Arno Becker · Marcus Pant

Android

Grundlagen und Programmierung

Die 2. Auflage dieses Buchs
» Android 2«
erscheint Ende Mai 2010

dpunkt.verlag

Die 2. Auflage dieses Buchs

» Android 2«

erscheint Ende Mai 2010

Dipl.-Inform. Arno Becker ist als geschäftsführender Gesellschafter der visionera GmbH in Bonn verantwortlich für den Bereich »Mobile Business«. Er beschäftigt sich seit mehreren Jahren mit der Programmierung von Mobilgeräten und verfügt über umfangreiche Expertise im Bereich J2ME.

Dipl.-Inform. Marcus Pant ist geschäftsführender Gesellschafter der visionera GmbH in Bonn. Seine Schwerpunkte liegen in der Entwicklung von Java-EE-Systemen sowie dem agilen Projektmanagement.

Arno Becker · Marcus Pant

Android

Grundlagen und Programmierung

dpunkt.verlag

Lektorat: René Schönfeldt
Copy-Editing: Annette Schwarz, Ditzingen
Satz: Science & More, www.science-and-more.de
Herstellung: Nadine Berthel
Umschlaggestaltung: Helmut Kraus, www.exclam.de
Druck und Bindung: Koninklijke Wöhrmann B.V., Zutphen, Niederlande

Bibliografische Information Der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.ddb.de> abrufbar.

ISBN 978-3-89864-574-4

1. Auflage 2009

Ringstraße 19
69115 Heidelberg

5 4 3 2 1 0

Vorwort

Hinweis: Die 2. Auflage dieses Buchs erscheint unter dem Titel
»Android 2. Grundlagen und Programmierung«
Ende Mai 2010 im dpunkt.verlag, ISBN 978-3-89864-677-2

Am 12. November 2007 veröffentlichte Google eine Vorabversion des Android-SDK, einer Entwicklungsumgebung für die Android-Plattform. Die überwiegend positive Reaktion darauf verdeutlicht, wie groß das Interesse des Marktes und der Entwickler an einer offenen Plattform für mobile Computer mittlerweile ist. Computer und Mobiltelefon verschmelzen immer mehr zu einer Einheit. Die ersten Android-Geräte sind auf dem Markt. Seit kurzem gibt es auch Android-Netbooks mit UMTS-Modul.

Es findet ein Wandel statt, der das Mobiltelefon in einen tragbaren, internetfähigen Computer verwandelt, auf dem man ganz selbstverständlich und unkompliziert Anwendungen installiert. Diese Anwendungen schaffen den Mehrwert.

Apple hat mit dem iPhone gezeigt, dass ein Mobiltelefon gut bedienbar sein kann. Damit wurde ein Trend in Gang gesetzt, in den sich Android nahtlos eingereiht hat. Ein berührungsempfindlicher, großer Bildschirm und ein kostengünstiger Internetzugang führen zu neuen Geräten, die beim Kunden sehr gut ankommen.

»Mobiler Computer«...

Wir verzichten, wenn wir von Android-Geräten reden, bewusst auf das Wort »Mobiltelefon«. Stattdessen verwenden wir den Begriff »*Mobiler Computer*« (*MC*) in Anlehnung an *PC* (*Personal Computer*).

Ein Buch zu Android

Ziel: Grundprinzipien praktisch vermitteln

Wir werden in diesem Buch die Grundprinzipien von Android vorstellen. Dabei geht es uns nicht darum, die Dokumentation von Android abzuschreiben, sondern anhand von Codebeispielen einen zielgerichteten Blick auf die grundlegenden Themen der Softwareentwicklung mit dem Android-SDK zu werfen.

Wir konzentrieren uns auf Kernthemen, die fast jede Android-Anwendung benötigt: Oberflächen und Menüs, Datenübertragung, Datenspeicherung, Hintergrunddienste und lose Kopplung von Android-Komponenten. Weniger von Interesse sind für uns multimediale Themen, wie zum Beispiel Video, Audio oder die Verwendung der eingebauten Kamera. Wir werden die einzelnen Komponenten von Andro-

id kennenlernen und uns anschauen, wie diese miteinander interagieren. Wir erklären, was hinter den Kulissen von Android passiert und wie man mit diesem Wissen stabile und performante Anwendungen schreibt. Darüber hinaus werden wir weitere wichtige Themengebiete wie Location Based Services und Sicherheit behandeln und zeigen, wie man eine Android-Anwendung »marktreif« macht.

Warum dieses Buch?

Da Android noch relativ neu ist, sind die Quellen für Softwareentwickler noch etwas unübersichtlich. Einen ersten Einstieg in Android zu finden ist nicht einfach. Daher haben wir Bedarf gesehen, die Kernthemen von Android in einem deutschsprachigen Buch ausführlich vorzustellen.

Für wen ist dieses Buch?

Das Buch richtet sich in erster Linie an Entwickler. Grundkenntnisse der Programmiersprache Java sollten vorhanden sein. Alle Themen werden ausführlich in Theorie und Praxis behandelt, so dass eine solide Ausgangsbasis für die Entwicklung eigener Anwendungen vermittelt wird.

Neben Entwicklern wollen wir aber auch technische Projektleiter ansprechen. Viele Fragestellungen und Probleme des Mobile Business, wie z.B. die Themen Sicherheit und Verschlüsselung, werden in das Buch mit einbezogen.

Aufbau des Buchs

Teil I

Wir werden in Teil I des Buchs mit einem einfachen Beispiel beginnen, welches aber schon über die übliche Hello-World-Anwendung hinausgeht. Es stellt die wichtigsten Elemente einer Anwendung vor. Dem folgt ein wenig Theorie, die für das Verständnis von Android wichtig ist.

Teil II

In Teil II steigen wir weiter in die Praxis ein. An einem durchgängigen Beispiel stellen wir Kapitel für Kapitel wichtige Elemente des Android-SDK vor. Die Teile, die das Beispielprogramm um eine Fachlichkeit erweitern, haben wir als Iterationen gekennzeichnet. Themen, die nicht direkt Bestandteile des Beispielprogramms sind, als Exkurse.

Jede Iteration enthält einen theoretischen und einen praktischen Teil. Der theoretische Teil soll helfen, ein tieferes Verständnis für die Arbeitsweise der einzelnen Komponenten und Bestandteile von Android zu vermitteln. Im Praxisteil wenden wir dann das Wissen an.

Teil III

In Teil III befassen wir uns mit fortgeschrittenen Themen rund um die Android-Anwendungsentwicklung: Debugging, Anwendung »marktreif« machen, Sicherheit und Verschlüsselung, Optimierung und Performance sowie Aspekte der Kompatibilität.

Wie lese ich dieses Buch?

Wir empfehlen, das Einsteigerbeispiel in Teil I durchzugehen. Der Rest von Teil I ist theoretischer Natur und kann jederzeit separat gelesen werden.

Teil II sollte in der vorgegebenen Reihenfolge der Kapitel durchgearbeitet werden, da diese aufeinander aufbauen. Wir haben Wert auf eine ausführliche theoretische Behandlung der einzelnen Themen gelegt, damit ein Verständnis für die Funktionsweise der Android-Plattform entsteht.

Teil III kann isoliert betrachtet werden. Wer gleich von Beginn des Buchs an viel selbst mit den Codebeispielen experimentiert, kann sich ein paar gute Tipps in Kapitel 16 (Debugging und das DDMS-Tool) holen.

Zwei Emulatoren

Das Android-SDK bringt zwei verschiedene Emulatoren mit, die sich im Wesentlichen in der Bildschirmgröße unterscheiden. Wir haben in den Abbildungen jeweils den für die Darstellung des Programms besser geeigneten Emulator verwendet.

Zum leidigen Thema »Geschlechtsneutralität« halten wir es wie Peter Rechenberg in [21]: »Rede ich von ›dem Leser‹, meine ich ja keinen *Mann*, sondern einen *Menschen*, und der ist nun einmal im Deutschen grammatisch männlich. Selbstverständlich ist mit ›dem Leser‹ der männliche *und* der weibliche Leser gemeint.«

Die Website zum Buch

Auf der Website zum Buch `www.androidbuch.de` finden Sie den Quelltext der Programmierbeispiele, eventuelle Korrekturen, ein Glossar mit Android-Fachbegriffen sowie weiterführende Links zum Thema Android-Entwicklung.

Danksagung

Wir danken unseren Familien, Partnern, Freunden und Kollegen für die Unterstützung und die Geduld.

Ebenfalls danken möchten wir dem *dpunkt.verlag*, insbesondere Herrn Schönfeldt, für die angenehme und hilfreiche Zusammenarbeit.

Bedanken möchten wir uns bei allen Gutachtern, besonders bei Klaus-Dieter Schmatz, für die umfangreiche Begutachtung des Manuskripts und die vielen wertvollen und hilfreichen Hinweise.

Inhaltsverzeichnis

Hinweis: Die 2. Auflage dieses Buchs erscheint unter dem Titel
»Android 2. Grundlagen und Programmierung«
Ende Mai 2010 im dpunkt.verlag, ISBN 978-3-89864-677-2

Teil I

Einführung

1 Ein erstes Beispiel

Hinweis: Die 2. Auflage dieses Buchs erscheint unter dem Titel
»Android 2. Grundlagen und Programmierung«
Ende Mai 2010 im dpunkt.verlag, ISBN 978-3-89864-677-2

Ein Staumelder für die Hosentasche

In diesem Abschnitt werden wir ein erstes Android-Programm erstellen. Es dient dem schnellen Einstieg in die Programmierung von Android.

Es handelt sich um einen Staumelder für die Hosentasche. Die Funktionen dieses Programms beschreiben wir ausführlich in Kapitel 4. Hier reicht es uns erst mal zu wissen, dass wir mit dem Programm über Staus informiert werden, die auf dem Weg liegen, den wir gerade fahren. Zusätzlich gibt es die Möglichkeit, selbst Staus zu melden.

1.1 Projekt anlegen

Der generelle Umgang mit der Eclipse-Entwicklungsumgebung, dem Android-Plug-in und dem Emulator wird an dieser Stelle vorausgesetzt. Wir haben es uns nicht zum Ziel gesetzt, die vorhandenen Tutorials und von Google vorgegebenen Beispiele zu übersetzen, und verweisen gerne auf die gute Arbeit der Kollegen (siehe [18]).

Ein Projekt erstellen

Mit Hilfe des Android-Plug-in für Eclipse ist die Erstellung eines Projekts sehr einfach (Abb. 1-1). Zu Beginn legt man, wie bei jedem anderen Java-Projekt auch, den Namen des Pakets und die zu erstellende Klasse fest. In diesem Fall erzeugen wir eine sogenannte *Activity*. Activities implementieren die Bildschirmmasken einer Android-Anwendung. Die darstellbaren Elemente einer Seite werden als *Views* bezeichnet. Der Aufbau einer Seite wird meist in XML definiert. Die dort auszuführenden Aktionen werden in Java-Klassen implementiert. Insofern ähnelt der Aufbau von Android-Anwendungen dem von Webanwendungen.

Nach Beendigung des Eclipse-Wizard findet man die folgende Standard-Projektstruktur für Android-Projekte vor.

- *src*: Java-Quelltexte (u.a. auch unsere Acitvity `StaumeldungErfassen`)
- *res*: Ressourcen, d.h. alle Nicht-Java-Texte und alle Nicht-Programmelemente, wie zum Beispiel Bibliotheken. Hier werden u.a. die Dateien zur Definition der Oberflächen, Bilder oder Textdefinitionen abgelegt.

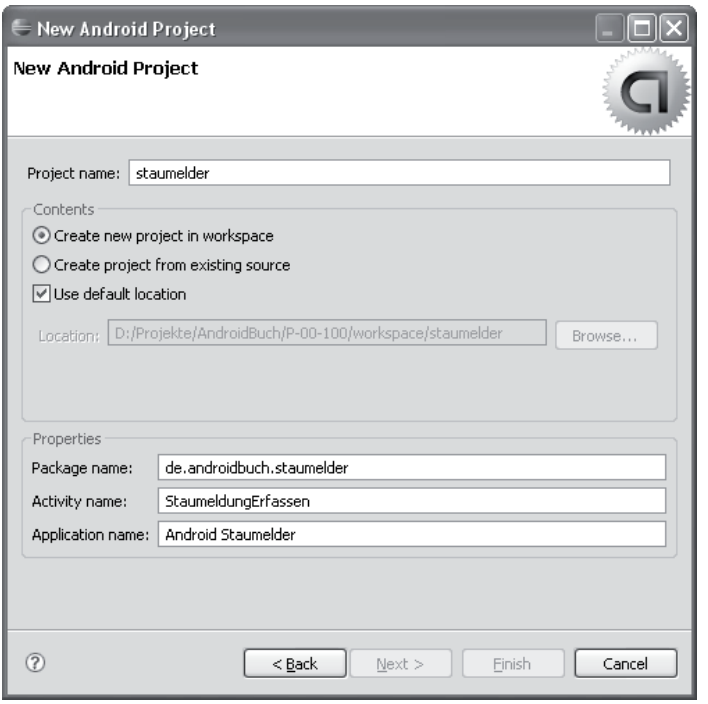


Abb. 1-1
Projekt anlegen

Im Wurzelverzeichnis befindet sich die zentrale Datei zur Definition von Metadaten der Anwendung: das `AndroidManifest.xml`.

1.2 Die erste Activity

Wir implementieren nun unsere erste Activity, die die Startseite unserer Anwendung anzeigen wird.

Listing 1.1
Activity »Staumeldung erfassen«

```
package de.androidbuch.staumelder;

import android.app.Activity;
import android.os.Bundle;

public class StaumeldungErfassen extends Activity {
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
  }
}
```

An dieser Stelle reicht es uns zu wissen, dass unsere eigenen Activities von der Android-API-Klasse *Activity* abgeleitet werden müssen. Activi-

ties implementieren die sichtbaren Bestandteile einer Anwendung und interagieren mit dem Anwender.

1.3 Layout definieren

XML GUI

Wenden wir uns nun der Erstellung unserer Eingabemaske zu. Die Maskenelemente werden in einer XML-Datei definiert. Der Vollständigkeit halber sei noch erwähnt, dass die Masken auch via Programmcode erstellt werden können. Dies ist aber, wie im Falle von Webanwendungen (JSP vs. Servlets), aus Gründen der Übersichtlichkeit und Wartbarkeit stets die zweite Wahl und wird daher nicht Thema dieses Buches sein.

Das Android-Plug-in hat bereits eine solche XML-Datei `res/layout/main.xml` erstellt, die in Listing 1.2 dargestellt ist.

Listing 1.2
Eine einfache main.xml

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

Ähnlich wie bei Swing-Anwendungen können verschiedene Layouts für den Aufbau der Maske verwendet werden. Beim Erstellen eines Android-Projekts wird automatisch ein *LinearLayout* mit vertikaler Ausrichtung gewählt, das wir zunächst übernehmen wollen.

Das XML-Tag `TextView` enthält ein Attribut `android:text`. Hier handelt es sich um einen Verweis auf eine Zeichenkettendefinition. Sie befindet sich im Ordner `values` des Android-Projekts und dort in der Datei `strings.xml`. Die Datei hat folgenden Inhalt:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="hello">Hello World,
        StaumeldungErfassen</string>
    <string name="app_name">Einstiegsbeispiel</string>
</resources>
```

Der Schlüssel für den Text, der in dem Anzeigeelement `TextView` dargestellt werden soll, lautet »`hello`«. Abbildung 1-2 zeigt die Ansicht der Start-Activity im Emulator.

Abb. 1-2
Die erste Activity im Emulator

Der nächste Schritt ist nun, dieses Layout für unsere Zwecke anzupassen. Dazu überlegen wir uns, welche Oberflächenelemente für die Erfassung einer Staumeldung nötig sind (siehe Tabelle 1-1).

Tab. 1-1
Feldliste »Staumeldung erfassen«

Feldname	Funktion	Darstellung
–	Funktionsbeschreibung	Text
position	»Stauanfang« oder »Stauende«	Radiobutton
stauUrsache	»Unfall«, »Baustelle«, »Gaffer«, »Überlastung«, »Sonstige«	Auswahlliste (nur, wenn Meldepunkt == »Stauanfang«)

Nun passen wir die Oberfläche an unsere Anforderungen an. Dazu definieren wir die Formularelemente aus Tabelle 1-1 in XML. Die Datei `main.xml` sieht nach der Änderung wie folgt aus:

Listing 1.3 *main.xml für Staumelder*

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

  <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/hello" />

  <RadioGroup android:id="@+id/position"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">

    <RadioButton android:id="@+id/stauAnfang"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Stauanfang" />

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

  <Spinner android:id="@+id/stauUrsache"
     android:layout_width="fill_parent"
     android:layout_height="wrap_content"
     android:drawSelectorOnTop="true" />
</LinearLayout>
```

Wir haben das Layout um eine `RadioGroup`, bestehend aus zwei `RadioButtons`, ergänzt. Zusätzlich ist ein `Spinner` dazugekommen. Ein Spinner ist eine aufklappbare Auswahlliste. In unserem Beispiel ist die Wertebelegung für den Spinner statisch, so dass wir sie in eine weitere XML-Datei im `values`-Verzeichnis auslagern können. Wir geben ihr den Namen `arrays.xml`.

Listing 1.4 *res/arrays.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <array name="stauUrsachen">
    <item>Unfall</item>
    <item>Baustelle</item>
```

```
    <item>Gaffer</item>
    <item>Überlastung</item>
    <item>Sonstige</item>
  </array>
</resources>
```

Zum Schluss muss die Layoutbeschreibung noch korrekt an die Activity angebunden werden. Zu diesem Zweck erweitern wir unsere Activity wie in Listing 1.5 beschrieben.

Listing 1.5 *Activity mit Listenauswahl*

```
package de.androidbuch.staumelder;

public class StaumeldungErfassen extends Activity {

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);

    // Listeneinträge konfigurieren
    Spinner s1 = (Spinner)findViewById(R.id.Stauursache);
    ArrayAdapter<CharSequence> adapter =
        ArrayAdapter.createFromResource(
          this, R.array.stauUrsachen,
          android.R.layout.simple_spinner_item);

    adapter.setDropDownViewResource(
        android.R.layout.simple_spinner_dropdown_item);
    s1.setAdapter(adapter);
  }
}
```

Geschafft! Unsere Dialogmaske zum Erfassen der Staumeldung ist fertig. Nun muss die Anwendung nur noch im Emulator gestartet werden. Auch dazu bedienen wir uns der vorhandenen Eclipse-Umgebung (*Run -> Android -> Application*). Nach einer Wartezeit sollte der folgende Dialog auf dem Bildschirm erscheinen (Abb. 1-3).

1.4 Activities aufrufen

Interaktionen zwischen Activities

Beschäftigen wir uns nun mit Interaktionen zwischen Activities. Wenn die Menüoption »Melden« gedrückt wird, soll eine neue Seite erscheinen, auf der die Anzahl der für diesen Stau bisher gemeldeten Hinweise sowie alle weiteren verfügbaren Daten zu diesem Stau ausgegeben

Abb. 1-3
Beispielanwendung im Emulator

werden. Es wird dazu eine neue Activity »Stauinformationen anzeigen« benötigt, über die die folgenden Staudaten angezeigt werden:

- Anzahl abgegebener Meldungen für diesen Stau
- Wahrscheinlicher Stauanfang (wenn bereits gemeldet)
- Stauursache (wenn bekannt)

Mehr Aktivität

Zunächst erstellen wir für die zweite Maske eine Activity `StauinfoAnzeigen`. Anhand dieser Maske demonstrieren wir

- die Verwendung von dynamischen Inhalten in Masken und
- die Interaktion zwischen Masken.

Beginnen wir mit der Definition der Oberfläche. Hierzu erzeugen wir die Maskenbeschreibungsdatei `stauinfoanzeigen.xml` und legen sie unterhalb von `res/layout` ab. Generell ist zu beachten, dass der Name einer solchen Beschreibungsdatei nur aus Ziffern und Kleinbuchstaben sowie dem Unterstrich bestehen darf. Der eingängigere Name `stauinfoAnzeigen.xml` wäre daher nicht erlaubt.

Listing 1.6
Table Layout, Stauinfos anzeigen

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

```
  android:layout_height="fill_parent">

  <TableRow>
    <TextView
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:text="Anzahl Meldungen:"
      android:padding="3dip" />
    <TextView
      android:id="@+id/anzahlMeldungen"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:padding="3dip" />
  </TableRow>

  <TableRow>
    <TextView
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:text="Stauende:"
      android:padding="3dip"/>
    <TextView
      android:id="@+id/wahrscheinlichesStauende"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:padding="3dip" />
  </TableRow>

  <TableRow>
    <TextView
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:text="Ursache:"
      android:padding="3dip" />
    <TextView
      android:id="@+id/stauUrsache"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:padding="3dip" />
  </TableRow>
</TableLayout>
```

Für diese View verwenden wir ein `TableLayout`, da die Maske in Tabellenform dargestellt werden soll. Ansonsten gleicht diese Maskendefinition der vorherigen.

1.5 Das Android-Manifest

Die mit dieser View verknüpfte Activity `StauinfoAnzeigen` wird ebenfalls als Unterklasse von Activity implementiert. Um sie dem System bekannt zu machen, muss sie im `AndroidManifest.xml` des Projektes registriert werden. Listing 1.7 zeigt das vollständige Android-Manifest der Einführungsanwendung.

Listing 1.7
AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/
    apk/res/android"
  package="de.androidbuch.staumelder">
  <application android:icon="@drawable/icon">
    <activity android:name=".StaumeldungErfassen"
        android:label="@string/app_name">
      <intent-filter>
        <action android:name="
            android.intent.action.MAIN" />
        <category android:name="android.intent.
            category.LAUNCHER" />
      </intent-filter>
    </activity>
    <activity android:name=".StauinfoAnzeigen" />
  </application>
</manifest>
```

Innerhalb der Deklaration der Activity `StaumeldungErfassen` wird ein Intent-Filter definiert. Mit Intents und Intent-Filtern werden wir uns in Kapitel 7 ausführlicher befassen. Ein Intent repräsentiert einen konkreten Aufruf einer anderen Activity, eines Hintergrundprozesses oder einer externen Anwendung. Wir können den englischen Begriff mit »*Absichtserklärung*« übersetzen. Der hier verwendete Intent-Filter sorgt dafür, dass die Staumelder-Anwendung gestartet wird, indem die Activity `StaumeldungErfassen` angezeigt wird.

Einen Stau melden

Zur Übertragung der Staumeldung an den Server benötigen wir einen Menüeintrag in der Activity `StaumeldungErfassen`. Dazu erweitern wir die Activity um eine zusätzliche Methode. Sie fügt dem Menü den Eintrag »*Melden*« hinzu (siehe Listing 1.8).

Listing 1.8
Implementierung Standardmenü

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
  boolean result = super.onCreateOptionsMenu(menu);
  menu.add(0, ABSCHICKEN_ID, "Melden");
  return result;
}
```

Als nächsten und für dieses Eingangsbeispiel letzten Schritt wollen wir die beiden Activities miteinander kommunizieren lassen. Konkret soll `StaumeldungErfassen` nach Auswahl der Menüoption »Melden« die Activity `StauinfoAnzeigen` aufrufen. Anschließend sollen die erforderlichen Daten vom Server geladen und dargestellt werden.

In unserem Fall muss `StaumeldungErfassen` einen Intent erzeugen, ihn mit Übergabeparametern versehen und anschließend ausführen. Die Operation soll ausgeführt werden, wenn »Melden« ausgewählt wurde. Listing 1.9 zeigt den dafür erforderlichen Code aus `StaumeldungErfassen`.

Listing 1.9 *Aufruf anderer Activities*

```
@Override
public boolean onOptionsItemSelected(Item item) {
  switch (item.getId()) {
    case ABSCHICKEN_ID:
      String stauId = stauMelden();
      Intent intent =
          new Intent(this,StauinfoAnzeigen.class);
      intent.putExtra(STAU_ID, stauId);
      startActivity(intent);
      return true;
  }

  return super.onOptionsItemSelected(item);
}
```

Auf der Gegenseite muss nun die Activity `StauinfoAnzeigen` erstellt werden. Da diese Activity immer nur als Folge einer von außen durchgeführten Operation erzeugt werden kann, wird bereits in der Methode `onCreate` versucht, den entsprechenden Intent entgegenzunehmen und seine Daten zu verarbeiten.

Listing 1.10 *Activity StauinfoAnzeigen*

```
public class StauinfoAnzeigen extends Activity {
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.stauinfoanzeigen);
    stauverwaltung = new DummyStauverwaltung();

