenannten *UI-Thread*. Dieser darf niemals blockiert werden, damit das Gerät nicht träge erscheint oder – noch schlimmer – Timeout-Meldungen angezeigt werden. Wenn also die Abarbeitung einer Benutzeraktivität längere Zeit in Anspruch nimmt,

weil sie beispielsweise eine Internetkommunikation erfordert, sollte sie in einen gesonderten Thread ausgelagert werden.

Bei Netzwerkaktionen ist es sogar Pflicht, sie außerhalb des UI-Threads zu behandeln, andernfalls gibt es bereits beim Kompilieren einen Fehler.

Folgendes Beispiel zeigt eine Funktion, die eine durch einen Button-Klick ausgelöste Aktion in einen gesonderten Thread verlagert:

```
import kotlin.concurrent.*
. . .
fun onClick(v:View) {
  thread {
      // hier die längere Aktion ausführen
  }
}
```

Dieses Vorgehen reicht aus, wenn Sie damit eine Aktion starten, die keine Rückmeldung auf der Oberfläche erfordert, z.B. das Abspielen von Hintergrundmusik oder das Hochladen einer Datei.

Wenn Sie jedoch am Ende der Aktion ein Ergebnis anzeigen wollen, müssen Sie berücksichtigen, dass die Klassen der Android-Anwendungsoberfläche nicht threadsicher sind. Daher dürfen Sie niemals aus einem anderen als dem UI-Thread auf sie zugreifen. Das wiederum bedeutet, dass Sie in der obigen `run()`-Funktion beispielsweise nicht einfach ein Oberflächenelement holen und dort etwas hinschreiben dürfen.

Als Lösung für dieses Problem bietet Android verschiedene Möglichkeiten an. Einerseits gibt es einige Funktionen, mit denen man aus einem Extra-Thread heraus `Runnable`-Objekte in den GUI-Thread einspeisen kann, zum Beispiel die Funktion `runOnUiThread()` der `Activity`-Klasse.

Daneben gibt es eine Klasse `android.os.AsyncTask`, mit deren Hilfe Aktionen in den Hintergrund verlegt werden können, ohne dass man sich selbst um das Thread-Handling kümmern muss. Diese ist aber umständlich anzuwenden und gilt unter bestimmten Bedingungen als problematisch. Generell ist es sinnvoller, die Klassen des

Packages java.util.concurrent zu nutzen oder auf ein externes Concurrency-Framework zurückzugreifen

Multithreading mit Anko

Eine elegante Möglichkeit bietet das Coroutines-Paket der Anko-Bibliothek, das auf ein neues, noch experimentelles Feature von Kotlin zurückgreift.

Sie müssen dieses Paket in das modulspezifische Gradle-Script einbinden, wenn Sie nicht die Abhängigkeit von der gesamten Anko-Bibliothek bereits definiert haben.

```
 dependencies {
    implementation
        "org.jetbrains.anko:anko-coroutines:$anko_version"
}
```

Um eine Aktion im Hintergrund auszuführen, deren Ergebnis dann auf dem Bildschirm dargestellt wird, verwenden Sie die Anko-Funktion async().

```
import kotlinx.coroutines.experimental.*
import kotlinx.coroutines.experimental.android.*
import org.jetbrains.anko.coroutines.experimental.bg
. . .
button.setOnClickListener {
  async(UI) {
    val dd: Deferred<Daten> = bg {
      holeDaten()
    }
    zeigeDaten(dd.await())
  }
}
```

Daten ist hier ein beliebiger Wert-Datentyp. Die Funktion holeDaten() läuft im Hintergrund und besorgt diese Daten beispielsweise aus dem Internet. Die Funktion bg() liefert sie in Form eines Deferred-Objekts. Die Funktion zeigeDaten() wird wiederum im UI-Thread aufgerufen und zeigt die Daten auf dem Bildschirm an, sobald sie angekommen sind und dem Deferred entnommen werden können.

Das obige Beispiel soll die Zusammenhänge in vereinfachter Form erklären. Die Anko-Bibliothek stellt im Layouts-Paket darüber hinaus Erweiterungsfunktionen zur Verfügung, mit denen eine derartige Hintergrundverarbeitung direkt der Ereignisquelle (hier `button`) zugewiesen werden kann.