    Bundle extras = getIntent().getExtras();
    if (extras != null) {
      String stauId =
          extras.getString(StaumeldungErfassen.STAU_ID);
      zeigeStaudaten(stauId);
    }
  }
}
```

Das Prinzip der losen Kopplung von Komponenten, hier zwei Activities, wurde anhand dieses Beispiels verdeutlicht. Wir haben gesehen, wie eine Activity einen Intent verschickt, sobald der Menüeintrag »*Melden*« gedrückt wurde. Die auf Intents basierende offene Kommunikationsarchitektur stellt eine der Besonderheiten von Android dar.

Das Ergebnis im Emulator

Zum Abschluss dieser Einführung schauen wir uns das Ergebnis im Emulator an (siehe Abb. 1-4). Der vollständige Quellcode steht unter `http://www.androidbuch.de` zum Herunterladen bereit. Es empfiehlt sich, dieses Beispiel auf eigene Faust zu verändern und die Resultate unmittelbar im Emulator zu betrachten.

Abb. 1-4
Das Standardmenü und die Maske »StauinfoAnzeigen«

1.6 Fazit

In diesem Abschnitt gaben wir Ihnen einen kurzen Einblick in die Programmierung von Android-Geräten. Kennengelernt haben wir die Minimalbestandteile

- Maskenerstellung
- Dialogverarbeitung
- Interaktion zwischen Masken
- Start der Laufzeitumgebung und des Emulators

sowie die Android-Artefakte

- Activity
- View
- Intent
- Manifest

Nun ist es an der Zeit, sich ein wenig mit der Theorie zu befassen. Der Rest dieses ersten Teils beschäftigt sich mit den Konzepten hinter Android. Dabei wird an geeigneter Stelle auf das vorliegende Eingangsbeispiel verwiesen.

2 Systemaufbau

Hinweis: Die 2. Auflage dieses Buchs erscheint unter dem Titel
»Android 2. Grundlagen und Programmierung«
Ende Mai 2010 im dpunkt.verlag, ISBN 978-3-89864-677-2

In diesem Kapitel beschäftigen wir uns mit der Systemarchitektur von Android, die in Abbildung 2-1 skizziert ist.

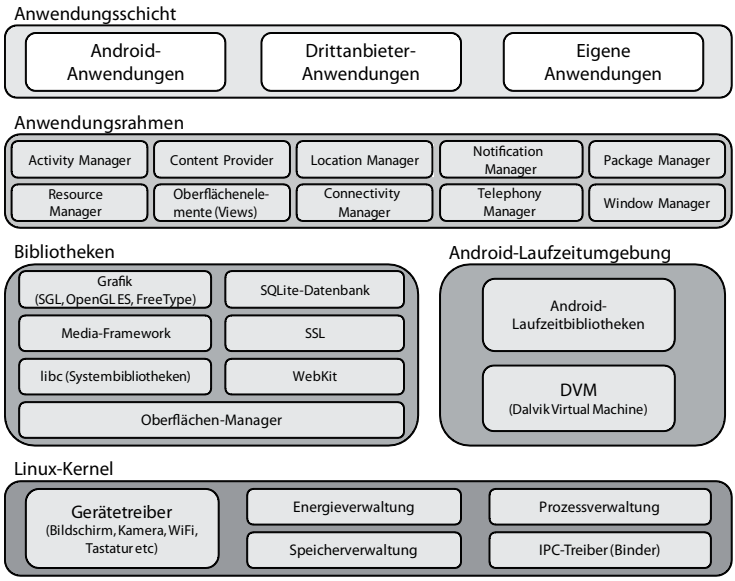


Abb. 2-1
Die Android-Systemarchitektur

2.1 Architekturübersicht

Linux Basis von Android ist ein Linux-Kernel (aktuell 2.6). Dieser stellt die erforderlichen Hardwaretreiber zur Verfügung und bildet eine bewährte Betriebssystemgrundlage.

Android-Laufzeitumgebung Kern der Laufzeitumgebung bildet die *Dalvik Virtual Machine* (DVM). Wird eine Android-Anwendung gestartet, so läuft sie in einem eigenen Betriebssystemprozess. Aber nicht nur das. Die DVM ist so klein und performant, dass Android jeder Anwendung darüber hinaus noch eine eigene DVM spendiert. Dies kostet

zwar extra Ressourcen, gibt aber erheblichen Auftrieb in puncto Sicherheit und Verfügbarkeit, da sich die Anwendungen keinen gemeinsamen Speicher teilen und ein sterbender Prozess nur eine Anwendung mit in die Tiefe reißt.

Die Anwendungen selbst werden in Java geschrieben und auch zur Entwicklungszeit von einem normalen Java-Compiler in Java-Bytecode übersetzt. Die Transformation des Java-Bytecode in DVM-kompatiblen *.dex-Code übernimmt das `dx`-Tool, welches im Lieferumfang des Android Development Kit enthalten ist. Es kommt immer dann zum Einsatz, wenn eine lauffertige Anwendung im Dalvik-Bytecode erzeugt werden soll. Dank des Eclipse-Plug-ins bekommt man davon jedoch als Entwickler meist nichts mit.

Standardbibliotheken Die Anwendungsschicht und der Anwendungsrahmen von Android greifen auf eine Menge von Basisbibliotheken zu, die in den folgenden Abschnitten detailliert beschrieben werden. Diese C/C++-Bibliotheken stellen alle zum Betrieb von Android-Anwendungen erforderlichen Funktionalitäten (Datenbank, 3D-Grafikbibliotheken, Webzugriff, Multimedia-Verwaltung, Oberflächengestaltung etc.) bereit. Die Standardbibliotheken sind fester Bestandteil des Systems und können von Anwendungsentwicklern nicht geändert werden.

Programmierschnittstelle/Anwendungsrahmen Der Anwendungsrahmen ist die für Android-Entwickler interessanteste Schicht des Systems. Android stellt verschiedene Programmierschnittstellen bereit, die eine Kommunikation zwischen einzelnen Anwendungen sowie zwischen Endanwender und Anwendung realisieren. Der Umgang mit diesen Manager-Komponenten wird uns im weiteren Verlauf des Buches noch beschäftigen.

Anwendungsschicht Auf dieser Ebene des Systems befinden sich die Android-Anwendungen. Dabei kann es sich um Eigenentwicklungen oder die von Google mitgelieferten Standardanwendungen handeln. Innerhalb der Anwendungsschicht findet die Kommunikation zwischen Mensch und Maschine sowie Interaktionen zwischen Anwendungen statt. Jede Anwendung bedient sich dabei der darunterliegenden Programmierschnittstelle.

2.2 Die Dalvik Virtual Machine

Die Dalvik Virtual Machine wurde von einem Google-Mitarbeiter namens Dan Bornstein entwickelt. Benannt ist sie nach dem isländischen Ort Dalvík, in dem Verwandte von Bornstein lebten. Sie stellt das Herzstück der Android-Laufzeitumgebung dar und basiert auf der quelloffenen JVM *Apache Harmony*, wurde aber in Aufbau und Funktionsumfang an die Anforderungen mobiler Endgeräte angepasst. Wir werden an dieser Stelle lediglich die Besonderheiten der VM darstellen. Für weiterführende Recherchen zur Dalvik VM sei ein Blick in [2] empfohlen.

JVM == DVM?

Wie unser Eingangsbeispiel bereits gezeigt hat, lässt sich Android komplett in Java programmieren. Dies hat den großen Vorteil, dass vorhandenes Wissen genutzt und vorhandene Entwicklungsumgebungen weiterverwendet werden können. Es stellt sich nun die Frage, wie der Java-Code Schritt für Schritt in ablauffähigen Code transformiert wird und worin die Unterschiede zwischen DVM und JVM liegen. Wir wollen dabei aber nicht zu tief in technische Details abtauchen, da das den Rahmen dieses Buches sprengen würde.

Vom Code zum Programm

Das Schaubild in Abb. 2-2 skizziert den Weg des Java-Codes von der Erstellung bis zur Ausführung im Android-Endgerät. Oberhalb der gestrichelten Linie findet die Entwicklung in der IDE auf dem PC statt. Der Pfeil nach unten deutet den Deployment-Prozess auf das Mobilgerät an.

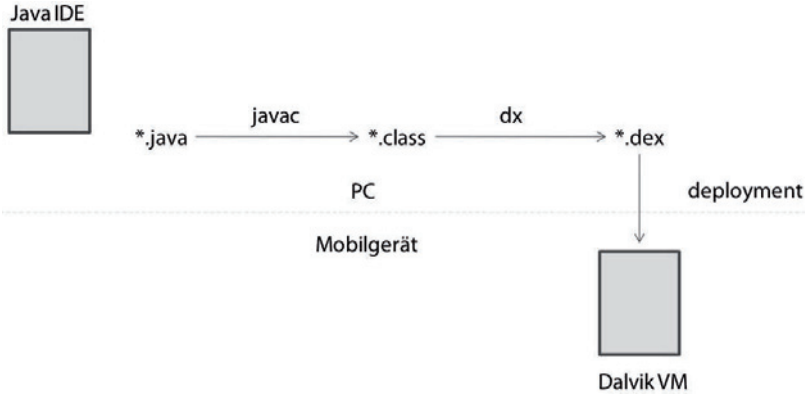


Abb. 2-2
*Von *.java zu *.dex*

DVM != JVM

Unterscheiden sich die »klassischen« JVMs, insbesondere die in der J2ME eingesetzten, von der Dalvik VM? Und wenn ja, inwieweit?

Zunächst mal hat Sun bei der Entwicklung der JVM für J2ME wenig Wert auf echte Neuerungen gelegt. J2ME bietet nur eine Untermenge der Java-Klassen des SDK. Einige Bestandteile der Sprache, wie z.B. *Reflection*, wurden weggelassen, da sie relativ viele Ressour-

cen verbrauchen. Im Grunde wurde nur abgespeckt, um Java auf mobilen Endgeräten mit wenig Speicher und langsamen Prozessoren zum Laufen zu bringen. Klassische JVMs nutzen in ihrer virtuellen Prozessorarchitektur nicht aus, dass Mikroprozessoren Register haben. Register sind Zwischenspeicher direkt im Mikroprozessor, die Berechnungen über mehrere Zwischenergebnisse extrem schnell machen. Google hat mit der DVM eine Java-Laufzeitumgebung geschaffen, die Registermaschinencode verarbeitet, also die Möglichkeiten moderner Prozessoren gut ausnutzt. Hinzu kommt, dass im Mobilfunkbereich Prozessoren der britischen Firma ARM Ltd. sehr verbreitet sind. Diese registerbasierten Prozessoren sind dank ihrer speziellen RISC-Architektur sehr sparsam und zugleich schnell. Die DVM ist sehr gut an diese Art von Prozessoren angepasst und läuft darauf ebenfalls schnell und ressourcenschonend. Aufgrund dieser Tatsache kann es sich Android leisten, pro Anwendung bzw. pro Prozess eine DVM zu starten. Dies ist ein sehr großer Vorteil gegenüber J2ME, insbesondere in Bezug auf die Sicherheit der auf dem Android-Gerät gespeicherten Daten, da der Zugriff auf Dateiressourcen innerhalb der VM nur mit Aufwand zu realisieren ist. Da aber sogar normalerweise pro Android-Anwendung ein eigener Betriebssystem-User verwendet wird (ein Prozess, ein User, eine DVM), sind gespeicherte Daten zum einen über die Berechtigungen des Betriebssystems geschützt und zum anderen über die Sandbox, in der die Anwendung innerhalb der VM ausgeführt wird. Es ist daher nicht möglich, unerlaubt aus einer Android-Anwendung heraus auf die gespeicherten Daten einer anderen Anwendung zuzugreifen.

DVM nutzt Register.

Eine DVM pro Anwendung

Kein unerlaubter Zugriff

Google hat nun einen klugen lizenzrechtlichen Trick angewandt. Für Android steht ein Großteil der API der Java Standard Edition (J2SE) zur Verfügung. Dadurch gewinnt es deutlich an Attraktivität gegenüber der stark eingeschränkten Java Mobile Edition (J2ME) von Sun. Die der DVM zugrunde liegende Standard-JVM *Apache Harmony* wird unter der Apache License vertrieben, so dass Änderungen am Code der JVM von den Geräteherstellern nicht im Quellcode ausgeliefert werden müssen.

Apache Harmony JVM

Die DVM selbst wird explizit nicht als Java-VM dargestellt, da der Name »Java« von Sun geschützt ist. Da sie auch keinen Java-Bytecode verarbeitet, sondern Android-eigenen DEX-Bytecode, fällt sie nicht unter die Lizenz von Sun. Der DEX-Code wird durch den Cross-Compiler, das *dx-Tool* im Android-SDK, erzeugt. Hier liegt der ganze Trick: Man programmiert in Java, erzeugt wie gewohnt mit Hilfe des Java-Compilers des Java-SDKs von Sun den Bytecode in Form von `.class`-Dateien und wendet darauf dann das dx-Tool an. Dieses liefert DEX-Bytecode-Dateien mit der Endung `.dx`, die zu einer fertigen Anwendung zusammengepackt werden. Das Ergebnis ist schließlich eine `.apk`-Datei,

dx-Tool erzeugt Bytecode.

die fertige Anwendung. Da die Programmierschnittstelle der J2SE von Sun bisher nicht patentrechtlich geschützt ist, liegt auch hier aktuell keine Verletzung bestehender Rechte vor.

2.3 Standardbibliotheken

Standardbibliotheken in C

Die Kernfunktionalität von Android wird über C/C++-Standardbibliotheken bereitgestellt, die von der Anwendungsschicht genutzt werden. Die folgenden Abschnitte stellen einige dieser Bibliotheken kurz vor. Im Praxisteil des Buches werden wir uns etwas intensiver damit befassen.

Browser-Grundlage

LibWebCore Android stellt eine auf der quelloffenen Bibliothek *WebKit* (`www.webkit.org`) basierende Webbrowser-Umgebung zur Verfügung. WebKit ist Grundlage vieler Browser auf Mobiltelefonen und wird z.B. in Nokias Symbian-S60-Betriebssystem, in Apples iPhone oder aber im Google-Chrome-Browser eingesetzt.

Leichte Datenbank

SQLite Als Datenbanksystem kommt das im mobilen Bereich bewährte *SQLite* (`www.sqlite.org`) zum Einsatz, welches uns ein längeres Kapitel wert ist (siehe Kap. 11).

Medien darstellen

Media Framework Das Android Media Framework basiert auf dem quelloffenen Multimedia-Subsystem *OpenCORE* der Firma PacketVideo. Diese Bibliothek ist für die Darstellung und Verarbeitung der gängigen Multimediaformate auf dem Gerät verantwortlich. Für die Grafikdarstellung und -verarbeitung werden die Bibliotheken *SGL* (2D) und *OpenGL 1.0* (3D) genutzt.

2.4 Der Anwendungsrahmen

Unterbau für Anwendungen

Als *Anwendungsrahmen* bezeichnet man eine Schicht im Android-Systemaufbau, die den Unterbau für die Anwendungen bildet. Der Anwendungsrahmen enthält die Android-spezifischen Klassen und abstrahiert die zugrunde liegende Hardware. Wir werden hier nicht die einzelnen Komponenten und Manager-Klassen vorstellen, da wir die wichtigsten im Verlauf des Buchs noch kennenlernen werden bzw. sie verwenden, ohne es unbedingt zu merken. Wenn wir zum Beispiel in Kapitel 13 über den Lebenszyklus von Prozessen reden, dann spielt der Activity Manager eine große Rolle, da er den Lebenszyklus der Activities verwaltet und sich eine Anwendung und damit ein Prozess (unter

anderem) aus Activities zusammensetzt. Uns interessiert aber nicht das »*Wie*«, sondern die Konsequenzen für die Anwendungsentwicklung, die sich aus der Funktionsweise des Anwendungsrahmens ergibt. Wir gehen daher einfach einen Abschnitt weiter und abstrahieren den Anwendungsrahmen in »Komponenten«.

2.5 Android-Komponenten

Wir werden im weiteren Verlauf oft von »*Android-Komponenten*« oder schlicht »*Komponenten*« sprechen. Dies hat einen ganz bestimmten Grund: Die Sprechweise soll immer wieder bewusst machen, dass es sich bei Android um eine sehr moderne Plattform für komponentenbasierte Anwendungen handelt. Ziel der Softwareentwicklung unter Android soll es sein, nicht jedesmal das Rad neu zu erfinden. Ausgehend von einigen vorinstallierten Standardanwendungen lassen sich Anwendungen entwickeln, die Teile der Standardanwendungen verwenden. Daraus entstehen wieder neue Anwendungen, die ihrerseits vielleicht wieder zum Teil von anderen Anwendungen genutzt werden.

Anwendungen bestehen aus Komponenten.

Ein Beispiel: Wer ein Android-Mobiltelefon kauft, findet darauf die vorinstallierten Anwendungen »*Contacts*« und »*Phone*«. Wenn Sie selbst Anwendungen für Android schreiben, können Sie sich aus der Datenbank der Anwendung »Contacts« eine Telefonnummer holen und sie in der Activity der Anwendung »Phone« wählen lassen. Die Anwendungen sind »offen«. Sie können über das Berechtigungssystem anderen Anwendungen erlauben, einige ihrer Komponenten zu verwenden. Was sind nun diese Komponenten?

Als Komponenten werden wir in Zukunft die folgenden Begriffe bezeichnen, die in späteren Kapiteln wesentlich detaillierter beschrieben werden:

Activity Anwendungen, die mit dem Anwender interagieren, brauchen mindestens eine Activity, um eine Oberfläche darzustellen. Activities sind sichtbar und können miteinander zu einer komplexeren Anwendung verknüpft werden. Sie kümmern sich um die Darstellung von Daten und nehmen Anwendereingaben entgegen. Sie sind jedoch Komponenten einer Anwendung, die mehr machen als die reine Darstellung von Daten und Formularen. Genaueres erfahren wir in Kapitel 5.

View

Service Nicht jeder Teil einer Anwendung braucht eine Oberfläche. Wenn wir Musik im Hintergrund abspielen wollen, können wir die Bedienung des Players einer Activity überlassen und das Abspielen der

Controller

Musik durch einen Service erledigen lassen, auch wenn die Bedienoberfläche schon geschlossen wurde. Ein Service erledigt Hintergrundprozesse und wird in Kapitel 8 näher erklärt.

Content Provider Nicht jede, aber viele Anwendungen bieten die Möglichkeit, Daten zu laden oder zu speichern. Ein Content Provider verwaltet Daten und abstrahiert die darunterliegende Persistenzschicht. Er kann über Berechtigungen seine Daten einer bestimmten Anwendung oder auch vielen Anwendungen zur Verfügung stellen. Er hat eine definierte Schnittstelle und wird darüber lose an Anwendungen gekoppelt. Wir beschäftigen uns in Kapitel 12 mit Content Providern.

Model

Broadcast Receiver Durch die komponentenbasierte Anwendungsentwicklung unter Android ist es notwendig, zwischen Betriebssystem und Anwendungen zu kommunizieren. Auch die Kommunikation zwischen Anwendungen ist möglich. Intents (»Absichtserklärungen«) als Objekte zur Nachrichtenübermittlung haben wir schon kurz in Listing 1.9 kennengelernt und werden das Thema später noch vertiefen. Broadcast Receiver lauschen jedoch als Komponente auf Broadcast Intents, die auf Systemebene verschickt werden und z.B. über Störungen der Netzwerkverbindung informieren oder über einen schwachen Akku. Mehr dazu erfahren wir in Kapitel 9.

Verbindung zum System

2.6 Die Klasse Context

Die Klassen `Activity` und `Service` sind von der abstrakten Klasse `android.content.Context` abgeleitet. `Context` gehört zur Android-Plattform und bildet eine Schnittstelle für Activities und Services zur Laufzeitumgebung. Über Methoden der Klasse `Context` lassen sich allgemeine Informationen über die Anwendungen erhalten und Methoden auf Anwendungsebene aufrufen. Damit ist zum Beispiel das Starten eines Service gemeint oder das Schreiben in eine Datei.

Schnittstelle zur Laufzeitumgebung

In der Anwendungsentwicklung verwendet man oft Methoden der Klasse `Context`, wenn man Activities oder Services implementiert. Der Begriff »*Context*« wird dabei immer wieder verwendet, wenn man die Umgebung der Anwendung meint. Zu dieser Umgebung gehören unter anderem

- der Classloader,
- das Dateisystem,
- die Berechtigungen der Anwendung,

- die verfügbaren Datenbanken,
- die anwendungseigenen Bibliotheken,
- die anwendungseigenen Ressourcen (Bilder etc.),
- die Bildschirmhintergrund,
- der Zugriff auf andere Komponenten der Anwendung
- etc.

Context via `this`

Man spricht in diesem Zusammenhang meist von dem Context, wenn man das `this`-Attribut eines Service oder einer Activity meint. Aber wie man an der Auflistung sieht, ist Context ein recht komplexer Begriff und steht für die Verbindung der Anwendung zu ihrer Laufzeitumgebung (und darüber hinaus, wie wir später sehen werden!).

Der Context einer Anwendung wird uns noch oft begegnen, und es ist wichtig, sich diesen Begriff zu merken und ein Gefühl für seine Bedeutung zu entwickeln.

3 Sicherheit

Hinweis: Die 2. Auflage dieses Buchs erscheint unter dem Titel
»Android 2. Grundlagen und Programmierung«
Ende Mai 2010 im dpunkt.verlag, ISBN 978-3-89864-677-2

Wir wollen in diesem Kapitel deutlich machen, dass Android-Anwendungen relativ sicher sind, wenn man einige Dinge berücksichtigt. Klar ist allerdings auch, dass eine komplexe Plattform wie Android Sicherheitslücken aufweisen kann. Die Autoren empfehlen, für sicherheitskritische Anwendungen noch eine eigene Sicherheitsstufe einzubauen, indem die sensiblen Daten nur verschlüsselt auf dem mobilen Computer gespeichert und verschlüsselt über das Netzwerk übertragen werden. Eine umfassende Einführung mit lauffähigem Beispiel dazu findet sich in Teil III des Buchs. Ein recht guter Artikel zu dem Thema findet sich auch im Android Developer Forum ([16]).

3.1 Die Sandbox

Von Haus aus darf man nix...

Android führt Programme, die keine speziellen Berechtigungen haben, in einer Sandbox aus. Bei einer Sandbox handelt es sich um einen eingeschränkten Bereich des Gesamtsystems, in dem das Java-Programm laufen darf. Die Sandbox regelt den Zugriff des Programms auf Ressourcen und Bestandteile des Betriebssystems. Zu diesen Ressourcen gehören der Arbeitsspeicher, Speichermedien, andere Anwendungen, das Netzwerk, Telefonfunktionen, SMS etc. Berechtigungen werden explizit in der Anwendung vergeben, und ohne diese Berechtigungen kann ein Programm die Sandbox nicht verlassen. Es kann also keine sicherheitsrelevanten Aktionen durchführen und darf nur seine eigenen Ressourcen verwenden.

Grundlegend basiert das Sicherheitskonzept der Sandbox auf Betriebssystemfunktionen, wie Gruppen- und Benutzerberechtigungen. Zudem wird normalerweise jede Anwendung als eigener Prozess gestartet und ist genau einem Benutzer auf Betriebssystemebene zugeordnet (dem »User« auf Linux-Ebene, weshalb wir hier diesen Begriff verwenden werden). Normalerweise wird für jede Anwendung (in Form einer `.apk`-Datei) durch das Betriebssystem automatisch ein User angelegt, dem eine eindeutige Linux-User-Id zugewiesen wird. Auf diese Weise

nutzt Android für die Sandbox das Berechtigungskonzept des Betriebssystems.

3.2 Signieren von Anwendungen

Während der Entwicklung ist es nicht nötig, sich mit dem Signieren von Anwendungen zu beschäftigen. Android bringt ein Standardzertifikat mit, welches im Android-Plug-in für Eclipse automatisch verwendet wird. Beim Starten der Anwendung im Emulator oder auf dem per USB-Kabel angeschlossenen mobilen Computer wird die .apk-Datei mit dem Standardzertifikat signiert und ist sofort lauffähig.

Signieren bedeutet hier, dass der Anwendung mit Hilfe des Zertifikats eine »digitale Unterschrift« hinzugefügt wird. Diese Unterschrift kann nur mit dem Zertifikat erzeugt werden und ist eindeutig. Dadurch lässt sich später der Ersteller der Anwendung identifizieren. Das mitgelieferte Standardzertifikat ist jedoch nicht eindeutig, weshalb es nur zum Entwickeln und Testen von Anwendungen verwendet werden sollte. Anwendungen während der Entwicklungs- und Testphase können nur direkt, z.B. über das Eclipse-Plug-in, auf dem angeschlossenen Android-Endgerät installiert werden. Später wird man aber meist die Anwendung einer größeren Zahl von Nutzern zur Verfügung stellen wollen und eine Installation über das Internet ermöglichen. Dann muss die Anwendung mit einem eigenen Zertifikat signiert werden. Dies ist ebenfalls nötig, wenn man sie im Android-Market zugänglich machen möchte. Wie man ein eigenes Zertifikat erstellt und die Anwendung »marktreif« macht, erklären wir im dritten Teil des Buchs in Kapitel 17.

Zum Entwickeln reicht das Standardzertifikat.

3.3 Berechtigungen

Berechtigungen gewähren Anwendungen Zugriff auf Systemfunktionen und Ressourcen außerhalb der Sandbox. Es ist beispielsweise nicht möglich, eine Internetverbindung aufzubauen oder eine SMS zu verschicken, wenn man nicht explizit die dafür zuständige Berechtigung vergeben hat. Berechtigungen werden im Android-Manifest gepflegt, damit sie zum Zeitpunkt der Installation bekannt sind. Dies ist wichtig, da eine Anwendung bzw. Bestandteile von ihr von anderen Anwendungen genutzt werden können, ohne dass sie läuft.

Berechtigungen weichen die Sandbox auf und öffnen die Anwendung kontrolliert nach außen. Ein Prinzip, welches schon von J2ME bekannt ist, dort aber trotz aller vergebenen Berechtigungen mit allerlei Restriktionen behaftet ist, die Anwendungsentwicklern das Leben schwer machen.

Ca. 100 Berechtigungen

Wir werden hier nicht alle Berechtigungen vorstellen, da es davon an die einhundert Stück gibt. Wir werden vielmehr zeigen wie man sie setzt. Ein paar prominente Beispiele werden wir uns aber rauspicken und in Tabelle 3-1 näher erklären.

Während der Entwicklung wird man bisweilen auf das Problem stoßen, dass man vergessen hat, die nötigen Berechtigungen zu setzen. Glücklicherweise wird dann zur Laufzeit eine `java.lang.SecurityException` geworfen, und der Fehlertext ist recht aussagekräftig. Wollen wir z.B. unsere aktuelle Position über das GPS-Modul bestimmen, verwenden wir den Location Manager, eine Android-Komponente zur Bestimmung der aktuellen Position. Beim ersten Testlauf erhalten wir einen Hinweistext der Form:

```
java.lang.SecurityException: Requires
    ACCESS_FINE_LOCATION permission
```

Berechtigungen vergeben

Diesen Fehler beheben wir, indem wir das Android-Manifest öffnen und gleich nach dem Manifest-Tag unsere Berechtigung einfügen. Dabei wird den Permissions immer ein »`android.permission.`« vorangestellt.

```
<manifest
  xmlns:android="http://schemas.android.com/apk/
      res/android"
  package="de.androidbuch.staumelder"
  android:versionCode="1"
  android:versionName="1.0.0">

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

Alle vorhandenen Berechtigungen finden sich unter [14]. Tabelle 3-1zeigt jedoch einige prominente Vertreter, die uns zum Teil später noch begegnen werden.

3.4 Anwendungsübergreifende Berechtigungen

Es ist möglich, mehrere Anwendungen in einer gemeinsamen Sandbox laufen zu lassen. Die Anwendungen können lesend und schreibend auf die gleichen Ressourcen zugreifen, so als würde es sich um eine Anwendung handeln.

Tab. 3-1 Einige wichtige Android-Berechtigungen

Berechtigung	Beschreibung
ACCESS_FINE_LOCATION	Berechtigung zum Abfragen der aktuellen Position über das GPS-Modul
ACCESS_NETWORK_STATE	Eine Internetverbindung kann abreißen, z.B. wenn Sie in ein Funkloch geraten. Dank dieser Berechtigung können Sie Informationen über die zur Verfügung stehenden Netzwerke abrufen (siehe auch die nächste Berechtigung).
CHANGE_NETWORK_STATE	Eine Verbindung mit dem Internet über WLAN kann billiger sein als über Ihren Netzbetreiber. Die Anwendung kann registrieren, wenn ein WLAN in Reichweite ist, und versuchen, sich darüber zu verbinden.
INTERNET	Erlaubt das Verbinden mit einem Server via Socket-Verbindung
RECEIVE_BOOT_COMPLETED	Erlaubt der Anwendung, den Intent `android.intent.action.BOOT_COMPLETED` zu empfangen. Dadurch kann sich eine Anwendung automatisch starten, nachdem das Android-Gerät eingeschaltet wurde.
RECEIVE_SMS	Erlaubt einer Anwendung, SMS zu empfangen. Damit lassen sich sehr interessante Programme entwickeln, da SMS genutzt werden können, um ein Android-Gerät zu erreichen, welches nicht über das Netzwerk mit einem Server verbunden ist.

sharedUserId

Um dies zu erreichen, vergibt man allen beteiligten Anwendungen im Android-Manifest eine gemeinsame `sharedUserId`. Bei der Installation der Anwendungen auf einem Android-Gerät erkennt Android die zusammengehörenden Anwendungen und teilt ihnen den gleichen User auf Betriebssystemebene zu. Die Sandbox umschließt dann bildlich gesprochen alle Anwendungen und Komponenten, die dieselbe `sharedUserId` besitzen. Ein Beispiel:

```
<manifest
  xmlns:android="http://schemas.android.com/apk/
      res/android"
  package="de.androidbuch.staumelder"
  android:versionCode="1"
```

```
android:versionName="1.0.0"
android:sharedUserId="de.androidbuch">
...
```

Hinweis

Die `sharedUserId` muss mindestens einen Punkt enthalten. Eine gute Wahl für eine `sharedUserId` ist der Domänenname der Firma, für die man das Programm entwickelt, oder die oberste Paketebene des Android-Projekts.

Sicherheit dank Zertifikat

Nun könnte jedoch jede Anwendung, die ebenfalls diese `sharedUserId` besitzt, auf die Daten der anderen Anwendungen zugreifen. Da dies ein Sicherheitsrisiko darstellt, wird bei der Installation der Anwendung auf dem Android-Gerät geprüft, ob die Anwendung mit dem gleichen Zertifikat signiert wurde wie die anderen schon installierten Anwendungen, die die gleiche `sharedUserId` verwenden (siehe Kap. 17). Da das Zertifikat beim Hersteller der Software liegt und nicht öffentlich ist, kann man keine Anwendung mit »gestohlener« `sharedUserId` zum Ausspähen von privaten Daten erstellen.

Für Anwendungen, die aus der Entwicklungsumgebung heraus getestet werden, gilt dies nicht. Hier wird sowieso für alle Anwendungen das gemeinsame Standardzertifikat verwendet.

Teil II

Android in der Praxis

In Teil II dieses Buches werden wir uns mit den Möglichkeiten und Grenzen von Android in der Praxis auseinandersetzen. Wir werden unser Eingangsbeispiel »Staumelder« von seinem Prototypstatus befreien und die Anwendung Schritt für Schritt für die raue Wirklichkeit auf mobilen Endgeräten vorbereiten.

Nach einer kurzen Einführung über Ziele und fachliche Inhalte des Projektes »Staumelder« werden wir die technischen Themengebiete in Iterationen vorstellen und unsere Software erweitern. Auch hier erheben wir nicht den Anspruch, jede noch so kleine Nuance der Android-API auszuloten und vorzustellen. Vielmehr ist es uns wichtig, ein kleines »Kochbuch« für den schnellen Einstieg zu liefern, welches jeweils von einem kurzen Theorieteil und einem Praxisteil mit ausführlichen Codebeispielen flankiert wird.

Ziel dieses zweiten Teils ist es, Sie in die Lage zu versetzen, ohne umfangreiche Online-Recherchen ein Android-Projekt erfolgreich umzusetzen. Die vorherige Lektüre des ersten Buchteils wird empfohlen, ist aber nicht unbedingt notwendig. Sämtliche hier erwähnten Quelltexte stehen auf der Website zum Buch (`www.androidbuch.de`) zum Herunterladen bereit.

4 Projekt »Staumelder«

Hinweis: Die 2. Auflage dieses Buchs erscheint unter dem Titel
»Android 2. Grundlagen und Programmierung«
Ende Mai 2010 im dpunkt.verlag, ISBN 978-3-89864-677-2

Wir beschreiben in diesem Kapitel den *Staumelder*. Dieses Android-Programm dient als roter Faden für den Teil II dieses Buchs. Wir werden anhand mehrerer Iterationen nach und nach ein vollständiges Android-Programm entwickeln, welches die wesentlichen Android-Komponenten enthält. Um das Programm zu verstehen und die in den folgenden Iterationen vorkommenden Quellcode-Auszüge richtig einordnen und verstehen zu können, ist es wichtig zu wissen, was der Staumelder für Funktionen hat. Daher beschreiben wir nun, was das Programm fachlich gesehen für Funktionen hat, bevor wir uns später um die technische Umsetzung kümmern.

4.1 Aus Sicht des Anwenders

Auf dem Weg zur Arbeit...

Stellen Sie sich vor, Sie steigen morgens in Ihr Auto und wollen ins Büro fahren. Sie haben sich gestern Abend den Staumelder auf Ihr Android-Gerät installiert und möchten ihn nun nutzen, um über die Staus auf Ihrem Weg zur Arbeit informiert zu werden. Natürlich sind Sie auch bereit, einen Stau zu melden, wenn Sie das Pech haben, in einem zu stehen. Eine kleine Vorarbeit mussten Sie aber noch erledigen: Die Routen, für die Sie Stauinformationen erhalten möchten, müssen dem Server bekannt gemacht werden. Daher müssen Sie sie über eine Weboberfläche eingeben. Die Eingabe der Routeninformation ist rein serverseitig. Die Routen werden aber an den Staumelder auf dem Android-Gerät übertragen. Da wir lernen wollen, wie man Android-Programme schreibt, setzen wir einfach voraus, dass die Routen auf dem Server vorliegen.

Sie starten nun das Staumelder-Programm und sehen den Startbildschirm der Anwendung (siehe Abb. 4-1 (a)).

Eine Route auswählen

Sie drücken auf den ersten Menüpunkt, um eine Route zu wählen. Der Staumelder verbindet sich mit dem Server und aktualisiert die Routeninformationen. Sie wählen nun die gewünschte Fahrtroute aus (siehe Abb. 4-1 (b)) und starten die Anzeige der Routeninformation. Der Staumelder kennt nun die Route, auf der Sie fahren möchten, und fragt beim Server nach, ob es auf dieser Strecke Staus gibt. Während-

Abb. 4-1
Startbildschirm des Staumelders (a) und Auswahl einer Route (b)

dessen wechselt die Anzeige, und sobald Stauinformationen vorliegen, werden diese angezeigt.

Wie ist die Verkehrslage?

Abbildung 4-2 zeigt die Stauinformationen zur gewählten Route. Entweder liegen keine Meldungen vor (a) oder es sind Staus gemeldet worden (b). Zu jeder der angezeigten Staumeldungen lassen sich Detailinformationen abrufen, indem man die Meldung anwählt.

Den Stau umfahren

Die Fahrt kann nun beginnen. Sind Staus auf der Strecke gemeldet, hilft die Straßenkarte des Staumelders weiter, um sich zu orientieren und eine alternative Route zu finden. Im Hauptmenü wählt man einfach den Menüpunkt »*Straßenkarte anzeigen*«. Mit dieser Option kann sich der Anwender eine Straßenkarte der Region, in der er sich gerade befindet, mit Hilfe von Google Maps anzeigen lassen. Die aktuelle Position wird mit einem roten Punkt kenntlich gemacht und die Karte auf diesen Punkt zentriert. Es ist möglich, die Karte zu vergrößern oder zu verkleinern. Mit Hilfe der Karte kann der Anwender einen Stau umfahren, ohne auf ein Navigationsgerät zurückgreifen zu müssen. Abbildung 4-3 (a) zeigt die Straßenkartenansicht im Staumelder.

Alle machen mit: Stau melden!

Aber bevor wir einen Stau umfahren, sollten wir ihn melden. Ein Stau ist eine dynamische Sache. Er kann länger oder kürzer werden, und irgendwann löst er sich auf. Als Anwender der Staumelder-Anwendung

Abb. 4-2
Anzeige der Staus auf einer Route

ist man gehalten, Staus zu melden, wenn man welche sieht. Je mehr Meldungen über einen Stau in einem Zeitraum eingehen, desto genauer und aktueller sind die Informationen über den Stau auf dem Server, und nachfolgende Autofahrer können sich eine Alternativroute suchen.

Daher gibt es im Hauptmenü noch die Option »*Meldung erfassen*«. Da Android über ein Modul zur Ermittlung der Ortsposition verfügt (den *Location Manager*), weiß der Staumelder immer, wo er sich gerade befindet. Eine Staumeldung an den Staumelderserver zu schicken ist daher einfach. Man gibt an, wo man sich gerade im Stau befindet (am Anfang, am Ende oder mittendrin), und, falls bekannt, einen Grund für den Stau. Ein Klick auf die Schaltfläche »*Melden*« und die Meldung wird zum Server geschickt. Abbildung 4-3 (b) zeigt die Maske zur Erfassung einer Staumeldung.

Der Staumelder weiß, wo er ist.

Zwei Menüpunkte, die in Abbildung 4-1 (a) zu sehen sind, bleiben noch offen. Mittels »*Einstellungen*« kann man einige anwendungsspezifische Parameter pflegen. Dazu gehören z.B. der Anmeldename und das Passwort für den Staumelderserver, damit dieser weiß, wer die Meldung gemacht hat. Mittels *Beenden* kann der Staumelder verlassen werden. Die Netzwerkverbindung wird getrennt und belegte Ressourcen freigegeben.

Abb. 4-3
Straßenkarte mit eigener Position (a) und Erfassen einer Staumeldung (b)

4.2 Iterationsplan

Nachdem die fachlichen Ziele definiert sind, wenden wir uns der Umsetzung in die Praxis zu. Der in diesem Abschnitt skizzierte Iterationsplan gibt einen Überblick über die Themengebiete, mit denen wir uns auf den folgenden Seiten beschäftigen wollen.

Unser Projekt wird in mehrere *Iterationen* aufgeteilt. Eine Iteration ist hier als Synonym für ein *Lerngebiet* aufzufassen. Jede Iteration ist nach folgendem Schema aufgebaut:

- *Iterationsziel*: Beschreibung des Lernziels dieses Kapitels
- *Theoretische Grundlagen*: Theoretische Einführung in die Themen, die zur Erreichung des Iterationsziels erforderlich sind
- *Hauptteil/Implementierungsphase*: Iterationsabhängige Beschreibung der Implementierungsschritte
- *Fazit*: Zusammenfassung, Referenzen, Schnelleinsteiger-Tipps

Zwischendurch werden wir *Exkurse* in Themengebiete durchführen, die für das Verständnis allgemein wichtig sind, sich aber nicht als Code im Staumelder wiederfinden. Die Exkurse sind etwas theoretischer und für das Verständnis von Android sehr wichtig.

Im Einzelnen sind folgende Iterationen geplant:

1. Dialoge und Gestaltung von Oberflächen
2. Interaktionen zwischen Programm und Oberfläche
3. Hintergrundprozesse
4. Persistenz
5. Unterbrechungsbehandlung
6. Netzwerke
7. Location Based Services

Den Themen »Intents«, »Systemnachrichten«, »Dateisystem« und »Lebenszyklus von Prozessen« haben wir Exkurse gewidmet. Die Kapitel dieses Buchteils bauen auf dem im ersten Teil erarbeiteten Einstiegsbeispiel des Staumelders auf. Der im Einstiegsbeispiel verwendete Code wird nicht weiterverwendet, da er ein anderes Ziel (Prototyperstellung) verfolgte.

5 Iteration 1 – Oberflächengestaltung

Hinweis: Die 2. Auflage dieses Buchs erscheint unter dem Titel
»Android 2. Grundlagen und Programmierung«
Ende Mai 2010 im dpunkt.verlag, ISBN 978-3-89864-677-2

Die Qualität der Benutzerschnittstelle, d.h. der Bildschirmseiten und -dialoge, ist ein wichtiges Kriterium für die Akzeptanz einer Anwendung.

Wir wollen uns daher in der ersten Iteration unseres Staumelder-Projektes mit den Grundlagen der Oberflächengestaltung in Android befassen. Das Thema allein könnte ein eigenes Buch füllen. Bitte sehen Sie es uns nach, dass wir in diesem Einsteigerbuch nur die elementaren Konzepte vorstellen.

Dieses Kapitel führt in die Erstellung von Bildschirmseiten und Menüs ein. Das nächste Kapitel befasst sich mit Dialogen und Formularverarbeitung.

5.1 Iterationsziel

Erstes Ziel: die Startseite

Ziel der Iteration ist es, den Startbildschirm des Staumelders zu implementieren. Auf diesem werden Schaltflächen für die Operationen »*Route auswählen*«, »*Staumeldung erfassen*« und »*Straßenkarte anzeigen*« dargestellt. Über die Menütaste des Geräts können die Funktionen »*Staumelder beenden*« und »*Einstellungen bearbeiten*« aufgerufen werden. Die Schaltflächen und Menüs sind noch ohne Funktion. Für alle Bildschirmseiten des Staumelders soll ein einheitliches Erscheinungsbild festgelegt werden.

5.2 Activities, Layouts und Views

In diesem Abschnitt werden die Grundbegriffe der Oberflächengestaltung vorgestellt. Anschließend wird beschrieben, wie man diese Bausteine zu einer lauffähigen Anwendung verknüpft.

5.2.1 Grundbegriffe der Oberflächengestaltung

Activity = Kontrolle

Wir wollen nach dem Start der Staumelder-Anwendung eine Bildschirmseite mit mehreren Schaltflächen, etwas Text und einem Menü angezeigt bekommen. Dazu benötigen wir eine »Kontrollinstanz«, deren Aufgabe die Darstellung der Texte, Schaltflächen und Menüoptionen ist. Des Weiteren muss diese auf Eingaben des Anwenders reagieren und die Kontrolle an andere Bildschirmseiten übergeben können. Diese Aufgaben werden von speziellen Java-Klassen, den *Activities*, übernommen. Pro Bildschirmseite muss eine Activity implementiert werden.

View = Darstellung

Jedes an der Oberfläche darstellbare Element ist von der Klasse `android.view.View` abgeleitet. Die klassischen Oberflächenelemente wie Eingabefelder und Auswahlboxen werden wir ab jetzt als *Oberflächen-Komponenten* oder *Widgets* bezeichnen.

Viewgroups gruppieren Views

Darüber hinaus gibt es Views, die als Rahmen für andere Views dienen. Betrachten wir beispielsweise eine View zur Anzeige einer Tabelle. Sie enthält ihrerseits Views, die Text- und Formularfelder darstellen. Die Rahmen-View wird als *Viewgroup* bezeichnet und erbt von (`android.view.Viewgroup`).

Layouts

Wir unterscheiden Viewgroups, die selbst als Formularelemente dienen, von denen, die »im Hintergrund« die Anordnung (das »Layout«) ihrer untergeordneten Views festlegen. Ein Beispiel für eine Formular-Viewgroup wäre eine Auswahlbox, die ihre Wahlmöglichkeiten als geschachtelte Views enthält. Eine Tabellen- oder Listendarstellung wäre hingegen ein Vertreter einer Layout-Viewgroup, die wir im Folgenden auch kurz als *Layout* bezeichnen wollen.

Bildschirmseite := Baum von Views

Eine *Bildschirmseite* wird als Baum von Viewgroups und Views definiert. Als Wurzel dient in den meisten Fällen eine Layout-Viewgroup, welche die restlichen Oberflächenelemente auf der Seite anordnet.

Bildschirmdialog := Activity + View

Als *Bildschirmdialog* definieren wir die Verbindung einer Activity mit einer Bildschirmseite. Der Bildschirmdialog ist also der Teil der Benutzeroberfläche, mit dem ein Anwender in Kontakt tritt. Über ihn erhält er die Kontrolle über die Anwendung.

5.2.2 Zusammenwirken der Elemente

Wenden wir diese Begriffe auf unsere Aufgabenstellung an. Wir wollen einen Bildschirmdialog »*Startseite*« definieren. Dazu benötigen wir eine Activity `StartseiteAnzeigen`, die ein Menü mit den Optionen »*Staumelder beenden*« und »*Einstellungen bearbeiten*« erstellt und alle Klicks auf die Schaltflächen entgegennimmt. Die Bildschirmseite besteht aus einer Viewgroup, welche Widgets für Schaltflächen und den Begrüßungstext

in einem einfachen Layout untereinander gruppiert. Die Abbildung 5-1 zeigt den Bildschirmdialog aus Anwendersicht.

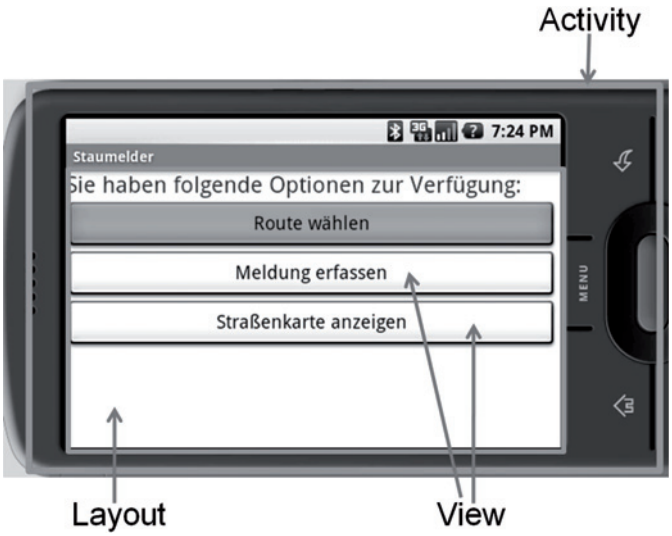


Abb. 5-1
Bildschirmaufbau

5.2.3 Theorie der Implementierung

Nachdem wir die Grundbausteine von Android-Oberflächen kennengelernt haben, wenden wir uns deren Implementierung zu.

Java-Activities

Activities werden von `android.app.Activity` abgeleitet. Neben dieser Standard-Implementierung stellt die Android-API Activities für unterschiedliche Aufgaben bereit. So übernimmt z.B. die `android.app.ListActivity` die Verwaltung von Listendarstellungen und die `android.app.PreferenceActivity` die Verwaltung von Systemeinstellungen. Eine Übersicht aller Activities kann der Online-Dokumentation zur Klasse `android.app.Activity` entnommen werden.

Layouts und Komponenten

Für jede View existiert ein XML-Element und eine Java-Klasse. Bildschirmseiten können also als XML-Dateien, als Java-Code oder als Kombination von beidem definiert werden. Wir empfehlen aus Gründen der Übersichtlichkeit die Verwendung von XML. Die Java-Komponenten nutzt man zum Auslesen von Eingabefeldern oder allgemein für Zugriffe auf Seitenelemente aus dem Programmcode heraus.

Pro Bildschirmseite muss eine XML-Datei im Verzeichnis `res/layout` erstellt werden. Von dort wird sie, identifiziert durch ihren Dateinamen, beim Start »ihrer« Activity eingelesen und angezeigt. Betrachten wir in Listing 5.1 die Definition des Staumelder-Startbildschirms.

Listing 5.1 *Bildschirmlayout startseite.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/
      res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:layout_margin="2dp">

  <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    style="@style/TextGross"
    android:text="@string/startseiteanzeigen_intro"
    android:lineSpacingExtra="2dp"
  />
  <Button
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    style="@style/SchaltflaechenText"
    android:id="@+id/sf_starte_routenauswahl"
    android:text="@string/app_routeFestlegen"
  />
  ...
</LinearLayout>
```

Die Activity `StartseiteAnzeigen` liest das Layout als Wurzel-View während ihrer Erzeugung ein (Listing 5.2). Der Bildschirmdialog ist fertig!

Listing 5.2 *Verknüpfung View mit Activity*

```
package de.androidbuch.staumelder.mobilegui;

public class StartseiteAnzeigen extends Activity {
  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.startseite_anzeigen);
  }
}
```

Ressourcen

Beim Blick auf die Layoutdefinition fällt auf, dass Attributwerte häufig in der Form

```
@style/SchaltflaechenText
@+id/sf_starte_routenauswahl
```

definiert sind. Die durch @ eingeleiteten Bezeichner verweisen auf sogenannte *Ressourcen*. Dabei handelt es sich um Texte, Formatierungsanweisungen, Bilder oder ähnliche Nicht-Java-Bestandteile der Anwen-

dung, die durch einen eindeutigen Schlüssel identifiziert werden. Ressourcen spielen bei der Oberflächendefinition eine wichtige Rolle. Die nächsten Abschnitte befassen sich mit der Definition von Ressourcen und stellen die wichtigsten Ressourcen-Arten und ihre Anwendungsgebiete vor.

5.3 Ressourcen

Benutzeroberflächen lassen sich nicht allein durch Java-Code umsetzen. Wir haben bereits die Java-freie Definition von Bildschirmseiten in XML-Dateien kennengelernt.

Wenn eine Anwendung mehrere Landessprachen unterstützen soll, müssen alle sichtbaren Texte mehrsprachig vorliegen. All diese Nicht-Java-Elemente (Texte, Grafiken) werden als *Ressourcen* bezeichnet. Je nach Format und Verwendungszweck unterscheiden wir folgende *Ressourcen-Arten*:

- Texte
- Grafiken
- Farbdefinitionen
- Multimediadateien
- Styles
- Themes
- Bildschirmseiten-Definitionen
- Menüdefinitionen
- XML-Konfigurationsdateien

5.3.1 Definition von Ressourcen

Kompilierte Ressourcen

Ressourcen werden entweder in XML-Dateien oder in Binärdateien definiert. Unmittelbar nach Erstellung oder Änderung einer Ressourcendatei startet das Eclipse-Plug-in den *Ressourcencompiler* `aapt` des Android-SDK. Hinter dem Kürzel verbirgt sich das *Android Asset Packaging Tool*. Dieses wandelt die Inhalte der Ressourcendateien in Objekte bzw. Objektbäume um. Dadurch werden alle Ressourcen in Bytecode übersetzt und in die Gesamtanwendung integriert. Zugriffe auf Ressourcen sind also ohne zeitaufwendige XML-Transformationen oder -Lesevorgänge möglich.

Schlüsselwert-Speicher R

Auf ein Ressourcen-Objekt wird vom Java-Code oder anderen Ressourcen anhand eines aus Ressourcen-Art und Ressourcen-Name abgeleiteten Schlüssels zugegriffen. Das `aapt` erzeugt automatisch im Paket

der Anwendung eine Klasse `R`, welche die Schlüssel aller für die Anwendung definierten Ressourcen enthält. Manuelle Änderungen an der Klasse werden bei jedem Lauf von `aapt` überschrieben. Wir bezeichnen `R` im weiteren Verlauf auch als *Schlüsselwert-Speicher*.

Ressourcen-Namen

Zur Definition einer Ressource muss ein Ressourcen-Name vergeben werden. Dieser muss innerhalb einer Ressourcen-Art eindeutig sein. Er darf nur aus alphanumerischen Zeichen und den Zeichen Unterstrich (_) oder Punkt (.) bestehen. Der `aapt` wandelt Punkte bei Erstellung des Schlüsselwert-Speichers in Unterstriche um. Die Ressource `startseite.titel` würde also über `R.string.startseite_titel` verfügbar gemacht. Wir werden in diesem Buch den Unterstrich der Punktnotation vorziehen.

Verzeichnisstruktur

Ressourcen werden per Konvention unterhalb des Verzeichnisses `res` gespeichert. Dieses Verzeichnis wird beim Anlegen eines Eclipse-Android-Projektes automatisch vom Plug-in erzeugt. Die Namen der Unterverzeichnisse setzen sich aus der dort gespeicherten Ressourcen-Art und weiteren optionalen Detailangaben wie Sprache, Bildschirmgröße etc. zusammen (s. Abschnitt 5.7.2 auf Seite 71). So findet man die Definitionen von Bildschirmseiten in den Verzeichnissen `res/layout` und die Definitionen von Menüs unter `res/menu`.

Nachdem geklärt ist, wo man Ressourcen definiert und speichert, wollen wir zeigen, wie man auf sie zugreift.

5.3.2 Zugriff auf Ressourcen

Der Zugriff auf eine Ressource, z.B. einen bestimmten Oberflächentext, kann auf dreierlei Weise erfolgen:

- Per Referenz innerhalb einer Ressourcendefinition
- Per Referenz innerhalb des Java-Codes
- Per Direktzugriff innerhalb des Java-Codes

Wir schauen uns die Zugriffsvarianten anhand eines Ausschnitts aus unserem Eingangsbeispiel an (Listing 5.3).

Zugriff innerhalb einer Ressourcendefinition

Listing 5.3 *Ressourcenzugriff mit @*

```
<TextView android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:theme="@android:style/Theme.Dialog"
  android:text="@string/startseite_anzeigen_titel"/>
```

Ressourcen-Schlüssel

Der Zugriff auf eine Ressource erfolgt über ihren *Ressourcen-Schlüssel*, den wir folgendermaßen definieren:

```
@[Package:]Ressourcen-Art/Ressourcen-Name
```

Beispiele für Ressourcen-Schlüssel sind:

```
android:style/Theme.Dialog
string/startseite_anzeigen_titel
```

Die Package-Angabe ist nur erforderlich, wenn auf freigegebene Ressourcen anderer Anwendungen oder auf systemweit gültige Ressourcen (Package `android`) zugegriffen werden soll. Listing 5.3 zeigt die Referenz auf eine für alle Anwendungen einheitlich definierte Formatvorlage `android:style/Theme.Dialog`. Zur Angabe der Ressourcen-Art dient der Name des Ressourcenverzeichnisses.

Anhand des Ressourcen-Schlüssels erfolgt nun der Zugriff auf eine Ressource auf eine der oben beschriebenen Arten.

Indirekter Zugriff auf Ressourcen Die meisten Methoden der Android-API erwarten für die Nutzung einer Ressource lediglich deren Ressourcen-Schlüssel. So wird die Titelzeile des Bildschirmfensters einer Activity z.B. mittels der in Listing 5.5 beschriebenen Codezeile auf den Wert der Text-Ressource `startseite_anzeigen_titel` gesetzt. Diese Ressource ist in der Datei `res/values/strings.xml` definiert (Listing 5.4).

Listing 5.4 *Definition von Ressourcen*

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="startseite_anzeigen_titel">
    Startseite
  </string>
</resources>
```

Listing 5.5 *Verwendung eines Ressourcen-Schlüssels*

```
@Override
public void onCreate(Bundle icicle) {
  ...
  setTitle(R.string.startseiteanzeigen_titel);
}
```

Direkter Zugriff auf Ressourcen Nehmen wir an, wir wollen eine Text-Ressource für Vergleiche oder andere Text-Operationen verwenden. Oder ein Bild nach dem Laden skalieren oder anderweitig ändern. Dann benötigen wir den Zugriff auf das Ressourcen-Objekt selbst und nicht nur eine Referenz auf dieses.

Dieser Direktzugriff wird über Methoden der Klasse *Resources* `android.content.res.Resources` ermöglicht. Die für eine Anwendung passende Instanz dieser Klasse ist jeder Activity bekannt.

Listing 5.6 zeigt, wie der Titeltext einer Activity mit Hilfe der Methode `getResources` ausgelesen und weiterverwendet wird.

Listing 5.6 *Laden einer Text-Ressource*

```
@Override
public void onCreate(Bundle icicle) {
  String titel =
    getResources().getString(
      R.string.startseiteanzeigen_titel);
  if( titel.indexOf(".") >= 0 ) {
    ...
  }
}
```

Zieldatentyp

Das Ressourcen-Objekt ist von dem Datentyp, in den die Ressource vom Ressourcencompiler übersetzt worden ist. Diesen Typ bezeichnen wir als *Zieldatentyp* der Ressource. So würde z.B. der Zugriff auf eine Grafik-Ressource `res/drawable/hintergrund.png` folgendermaßen implementiert:

Listing 5.7 *Laden einer Grafik-Ressource*

```
Drawable hintergrund = getResources().getDrawable(
    R.drawable.hintergrund);
```

Zwischenstand

Wir sind nun in der Lage, Ressourcendateien zu speichern und auf deren Inhalte zuzugreifen. Doch wie definiert man eine Ressource konkret? Was sind die Besonderheiten einzelner Ressourcen-Arten? Diese und andere Fragen wollen wir in den nächsten Abschnitten beantworten.

Wir stellen die Ressourcen-Arten kurz vor und geben Beispiele für ihre Verwendung.