Logging

Insbesondere beim Debuggen von Anwendungen, die sich in der Entwicklung befinden, ist es sinnvoll, an kritischen Stellen des Programms eine Meldung auszugeben, damit man verfolgen kann, was die Anwendung gerade tut und in welcher Reihenfolge bestimmte Aktionen ausgeführt werden. Die `println()`-Funktion ist dafür aber ebenso wenig geeignet wie bekannte Java-Logger wie *Log4J*.

Stattdessen bietet das Android SDK ein eigenes Logging-Objekt namens `android.util.Log`. Man benutzt es, indem man eine seiner statischen Funktionen aufruft. Wir erläutern das am Beispiel der Funktion `Log.d()` für Debug-Meldungen.

`fun d (tag:String, msg:String, thr:Throwable?=null):Int` meldet einen Fehler mit dem Kennzeichen `tag` und der Meldung `msg`. Zusätzlich kann noch eine Exception angegeben werden, die durch den Fehler ausgelöst worden ist.

Die Kennung `tag` dient dazu, den Ort zu identifizieren, an dem die Meldung abgegeben wird, typischerweise also die Klasse und/oder die Funktion. Ein typisches Muster für das Logging sieht so aus:

```
class MainActivity : Activity() {
  companion object {
     const val TAG = "MainActivity"
  }
  . . .
  override fun onCreate(savedInstanceState:Bundle?) {
    super.onCreate(savedInstanceState)
    Log.d(TAG,"Activity wird erzeugt")
    . . .
  }
}
```

Die folgenden sechs Logfunktionen sind definiert, hier mit steigender Priorität aufgelistet:

- `Log.v()` (*verbose*) für besonders gesprächige Meldungen.
- `Log.d()` (*debug*) für Meldungen zum Debuggen.
- `Log.i()` (*information*) für normale Hinweise zum Ablauf.
- `Log.w()` (*warning*) für Hinweise auf verdächtige Situationen.
- `Log.e()` (*error*) für Fehlermeldungen.
- `Log.wtf()` (*what a terrible failure*) für fatale Fehler, die zu einem Abbruch der Anwendung führen.

Jede dieser Funktionen ist mit mindestens einer Variante überladen, in der die Nachricht oder die Exception weggelassen werden kann.

Die Logmeldungen können mit *Logcat* sichtbar gemacht werden, wobei man die anzuzeigenden Meldungen nach Priorität und Kennung filtern kann.

`Log.v()`-Aufrufe sollten vor der Veröffentlichung einer App entfernt werden. `Log.d()`-Aufrufe werden nur im Debug-Modus ausgeführt.

Logging mit Anko

Die Anko-Bibliothek unterstützt das Logging mit einigen Funktionen. Wenn Sie sie verwenden möchten, müssen Sie das betreffende Anko-Paket folgendermaßen im Gradle-Script anmelden:

```
dependencies {
    implementation
        "org.jetbrains.anko:anko-commons:$anko_version"
}
```

Am einfachsten nutzen Sie diese Erweiterung, indem Sie das Interface `AnkoLogger` verwenden, z. B. so:

```
import org.jetbrains.anko.*
. . .
class SomeActivity : Activity(), AnkoLogger { . . . }
```

Innerhalb der Klasse (es muss keine Activity sein) stehen dann Funktionen wie `verbose()`, `debug()`, `info()` usw. zur Verfügung. Die

Kennung wird bei diesen Funktionen nicht angegeben; sie ist standardmäßig gleich dem aktuellen Klassennamen, kann aber mit der Property `loggerTag` überschrieben werden.

```
info("Nachricht")
error("Fehler in...",exception)
```

Wenn Sie einen Text loggen wollen, dessen Zusammenstellung aufwendig ist, können Sie auch die *lazy* Form verwenden, bei der Sie die Nachricht als Lambda-Ausdruck angeben, der nur bei Bedarf ausgewertet wird.

```
debug { "Objekt ${meinObjekt.toString()} geladen." }
```