5.3.3 Text-Ressourcen

Texte sind Grundbaustein einer jeden Anwendung. Sie sollten einfach zu pflegen sein und manchmal auch in mehreren Sprachen vorliegen.

Textbausteine auslagern

Das Konzept, Textbausteine vom Java-Code zu trennen, hat sich in zahlreichen JEE-Frameworks (Struts, JSF, Spring) bewährt. Android verwendet Ressourcen der Art `string`, um die Texte einer Anwendung in einer separaten Datei vorzuhalten. Die Bildschirmseiten-Überschrift in Listing 5.1 auf Seite 40 ist eine solche Text-Ressource. Tabelle 5-1 stellt die Ressourcen-Art `string` vor.

Der folgende Ausschnitt aus der Definitionsdatei `res/values/strings.xml` verdeutlicht die Möglichkeiten und Grenzen von Text-Ressourcen.

Ressourcen-Art:	string
Definitionsdatei:	res/values/strings.xml
Zieldatentyp:	java.lang.CharSequence
Zugriffsmethode:	Resources::getString(id)
Definition:	`<string name="Rsrc-Name">Wert</string>`

Tab. 5-1
Ressourcen-Art string

```
<resources>
  <string name="einfach">
    Ein einfacher Text
  </string>
  <string name="quote_ok1">
    Ein \'einfacher\' Text
  </string>
  <string name="quote_ok2">
    "Ein 'einfacher' Text"
  </string>
  <string name="quote_falsch">
    Ein 'einfacher' Text
  </string>
  <string name="html_formatierung">
    Ein <i>einfacher</i> Text
  </string>
</resources>
```

5.3.4 Farb-Ressourcen

Farbdefinitionen werden für das Einfärben von Texten, Bildschirmvorder- und -hintergründen etc. benötigt. Sie werden wie aus HTML bekannt codiert:

- #RGB
- #ARGB
- #RRGGBB
- #AARRGGBB

Dabei bedeuten: R = Rot-Anteil, G = Gelb-Anteil, B = Blau-Anteil, A = Alphakanal.

Farb-Ressourcen definieren »sprechende« Namen für Farbwerte. Das nächste Beispiel nutzt die Ressource `fehler` als Synonym für den Farbwert `#FF0000`. So wird die Darstellung von Fehlermeldungen vereinheitlicht und die Wahl der »Fehler-Farbe« flexibel gehalten.

```
<resources>
  <color name="fehler">#FF0000</color>
</resources>
```

Durch Farb-Ressourcen werden Farbzuweisungen bei der Oberflächengestaltung lesbarer. So kann z.B. das Farbschema der Corporate Identity eines Unternehmens für dessen Anwendungen einheitlich definiert werden. Tabelle 5-2 fasst die Ressourcen-Art `color` kurz zusammen.

Tab. 5-2
Ressourcen-Art color

Ressourcen-Art:	color
Definitionsdatei:	res/values/colors.xml
Zieldatentyp:	int
Zugriffsmethode:	Resources::getColor(id)
Definition:	`<color name="Rsrc-Name">#Farbwert</color>`

5.3.5 Formatvorlagen: Styles und Themes

Die Bildschirmseiten einer Anwendung sollten in einem einheitlichen Erscheinungsbild gestaltet werden. Dazu gehören einheitliche Schriftarten und -größen, Vorder- und Hintergrundfarben sowie die Ausrichtung einzelner Views (zentriert, links-, rechtsbündig). Das Erscheinungsbild wird durch die Attributwerte der Views beeinflusst. Gesucht ist also ein Weg, diese Attribute innerhalb einer Anwendung einheitlich zu definieren und zuzuweisen.

Style := Gruppe von Attributwerten

Ähnlich wie *Cascading Style Sheets* (CSS) für Weboberflächen, gibt es auch für Android-Bildschirmseiten *Styles*, die als Formatvorlagen für View-Elemente (Widgets, Layouts etc.) genutzt werden. Eine Formatvorlage wird einer View über ihr Attribut `style` zugewiesen. Dadurch erhält die View alle Attributwerte, die durch die Formatvorlage vorgegeben werden. Wenn eine solche Vorlage auf alle Views gleichen Typs angewandt wird, lassen sich z.B. Textgröße, Schriftart etc. anwendungsübergreifend definieren.

Um beispielsweise alle Schaltflächentexte des Staumelders einheitlich zu formatieren, definieren wir eine Vorlage mit dem Ressourcen-Namen `SchaltflaechenText`, mit deren Hilfe wir Schriftgröße und -farbe auf einen gewünschten Wert festlegen (Listing 5.8). Alle Formatvorlagen einer Anwendung werden in der Datei `res/values/styles.xml` definiert.

Listing 5.8 *Definition von Formatvorlagen*

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="SchaltflaechenText">
    <item name="android:textSize">18sp</item>
    <item name="android:textColor">#EEEEFF</item>
  </style>
</resources>
```

Der folgende Ausschnitt einer Bildschirmseiten-Definition demonstriert, wie die Vorlage einer View zugewiesen wird (Listing 5.9).

Listing 5.9 *Stilvolle Schaltfläche*

```
<Button
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  style="@style/SchaltflaechenText"
  android:id="@+id/sf_starte_routenauswahl"
  android:text="@string/app_routeFestlegen"
/>
```

Größenbezeichnungen

Die Schriftgröße wird hier in der Maßeinheit `sp` angegeben. Es handelt sich dabei um *scale independent pixel*. Ein `sp` entspricht einem Pixel auf einem 160-dpi-Bildschirm. Bei Änderung der Bildschirmgröße skaliert die Maßeinheit mit. Neben der Bildschirmgröße wird die vorgegebene Schriftgröße als Grundlage für die Skalierung verwendet. Scale independent pixel werden daher für Schriftgrößendefinitionen empfohlen.
Als weitere Größenbezeichnung wird der *density independent pixel* (`dp` oder `dip`) für textunabhängige Größen angeboten. Auch diese Größeneinheit passt sich automatisch einer geänderten Bildschirmgröße an, skaliert allerdings Schriften nicht optimal.
Der Vollständigkeit halber seien noch die in der Praxis seltener verwendeten Maßeinheiten `px` (Pixel), `mm`, `in` (Inch) und `pt` (Punkte) genannt.

Vererbung von Formatvorlagen

Formatvorlagen können ihre Attributwerte von anderen Vorlagen erben. Auf diesem Weg lassen sich allgemeine Vorgaben definieren, die dann in Unterformaten verfeinert werden. Denkbar wäre beispielsweise ein in der gesamten Anwendung einheitlicher Wert für die Schriftart, der für Überschrift-Formatvorlagen oder Fließtext-Formatvorlagen um eine Größenangabe erweitert wird. Die Vererbung drückt man durch Angabe der Vater-Formatvorlage im Attribut `parent` aus.

In Listing 5.10 wird die aus dem letzten Beispiel bekannte Formatvorlage für Schaltflächen um ein Sonderformat für »wichtige« Schaltflächen erweitert. Dieses erbt die Schriftgröße von ihrem »Vater«, überschreibt aber dessen Farbwert und fügt eine Textzentrierung hinzu.

Listing 5.10 *Vererbung von Formatvorlagen*

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="SchaltflaechenText">
    <item name="android:textSize">18sp</item>
    <item name="android:textColor">#EEEEFF</item>
  </style>
  <style name="WichtigeSchaltflaechenText"
        parent="SchaltflaechenText">
    <item name="android:textColor">#FF0000</item>
    <item name="android:textAlign">center</item>
  </style>
</resources>
```

Themes

Während Styles das Format von Views vorgeben, beeinflussen sogenannte *Themes* das Erscheinungsbild kompletter Bildschirmfenster (Vorder-/Hintergrundfarbe, Titelzeile aktiv/inaktiv etc.). Themes sind also Formatvorlagen für die Bildschirmfenster, während Styles Formatvorlagen für die Elemente auf den Seiten sind.

Konsequenterweise werden Themes daher nicht pro View-Element bei der Bildschirmseiten-Definition zugewiesen, sondern im Android-Manifest. Jede dort registrierte Anwendung und Activity kann mit einem Theme versehen werden. Das folgende Beispiel demonstriert die Definition (Listing 5.11) und Zuweisung (Listing 5.12) eines anwendungsweit gültigen Themes.

Listing 5.11 *Definition eines Themes*

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="StaumelderDesign">
    <item name="windowFrame">
      @drawable/scr_sm_hintergrund
    </item>
    <item name="panelForegroundColor">
      #FF000000
    </item>
    <item name="panelTextColor">
      ?panelForegroundColor
    </item>
  </style>
</resources>
```

Interne Referenzen

Durch die Schreibweise `?panelForegroundColor` erhält das Attribut `panelTextColor` den gleichen Wert wie das Attribut `panelForegroundColor`.

```
<manifest xmlns:android=
   "http://schemas.android.com/apk/res/android"
  package="de.androidbuch.staumelder"
  android:versionCode="1"
  android:versionName="0.1.0">
  <application
      android:theme="@style/StaumelderDesign">
  ...
  </application>
</manifest>
```

Listing 5.12
Zuweisung des Themes im Manifest

Standard-Formatvorlagen

Die Android-API bietet mehrere Standard-Formatvorlagen zur Verwendung als Themes oder Styles an [19]. Diese lassen sich problemlos für eigene Anwendungen nutzen oder mit geringem Aufwand als »parent« eines eigenen Themes oder Styles wiederverwenden. Bei Verwendung der Standardvorlagen darf die Package-Angabe `android:` nicht fehlen (z.B. `android:style/Theme.Light`).

Wir wollen nun im Schnelldurchlauf einen Überblick über die von Android unterstützten Binär-Ressourcen geben.

5.3.6 Bilder

Bilder werden als Teil von Bildschirmseiten oder als Piktogramme für Schaltflächen und Menüs verwendet. Sofern sie fester Bestandteil der Anwendung sind, werden sie als Ressourcen verwaltet.

Im Zweifel PNG

Android unterstützt die Bildformate PNG, JPG und GIF. PNG weist das beste Verhältnis von Dateigröße zu Bildqualität auf und sollte daher den anderen Formaten vorgezogen werden.

Size matters

Bevor man Bilddateien verwendet, sollte man sie auf die erforderliche Größe zuschneiden. Große Bilder verschlechtern die Lade- und somit Reaktionszeiten einer Anwendung.

Tabelle 5-3 stellt die Ressourcen-Art *drawable* vor, zu der alle Bildformate gehören.

Der Dateianhang (z.B. »`.png`«) gehört nicht zum Ressourcen-Namen. Daher muss der Basis-Dateiname innerhalb der Anwendung eindeutig sein.

Tab. 5-3
Ressourcen-Art drawable

Ressourcen-Art:	drawable
Definitionsdatei:	z.B. res/drawable/Datei.png
Definition:	Die Ressource wird im o.g. Verzeichnis gespeichert. Der Ressourcen-Name leitet sich aus dem Basisnamen der Datei ab. `res/drawable/hintergrund.png` hätte also den Ressourcen-Namen `R.drawable.hintergrund`.
Zieldatentyp:	android.graphics.drawable.BitmapDrawable
Zugriffsmethode:	Resources::getDrawable(id)

5.3.7 Multimediadaten

Audio- und Videodaten spielen im Mobiltelefonbereich eine große Rolle. Das Gerät wird als MP3-Player oder zum Abspielen heruntergeladener Videos genutzt. Aus diesem Grund bietet Android für derartige Inhalte ebenfalls eigene Ressourcen-Arten an.

Vom Android-SDK werden die folgenden Musikformate unterstützt: MP3 (CBR und VBR bis zu 320Kbps), M4A (AAC LC, AAC, AAC+), OGG, 3GP (AMR-NB and AMR-WB), WAVE (8/16-bit PCM) und MIDI (SMF0/1, XMF0/1, RTTTL/RTX, OTA, iMelody).

Die Tabellen 5-4 und 5-5 liefern eine Kurzdarstellung der Multimedia-Ressourcen. Definition und Benennung der Ressourcen erfolgt analog zu Ressourcen der Art `drawable`.

Tab. 5-4
Ressourcen-Art raw für Audiodateien

Ressourcen-Art:	raw
Definitionsdatei:	z.B. res/raw/Audiodatei.wav
Zieldatentyp:	keiner
Zugriffsmethode:	keine

Tab. 5-5
Ressourcen-Art raw für Videodateien

Ressourcen-Art:	raw
Definitionsdatei:	z.B. res/raw/Video.mpeg
Zieldatentyp:	android.graphics.Movie
Zugriffsmethode:	Resources::getVideo(id)

5.4 Menüs

Mit zunehmender Beliebtheit von berührungsempfindlichen Bildschirmen (engl. *touch screens*) im Mobilgerätebereich stellt sich für Entwickler die Frage, auf welche Weise sie die Navigation zwischen den Bildschirmseiten am benutzerfreundlichsten gestalten.

Qual der Wahl

Es gibt zwar nach wie vor die vom Mobiltelefon bekannte Navigation über die Menütaste des Telefons. Die Steuerung über Schaltflächen in Verbindung mit den neuartigen Bildschirmen oder Steuerungskomponenten (Cursor, Track-Ball etc.) ist aber in den meisten Fällen einfacher zu bedienen. Probleme treten hier erst auf, wenn zu viele Auswahlmöglichkeiten angeboten werden müssen. Eine Bildschirmseite mit zu vielen Schaltflächen wird schnell unübersichtlich. Für unser Projekt legen wir daher die folgende Implementierungsrichtlinie fest:

> Wir ziehen die Verwendung von Schaltflächen zur Bildschirmnavigation der klassischen Menütasten-Navigation vor, wenn auf dem Bildschirm genug Platz dafür ist. Die Steuerung über Menütasten sollte in jedem Fall für Standardfunktionen (z.B. Einstellungen ändern, Programm beenden etc.) verwendet werden.

Der vorliegende Abschnitt befasst sich mit den Erscheinungsformen und der Definition von Menüs im Allgemeinen. In Android werden zwei Arten von Menüs angeboten:

Optionsmenüs sind Menüs, die über die Menütaste des Geräts aktiviert werden. Pro Activity existiert ein Optionsmenü (Abb. 5-2 links).

Kontextmenüs lassen sich durch längeres Anklicken von Bildschirmelementen (Schaltflächen, Eingabefelder etc.) aktivieren. Für jede View kann ein Kontextmenü definiert werden (Abb. 5-2 rechts).

5.4.1 Allgemeine Menüdefinition

Menüs werden als XML-Ressourcen oder als Java-Code definiert. Wird die Definition per XML durchgeführt, muss pro Menü eine Datei im Ressourcenverzeichnis `res/menu` angelegt werden. Listing 5.13 zeigt die Definition für das Menü der Staumelder-Startseite (Abb. 5-2 links).

Abb. 5-2
Ein einfaches Optionsmenü (links), Kontextmenü für Schaltfläche (rechts)

Listing 5.13
Definition Hauptmenü Staumelder

```
<menu xmlns:android=
  "http://schemas.android.com/apk/res/android">
  <item
    android:id="@+id/opt_einstellungenAnzeigen"
    android:title="@string/app_einstellungenAnzeigen"
  />
  <item
    android:id="@+id/opt_staumelderBeenden"
    android:title="@string/app_staumelderBeenden"
    android:icon="@drawable/icoBeenden"/>
  />
</menu>
```

Diese Definition der Menüinhalte ist sowohl für ein Options- als auch für ein Kontextmenü nutzbar. Pro `<item>`-Element muss das Attribut `android:id` mit dem Wert belegt werden, über den der beschriebene Menüeintrag später ausgewertet werden soll.

Sagen Bilder mehr als Worte?

Als Alternative oder Ergänzung der Textdarstellung können für die ersten fünf Einträge eines Optionsmenüs Piktogramme (engl. *icons*) verwendet werden. Man sollte nur kleine, klar unterscheidbare Grafiken einsetzen.

Die Tabelle 5-6 gibt eine Übersicht wichtiger Attribute des `<item>`-Elements. Die vollständige Attributliste ist Teil der API-Dokumentation.

Attribut	Beschreibung	gültig für (Anzahl)
id	Schlüssel des Menüeintrags	Optionsmenüs, Kontextmenüs
title	Langtext Menüoption	Optionsmenüs (6-n), Kontextmenüs
condensedTitle	Abkürzung Menüoption	Optionsmenüs (1-5)
icon	Referenz auf Icon-Ressource	Optionsmenüs (1-5)

Tab. 5-6
Attribute von <item>

Nach dieser allgemeinen Einleitung wollen wir uns die beiden Menüarten genauer ansehen.

5.4.2 Optionsmenüs

Jede Activity erhält vom Betriebssystem eine Referenz auf ihr Optionsmenü. Sie kann dieses dann um weitere Menüoptionen ergänzen.

Platzprobleme

Menüeinträge werden in der Reihenfolge ihrer Definition angezeigt. Falls mehr als fünf Einträge dargestellt werden sollen, wird automatisch ein künstliches »More…[1]«-Element eingefügt, welches die verbleibenden Auswahloptionen als Liste sichtbar macht. Abbildung 5-3 zeigt eine solche lange Optionsliste. Lange Optionsmenüs lassen sich nicht so schnell und komfortabel bedienen wie kürzere. Man sollte daher weitgehend auf sie verzichten.

Abb. 5-3
Optionsmenü mit more

Zuweisung und Auswertung

Die Definition der Menüeinträge des Optionsmenüs einer Activity findet in der Methode `onCreateOptionsMenu` der Activity statt (Listing 5.14). Dabei muss lediglich die Referenz auf die Menü-Ressource angegeben werden. Der Activity Manager des Betriebssystems ruft `onCreateOptionsMenu` nach Erstellung der Activity durch `onCreate` auf.

[1] Der Text hängt von der Spracheinstellung des Geräts ab.

Abb. 5-4 *Ein langes Optionsmenü*

Listing 5.14 *Zuweisung Optionsmenü*

```
public boolean onCreateOptionsMenu(Menu menu) {
  super.onCreateOptionsMenu(menu);
  MenuInflater mi =
    new MenuInflater(getApplication());
  mi.inflate(R.menu.hauptmenue, menu);
  return true;
}
```

Wählt der Anwender eine Menüoption aus, wird die Methode `Activity::onOptionsItemSelected` aufgerufen. Dort wird das gewünschte `<item>` ermittelt und auf die Auswahl reagiert (Listing 5.15).

Listing 5.15 *Auswertung Optionsauswahl*

```
public boolean onOptionsItemSelected(MenuItem item) {
  switch (item.getItemId()) {
  ...
  }
}
```

5.4.3 Kontextmenüs

Für jedes Element einer Bildschirmseite kann ein Kontextmenü definiert werden. Dieses erscheint, sobald der Anwender mit der Auswahltaste länger auf das betroffene Bildschirmelement klickt (*Long-Click-Events*). Die Verknüpfung zwischen Kontextmenü und Bezugsobjekt wird in der zugrunde liegenden Activity hergestellt.

> Kontextmenüs werden pro Aufruf neu gefüllt, Optionsmenüs nur einmal bei Erzeugung ihrer Activity.

Es werden immer alle Einträge eines Kontextmenüs angezeigt. Reicht der Platz nicht aus, so wird automatisch eine Bildlaufleiste (engl. *scroll-*

bar) aktiviert. Das erschwert die Bedienung der Anwendung, so dass man lange Kontextmenüs vermeiden sollte.

Menü zu Kontext

Die Zuweisung eines Kontextmenüs zu einer View erfolgt in zwei Stufen. Als Erstes wird die View bei ihrer Activity für die Verwendung eines Kontextmenüs registriert. In Listing 5.16 wird die Schaltfläche mit dem Schlüssel `starte_routenauswahl` registriert.

Listing 5.16
Registrierung Kontextmenü für eine View

```
public void onCreate(Bundle icicle) {
  registerForContextMenu(
      findViewById(R.id.starte_routenauswahl));
}
```

Damit allein ist das Menü allerdings noch nicht aktiviert. Die Menüoptionen müssen noch zugewiesen werden. Dieser Schritt ist in Listing 5.17 beschrieben. Jede Activity bietet dazu die Methode `onCreateContextMenu` an. Diese ist für die »Befüllung« aller Kontextmenüs der ihr zugewiesenen Bildschirmseite zuständig. `onCreateContextMenu` wird aufgerufen, sobald ein Long-Click-Event auf einer für Kontextmenüs registrierten View ausgelöst wurde.

Listing 5.17
Erzeugung von Kontextmenüs

```
public void onCreateContextMenu(
    ContextMenu menu,
    View v,
    ContextMenuInfo menuInfo) {
  super.onCreateContextMenu(menu, v, menuInfo);
  MenuInflater mi =
     new MenuInflater(getApplication());
  if (v.getId() == R.id.starte_routenauswahl) {
    mi.inflate(R.menu.demo_langes_menue, menu);
  }
}
```

Wird nun ein `<item>` dieses Kontextmenüs ausgewählt, so wird vom System die Methode `Activity::onContextItemSelected` aufgerufen. Dort kann auf die gewählte Option reagiert werden (Listing 5.18).

Listing 5.18
Auswertung eines Kontextmenüs

```
public boolean onContextItemSelected(MenuItem item) {
  switch (item.getItemId()) {
    case R.id.opt_einstellungenAnzeigen: {
      macheEtwasSinnvolles();
      return true;
    }
  }
  return super.onContextItemSelected(item);
}
```

5.4.4 Dynamische Menügestaltung

In manchen Fällen reicht die statische Definition der Menüs nicht aus. Es wäre wünschenswert, wenn man einzelne Optionen während des Anwendungslaufs aktivieren, deaktivieren oder komplett verbergen könnte. Die Darstellungsform einzelner `<item>`-Elemente sollte also während des Programmlaufs verändert werden können.

Die Lösung besteht darin, Menüs nicht als Ressource, sondern direkt im Java-Code zu definieren und zu verändern. Die Android-API sieht dafür die Klasse `android.view.Menu` vor. Das folgende Listing zeigt, wie ein Optionsmenü anhand dieser Klasse definiert wird.

```
public boolean onCreateOptionsMenu(Menu menu) {
  final boolean result =
    super.onCreateOptionsMenu(menu);
  menu.add(
    0,
    ROUTE_STARTEN_ID,
    0,
    R.string.routefestlegen_routenStarten);
  return result;
}
```

Menüs führen den Nutzer durch die Anwendung. Sie sollten deshalb normalerweise von vornherein klar definiert werden können. Daher empfehlen wir, Menüs weitgehend in XML zu definieren und sie nur in Ausnahmefällen im Java-Quelltext zu realisieren.

5.5 Das Android-Manifest

Wir haben jetzt alle Bestandteile eines Bildschirmdialogs kennengelernt. Um daraus eine lauffähige Anwendung zu machen, benötigen wir noch eine weitere Datei: das *Android-Manifest*. Diese teilt der Android-Laufzeitumgebung mit, aus welchen Komponenten sich die Staumelder-Anwendung zusammensetzt. Beim Android-Manifest handelt es sich um eine XML-Datei `AndroidManifest.xml` im Wurzelverzeichnis der Anwendung. Dort

- wird der Package-Name der Anwendung festgelegt,
- werden alle Komponenten (Activities, Services) der Anwendung aufgelistet,
- werden Berechtigungen auf Anwendungs- und Prozessebene vergeben.

Vom Konzept her erinnert das Android-Manifest an die zahlreichen Deployment-Deskriptoren im Java-EE-Umfeld. Die formale Definition des Manifests soll hier nicht aus der Online-Dokumentation kopiert werden [10]. Wir werden im Verlauf des Buches viele Elemente des Manifests kennenlernen.

Mit diesem Abschnitt endet der Theorieblock zum Thema »Oberflächen«. Nun wollen wir uns den Umgang mit Activities, Views und Ressourcen in der Praxis anschauen und beginnen mit der Implementierung des Staumelders.

5.6 Implementierung eines Bildschirmdialogs

Starten wir mit der Fortsetzung unseres am Kapitelanfang begonnenen Vorhabens, den Startbildschirm als ersten Bildschirmdialog des Projekts zu implementieren. Wir legen als Erstes ein Android-Projekt `staumelder` im Package `de.androidbuch.staumelder` an. Das Vorgehen dazu haben wir bereits im Eingangsbeispiel beschrieben. Rufen wir uns nun noch einmal die Iterationsziele ins Gedächtnis:

- Es soll ein Startbildschirm mit mehreren Schaltflächen angezeigt werden.
- Über das Optionsmenü der Startseite sollen die Funktionen »Staumelder beenden« und »Einstellungen bearbeiten« angeboten werden.
- Für alle Masken soll ein einheitliches Erscheinungsbild definiert werden.

5.6.1 Checkliste Dialogerstellung

Wir müssen also als Erstes ein Layout für die Startseite definieren und im Ressourcenverzeichnis ablegen. Anschließend implementieren wir eine Activity, die als Einstiegspunkt der Anwendung dient. Durch den Einsatz von Formatvorlagen stellen wir das einheitliche Erscheinungsbild sicher.

Die folgenden Schritte sind zur Erstellung eines Bildschirmdialogs erforderlich.

1. Text-Ressourcen für die Beschriftung definieren,
2. Multimedia-Ressourcen definieren,
3. Bildschirmseite definieren,
4. Menüs definieren,

5. Activity implementieren, Bildschirmseiten-Definition und Menüs einbinden,
6. Android-Manifest um neue Activity erweitern,
7. Bildschirmdialog im Emulator testen.

Zur Verdeutlichung der einzelnen Schritte werden wir die erste Maske unseres Projektes Schritt für Schritt erstellen.

5.6.2 Texte für Bildschirmseiten definieren

Für die Startseite benötigen wir einen Titel, einen Einführungstext sowie die Bezeichnungen der Schaltflächen und Menüoptionen. Wir erstellen also im Verzeichnis `res/values` eine Datei `strings.xml` mit folgendem Inhalt:

Listing 5.19 *strings.xml für die Startseite*

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="startseiteanzeigen_titel">
    Staumelder
  </string>
  <string name="startseiteanzeigen_intro">
    Sie haben folgende Optionen zur Verfügung:
  </string>
  <string name="app_routeFestlegen">
    Route wählen
  </string>
  <string name="app_staumeldungErfassen">
    Meldung erfassen
  </string>
  <string name="app_strassenkarteAnzeigen">
    Straßenkarte anzeigen
  </string>
  <string name="app_staumelderBeenden">Beenden</string>
  <string name="app_einstellungenAnzeigen">
    Einstellungen
  </string>
  <string name="app_hilfe">Hilfe</string>
</resources>
```

Beim Speichern dieser Datei wird der Schlüsselspeicher `de.androidbuch.staumelder.R.java` aktualisiert. Der Compiler überwacht die Einhaltung der Namensregeln für Ressourcen-Namen. Probieren Sie es aus. Versuchen Sie, der Ressource für den Startseitentitel den Namen `startseite-anzeigen_titel` zu geben. Sie erhalten nach dem Speichern prompt eine Fehlermeldung in der Klasse `R`.

Da `R.java` vom Eclipse-Plug-in generiert wird, sollte man diese Klasse nicht unter Versionskontrolle stellen. Die Ressourcen sollten versioniert werden.

In der Praxis hat es sich als effizient erwiesen, zu Projektbeginn die Texte für *alle* Bildschirmseiten zu definieren. Dadurch fällt eine einheitliche Namensgebung der Elemente leichter, und das zeitaufwendige Wechseln der Editoren wird reduziert.

5.6.3 Multimedia-Ressourcen definieren

Auf die Texte können wir nun zugreifen. Wenden wir uns der nächsten Ressource zu. Die Datei `autobahn.png` soll Hintergrundbild der Startseite werden. Wir legen also ein Verzeichnis `res/drawable` an und kopieren `autobahn.png` dorthin. Sie ist jetzt über den Ressourcen-Schlüssel `drawable/autobahn` erreichbar.

5.6.4 Bildschirmseite definieren

Wahl des Layouts

Als nächsten Schritt legen wir das Design der Bildschirmseite fest. Dieses definieren wir in der Datei `res/layout/startseite_anzeigen.xml`. Wir suchen zuerst ein passendes Layout, das die Anordnung der Oberflächenelemente vorgibt. So können die Schaltflächen, Texte und Bilder entweder untereinander, in Listenform oder als Tabelle dargestellt werden. Die für unsere Anwendung benötigten Basislayouts der Android-API stellen wir kurz in den Tabellen 5-7 auf Seite 60 und 5-8 auf Seite 61 vor. Eine Liste aller Layouts ist Teil der Online-Dokumentation [11].

Für die Startseite des Staumelders verwenden wir ein `<LinearLayout>`, da wir lediglich etwas Text und einige Schaltflächen untereinander anzeigen lassen wollen. Wir wählen die vertikale Ausrichtung der Oberflächenelemente über das Attribut `android:orientation` (s. Listing 5.1 auf Seite 40).

Seiteninhalte definieren

Innerhalb des Layouts setzen wir jetzt die gewünschten Views als Oberflächenelemente ein. Die Android-API bietet auch hier einen breiten Fundus an Views für nahezu alle Anwendungsfälle. Dieser »Baukasten« lässt sich noch durch selbst programmierte Views erweitern. Wir stellen die für den Staumelder genutzten Oberflächenelemente in der Tabelle 5-9 auf Seite 62 kurz vor. Für die Liste aller Oberflächenelemente verweisen wir auf die Online-Dokumentation [13].

Tab. 5-7
Basislayouts

Name	Beschreibung
`LinearLayout` 	Die Oberflächenelemente werden horizontal oder vertikal zueinander ausgerichtet.
`TableLayout` 	Jedes Oberflächenelement wird in einem Feld einer beliebig großen Tabelle dargestellt. Dieses Layout eignet sich gut für komplexere Eingabeformulare.
`ListView` 	Listendarstellung von Oberflächenelementen. Bei Bedarf wird automatisch eine Bildlaufleiste ergänzt. Das Format der Listenelemente wird für die gesamte `ListView` einheitlich festgelegt.

Name	Beschreibung
`Gallery` 	Stellt eine Liste von Bildern horizontal nebeneinander dar. Scrolling erfolgt automatisch.
`TabHost` 	Layout für Verwendung von `<TabWidget>`-Views, um Reiter-Strukturen darzustellen

Tab. 5-8
Speziellere Layouts

Hinweis

Für jede View existiert eine XML- und eine Java-Repräsentation. Die zu einem XML-Element passende Java-Klasse befindet sich im Package `android.widget` (z.B. `<TextView>` -> `andoid.widget.TextView`).
Beide Repräsentationen haben im Wesentlichen die gleichen Attribute. Somit lässt sich die Liste der Attribute einer View schnell durch einen Blick ins JavaDoc der korrespondierenden Java-Klasse ermitteln.

Tab. 5-9
Elementare Views

Name	Beschreibung
`<TextView>`	Einfache Textausgabe
`<Button>`, `<ImageButton>`	Schaltfläche
`<EditText>`	Formularelement zur Texteingabe
`<CheckBox>`	Ankreuzfeld
`<RadioGroup>`, `<RadioButton>`	Auswahlschalter; von allen `<RadioButton>` einer `<RadioGroup>` kann genau eines ausgewählt werden.
`<Spinner>`	Auswahlliste
`<TabWidget>`	Reiter
`<MapView>`	Darstellung einer Kartenansicht (Google Maps)
`<WebView>`	Darstellung einer HTML-Seite (Browser-Ansicht)

Die ersten Views...

Den einführenden Textabschnitt stellen wir mit einer `<TextView>` dar. Für die Schaltflächen wählen wir `<Button>`-Views. Die Texte wurden bereits als Ressourcen definiert. Wir können also durch Angabe ihrer Ressourcen-Schlüssel auf sie verweisen. Ergänzen wir unsere Seitendefinition `startseite_anzeigen.xml` durch

```
<TextView
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:text="@string/startseiteanzeigen_intro"
/>
<Button
  android:id="@+id/sf_starte_routenauswahl"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:text="@string/app_routeFestlegen"
/>
```

Schlüssel sind Ressourcen.

Da wir die `TextView` im weiteren Code nicht benötigen, kann die Angabe des Schlüsselattributs `android:id` entfallen. Für den `Button` definieren wir implizit eine neue Ressource der Art `id`, um später auf Klicks auf die Schaltfläche reagieren zu können. Schlüssel von Views sind also ebenfalls Ressourcen. Durch die Plus-Notation `@+id/sf_starte_routenauswahl` wird eine *neue* Ressource `sf_starte_routenauswahl` definiert.

Einheitliches Format

Um die Forderung nach einheitlicher Darstellung der Oberflächenelemente zu erfüllen, nutzen wir die in Abschnitt 5.3.5

auf Seite 46 erzeugten Formatvorlagen für den Text und die Schaltflächen. Die vollständige Bildschirmseiten-Definition `res/layout/startseite_anzeigen.xml` ist in Listing 5.20 zusammengefasst.

Listing 5.20 *Bildschirmseite `startseite_anzeigen.xml`*

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

  <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    style="@style/TextGross"
    android:text="@string/startseiteanzeigen_intro"
    android:lineSpacingExtra="2dp"
  />
  <Button
    android:id="@+id/sf_starte_routenauswahl"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    style="@style/SchaltflaechenText"
    android:text="@string/app_routeFestlegen"
  />
  <Button
    android:id="@+id/sf_erfasse_staumeldung"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    style="@style/SchaltflaechenText"
    android:text="@string/app_staumeldungErfassen"
  />
</LinearLayout>
```

5.6.5 Menüs definieren

Nun definieren wir die Menüs für den Startseiten-Dialog. Wir wollen ein Optionsmenü mit zwei Auswahlmöglichkeiten anbieten. Darüber hinaus soll durch langes Klicken auf eine Schaltfläche ein Verweis auf einen Hilfetext angezeigt werden. Wir erstellen also zwei Menüdefinitionen: eine für das Optionsmenü, die andere für alle Kontextmenüs. Das Kontextmenü selbst ist ja für alle Schaltflächen gleich.

XML-Definitionen erstellen

Wir definieren also die Ressourcen `res/menu/hauptmenue.xml` (Listing 5.21) und `res/menu/hilfemenue.xml` (Listing 5.22)

Listing 5.21 *Optionsmenü für `StartseiteAnzeigen`*

```
<menu xmlns:android=
    "http://schemas.android.com/apk/res/android">
  <item
    android:id="@+id/opt_einstellungenAnzeigen"
    android:title="@string/app_einstellungenAnzeigen"
  />
  <item
    android:id="@+id/opt_staumelderBeenden"
    android:title="@string/app_staumelderBeenden"
  />
</menu>
```

Listing 5.22 *Kontextmenü für Schaltflächen*

```
<menu xmlns:android=
    "http://schemas.android.com/apk/res/android">
  <item
    android:id="@+id/opt_hilfe"
    android:title="@string/app_hilfe"
  />
</menu>
```

5.6.6 Activity implementieren

Es wächst zusammen...

Activities verknüpfen die bisher definierten Ressourcen mit der Anwendung. Sie implementieren die Geschäftsprozesse, werten Dialogeingaben aus und reagieren auf Menüauswahl und andere Steuerungsoperationen des Anwenders.

Die Startseite soll ohne Listenfunktionalität oder ähnliche Besonderheiten gestaltet sein. Es reicht also aus, die Klasse `de.androidbuch.staumelder.mobilegui.StartseiteAnzeigen` von `android.app.Activity` abzuleiten, um die Ausgangsbasis für die erste Activity des Staumelders zu erhalten.

Bildschirmseite zuweisen

Als Nächstes wollen wir der Activity die Bildschirmseite `startseite_anzeigen.xml` zuweisen. Zum Erzeugen einer Activity ruft das Betriebssystem deren Methode `onCreate` auf. Dort erfolgt die Verknüpfung zwischen View und Activity.

```
public void onCreate(Bundle icicle) {
  super.onCreate(icicle);
  setContentView(R.layout.startseite_anzeigen);
}
```

Initialisierung fortsetzen

Des Weiteren setzen wir in dieser Methode den Seitentitel und das Hintergrundbild der Startseite. Auch hier referenzieren wir einfach die vorhandenen Ressourcen.

```
public void onCreate(Bundle icicle) {
  super.onCreate(icicle);
  setContentView(R.layout.startseite_anzeigen);
  setTitle(R.string.startseiteanzeigen_titel);
  getWindow().setBackgroundDrawableResource(
      R.drawable.autobahn);
}
```

Optionsmenü laden

Nach Erstellung einer Activity weist das Betriebssystem dieser unter anderem auch »sein« Optionsmenü zu. Wir können durch Überschreiben der Methode `onCreateOptionsMenu` den Aufbau des Menüs frei gestalten.

```
public boolean onCreateOptionsMenu(Menu menu) {
  super.onCreateOptionsMenu(menu);

  getMenuInflater().inflate(       // (1)
    R.menu.hauptmenue, menu);

  return true;
}
```

Das als XML-Ressource definierte Hauptmenü wird über einen `android.view.MenuInflator` in ein `android.view.Menu`-Objekt umgewandelt. Schritt (1) zeigt, wie auf diese Weise das vorhandene Menü erweitert wird. Das Optionsmenü ist nun vollständig angebunden.

Kontextmenüs registrieren

Die `StartseiteAnzeigen`-Activity soll drei Kontextmenüs verwalten, um die Hilfefunktion zu realisieren. Dazu melden wir zunächst bei der Activity an, dass wir für bestimmte Views Kontextmenüs verwenden wollen. Diese Anmeldung ist im Leben einer Activity nur einmal notwendig, wir machen sie also in der `onCreate`-Methode.

```
public void onCreate(Bundle icicle) {
  ...
  registerForContextMenu(
    findViewById(R.id.sf_starte_routenauswahl));
  registerForContextMenu(
    findViewById(R.id.sf_strassenkarte_anzeigen));
  registerForContextMenu(
    findViewById(R.id.sf_erfasse_staumeldung));
  ...
}
```

Wertebelegung

`findViewById` lernen wir im nächsten Kapitel näher kennen. Die Methode ermittelt eine View anhand ihres Ressourcen-Schlüssels. Nach der

Anmeldung muss die Activity noch wissen, mit welchen <item>-Werten sie die Kontextmenüs der registrierten Views füllen soll. Wir füllen also in onCreateContextMenu die Kontextmenüs aller Schaltflächen und bedienen uns dabei, wie bei den Optionsmenüs, des MenuInflator.

```
public void onCreateContextMenu(
    ContextMenu menu,
    View v,
    ContextMenuInfo menuInfo) {
  super.onCreateContextMenu(menu, v, menuInfo);
  switch( v.getId() ) {
    case R.id.sf_erfasse_staumeldung :
      R.id.sf_strassenkarte_anzeigen :
      R.id.sf_erfasse_staumeldung : {
        getMenuInflater().inflate(
            R.menu.demo_kurzes_menue, menu);
        return;
    }
  }
}
```

Dieser Codeabschnitt verdeutlicht noch einmal, dass ein Kontextmenü *pro View* existiert und somit auch pro View definiert werden muss. Wir haben nun die Anbindung aller Menüs an die Activity abgeschlossen. In Listing 5.23 haben wir die erste Activity unserer Anwendung zusammengefasst.

Listing 5.23 *Die vollständige Activity StartseiteAnzeigen*

```
package de.androidbuch.staumelder.mobilegui;

public class StartseiteAnzeigen extends Activity {
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.startseite_anzeigen);
    setTitle(R.string.startseiteanzeigen_titel);
    getWindow().setBackgroundDrawableResource(
        R.drawable.autobahn);
    registerForContextMenu(
      findViewById(R.id.sf_starte_routenauswahl));
    registerForContextMenu(
      findViewById(R.id.sf_strassenkarte_anzeigen));
    registerForContextMenu(
      findViewById(R.id.sf_erfasse_staumeldung));
  }

  public boolean onCreateOptionsMenu(Menu menu) {
    super.onCreateOptionsMenu(menu);
    getMenuInflater().inflate(
```

```
      R.menu.hauptmenue, menu);
    return true;
  }

  public void onCreateContextMenu(
      ContextMenu menu,
      View v,
      ContextMenuInfo menuInfo) {
    super.onCreateContextMenu(menu, v, menuInfo);
    switch( v.getId() ) {
      case R.id.sf_erfasse_staumeldung :
          R.id.sf_strassenkarte_anzeigen :
          R.id.sf_erfasse_staumeldung : {
        getMenuInflater().inflate(
        R.menu.demo_kurzes_menue, menu);
        return;
      }
    }
  }
}
```

5.6.7 Android-Manifest anpassen

Bevor wir das Ergebnis unserer Arbeit im Emulator ausführen können, müssen wir noch das *Android-Manifest* anpassen. Wir wollen die Activity `StartseiteAnzeigen` als Teil der Anwendung deklarieren. Dazu passen wir das vom Eclipse-Plug-in generierte `AndroidManifest.xml` wie folgt an:

Listing 5.24
AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
  xmlns:android=
    "http://schemas.android.com/apk/res/android"
  package="de.androidbuch.staumelder"
  android:versionCode="1"
  android:versionName="0.1.0"
  androidName="Androidbuch Staumelder">

  <application android:icon="@drawable/icon"
              android:label="@string/app_name"
              android:debuggable="true"
              android:theme="@android:style/Theme.Light">
    <activity
      android:name=".mobilegui.StartseiteAnzeigen"
      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>
```

Markierung des Startpunkts

Mit dem hinter `<intent-filter>` verborgenen Themenbereich der *Intents* befassen wir uns in Kapitel 7. Das hier verwendete Konstrukt qualifiziert eine Activity als Start-Activity. Beim Aufruf der Anwendung kann im Run-Dialog von Eclipse angegeben werden, ob eine bestimmte Start-Activity genutzt werden soll. Anderenfalls startet die Anwendung mit der ersten Activity, die mit dem oben beschriebenen `<intent-filter>` markiert wurde.

Einheitliches Theme

Ein schwarzer Hintergrund erschwert die Lesbarkeit der Bildschirminhalte auf Buchseiten. Daher nutzen wir die von Android gelieferte Formatvorlage `android:style/Theme.Light` als Theme für die komplette Anwendung.

5.6.8 Bildschirmdialog im Emulator testen

Starten wir nun den ersten Bildschirmdialog im Emulator. Die dazu erforderlichen Schritte wurden im Eingangsbeispiel erläutert. Wir erhalten das in Abbildung 5-5 auf Seite 69 dargestellte Resultat.

Nun könnten wir eigentlich zufrieden sein. Aber wir müssen uns noch mit einer Besonderheit der Android-Geräte befassen, dem automatischen Wechsel der Bildschirmperspektive (engl. *orientation*). Sobald z.B. beim von uns getesteten Gerät die Tastatur ausgeklappt wird, wechselt die Perspektive automatisch vom Hoch- in Querformat. Wir müssen also unsere Bildschirmseiten immer für beide Perspektiven testen. Dazu drücken wir im Emulator die Tastenkombination `<STRG>+F11` (Abbildung 5-6 auf Seite 69).

Scrolling

Durch den Perspektivenwechsel ist ein Teil unserer Bildschirmseite nicht mehr sichtbar! Eine Lösung dafür ist Scrolling, welches in Abschnitt 5.7.1 auf Seite 70 beschrieben wird.

Sie haben Ihr Ziel erreicht!

Wir haben die Iterationsziele alle erreicht. Ein erster Bildschirmdialog wird im Emulator angezeigt. Wir haben die Zusammenhänge zwischen Oberflächengestaltung und ausführbaren Komponenten einer Android-Anwendung in Theorie und Praxis kennengelernt.

Abb. 5-5
Startseite Hochformat

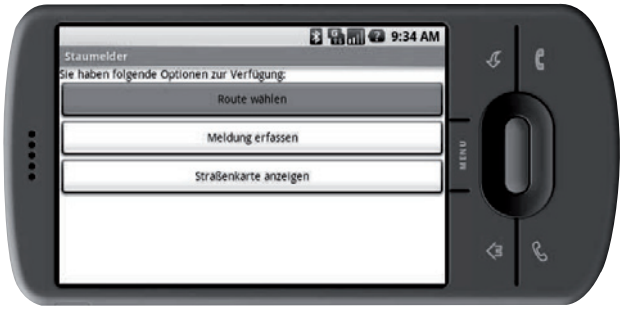


Abb. 5-6
Startseite Querformat

Dennoch ist das Kapitel noch nicht abgeschlossen. In der Praxis stellt man schnell fest, dass das bisher Gelernte nicht ausreicht, um robuste Oberflächen zu erstellen. Was wäre zum Beispiel, wenn der Kunde eine Anwendung mit mehrsprachigen Oberflächen wünscht? Außerdem haben wir schon festgestellt, dass uns der spontane Wechsel der Bildschirmperspektive vor neue Herausforderungen stellt. Aus diesem Grund schließen wir das Kapitel mit einem Abschnitt über fortgeschrittenere Themen der Oberflächengestaltung.

5.7 Tipps und Tricks

In diesem Abschnitt greifen wir einige Punkte auf, die bei der Erstellung von Bildschirmseiten sehr hilfreich sind. Für den Staumelder wer-

den wir sie nicht benötigen, daher verlassen wir kurz das übergreifende Beispiel und stellen die Themen einzeln vor.

5.7.1 Scrolling

Android-Geräte haben kleine Bildschirme. Um größere Datenmengen darstellen zu können, ist Scrolling unverzichtbar. Dies kann den kompletten Bildschirm oder nur einzelne Oberflächenelemente betreffen. Einige Layouts und Views (z.B. `ListView`, `TextView`, `PreferenceView`) haben eine vertikale Bildlaufleiste (engl. *scrollbar*) bereits »serienmäßig« eingebaut. Was aber, wenn nicht mehr alle Elemente eines `LinearLayout` angezeigt werden können?

ScrollView

Für diesen Fall stellt die Android-API ein eigenes Layout, die `android.widget.ScrollView`, bereit. Dieses ergänzt bei Bedarf die vertikale Bildlaufleiste für die darunterliegenden Oberflächenelemente. Im vorliegenden Fall würden wir also die Bildschirmseite wie folgt definieren:

```
<ScrollView xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

  <LinearLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    ...
  </LinearLayout>
</ScrollView>
```

Aber Vorsicht! Eine `ScrollView` sollte nur Elemente enthalten, die *kein* eigenes Scrolling implementieren. In einer `ListView` ist beispielsweise bereits eine eigene, optimierte Scrolling-Logik vorgesehen, die durch die Einbettung in eine `ScrollView` außer Kraft gesetzt würde.

Horizontales Scrolling

Bisher war nur von vertikalem (hoch/runter) Scrolling die Rede. Doch wie erreicht man die Darstellung von Bildschirmelementen, die über den Seitenrand des Bildschirms hinausragen? Leider bietet die Standard-API derzeit noch keine Lösung dieses Problems. Man müsste daher eine eigene View-Klasse von der `ScrollView`-Klasse ableiten und dort das horizontale Scrolling implementieren. Die dafür erforderlichen Schritte führen an dieser Stelle zu weit. Auf der Website `www.androidbuch.de` haben wir eine Musterlösung bereitgestellt.

5.7.2 Umgebungsabhängige Ressourcen

Ressourcen sind fester Bestandteil einer Anwendung. Wenn diese unter verschiedenen Umgebungen (z.B. Landessprachen, Bildschirmgrößen etc.) nutzbar sein soll, hat das Auswirkungen auf die Ressourcen-Definitionen. Bisher haben wir Ressourcen nur für genau eine Umgebung definiert, die wir als *Standardumgebung* bezeichnen wollen. Das Ressourcenverzeichnis war folgendermaßen aufgebaut:

```
/res/drawable
    /layout
    /menu
    /values
```

Android erlaubt uns nun, eigene Umgebungen anhand verschiedener Einschränkungskriterien (engl. *qualifier*) zu definieren. Die aus unserer Sicht häufigsten Kriterien sind in Tabelle 5-10 beschrieben. Wollen wir beispielsweise die Anwendung für verschiedene Sprachvarianten entwickeln, so definieren wir uns anhand des Einschränkungskriteriums »Sprachvariante« je eine Umgebung für die englische und deutsche Fassung der Anwendung.

Kriterium	Beschreibung	Kennung
Sprache	ISO 639-1 Sprachcodierung	en, fr, de
Region	ISO 3166-1-alpha-2 Sprachcodierung mit Präfix »r«	rUS, rDE, rFR
Perspektive	Hochformat (*portrait*), Querformat (*landscape*), Quadratisch (*square*)	port, land, square
Bildschirmgröße	beachte: 640x480 statt 480x640	320x240, 640x480

Tab. 5-10 *Einschränkungskriterien für Ressourcen*

Ein Verzeichnis pro Umgebung

Für jede unterstützte Umgebung legen wir ein eigenes Ressourcenverzeichnis an, wenn wir für die Ressourcen-Art umgebungsabhängige Ressourcen definieren wollen. Die Mehrsprachigkeit wird sich auf die Definition der Text-Ressourcen auswirken. Daher ändern wir unser Ressourcenverzeichnis ab in:

```
/res/drawable
    /layout
    /menu
    /values
    /values-en
```

Im Zweifel zurück zum Standard

Falls in `values-en` kein Wert für einen Ressourcen-Schlüssel definiert ist, greift Android auf die Standardumgebung zurück. Ressourcen müssen also nicht für jede Umgebung kopiert werden.

Flexible Perspektiven

Mit Hilfe umgebungsabhängiger Ressourcen lässt sich auch unser auf Seite 68 aufgetretenes Problem noch eleganter lösen. Wir können für Hoch- und Querformat eigene Layouts definieren. Wir erweitern zur Verdeutlichung des Problems die Startseite um mehrere Schaltflächen (Abbildung 5-7).

Abb. 5-7
Startseite Hochformat

Wechselt die Perspektive auf Querformat, so sehen wir nicht mehr alle Schaltflächen auf dem Bildschirm (Abbildung 5-8).

Abb. 5-8
Startseite Querformat normal

Wir wählen das Hochformat als Standardumgebung, definieren eine eigene Bildschirmseite `startseite_anzeigen.xml` für die Querformat-

Darstellung (Listing 5.25 auf Seite 73) und erweitern das Ressourcenverzeichnis noch einmal:

```
/res/drawable
    /layout
    /layout-land
    /menu
    /values
    /values-en
```

Unter `layout-land` legen wir die neue Seitendefinition an. Wir nutzen diesmal ein `<TableLayout>`, so dass im Querformat immer je zwei Schaltflächen nebeneinander dargestellt werden.

Listing 5.25
Startseite Querformat (vereinfacht!)

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout>
  <TextView
    android:id="@+id/kurzbezeichnung"
  />
  <TableLayout>
    <TableRow>
      <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        style="@style/SchaltflaechenText"
        android:id="@+id/sf_starte_routenauswahl"
        android:text="@string/app_routeFestlegen"
      />
      <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        style="@style/SchaltflaechenText"
        android:id="@+id/sf_erfasse_staumeldung"
        android:text="@string/app_staumeldungErfassen"
      />
    </TableRow>
    ...
  </TableLayout>
</LinearLayout>
```

Das Ergebnis ist in Abbildung 5-9 zu sehen. Alle Schaltflächen werden in Querformat-Darstellung angezeigt.

Abb. 5-9
Startseite Querformat optimiert

5.8 Fazit

In dieser Iteration haben wir gezeigt, dass die Erstellung einfacher Bildschirmseiten für Android nicht übermäßig schwierig ist. Der Abschnitt über Ressourcen deutet aber schon an, dass Oberflächen mobiler Computer nicht mit Oberflächen für Webbrowser oder Rich Clients vergleichbar sind. Es gilt neue Herausforderungen anzunehmen. Dazu gehören wechselnde Bildschirmperspektiven, verschiedene Eingabemedien (berührungsempfindlicher Bildschirm, Track-Ball, Tastatur) und ein Oberflächendesign, das auf einer Vielzahl möglicher Endgerätetypen immer noch seinen Zweck erfüllen muss.

Alles wird gut...

Aber lassen Sie sich nicht beunruhigen. Es bleibt abzuwarten, welche der vielen Konfigurationsparameter in der Praxis relevant sein werden.

Mit der Zeit werden wahrscheinlich weitere Oberflächeneditoren auf den Markt kommen. Der GUI-Editor des Eclipse-Plug-ins ist derzeit noch nicht so gut entwickelt, als dass wir ihn hier hätten vorstellen wollen.

6 Iteration 2 – Oberflächen und Daten

Hinweis: Die 2. Auflage dieses Buchs erscheint unter dem Titel
»Android 2. Grundlagen und Programmierung«
Ende Mai 2010 im dpunkt.verlag, ISBN 978-3-89864-677-2

Im letzten Kapitel haben wir uns mit der Erstellung von Bildschirmseiten beschäftigt. Wir haben Oberflächenelemente auf einer Seite angeordnet und haben diese von einer Activity auf dem Bildschirm darstellen lassen.

In diesem Kapitel lernen wir, wie man die Bildschirmseiten mit Daten füllt. Ein wichtiger Teil einer Anwendung ist die Interaktion mit dem Nutzer über Bildschirmformulare und Schaltflächen oder Menüs. Wir lernen, wie man Eingaben des Nutzers erfasst und auf sie reagieren kann.

6.1 Zielsetzung

Aufgaben

Wir erstellen in diesem Kapitel zwei weitere Bildschirmdialoge: die Übersicht aller Staumeldungen einer Route und den Dialog zur Änderung von Anwendungseinstellungen.

Lernziele

Dabei lernen wir, wie man im Java-Code auf Views zugreift, um ihre Attributwerte zu ändern oder anzupassen. Wir reagieren auf Ereignisse (engl. *events*), die von Views oder Menüs ausgelöst werden. Schritt für Schritt lernen wir weitere Views und Activities, z.B. zur Darstellung von Listen und Systemeinstellungen kennen. Am Ende dieses Kapitels können Sie interaktive Benutzer-Dialoge für nahezu alle Anwendungsbereiche erstellen.

6.2 Arbeiten mit Views

Views sind Objekte!

Bisher haben wir Views nur auf Bildschirmseiten angezeigt. Wir haben sie dazu als XML-Elemente einer Bildschirmseiten-Definition betrachtet. Wir wollen jetzt aber ihre Attributwerte bei laufender Anwendung auswerten und anpassen.

Für diese Anforderungen reicht das statische XML-Format nicht mehr aus. Wir greifen daher auf die View-Objekte zu, die vom Ressourcencompiler aus den XML-Definitionen erzeugt wurden.

Datenträger-View

In diesem Kapitel konzentrieren wir uns auf Views, die Daten aus Datenbanken, Dateisystem oder externen Anwendungen auf der Oberfläche darstellen. Diese Views bezeichnen wir als *Datenträger-Views*.

6.2.1 Zugriff auf Views

View-Schlüssel

Bevor man auf eine View zugreifen kann, muss man sie finden. Als Suchkriterium dient der eindeutige Schlüsselwert, der dem Attribut `id` bei der Definition der View zugewiesen wurde. Diesen Wert bezeichnen wir als *View-Schlüssel*.

`findViewById`

Die Views einer Bildschirmseite sind als Baumstruktur organisiert. Jede View `v` kann über die Methode `v.findViewById` nach einer View suchen, die sich innerhalb des Teilbaumes befindet, dessen Wurzel `v` ist.

Wenn wir beispielsweise auf eine der Schaltflächen der Startseite (definiert in Listing 5.1 auf Seite 40) zugreifen wollen, schreiben wir

```
public class StartseiteAnzeigen extends Activity {
  public void onCreate(Bundle icicle) {
    ...
    Button sfStarteRoutenauswahl =
      (Button)findViewById(R.id.sf_starte_routenauswahl);
  }
}
```

Achtung Laufzeit!

Hier wird `findViewById` von der Activity aus (Activities sind auch Views) verwendet. Die einfache Handhabung dieser Methode täuscht darüber hinweg, dass bei jedem Aufruf der View-Baum von der Wurzel bis zum Ziel durchlaufen werden muss. Daher sollte man zwei Regeln für die Suche nach Views beachten:

Wiederholung vermeiden Das Ergebnis einer View-Suche sollte immer in einer lokalen Variable gespeichert werden, um wiederholte Zugriffe auf die gleiche View zu vermeiden.

Nicht zu früh beginnen *Jede* View kann als Ausgangspunkt der Suche dienen. Manchmal macht es Sinn, erst den Ausgangspunkt zu suchen, um dann weitere Suchen von ihm aus zu beginnen. Betrachten wir die Bildschirmseite in Listing 6.1. Wenn wir nacheinander auf die Views `f2,f3,f4` zugreifen wollen, ist es sinnvoll, erst das `TableLayout` mit Schlüssel `tabelleZwei` zwischenzuspeichern und über dessen `findViewById` auf die Textfelder zuzugreifen.

Listing 6.1 *Suche in komplexen Layouts*

```
<LinearLayout>
  <TextView android:id="@+id/text1" />
  <TextView android:id="@+id/text2" />

  <TableLayout android:id="@+id/tabelleEins">
    <TableRow>
      <TextView android:text="@string/feld1" />
      <TextView android:id="@+id/f1" />
    </TableRow>
  </TableLayout>

  <TextView android:id="@+id/text3" />

  <TableLayout android:id="@+id/tabelleZwei">
    <TableRow>
      <TextView android:text="@string/feld2" />
      <TextView android:id="@+id/f2" />
    </TableRow>
    <TableRow>
      <TextView android:text="@string/feld3" />
      <TextView android:id="@+id/f3" />
    </TableRow>
    <TableRow>
      <TextView android:text="@string/feld4" />
      <TextView android:id="@+id/f4" />
    </TableRow>
  </TableLayout>
</LinearLayout>
```

Einfache Views

Die Attributwerte einer View können innerhalb der Activity beliebig ausgelesen oder geändert werden. Auf diese Weise werden *einfache* Views wie Textfelder, Kontrollkästchen etc. mit Daten versorgt, die sie an der Oberfläche darstellen.

```
TextView eingabeFeld =
  findViewById(R.id.staumeldung.kommentar);
eingabeFeld.setText("Stauende in Kurve");
```

AdapterViews

Doch wie geht man mit Viewgroups um, die Datenträger-Views beinhalten? Wie füllt man beispielsweise Listen (`ListView`) oder Auswahlboxen (`Spinner`) mit Daten? Diese speziellen Viewgroups werden in Android als *AdapterViews* (`android.widget.AdapterView`) bezeichnet.

6.2.2 AdapterViews und Adapter

Aufgaben des Adapters

Als Bindeglied zwischen einer Datenmenge und einer `AdapterView` dienen *Adapter*. Ein Adapter erfüllt zwei Aufgaben:

- Er füllt die `AdapterView` mit Daten, indem er ihr eine Datenquelle liefert.
- Er definiert, welche View bzw. Viewgroup zur Darstellung der einzelnen Elemente der Menge verwendet wird.

Wahl der Adapter

Anhand der Art und Weise, wie die Datenmenge definiert ist, hat man die Wahl zwischen verschiedenen Implementierungen des Interface `android.widget.Adapter`. In diesem Kapitel werden wir unsere Daten in Arrays speichern, daher stellen wir in Listing 6.2 den `ArrayAdapter` vor.

Adapter Schritt für Schritt

Zunächst wird die `AdapterView` ermittelt. Danach werden die anzuzeigenden Daten geladen. Schließlich verbindet der Adapter die View mit den Daten und gibt noch den Ressourcen-Schlüssel des Layouts mit, das die einzelnen Array-Elemente formatiert. Im letzten Schritt wird der Adapter an die `AdapterView` übergeben und die Daten werden auf der Oberfläche angezeigt.

Listing 6.2 *Nutzung eines ArrayAdapter*

```
private void zeigeStaubericht(long routenId) {
  AdapterView listView = findViewById(
    R.id.staubericht_liste);
  String[] staubericht Daten =
    leseStaubericht(routenId);

  ListAdapter staubericht Adapter =
    new ArrayAdapter<String>(
      listView,
      android.R.layout.simple_list_item_1,
      staubericht Daten);
  listView.setListAdapter(
    staubericht Adapter);
}
```

Wo bin ich?

Jede `AdapterView` erlaubt den Zugriff auf die Elemente seiner Datenmenge (`getItemAtPosition(int)`). Die Methoden `getSelectedItem` und `getSelectedItemPosition` liefern Informationen über das aktuell markierte Element der Datenmenge.

Reaktion auf Interaktion

Eine weitere Aufgabe einer `AdapterView` ist es, auf Nutzereingaben, die in ihren Anzeigebereich fallen, zu reagieren. Sie tun dies, indem sie je nach Aktion (Einfachklick, langer Klick etc.) des Nutzers ein Ereignis auslösen, auf das die Anwendung reagieren kann. Damit wären wir beim Thema des nächsten Abschnitts.

6.3 Oberflächenereignisse

Callbacks

Eine `AdapterView` ist nicht die einzige Komponente, die nach Nutzereingaben Oberflächenereignisse auslösen kann. Im letzten Kapitel haben wir uns mit den Methoden befasst, die nach einem Klick auf ein Element eines Kontext- oder Optionsmenüs aufgerufen werden, um auf das Ereignis zu reagieren. Diese Methoden bezeichnen wir als *Callback-Methoden*.

Event-Handler

Callback-Methoden werden von Ereignis-Verarbeitungsklassen, sogenannten *Event-Handlern*, implementiert. Um beim Beispiel von Optionsmenüs zu bleiben, wäre die Activity der Event-Handler für Ereignisse, die vom Menü ausgelöst werden. Sobald eine Menüoption ausgewählt wurde, wird die Callback-Methode `onOptionsItemSelected` des Event-Handlers aufgerufen.

Ereignisbehandlung

Wir demonstrieren Ereignisbehandlung am besten anhand eines Beispiels. Wenn auf die Schaltfläche »Staumeldung erfassen« geklickt wird, soll der entsprechende Bildschirmdialog angezeigt werden.

Event-Handler definieren

Als Erstes definiert man den Event-Handler. Nach einem Blick auf Tabelle 6-1 auf Seite 80 wählen wir den `View.OnClickListener` als auf unser gewünschtes Ereignis passenden Event-Handler aus und erstellen ein Exemplar für unsere Zwecke.

```
View.OnClickListener eventHandler =
  new View.OnClickListener() {
    public void onClick(View ausloeser) {
      // rufe Dialog auf...
    }
  });
```

Und registrieren

Damit die Anwendung auf das `onClick`-Ereignis des `Button` reagieren kann, müssen wir den Event-Handler noch bei der View bekannt machen.

```
Button sfStaumeldung =
  (Button) findViewById(R.id.sf_erfasse_staumeldung);
sfStaumeldung.setOnClickListener(eventHandler);
```

Beispiele folgen …

Nach diesem Prinzip funktioniert die Behandlung von Oberflächenereignissen in Android. Wir werden in den nächsten Abschnitten zeigen, wie Event-Handler für Navigationsaufgaben und die Auswahl von Listen- und Formularelementen eingesetzt werden.

Tab. 6-1 Einige Event-Handler der Android-API

Event-Handler	wird aktiviert, wenn...
View.OnClickListener	die View angeklickt wird
View.OnLongClickListener	die View länger angeklickt wird
View.OnKeyListener	die View eine Eingabe über die Tastatur erkennt
View.OnTouchListener	der berührungsempfindliche Bildschirm (Touch-Screen) einen Klick auf die View meldet
AdapterView.OnItemClickListener	ein Datenelement kurz angeklickt wird
AdapterView.OnItemLongClickListener	ein Datenelement für längere Zeit angeklickt wird
AdapterView.OnItemSelectedListener	ein Datenelement ausgewählt wird

6.4 Implementierung von Listendarstellungen

Ziel: Staubericht anzeigen

Über das Optionsmenü der Staumelder-Startseite werden wir einen neuen Bildschirmdialog aufrufen: den *Staubericht*. Dort wird eine Liste aller der Anwendung bekannten Staumeldungen angezeigt. Da wir im Moment noch nicht wissen, wie man eine Activity aus einer anderen aufruft (Inhalt des Kapitels 7 ab Seite 93), konzentrieren wir uns auf die Erstellung der Listenanzeige.

AdapterView in der Praxis

Dabei lernen wir den Umgang mit einer `AdapterView` und der Ereignisbehandlung in der Praxis kennen.

6.4.1 Bildschirmdialog definieren

`ListActivity`

Wenn ein Bildschirmdialog ausschließlich zur Darstellung einer Liste von Daten benötigt wird, sollte man eine `android.app.ListActivity` als Basis verwenden. Diese erweitert die Funktionalität einer normalen `Activity` durch folgende Punkte:

Layout serienmäßig Die `ListActivity` verfügt über eine implizite `android.widget.ListView` als Wurzel ihrer Bildschirmseite. Eine eigene Layoutdefinition ist möglich, aber nicht notwendig.

Vordefinierte Callbacks Die Callback-Methoden zur Behandlung typischer Ereignisse für Listen (z.B. ein Datenelement wird ausgewählt) sind bereits in der Activity enthalten. Es müssen keine separaten

Event-Handler deklariert werden.

Hilfestellung für Listenzugriffe Jede `ListActivity` bietet Methoden, mit deren Hilfe Informationen über die aktuelle Listenposition des Anzeige-Cursors oder das aktuell ausgewählte Listenelement abgefragt werden können.

Implizites Layout

Man sollte von dem vordefinierten Layout einer `ListActivity` nur in Ausnahmefällen abweichen. Es sorgt dafür, dass die Listenelemente optimal angezeigt werden. Bei Bedarf wird eine vertikale Bildlaufleiste (Scollbar) automatisch ergänzt.

Der Rumpf unserer neuen Activity ist in Listing 6.3 beschrieben. Aus Platzgründen lassen wir hier die Definition und Verwaltung von Menüs weg.

Listing 6.3 *ListActivity StauberichtAnzeigen*

```
public class StauberichtAnzeigen
     extends ListActivity {

  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);

    setTitle(R.string.stauberichtanzeigen_titel);
  }
}
```

6.4.2 Liste mit Daten füllen

Adapter nutzen

Wir gehen in diesem Beispiel davon aus, dass uns ein Staubericht als Array von Staubeschreibungen geliefert wird. Da die `ListView` eine `AdapterView` ist, müssen wir ihr Inhalt und Erscheinungsform ihrer Datenelemente über einen Adapter mitteilen. Dafür definieren wir uns eine Methode `zeigeStaubericht`, in der die Daten angefordert und dargestellt werden.

```
private void zeigeStaubericht() {
  String[] stauberichtDaten =
    leseStaubericht();

  ListAdapter stauberichtAdapter =
    new ArrayAdapter<String>(
      this,
      android.R.layout.simple_list_item_1,
      stauberichtDaten);
```

```
    setListAdapter(
      stauberichtAdapter);
  }
```

Layout der Daten

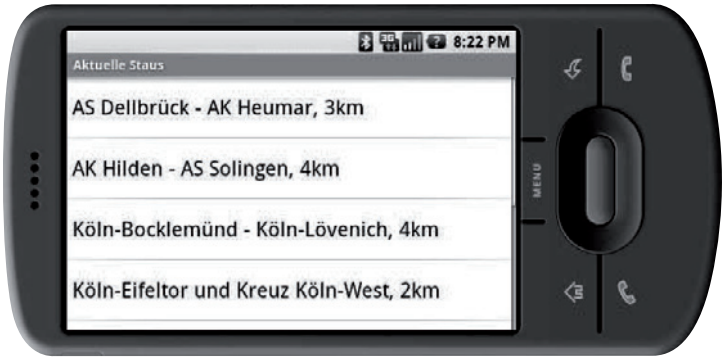
Die Android-API bietet vorgefertigte Layouts zur Darstellung der Daten einer `ArrayView` an (Tabelle 6-2). Bei Bedarf können auch eigene Layoutdefinitionen verwendet werden (s. Online-Dokumentation). Man sollte aber bedenken, dass auf den kleinen Bildschirmen wenig Platz für eigene Kreativität vorhanden ist.

Tab. 6-2
Datenlayouts für ArrayViews

android.R.layout.	Beschreibung
simple_gallery_item	Einzelbild in Gallery-View
simple_list_item_1	Ein-Elementige Liste
simple_list_item_2	Zwei-Elementige Liste
simple_list_item_checked	Liste mit Checkbox
simple_list_item_multiple_choice	Liste mit Mehrfachauswahl
simple_list_item_single_choice	Liste mit Checkbox

Für die Staudarstellung reicht uns die Anzeige eines Elementes pro Listeneintrag völlig aus. Wir wählen also den Schlüssel `android.R.layout.simple_list_item_1`. Abbildung 6-1 zeigt das Ergebnis.

Abb. 6-1
Der Staubericht



Leere Datenmenge

Falls die Datenmenge leer ist, wird die View standardmäßig ohne Werte angezeigt. Es ist jedoch möglich, mittels `AdapterView::setEmptyView` eine Referenz auf eine View zu übergeben, die in diesem Fall als Platzhalter angezeigt wird. Auf diese Weise ließe sich beispielsweise eine `TextView` mit einem Hinweis darstellen.

```
TextView hinweisKeinStau = new TextView(this);
hinweisKeinStau.setText(R.string.keine_staus);
getListView().setEmptyView(hinweisKeinStau);
```

Die Definition der `emtpyView` scheint derzeit nur dann zu funktionieren, wenn man ein *eigenes* Layout für die `ListActivity` definiert.

6.4.3 Auf Listenauswahl reagieren

Wenn der Nutzer ein Listenelement auswählt, soll dessen Inhalt kurz in einem kleinen Fenster angezeigt werden. Wir müssen also erkennen, dass ein Auswahl-Ereignis aufgetreten ist, und durch Anzeige einer anderen View darauf reagieren.

Callback implementieren

Die `ListActivity` sieht eine Callback-Methode `onListItemClick` vor, die wir jetzt für unsere Zwecke implementieren wollen. Die Callback-Methode kennt die `ListView`, die das Ereignis ausgelöst hat, und die View des gewählten Datenelementes.

Ein Toast

Die ausgewählten Daten sollen kurz in einem kleinen Fenster angezeigt werden. Android bietet dafür ein Oberflächenelement `android.widget.Toast` an. Ein Toast ist für die kurze Darstellung von Hinweistexten geeignet. Die Verwendung wird im folgenden Beispiel deutlich.

```
protected void onListItemClick(ListView liste,
    View datenElement, int position, long id) {
  super.onListItemClick(
    liste, datenElement, position, id);

  final Toast hinweis = Toast
      .makeText(this, "Element "
          + ((TextView) datenElement).getText(),
          Toast.LENGTH_LONG);
  hinweis.show();
}
```

Abbildung 6-2 zeigt den Bildschirm nach Auswahl eines Listenelementes. Der Bildschirmdialog ist fertiggestellt.

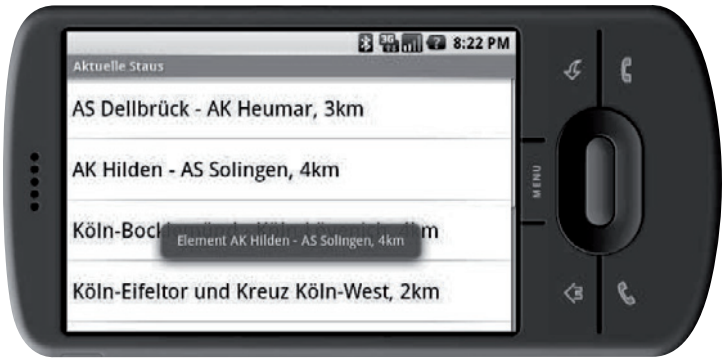


Abb. 6-2
Ein Toast

6.5 Anwendungseinstellungen

Konfiguration

Viele Anwendungen benötigen nutzer- oder gerätespezifische Konfigurationen. Diese müssen während einer Anwendungssitzung angezeigt und bearbeitet werden können. Nach Beendigung der Anwendung darf deren Konfiguration nicht gelöscht werden. Diese muss beim nächsten Start unverändert wieder zur Verfügung stehen.

Noch keine Datenbanken

Die Android-API unterstützt uns bei der Verwaltung und Darstellung dieser Konfigurationseinstellungen, ohne dass dazu Datenbankkenntnisse notwendig sind. Sie stellt spezielle Oberflächenelemente und eine eigene Activity bereit, um die Darstellung von Einstellungen zu vereinfachen. Diese Komponenten sowie die zur Verwaltung der Konfigurationsdaten empfohlenen Klassen werden wir in diesem Abschnitt kennenlernen.

Staumelder-Konfiguration

Der Staumelder benötigt unter anderem Informationen darüber, wie er die Internetverbindung zum Staumelder-Server aufbauen soll. Diese Parameter sollten nicht als Java-Code oder Ressourcen definiert werden, da sie sich ändern können, ohne dass eine Neuinstallation der Anwendung gewünscht wird.

Ziel: Einstellungen bearbeiten

Unser Ziel ist es, zur Verwaltung der Anwendungseinstellungen des Staumelders einen Bildschirmdialog zu erstellen, der an das Hauptmenü angebunden werden kann.

6.5.1 Begriffsdefinitionen

Einstellungsparameter

Als *Einstellungsparameter* (engl. *preference*) definieren wir ein Schlüssel-Wert-Paar, das zur Konfiguration einer Anwendung dient. Der Schlüssel eines Einstellungsparameters wird bei Erstellung der Anwendung definiert. Sein Wert wird dagegen erst während einer Anwendungssitzung vergeben. Der Wert kann jederzeit vom Nutzer der

Anwendung geändert werden und bleibt auch nach Anwendungsende gespeichert.

Parametergruppen

Einstellungsparameter können in *Parametergruppen* organisiert werden. Für jede Parametergruppe kann ein innerhalb der Anwendung eindeutiger Name vergeben werden. Eine Anwendung kann dann anhand dieses Namens auf die Parametergruppe zugreifen.

Anwendungseinstellungen

Als *Anwendungseinstellungen* bezeichnen wir die Einstellungsparameter, die für die gesamte Anwendung definiert und zugreifbar sind. Anwendungseinstellungen sind also eine Parametergruppe, die meist den Namen der Anwendung trägt.

Einstellungen immer lokal

Anwendungseinstellungen sind nur *innerhalb einer Anwendung* gültig. Sollen einzelne Parameter auch für andere Anwendungen sichtbar gemacht werden, muss ein *Content Provider* implementiert werden (s. Kapitel 12 auf Seite 189).

6.5.2 Einstellungen definieren

PreferenceScreen

Definieren wir nun die Anwendungseinstellungen für den Staumelder. Über einen `android.preference.PreferenceScreen` werden alle Einstellungsparameter einer Parametergruppe zusammengefasst. Wir definieren die Staumelder-Einstellungen in einer XML-Datei `staumelder_einstellungen.xml` im Ressourcenverzeichnis `res/xml`. Listing 6.4 zeigt einen Auszug der Staumelder-Einstellungen.

Einstellungsparameter sind keine Views.

Zusammen mit einem Einstellungsparameter definiert man auch dessen Oberflächendarstellung (Textfeld, Auswahlliste etc.). Auch wenn der Aufbau des `PreferenceScreen` an den einer `ViewGroup` erinnert, handelt es bei ihm und seinen Elementen *nicht* um Views. Daher haben wir den `PreferenceScreen` auch nicht unterhalb von `res/layout`, sondern in `res/xml` definiert.

Listing 6.4 *Beispiel für einen PreferenceScreen*

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android=
    "http://schemas.android.com/apk/res/android"
>
  <PreferenceCategory
    android:title="@string/cfg_verbindungsdatenTitel"
  >
    <EditTextPreference
      android:key="username"
      android:title="@string/cfg_verbindungsdatenLogin"
    />
```

```
    <EditTextPreference
      android:key="password"
      android:title="@string/cfg_verbindungsdatenPasswort"
    />
  </PreferenceCategory>
</PreferenceScreen>
```

Darstellung der Einstellungen

Im Listing 6.4 werden Einstellungsparameter vom Typ »Text« definiert. Die Abbildungen 6-3 und 6-4 zeigen, wie ein etwas komplexerer `PreferenceScreen` dargestellt wird und wie die Änderung eines Textparameters durchgeführt wird.

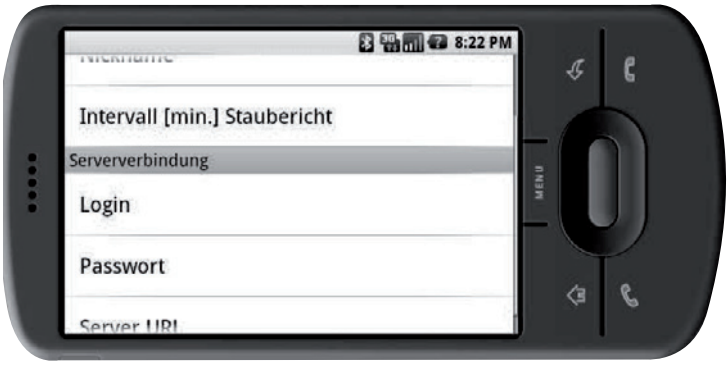


Abb. 6-3
Anwendungseinstellungen des Staumelders

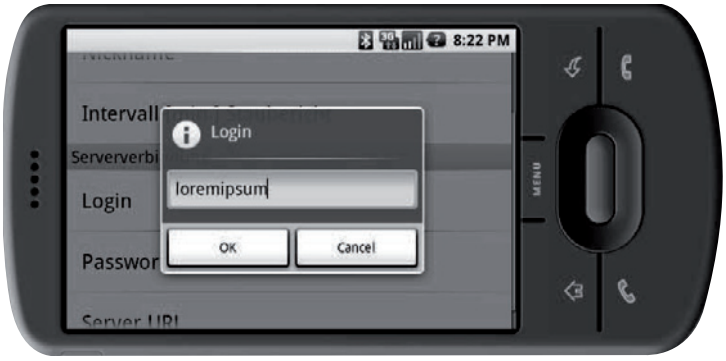


Abb. 6-4
Textparameter ändern

Weitere Möglichkeiten…

Für jeden Einstellungsparameter kann ein Standardwert vergeben werden. Neben Texteingabefeldern werden auch Auswahlboxen, Checkboxen etc. zur Darstellung der Einstellungen angeboten. Ein `PreferenceScreen` kann weitere `PreferenceScreens` als Elemente haben.

Einstellungen bekannt machen

Nachdem alle Einstellungsparameter definiert sind, wollen wir sie in der Anwendung verwenden.

6.5.3 Auf Einstellungen zugreifen

SharedPreferences

Zur Verwaltung von Anwendungseinstellungen dienen Implementierungen der Schnittstelle `android.content.SharedPreferences`. Diese bietet typsicheren Zugriff auf Einstellungsparameter.

Zugriff auf Anwendungseinstellungen

Um ein Exemplar der `SharedPreference` zu erhalten, greift man auf die Methode `getSharedPreferences(String name, int mode)` zurück, die vom Anwendungskontext angeboten wird. Es wird pro `name` ein Konfigurationsdatensatz erzeugt. Es empfiehlt sich also, für globale Anwendungseinstellungen den Namen der Gesamtanwendung, z.B. »Staumelder«, zu wählen.

Als mögliche Modi definiert Context die in Tabelle 6-3 aufgeführten Werte. Im Normalfall wird für diese Komponenteneinstellungen auf `MODE_PRIVATE` zurückgegriffen. Die anderen Modi (`MODE_WORLD_READABLE`, `MODE_WORLD_WRITEABLE`) beeinflussen den Zugriff auf die im Dateisystem abgelegte Datei mit den Einstellungen. Da Einstellungen derzeit nur innerhalb einer Anwendung sichtbar sein dürfen, sind die beiden letztgenannten Modi ohne Auswirkung.

Modus	Erlaubter Zugriff
`MODE_PRIVATE`	nur innerhalb der Anwendung
`MODE_WORLD_READABLE`	Lesezugriff durch andere Anwendungen
`MODE_WORLD_WRITEABLE`	Vollzugriff durch andere Anwendungen

Tab. 6-3
Zugriffsmodi für Konfigurationen

Zugriff auf Activity-Einstellungen

Die aktuellen Konfigurationseinstellungen einer Activity erhält man über `getPreferences(int mode)`. Allgemein werden Einstellungen von Android-Komponenten intern wie Anwendungseinstellungen gespeichert. Sie erhalten als Schlüsselnamen den Klassennamen der Activity.

Achtung!

Anwendungseinstellungen dürfen nicht den Namen einer der Android-Komponenten der Anwendung erhalten.

6.5.4 Einstellungen bearbeiten

Nun befassen wir uns damit, wie wir die Werte von Einstellungsparametern auf unsere Bedürfnisse anpassen und speichern können. Hier bietet uns die Android-API zwei Vorgehensweisen an:

Anpassung per Programmcode Dies ist sinnvoll, wenn wir den »Zustand« einer Activity (oder einer anderen Android-Komponente) zwischenspeichern wollen. Als *Zustand* definieren wir hier die Werte der Attribute einer Activity zu einem bestimmten Zeitpunkt. Activities haben meist eine kurze Lebensdauer und müssen ihren Zustand zwischen zwei Aufrufen häufig sichern (z.B. bei Unterbrechungen durch einen Telefonanruf).

Anpassung per Benutzer-Dialog Die nutzerabhängigen Einstellungsparameter sollten über die Oberfläche verändert werden können. Dabei nimmt uns eine spezielle Activity des Android-SDK, die auf den nächsten Seiten beschriebene `PreferenceActivity`, den Großteil der Arbeit ab.

Änderung von Einstellungsparametern

Zustand speichern SharedPreferences.Editor

Beginnen wir mit der Speicherung des Zustands einer Activity. Jede Implementierung von `SharedPreferences` verfügt über ein Exemplar von `SharedPreferences.Editor`. Über dieses können Änderungen an den Einstellungsparametern gemacht werden.

```
SharedPreferences einstellungen = getPreferences(MODE_PRIVATE);
SharedPreferences.Editor editor = einstellungen.edit();
```

Im Editor ändern…

Da wir den Zustand der Activity speichern wollen, definieren wir für jedes ihrer Attribute einen Einstellungsparameter und füllen diesen mit dem aktuellen Attributwert.

```
editor.putBoolean("nutzeVerschluesselung", this.verschluesseln);
```

commit

Wirksam werden die Änderungen allerdings erst nach dem Aufruf von `commit`. Von diesem Zeitpunkt an kann mit `getPreferences(MODE_PRIVATE).getBooleanValue(Schlüsselname, defaultWert)` auf den geänderten Wert zugegriffen werden. Listing 6.5 fasst die Schritte noch einmal zusammen.

Listing 6.5 *Ändern von Einstellungen*

```
public class BeispielActivity extends Activity {
  ...
  private Boolean verschluesseln = Boolean.TRUE;
  ...
  protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);

  SharedPreferences einstellungen = getPreferences(MODE_PRIVATE);
  SharedPreferences.Editor editor = einstellungen.edit();
```

```
  editor.putBoolean(
    "nutzeVerschluesselung", this.verschluesseln);
  editor.commit();
}
```

Einstellungen im Bildschirmdialog pflegen

PreferenceActivity

Wenn die Anwendungseinstellungen über einen Bildschirmdialog gepflegt werden sollen, sollte man eine `android.preference.PreferenceActivity` nutzen. Diese benötigt lediglich eine Referenz auf die XML-Datei mit den Einstellungsdefinitionen.

Listing 6.6 *Bearbeitung der Anwendungseinstellungen*

```
public class EinstellungenBearbeiten
       extends PreferenceActivity {
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    this.addPreferencesFromResource(
       R.xml.staumelder_einstellungen);
  }
}
```

Alles automatisch

Aus dieser Datei wird die Bildschirmseite aufgebaut (Abbildung 6-3 auf Seite 86). Alle Änderungen an den Einstellungen werden sofort gespeichert, ohne dass weiterer Implementierungsaufwand entsteht.

Vorsicht Falle!

In Listing 6.6 fällt auf, dass *kein Name* für die Anwendungseinstellungen vergeben wird. Dabei handelt es sich um eine, leider nicht gut dokumentierte, Besonderheit der `PreferenceActivity` (oder um einen Fehler). Alle `PreferenceActivities` einer Anwendung schreiben *alle* von ihr verwalteten Einstellungsparameter in *eine* Datei mit dem Namen `${package.name}_preferences.xml` im Verzeichnis `/data/data/${package.name}/shared_prefs`.

Unsere Staumelder-Einstellungen befinden sich also in der Datei `de.androidbuch.staumelder_preferences.xml`.

Hilfmethode

Damit wir diesen Namen nicht in jeder Activity, die die Anwendungseinstellungen nutzt, zusammenbasteln müssen, definieren wir eine weitere Methode in unsere Activity `EinstellungenBearbeiten`.

```
public static final SharedPreferences
    getAnwendungsEinstellungen(ContextWrapper ctx) {
  return ctx.getSharedPreferences(
    ctx.getPackageName()
    + "_preferences", 0);
}
```

6.6 Fortschrittsanzeige

Wartezeit überbrücken

Es wird sich nicht vermeiden lassen, dass der ein oder andere Prozess länger läuft, als zunächst erwartet wurde. Eine nutzerfreundliche Anwendung informiert die Wartenden, dass das System zwar beschäftigt ist, es aber weiter vorangeht.

Progress bars

Dazu bedient man sich gerne sogenannter Fortschritts- oder Verlaufsanzeigen (engl. *progress bars*). Zur Darstellung des Prozessfortschritts haben wir die Wahl zwischen den Views `android.widget.ProgressBar` und `android.app.ProgressDialog`.

ProgressBar

Eine `ProgressBar` verwendet man, wenn man die Statusanzeige fest in eine Bildschirmseite integrieren will. Während diese immer wieder aktualisiert wird, kann der Nutzer im Vordergrund z.B. Eingaben auf der Seite durchführen. Ein Codebeispiel für die Implementierung einer solchen Komponente ist auf der JavaDoc-Seite der API-Dokumentation enthalten und braucht daher hier nicht wiederholt zu werden.

ProgressDialog

Ein `ProgressDialog` zeigt für den Zeitraum der Hintergrundoperation eine modale Dialogbox mit passendem Informationstext an. Diese Komponente ist flexibel nutzbar, wie das folgende Beispiel in Listing 6.7 zeigt.

Listing 6.7
Beispiel Fortschrittsanzeige

```
public class RoutenManager extends Activity {
  private ProgressDialog verlauf;
  ...
  @Override
  public void onCreate(Bundle savedInstanceState) {
    ...
    Button sfDateiAnlegen =
      (Button) findViewById(R.id.sf_dateiAnlegen);
    sfDateiAnlegen
        .setOnClickListener(new View.OnClickListener() {
          public void onClick(View view) {
            verlauf = ProgressDialog.show(
                       RoutenManager.this,
                       "Bitte warten...",
                       "Routendatei wird erstellt",
                       true, // zeitlich unbeschränkt
                       false); // nicht unterbrechbar
            new Thread() {
              public void run() {
                speichereTestroute();
                verlauf.dismiss(); // dialog schließen
              }
            }.start();
          }
```

```
        });
    ...
  }
  ...
}
```

Die genauen Attribute zur Konfiguration der Dialog-Box sind hier nur angedeutet und können bei Bedarf dem Android-Java-Doc entnommen werden. Abbildung 6-5 zeigt eine Fortschrittsanzeige des Staumelders.

Abb. 6-5
Der `ProgressDialog` im Einsatz

6.7 Fazit

Mit diesem Kapitel ist der Einstieg in die Gestaltung von Bildschirmoberflächen für Android abgeschlossen. Wir können nun Bildschirmseiten erstellen und auf Eingaben des Nutzers reagieren.

Weitere Quellen

Wir können in diesem Buch die einzelnen Themen nur kurz umschreiben. Für das weitere Studium empfehlen wir die Online-Dokumentation von Google sowie die mit dem SDK gelieferten Codebeispiele `samples/ApiDemos`.

Verbindung herstellen

Im nächsten Kapitel befassen wir uns damit, wie Bildschirmdialoge miteinander verknüpft werden.

7 Exkurs: Intents

Hinweis: Die 2. Auflage dieses Buchs erscheint unter dem Titel
»Android 2. Grundlagen und Programmierung«
Ende Mai 2010 im dpunkt.verlag, ISBN 978-3-89864-677-2

Bereits in Teil I in Abschnitt 1.4 haben wir den Begriff des *Intent* als »Absichtserklärung« kennengelernt. Dort hat eine Activity (`StaumeldungErfassen`) eine andere Activity (`StauinfoAnzeigen`) gestartet. Dies ist auch der häufigste Einsatzzweck für Intents, aber nicht der einzige. Intents sind vielmehr Teil eines Mechanismus zum Austausch von Nachrichten und Daten zwischen Komponenten einer Anwendung, verschiedenen Anwendungen untereinander oder mit der Android-Plattform.

Intents verbinden unabhängige Komponenten, also Activities, Services, Content Provider oder Broadcast Receiver, untereinander zu einem Gesamtsystem und schaffen eine Verbindung zur Android-Plattform. Letzteres erfolgt über sogenannte Broadcast Intents. Es handelt sich dabei um Systemnachrichten, die wir im Exkurs über Systemnachrichten (siehe Kap. 9 auf Seite 141) gesondert betrachten.

Wir lernen nun auf einem allgemeinen Niveau das Konzept der Intents kennen. Intents tauchen jedoch im weiteren Projektverlauf immer wieder auf, so dass uns das Thema auch durch die weiteren Kapitel begleitet.

7.1 Warum gibt es Intents?

Android ist komponentenbasiert. Da liegt es nahe, die Komponenten mit einem Mechanismus zu verbinden, der standardisiert ist, sich ohne Quellcodeänderung nutzen lässt und einfach zu dokumentieren ist. Auf diese Weise kann eine Komponente auch von anderen Programmierern genutzt oder einfach gegen eine andere ausgetauscht werden.

Ähnlich wie die Android-Plattform verwenden die meisten mobilen Plattformen ein Sandbox-Modell. Eine Interaktion zwischen Anwendungen oder Teilen zweier verschiedener Anwendungen ist meist nicht möglich. Bei Android unterliegen die Intents dem Berechtigungssystem. Zugriff auf Komponenten außerhalb der eigenen Anwendung ist daher nicht erlaubt, solange man nicht explizit geeignete Berechtigungen vergibt.

Durch Intents erreichen wir eine lose Kopplung der Bestandteile einer Anwendung. Dies wird durch zwei verschiedene Arten von Intents verschieden stark unterstützt. Denn wir können explizite und implizite Intents verwenden.

7.2 Explizite Intents

Bei einem expliziten Intent ist die Empfängerkomponente bereits bei der Programmierung des Aufrufs bekannt und eindeutig identifiziert.

Klassenname angeben!

Die Empfängerkomponente wird im Allgemeinen bei Erzeugung des Intent durch ihren Namen oder die Klassenangabe übergeben. Wollen wir also eine Activity `RouteFestlegen` innerhalb unseres Pakets aufrufen, so schreiben wir:

```
Intent i = new Intent(this, RouteFestlegen.class);
i.putExtra("routenId", myRouteId);
startActivity(i);
```

Als ersten Parameter übergeben wir dem Konstruktor des Intent den Android-Context der Zielkomponente. Da sich die aufrufende Activity im gleichen Paket wie die Ziel-Activity befindet, können wir die Activity selbst als Context (»`this`«) verwenden. Der gleiche Intent ließe sich auch wie folgt abschicken:

```
Intent i = new Intent();
i.setComponent(new ComponentName(
    "de.androidbuch.staumelder.mobilegui",
    "RouteFestlegen"));
startActivity(i);
```

Laufzeitfehler möglich

Hier haben wir die Zielklasse über ihren Namen definiert. Nachteil dieser Adressierungsform ist, dass die Prüfung auf das korrekte Ziel nicht vom Compiler übernommen werden kann. Fehler treten daher erst zur Laufzeit, beim ersten Aufruf des Intent, auf.

7.3 Implizite Intents

Implizite Intents adressieren keinen bestimmten Empfänger und überlassen es den Komponenten, auf den Intent zu reagieren.

Implizite Intents spezifizieren keine Komponente, an die sie adressiert sind. Sie werden praktisch »ins Leere« abgeschickt, in der Hoffnung, dass es eine oder mehrere Komponenten gibt, die mit diesem Intent etwas anfangen können. Oft werden implizite Intents verwendet, wenn man Komponenten anderer Anwendungen nutzen möchte. Beispielsweise lässt sich über einen impliziten Intent die *Dialer*-Activity nutzen, die es anderen Komponenten erlaubt, eine Telefonnummer zu wählen.

Auf gut Glück

```
Intent intent = new Intent(Intent.ACTION_DIAL,
    Uri.parse("tel:(0228)1234567"));
startActivity(intent);
```

»*Dialer*« ist eine der vorinstallierten Standardanwendungen der Android-Plattform. Das Abschicken des Intent ruft die Activity zum Wählen einer Telefonnummer auf (Abb. 7-1).

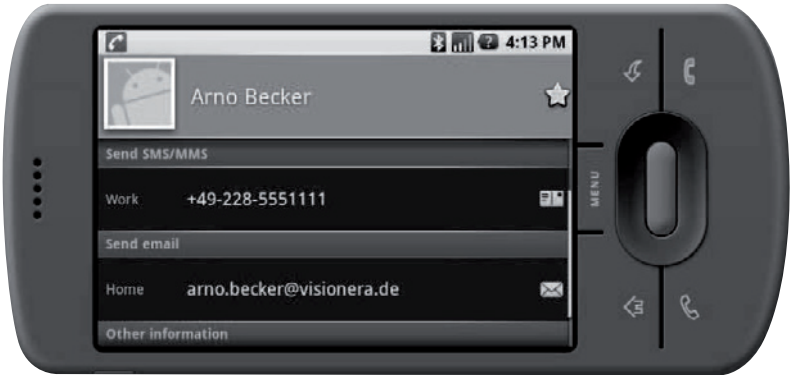


Abb. 7-1
Dialer-Activity im Emulator

Wie wir oben gesehen haben, wird bei impliziten Intents ein Intent-Bezeichner (`Intent.ACTION_DIAL`) mitgegeben. Die Zielkomponente legt selbst fest, auf welche Intents sie reagieren möchte. Im folgenden Abschnitt schauen wir uns an, wie man Intent-Filter deklariert.

Die Zielkomponente bestimmt ihre Intents.

7.4 Intent-Filter für implizite Intents

Intent-Filter werden im Android-Manifest deklariert. Einen Intent-Filter für implizite Intents kennen wir schon. Um eine Activity zur Start-Activity unserer Anwendung zu machen, müssen wir einen bestimmten Intent-Filter deklarieren. Unsere bekannte Activity `StartseiteAnzeigen` hatten wir mit einem Intent-Filter versehen.

Listing 7.1 *Definition eines Intent-Filters*

```
<activity
  android:name=".mobilegui.StartseiteAnzeigen"
  android:label="@string/app_name">
  <intent-filter>
    <action android:name="android.intent.action.MAIN"/>
    <category
      android:name="android.intent.category.LAUNCHER"/>
  </intent-filter>
</activity>
```

Implizite Intents dokumentieren!

Ein Intent-Filter besteht aus bis zu drei Elementen, mit deren Hilfe er genau festlegen kann, auf welche Intents er reagiert. Es ist allerdings auch wichtig, die Intent-Filter anderer Anwendungen zu kennen, wenn man deren Komponenten verwenden möchte. Daher ist es wichtig, die Intent-Filter gut zu dokumentieren, wenn man Teile seiner Anwendung öffentlich macht.

Was soll passieren?

action Das `android:name`-Attribut des `action`-Tags des Intent-Filters muss eine eindeutige Zeichenkette sein. Sie bezeichnet die Aktion, die stattfinden soll. In der Praxis verwendet man hier gerne den Paketnamen der Komponente, die den Intent empfangen soll, plus einen Bezeichner für die Aktion.

Die Klasse `android.content.Intent` besitzt einige generische Action-Intents, nach denen man im Intent-Filter filtern kann. Sie erlauben den Zugriff auf Komponenten, die Bestandteil der Anwendungen der Android-Plattform sind. Beispielsweise konnten wir oben mittels des Action-Intents `Intent.ACTION_DIAL` eine Activity zum Wählen einer Telefonnummer nutzen. Die generischen Action-Intents beginnen mit `ACTION_`.

Eine interessante Sammlung bekannter Intents und deren Adressierung ist unter [6] zu finden. Die Standard-Intents der Android-API sind in der Online-Dokumentation unter [15] aufgelistet.

Unter welchen Bedingungen?

category `category` legt fest, unter welchen Bedingungen der Intent ausgeführt werden soll. Die Kategorie `LAUNCHER` gibt es beispielsweise nur für Activities und legt fest, dass die Activity im Android-Gerät auf der Übersichtsseite aller installierten Anwendungen aufgeführt wird.

Wichtig ist auch die Kategorie »`DEFAULT`«. Sie legt fest, dass diese Activity ausgeführt werden soll, wenn der Intent empfangen wird. Denn es können beliebig viele Activities auf einen Intent reagieren. Werden Activities der eigenen Anwendung per implizitem Intent aufgerufen, muss die Activity einen Intent-Filter mir der `DEFAULT`-Kategorie besitzen.

data Bei Aufruf eines Intent wird dem Intent eine URI (*Uniform Resource Identifier*) mitgegeben. Diese URI ist das `data`-Attribut, welches eine Datenquelle spezifiziert. Wir haben oben schon einen Intent mit einer URI als `data`-Attribut kennengelernt: `new Intent(Intent.ACTION_DIAL, Uri.parse(""tel:(0228)1234567""))`. Die Komponente, die durch Intents gestartet wird, die der Intent-Filter passieren lässt, kann auf den Daten operieren, die das `data`-Attribut spezifiziert. Für das `data`-Attribut können folgende Attribute verwendet werden. Dabei kann ein Intent-Filter auch aus mehreren `data`-Attributen bestehen.

Mit welchen Daten?

- `android:scheme` Gibt das Schema für den Zugriff auf die Daten an. Im Zusammenhang mit Android sind `content`, `http` und `file` häufig gebrauchte Werte. Mindestens das `scheme`-Attribut muss innerhalb von `data` definiert sein. Es können ein oder mehrere `scheme`-Attribute gesetzt werden.
- `android:mimetype` Angabe eines Mime-Typen. Es können selbstdefinierte oder bekannte Mime-Typen verwendet werden. Zulässing ist z.B. `image/jpeg` oder der selbstdefinierte Mime-Typ `vnd.androidbuch.cursor.item/*`. Das Sternchen (»Asterisk«) bedeutet, dass jeder Subtyp akzeptiert wird. In Kapitel 12 über Content Provider werden wir eigene Mime-Typen definieren, um über den Content Provider auf Daten in einer eigenen Datenbank zuzugreifen.
- `android:host` Angabe eines Hostnamens (z.B. »`developer.android.com`«)
- `android:path` Setzt voraus, dass das `host`-Attribut gesetzt wurde. Gibt den Pfad zu den Daten an. Im Falle einer URL z.B. `reference/android/content/Intent.html`. Will man mehr Flexibilität und Zugriff auf bestimmte Daten in Teilbäumen der Pfadstruktur zulassen oder mit Wildcards arbeiten, kann man statt `path` die Attribute `pathPrefix` oder `pathPattern` verwenden. Näheres zu deren Verwendung findet sich in der Dokumentation des Android-SDK ([12]).
- `android:port` Angabe eine Ports (z.B. 8080). Wird ignoriert, wenn `host` nicht gesetzt wurde.

Schauen wir uns das an einigen weiteren Beispielen an. Die Android-Standardanwendung *Maps* kann zur Darstellung eines Ortspunkts genutzt werden.

```
Uri uri = Uri.parse("geo:50.7066272,7.1152637");
Intent intent = new Intent(Intent.ACTION_VIEW);
intent.setData(uri);
startActivity(intent);
```

Dieser Intent startet eine Activity der Anwendung *Maps*, wie in Abbildung 7-2 zu sehen. Wir haben hier die URI für den Ortspunkt über die Methode `setData` gesetzt und nicht über den Konstruktor des Intent. Dadurch wird deutlicher, dass wir mit der URI Daten an die Activity übergeben.

Abb. 7-2
Google-Maps-Activity

Mit dem folgenden Intent starten wir eine Activity der *Browser*-Anwendung und rufen darin die Internetadresse `www.visionera.de` auf. Abbildung 7-3 zeigt das Ergebnis des Intent im Emulator.

```
Uri uri = Uri.parse("http://www.visionera.de");
Intent intent = new Intent(Intent.ACTION_VIEW, uri);
startActivity(intent);
```

Abb. 7-3
Ansicht im Emulator

Nun kann es aber vorkommen, dass die Komponente, die unser Intent aufrufen soll, gar nicht existiert, z.B. weil wir die dazugehörige Anwendung gar nicht auf unserem Android-Gerät installiert haben. Falls keine passende Zielkomponente gefunden wird, wird ein Laufzeitfehler (z.B. `ActivityNotFoundException`) ausgelöst.

Hört mich jemand?

Um dem vorzubeugen, ist es gut, wenn man bereits vor dem Aufruf des Intent weiß, ob die Zielkomponente auf dem Gerät installiert ist. Mit dieser Kenntnis könnte man die `ActivityNotFoundException` vermeiden. Denn diese Exception wird nicht innerhalb der eigenen Anwendung, sondern innerhalb des Package Managers geworfen, der Teil der Android-Plattform ist. Denn der Intent wird fehlerfrei aus der Anwendung heraus verschickt. Jedoch kann die Android-Plattform keinen Empfänger finden und erzeugt die `ActivityNotFoundException`. Das Ergebnis ist eine Meldung auf dem Bildschirm des Android-Geräts, wie sie in Abbildung 7-4 zu sehen ist.

Abb. 7-4
Ergebnis eines Intent ohne Empfänger

Listing 7.2 demonstriert, wie man innerhalb einer Activity prüfen kann, ob es einen Empfänger für den Intent gibt. Dazu befragt man den Package Manager. Man erhält ihn aus dem `Context` der Activity, der gleichbedeutend mit `this` ist.

Listing 7.2
Verfügbarkeitsprüfung für Intents

```
private boolean isIntentErreichbar(Intent intent) {
  final PackageManager pm = this.getPackageManager();
  List<ResolveInfo> list =
      pm.queryIntentActivities(intent,
          PackageManager.MATCH_DEFAULT_ONLY);
  return list.size() > 0;
}
```

Fassen wir zusammen. Wir haben nun Intents und Intent-Filter kennengelernt. Wir wissen, dass wir vorhandene Komponenten verwenden können. Dabei ist es egal, ob wir sie selbst implementiert haben oder ob

sie zu einer anderen Anwendung gehören, die auf dem Android-Gerät installiert ist. Wenn wir den Intent-Filter der Komponente kennen, die wir verwenden wollen, können wir einen Intent implementieren, der vom Intent-Filter dieser Komponente durchgelassen wird.

Aber was macht die Komponente mit dem empfangenen Intent? Das werden wir uns im nächsten Abschnitt anschauen.

7.5 Empfang eines Intent

Anweisung für den Empfänger

Eine Komponente wird durch einen Intent gestartet. Aber *was* sie tun soll, weiß sie zunächst nicht. Diese Information steckt im Intent, und man kann ihn dazu nutzen, zusätzliche Informationen an die Komponente zu übermitteln. Wir haben oben gesehen, dass ein impliziter Intent meist aus einer Zeichenkette besteht, die die Aktion bezeichnet, und einer URI, die die Datenquelle spezifiziert. Nun kann diese URI aber eine Datenbanktabelle oder ein Dateiverzeichnis sein. Wenn dann die Komponente eine Activity ist, die eine einzelne Datei oder einen einzelnen Datensatz aus der Tabelle darstellt, müssen wir im Intent noch zusätzliche Informationen wie z.B. den Dateinamen oder den Schlüssel des Datensatzes (Primary Key) mitgeben.

```
Uri uri = Uri.parse("content://de.androidbuch");
Intent intent = new Intent(Intent.ACTION_PICK, uri);
intent.putExtra("dateiname", "meineDatei.txt");
startActivity(intent);
```

Zusätzliche Daten

Das Beispiel zeigt, wie wir mit Hilfe der Methode `putExtra` dem Intent einen Dateinamen hinzufügen. Der Intent besitzt eine Art Container für Schlüssel-Wert-Paare. Die Methode `putExtra` steht für fast jeden Datentyp zur Verfügung.

Wenn unsere Anwendung auf den Intent reagieren soll und eine Activity namens `ZeigeDateiinhaltActivity` zur Anzeige der Datei `meineDatei.txt` nutzen soll, dann sieht der Intent-Filter wie folgt aus.

```
<activity android:name=".ZeigeDateiinhaltActivity">
  <intent-filter>
    <action android:name="android.intent.action.PICK" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:path="de/androidbuch"
          android:scheme="content" />
  </intent-filter>
</activity>
```

Innerhalb der `onCreate`-Methode der Activity mit Namen `ZeigeDateiinhaltActivity` werten wir nun den Intent aus.

Listing 7.3
Empfang eines Intent

```
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  Intent intent = getIntent();
  String dateiname = intent.getStringExtra("dateiname");
  String aktion = intent.getAction();
  Uri uri = Uri.parse(intent.getDataString());
  ...
}
```

Wir haben die `onCreate`-Methode in Listing 7.3 nicht ausprogrammiert. Wir werden uns in Kapitel 10 mit dem Laden von Dateien aus dem Dateisystem befassen. Wir wollen hier nur zeigen, wie wir die Metadaten aus dem Intent auslesen.

7.6 Intent-Resolution

Empfänger ermitteln

Als *Intent-Resolution* wird der Prozess bezeichnet, den Android durchführt, um implizite Intents aufzulösen. Ziel dabei ist es, die Komponente zu finden, die am besten auf den Intent passt. Diese wird dann gestartet. In der Praxis wird man sich häufiger mal wundern, warum die gewünschte Komponente nicht startet. Dann hilft es, die Regeln der *Intent-Resolution* zu kennen, um den Fehler zu finden.

Regeln der Intent-Resolution

- **Regel 1**: Der Intent passt nicht zur Komponente, wenn keiner der `action`-Tags des Intent-Filters zum Intent passt.
- **Regel 2**: Der Intent passt nicht zur Komponente, wenn er Kategorien enthält, für die es keinen `category`-Tag im Intent-Filter gibt. Für alle Kategorien des Intent muss es Entsprechungen im Filter geben.
- **Regel 3**: Die URI des Intent muss mit mindestens einem der `data`-Tags übereinstimmen. Wurde im Intent keine URI als Verweis auf die Datenquelle angegeben, passiert der Intent den Intent-Filter und die Komponente wird aufgerufen.
- **Regel 4**: Werden zu einem Intent mehrere Komponenten gefunden, waren deren Intent-Filter vielleicht zu allgemein gefasst. Android präsentiert dann einen Bildschirm, der eine Auswahl der gewünschten Komponente zulässt (siehe Abb. 7-5).

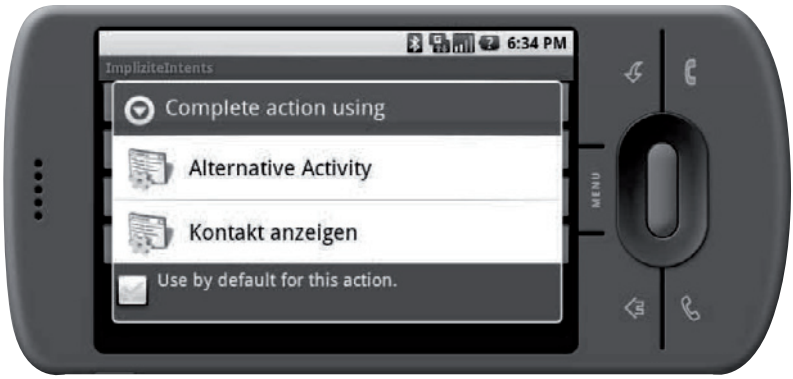


Abb. 7-5
Android-Dialog zur Auswahl der Komponente

7.7 Sub-Activities

Eine Sub-Activity ist zunächst mal eine ganz normale Activity. Sie wird im Android-Manifest wie jede andere Activity auch deklariert. Sub-Activities werden aber aus anderen Activities heraus gestartet und bleiben während ihrer Lebensdauer mit der Activity verbunden, die sie gestartet hat. Über einen Callback-Mechanismus kann die Sub-Activity Daten an die aufrufende Activity zurückgeben, wenn sie geschlossen wird.

Details anzeigen

Der häufigste Einsatzzweck für Sub-Activities sind Master-Detail-Szenarien. Beispielsweise stellt die Activity alle Kontakteinträge als Auswahlliste dar. Wählt der Anwender einen bestimmten Kontakt aus, startet die Anwendung eine Sub-Activity, die den Kontakt editierbar macht. Speichert man die Änderungen an dem Kontakt, schließt sich die Sub-Activity, und die Activity mit der Kontaktliste muss aktualisiert werden.

Eine Sub-Activity startet man mittels der Methode `startActivityForResult`, die einen Intent absetzt.

```
Uri uri = Uri.parse("content://contacts/people");
Intent intent = new Intent(Intent.ACTION_EDIT, uri);
startActivityForResult(intent, 101);
```

Im obigen Fall wird eine Activity aus der vorinstallierten Android-Anwendung *Contacts* zum Bearbeiten eines Kontakts gestartet. In der Methode `startActivityForResult` übergibt man als zweiten Parameter einen Request-Code. Dies kann z.B. die Id des Datensatzes sein, den man bearbeiten will, da dieser in der Sub-Activity aus der Datenbank geladen werden muss.

Drückt man in der Sub-Activity den Menüeintrag »*Speichern*«, wird in der aufrufenden Activity die Methode `onActivityResult` aufgerufen, die man überschreiben sollte. Ein Beispiel:

Listing 7.4 Ergebnis der Sub-Activity auswerten

```
public void onActivityResult(int requestCode,
    int resultCode, Intent data) {
  super.onActivityResult(requestCode, resultCode, data);

  switch (resultCode) {
    case Activity.RESULT_OK:
      Uri uri = data.getData();
      // lade Kontakt neu und aktualisiere Anzeige der
      // Kontaktliste
      break;
    case Activity.RESULT_CANCELED:
      // Abbruchmeldung anzeigen...
      break;
    ...
  }
}
```

Führen wir uns zunächst den kompletten Prozess vor Augen. Durch den Aufruf von `startActivityForResult` wird die Sub-Activity erzeugt und deren `onCreate` aufgerufen. Dort werden die im Intent übergebenen Parameter ausgelesen und interpretiert. Nun findet im Allgemeinen ein Dialog zwischen Anwender und Oberfläche statt, der irgendwann per Tasten- oder Schaltflächendruck beendet wird.

Sub-Activities beenden

Wir haben hier eine bestehende Activity aus der Anwendung *Contacts* verwendet. Implementiert man selbst eine Sub-Activity, beendet man diese durch Aufruf der Methode `finish`.

Vor dem `finish`-Aufruf teilt man der aufrufenden Activity mit Hilfe eines Intent das Resultat der Sub-Activity mit. Ein explizites Ziel braucht im Intent in diesem Fall nicht angegeben zu werden, da wir zur aufrufenden Komponente zurückkehren wollen. Dies wird durch die Methode `setResult(int resultCode, Intent intent)` ausgelöst. Als Werte für `resultCode` können die in der Klasse `Activity` bereits vordefinierten Werte `Activity.RESULT_OK` oder `Activity.RESULT_CANCELED` genutzt werden.

Der Activity Manager übergibt nach Beendigung der Sub-Activity die Kontrolle (und die Werte im Rückgabe-Bundle) an die aufrufende Activity. Dort wird die oben beschriebene `onActivityResult`-Methode aktiv, und der Subroutinen-Aufruf ist abgeschlossen. Listing 7.5 zeigt den Quelltext eines Beispiels dazu.

Listing 7.5 *Auslösen des Rückgabe-Intents*

```
speicherKontakt.setOnClickListener(
    new View.OnClickListener() {
  public void onClick(View view) {
    Bundle antwortBundle = new Bundle();
    antwortBundle.putString(
        "kontaktId", String.valueOf(mKontaktId));
    Intent antwortIntent = new Intent();
    antwortIntent.putExtras(antwortBundle);
    setResult(RESULT_OK, antwortIntent);
    finish();
  }
});
```

Zurück zur Praxis

Das Thema Intents ist damit bei weitem noch nicht abgeschlossen. Wir werden in den folgenden Iterationen noch häufiger in Theorie und Praxis damit konfrontiert werden. An dieser Stelle belassen wir es dabei, da wir möglichst schnell wieder in die Entwicklungspraxis unseres Staumelderprojekts einsteigen wollen.

7.8 Fazit

Wir haben in diesem Exkurs den »Leim« kennengelernt, der die einzelnen Komponenten einer Android-Anwendung zu einem Ganzen verbindet: den Intent. Mit Hilfe dieses Konstrukts können die Komponenten sich gegenseitig starten und Daten austauschen. Dabei können durchaus auch Anwendungsgrenzen überschritten werden, und wir können Komponenten anderer Anwendungen in unserer Anwendung verwenden.

8 Iteration 3 – Hintergrundprozesse

Hinweis: Die 2. Auflage dieses Buchs erscheint unter dem Titel
»Android 2. Grundlagen und Programmierung«
Ende Mai 2010 im dpunkt.verlag, ISBN 978-3-89864-677-2

Hinter der Oberfläche einer Anwendung tut sich manchmal so einiges. Wenn sich allerdings zu viel tut, kann es für den Anwender unangenehm werden. Er drückt eine Taste und nichts passiert, zumindest für eine kleine Ewigkeit. Daher sollten Anwendungen »responsive« sein, also ansprechbar bleiben. Dazu müssen wir die Oberfläche manchmal von den tieferliegenden Programmschichten trennen, indem wir sehr rechenintensive oder zeitraubende Prozesse auslagern und parallel zum restlichen Programm ausführen lassen. Dazu benötigen wir sogenannte Hintergrundprozesse.

8.1 Iterationsziel

Früher oder später wird man dazu kommen, Programmteile, die selbst keine Schnittstelle zur Oberfläche haben, als Hintergrundprozess laufen zu lassen. Ein gerne verwendetes Beispiel ist der »Music-Player«, der eine Bedienoberfläche hat, mit der er sich unter anderem starten und stoppen lässt, ansonsten aber im Hintergrund die Musik abspielt, auch wenn die Oberfläche nicht mehr im Vordergrund ist.

Aber auch langlaufende Aktionen, wie z.B. die Übertragung von Daten über das Internet, sollten im Hintergrund laufen, wie wir gleich sehen werden.

Lernziele

In dieser Iteration werden wir lernen, wie wir

- Services in unserem eigenen Prozess starten,
- Services in einem anderen Prozess starten,
- Threads oder Prozesse einsetzen,
- Inter Process Communication (IPC) einsetzen,
- Threads und Prozesse miteinander kommunizieren lassen.

8.2 Theoretische Grundlagen

Android bietet verschiedene Wege an, Hintergrundprozesse zu starten. Die Auswahl des richtigen Verfahrens hilft, eine stabile und performante Anwendung zu programmieren.

8.2.1 Prozesse und Threads

PID und UI-Thread

Für Android ist zunächst jede gestartete Anwendung ein eigener Betriebssystem-Prozess, erkennbar an der PID (Process ID). Dieser Prozess startet immer einen Thread, den »UI-Thread« oder »user interface thread«. Dieser Thread wird so bezeichnet, weil er für die Anzeige der sichtbaren Bestandteile der Anwendung zuständig ist. Im Android-Manifest müssen wir immer genau eine Activity als unsere Start-Activity kennzeichnen. Sie läuft automatisch im UI-Thread. Mit seinen Ressourcen sollte man sparsam umgehen, da er nicht nur für die Darstellung, sondern auch für die Anwendereingaben über z.B. Tastatur oder Touchscreen zuständig ist. Wenn der UI-Thread blockiert, reagiert weder Anzeige noch Eingabe.

Threads sind an Prozesse gekoppelt.

Threads und Prozesse sind nicht zu verwechseln. Man kann in einem Prozess mehrere Threads starten. Diese werden aber vom System automatisch beendet, sobald der Prozess stirbt. Das heißt, ein Thread ist immer an einen Prozess gekoppelt. Threads laufen parallel zur Hauptanwendung (dem UI-Thread) und verrichten im Hintergrund ihre Arbeit.

Durch die Kopplung der Threads an den Prozess werden alle gestarteten Threads beendet, sobald man die Anwendung verlässt. Als Programmierer muss man sich die Frage stellen, ob dies gewollt ist. Bei dem oben erwähnten Music-Player möchte man zum Beispiel die Oberfläche beenden können, aber die Musik soll weiterspielen.

Alternative: Service

In Abschnitt 2.5 auf Seite 20 haben wir Services als Komponente für Hintergrundprozesse vorgestellt. Ein Service läuft unsichtbar im Hintergrund. Er kann dies im Prozess der aufrufenden Komponente (oft einer Activity) tun, oder man kann ihn in einem eigenen Prozess starten. Mit Hilfe dieser zweiten Startvariante lassen sich Programmteile in einen Prozess auslagern, der den Lebenszyklus der Anwendung überdauert.

Bei der Implementierung einer Anwendung hat man die Wahl, ob man Prozesse durch den Application Manager verwalten lassen möchte. Sie werden dann automatisch beendet, sobald die Anwendung geschlossen wird. Oder man macht die Prozesse eigenständig (mit eigener PID), damit sie weiterlaufen, wenn die Anwendung beendet wird.

8.2.2 Langlaufende Prozesse

Langlaufende Programmteile sollten in eigene Threads oder Prozesse ausgelagert werden, damit die Anwendung möglichst schnell wieder auf Anwendereingaben reagiert und nicht »hängt«. Würde man z.B. das Herunterladen eines Videos in der `onCreate`-Methode einer Activity implementieren, die gerade ein Layout auf dem Bildschirm anzeigt, so würden alle Anwendereingaben abgeblockt, bis das Herunterladen beendet ist.

Reaktion auf Anwendereingaben sicherstellen

Die Lösung ist ein eigener Thread, der in der `onCreate`-Methode gestartet wird, den Download übernimmt und sich meldet, wenn das Video vollständig geladen und gespeichert ist. Andernfalls würde das System mit einem »ANR«, einem »Application Not Responding«-Ereignis, reagieren. Ein ANR wird ausgelöst, wenn

- eine Anwendereingabe nicht innerhalb von ca. 5 Sekunden,
- ein BroadcastReceiver nicht innerhalb von ca. 10 Sekunden

abgehandelt wird. Das Ergebnis ist, dass Android eine ANR-Meldung auf der aktuellen Bildschirmanzeige präsentiert (siehe Abb. 8-1). Der Thread sorgt dafür, dass die `onCreate`-Methode ohne Verzögerung durchläuft. Er fängt an, seine Arbeit parallel zur `onCreate`-Methode im Hintergrund zu erledigen.

Abb. 8-1
ANR-Dialog

8.2.3 Prozesse vs. Threads

Thread oder Prozess?

Wann verwendet man nun einen Thread und wann einen Prozess? Ein Thread ist fest an eine Anwendung gekoppelt, ein neuer Prozess dagegen nicht. Wenn man einen Dienst starten möchte, der eigenständig im Hintergrund läuft und beliebigen Anwendungen zur Verfügung steht, dann verwendet man einen Prozess. In Android realisiert man einen solchen Prozess durch eine spezielle Art von Service. Zur Unterscheidung verwenden wir in Zukunft die beiden Begriffe »*Local Service*« und »*Remote Service*«.

Local Service Service, der im gleichen Prozess wie die Anwendung läuft, die den Service startet. Im Android-Manifest deklariert man den Service, indem man

```
<service android:name=".LocalService" />
```

auf der gleichen XML-Ebene einfügt wie die Deklarationen für die Activities. `LocalService` sei dabei die Service-Klasse, die wir selbst implementiert haben. Wir werden in Zukunft von Service sprechen, wenn wir Local Service meinen.

Remote Service Ein Android-Service, der in einem eigenen Prozess läuft. Im Android-Manifest deklariert man den Service, indem man

```
<service android:name=".RemoteService"
    android:process=":remote" />
```

auf der gleichen XML-Ebene einfügt wie die Deklarationen für die Activities. `RemoteService` ist unsere Implementierung des Service.

Low-Level- vs. High-Level-Komponente

Ein Thread dagegen kann überall implementiert und gestartet werden, also auch in einem beliebigen Service. Threads sorgen dafür, dass eine Komponente »ansprechbar« (engl. *responsive*) bleibt und kein ANR auftritt. Mittels Threads kann dafür gesorgt werden, dass die Methoden des Lebenszyklus einer Komponente eine kurze Laufzeit haben und langwierige Aufgaben im Hintergrund erledigt werden.

Der Thread ist quasi ein Low-Level-Dienst, der eine langwierige technische Aufgabe ohne fachliche Logik erledigt, z.B. das Video aus dem Internet herunterladen.

Für Anwendungsfälle, die komplex sind, Geschäftslogik enthalten oder wiederverwendbar sein sollen, nimmt man besser einen Service. Services sind High-Level-Komponenten, die andere Komponenten verwenden und mit ihnen kommunizieren können. Services können Intents verschicken oder über Content Provider (siehe Kapitel 12) auf persistente Daten zugreifen.

Keine Angst vor Services

Auch wenn wir mit beiden Möglichkeiten der Implementierung von Hintergrundprozessen, Service und Thread, prinzipiell dasselbe erreichen können, ist es doch mehr als eine Frage des guten Geschmacks, welche der beiden Klassen man wann einsetzt. Wer aus der J2ME-Welt zu Android wechselt, wird vielleicht den Umgang mit Threads so verinnerlicht haben, dass er sich an die Services erst gewöhnen muss.

Das folgende Diagramm stellt beispielhaft die drei Arten von Hintergrundprozessen dar: Local Service, Thread und Remote Service. Es führt uns zum nächsten Schritt, nämlich der »Inter Process Communication« (IPC). IPC brauchen wir, wenn wir über Prozessgrenzen auf Betriebssystemebene mit einem Remote Service kommunizieren wollen.

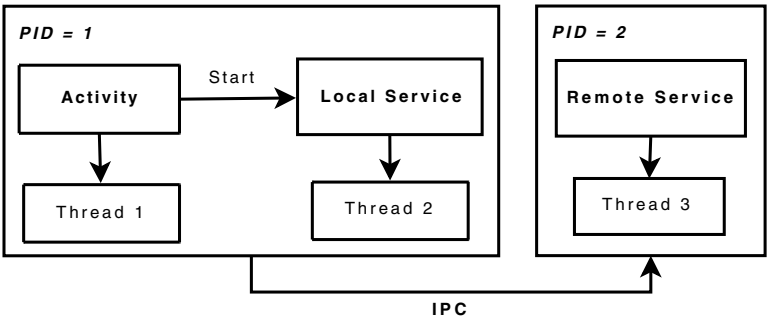


Abb. 8-2
Local und Remote Services

8.3 Implementierung

8.3.1 Services

Ein einfacher Android Service

Wir werden uns hier etwas genauer mit Android-Services beschäftigen. In unserer Staumelder-Anwendung werden wir später zum Beispiel einen Local Service verwenden, um GPS-Positionsdaten zu ermitteln und zu verarbeiten. Wir starten diesen Service mit der Staumelder-Anwendung und lassen ihn im Hintergrund laufen. Der Service hat eine klar umrissene Aufgabe und ist ein eigenständiger Programmteil, welcher für andere Anwendungen wiederverwendet werden kann.

Kommunikation mit Hilfe eines »Binders«

Wenn man einen Service im Hintergrund startet, muss man auch mit ihm kommunizieren können. Wir stellen nun zunächst das Konzept des *Binders* vor. Mit Hilfe von Bindern kann man Programmcode innerhalb des Service ausführen lassen und so Methoden oder Attribute des Service nutzen. Gleichzeitig zeigen wir in unserem ersten praktischen Beispiel, wie man einen Service implementiert und aus einer Activity heraus aufruft. Als Ergebnis haben wir einen Local Service, mit dem

wir über ein Binder-Objekt aus einer Activity heraus kommunizieren können. Später werden wir Remote Services kennenlernen und auch dort Binder verwenden.

Wir starten unser Beispiel mit der Deklaration des Service im Android-Manifest:

```
<service android:name=".location.
    GpsPositionsServiceLocal" />
```

Der Service selbst ist zunächst relativ einfach zu programmieren. Wir überschreiben lediglich drei Methoden der Oberklasse `Service`:

- `onCreate()`
- `onDestroy()`
- `onBind(Intent intent)`

Schauen wir uns den Quellcode des Service an:

Listing 8.1 *Lokaler Service zum Abrufen von GPS-Positionsdaten*

```
public class GpsPositionsServiceLocal
    extends Service { // (1)

  GpsData gpsData = null;
  private final IBinder gpsBinder =
      new GpsLocalBinder(); // (4)

  public class GpsLocalBinder extends Binder { // (3)
    GpsPositionsServiceLocal getService() {
        return GpsPositionsServiceLocal.this;
    }
    GpsData getGpsData() { // (6a)
      return gpsData;
    }
  }

  @Override
  public void onCreate() { // (2)
    gpsData =
        new GpsData(System.currentTimeMillis(), 50.7066272f,
            7.1152637f, 69.746456f);
  }

  @Override
  public void onDestroy() { }

  @Override
  public IBinder onBind(Intent intent) { // (5)
    return gpsBinder;
```

```
  }

  GpsData getGpsData() { // (6b)
      return gpsData;
  }
}
```

Implementiert man einen eigenen Service, so wird dieser von `Service` abgeleitet (1). Genau wie Activities besitzen auch Services eine `onCreate`-Methode. Wir haben dort ein `GpsData`-Objekt mit festen Werten zum Testen als Attribut unserer Klasse erschaffen (2), da wir hier noch keinen Zugriff auf Positionsdaten einführen wollen.

Soweit handelt es sich um die Implementierung eines Service. Wenn wir nun das erzeugte `GpsData`-Objekt nach außen geben wollen, können wir dies mit Hilfe eines `Binder`-Objekts tun. Ein `Binder` ist eine Klasse, die entfernte Zugriffe auf Objekte realisiert, die im selben Prozess wie die Hauptanwendung laufen. Wir implementieren hier als innere Klasse einen Binder mit Namen `GpsLocalBinder` (3). Der `GpsLocalBinder` stellt zwei Methoden zur Verfügung. Die erste gibt uns Zugriff auf den Service selbst, so dass wir in der aufrufenden Komponente seine Methoden verwenden können. Die zweite Methode, `getGpsData`, liefert uns den eben definierten Ortspunkt.

Binder für entfernte Zugriffe

Wir machen den `GpsLocalBinder` zu einem Attribut der Service-Klasse (4), damit wir ihn in der Methode `onBind` an die Komponente zurückgeben können, die den Service gestartet hat. Es fällt auf, dass wir nun das Interface `IBinder` für unsere von `Binder` abgeleitete Klasse `GpsLocalBinder` verwenden. Das liegt daran, dass `Binder` das `IBinder`-Interface implementiert.

Wir weisen noch darauf hin, dass es die Methode `getGpsData` zweimal gibt (5a und 5b). Der `GpsLocalBinder` liefert uns mittels seiner Methode `getService` Zugriff auf das komplette Service-Objekt. Darüber können wir jede öffentliche Methode des Service direkt verwenden, ohne den Umweg über den `Binder` zu gehen. Muss man in der aufrufenden Komponente sehr viele Methoden des Service nutzen, ist dieser Weg sinnvoll. Andernfalls muss man, wie im Beispiel an der Methode `getGpsData` gezeigt, jede Methode des Service und jeden Zugriff auf Attribute im `Binder` nachprogrammieren.

Zwei Möglichkeiten der Implementierung

Die hier verwendete Klasse `GpsData` ist ein einfacher Datencontainer, in dem wir einen beliebigen Ortspunkt samt Höhenabgabe und Zeitstempel abspeichern können. Der Zeitstempel dient dazu, festzuhalten, wann wir wo waren, wird also z.B. bei einer Staumeldung gebraucht, um zu dokumentieren, wann die Meldung gemacht wurde.

Listing 8.2 *Die Klasse GpsData*

```
public class GpsData {
  public long zeitstempel;
  public float geoLaenge;
  public float geoBreite;
  public float hoehe;

  public GpsData(long zeitstempel, float geoLaenge,
      float geoBreite, float hoehe) {
    this.zeitstempel = zeitstempel;
    this.geoLaenge = geoLaenge;
    this.geoBreite = geoBreite;
    this.hoehe = hoehe;
  }
}
```

Erst starten, dann verbinden...

Um den Service z.B. in einer `Activity` zu nutzen, müssen wir ihn starten und eine Verbindung zu ihm aufbauen. Um eine Verbindung aufzubauen, stellt Android die Klasse `ServiceConnection` zur Verfügung. Wir können es uns einfach machen und in unserer `Activity` eine Instanz der Klasse erstellen, indem wir die Methode `onServiceConnected(ComponentName className, IBinder binder)` überschreiben. Wie man an den Übergabeparametern sieht, bekommt die Methode einen `IBinder` übergeben. Dies ist in diesem Fall unser `GpsLocalBinder`, dessen Methode `getGpsData` wir nutzen können, um die GPS-Daten abzufragen.

Listing 8.3 *Activity zum Kommunizieren mit einem Local Service*

```
public class MainActivity extends Activity {

  private static final int VARIANTE_1 = 1;
  private static final int VARIANTE_2 = 2;

  private GpsPositionsServiceLocal localService;
  private GpsPositionsServiceLocal.GpsLocalBinder
      localServiceBinder;

  private ServiceConnection localServiceVerbindung =
      new ServiceConnection() {
    @Override
    public void onServiceConnected(ComponentName
        className, IBinder binder) { // (1)
      localServiceBinder =
          (GpsPositionsServiceLocal.GpsLocalBinder)binder;
      localService = binder.getService();
    }
  }
```

```
  @Override
  public void onResume() {
    ...
    verbindeMitService();
  }

  private void verbindeMitService() {
    Intent intent = new Intent(getApplicationContext(),
        GpsPositionsServiceLocal.class);

    bindService(intent, localServiceVerbindung,
        Context.BIND_AUTO_CREATE); // (2)
  }

  public GpsData ermittlePosition(int variante) { // (3)
    if (variante == VARIANTE_1) {
       return localService.getGpsData();
    }
    else if (variante == VARIANTE_2) {
       return localServiceBinder.getGpsData();
    }
  }

  @Override
  public void onPause() { // (4)
    Intent intent = new Intent(getApplicationContext(),
        GpsPositionsServiceLocal.class);
    stopService(intent);
  }
}
```

Die Activity initialisiert das Attribut `localServiceVerbindung` vom Typ `ServiceConnection`. Es wird dem Service übergeben, wenn sich die Activity mit ihm verbindet. In der `verbindeMitService`-Methode wird ein Intent zum Starten des Service erzeugt. Die Methode `bindService` (2) erhält als ersten Parameter den Intent, damit sie weiß, welchen Service sie starten soll. Als zweiten Parameter erhält sie `localServiceVerbindung`, dessen `onServiceConnected`-Methode (1) aufgerufen wird, sobald die Verbindung zum Service steht. Der dritte Parameter `Context.BIND_AUTO_CREATE` sorgt dafür, dass der Service automatisch gestartet wird, falls er noch nicht läuft. Dadurch spart man sich einen Aufruf der Methode `startService`.

Verbindung herstellen

Wenn die Verbindung zum Service steht, haben wir über die Attribute `localServiceBinder` und `localService` zwei verschiedene Zugriffsmöglichkeiten auf den Service: zum einen über den Binder und zum anderen direkt auf den Service selbst. Beide Möglichkeiten probieren

Zwei Zugriffsmöglichkeiten

wir in der Methode `ermittlePosition` (3) aus. Wir rufen einmal die Methode `getGpsData` im Service auf und einmal `getGpsData` im Binder des Service (siehe Listing 8.1).

Nun noch eine Bemerkung zum Service. Ein Local Service läuft immer im gleichen Prozess wie die Activity. Man muss einen solchen Service nicht stoppen oder die Verbindung beenden. Er wird automatisch beendet, wenn man die Activity beendet. Würde man dennoch die Verbindung zum Service unterbrechen wollen, so könnte man dies auf die folgende Art tun:

```
unbindService(localServiceVerbindung)
```

Wie man den Service ganz beendet, lässt sich in Methode `onPause` sehen (4).

IPC – Inter Process Communication

IPC ist nichts Android-spezifisches.

IPC steht für *Inter Process Communication* und ist in Android der bevorzugte Weg, wenn man mit einem Remote Service kommunizieren möchte. Ein Remote Service ist ein Service, der in einem eigenen Prozess läuft und die Lebensdauer der Komponente, die ihn gestartet hat, überdauern kann. Es erfordert etwas mehr Vorarbeit, mit einem Remote Service zu kommunizieren. Der Grund ist, dass Google hier einen bekannten Standard, (die *Interface Definition Language (IDL)*, siehe unten), verwendet hat. Um über Prozessgrenzen hinweg mit anderen Komponenten kommunizieren zu können, muss die Kommunikation auf Betriebssystemebene erfolgen. Sämtliche Parameter der Methodenaufrufe werden in Datentypen konvertiert, die das Betriebssystem versteht.

IDL: Interface Definition Language

IPC erfolgt mit Hilfe spezieller Interfaces, die in IDL angegeben werden. IDL ist eine allgemeine Spezifikationssprache, um den Datenaustausch zwischen Prozessen unabhängig von Betriebssystem und Programmiersprache zu ermöglichen. Zwar haben wir bei Android nicht die Möglichkeit, Programme in einer anderen Programmiersprache als Java zu installieren, jedoch sind viele Kernbibliotheken von Android in C/C++ geschrieben, und Android verwendet für viele seiner Standardkomponenten IDL für eigene interne Kommunikationszwecke.

Da sich die Datentypen von Java und C/C++ teilweise unterscheiden, wird dafür gesorgt, dass die Parameter und Rückgabewerte sämtlicher Methodenaufrufe, die man per IDL definiert hat, beim Serialisieren und Deserialisieren so in primitive Datentypen zerlegt werden, dass die Zielanwendung diese versteht, egal in welcher Programmiersprache sie geschrieben wurden.

Googles IDL: AIDL

Google hat für Android eine eigene Variante der IDL geschaffen: die *AIDL*, die Android-IDL. Zu ihr gehört das Programm (`/tools/aidl.exe` auf Windows-Systemen), welches den komplizierten Prozess des (De-)Serialisierens übernimmt, indem es automatisch Klassen generiert, die IPC-Methodenaufrufe über die vom Anwendungsentwickler zu definierende AIDL-Schnittstelle ermöglichen. Das Programm `aidl.exe` wird im Build-Prozess der Android-Anwendungserstellung benutzt und ist so in das Eclipse-Plug-in integriert, dass es automatisch ausgeführt wird.

Schauen wir uns das in der Praxis an. Wir bleiben bei unserem Staumelder-Beispiel und definieren den aus Listing 8.1 auf Seite 110 bekannten Service nun als Remote Service (`GpsLocationServiceRemote`). Für ihn legen wir eine Textdatei mit Namen `IGpsRemoteService.aidl` an. Diese Datei sollte man in dem Paket anlegen, in dem später auch die Klassendatei des Remote Service liegen wird.

Wir definieren zunächst für unseren Remote Service eine Methode, die uns die GPS-Daten in Form eines Strings zurückgibt. Dazu fügen wir den Quelltext aus Listing 8.4 in die Datei `IGpsRemoteService.aidl` ein. Die Syntax von AIDL werden wir am Ende des Abschnitts noch im Detail erklären.

Listing 8.4
AIDL-Interface für IPC zum Remote Service

```
package de.androidbuch.staumelder.location;

interface IGpsRemoteService {
  String getGpsDataAlsString();
}
```

Sobald wir die Datei speichern, erzeugt das Eclipse-Plug-in aus der `.aidl`-Datei automatisch ein Interface namens `IGpsRemoteService.java`. Darüber hinaus finden wir im `bin`-Verzeichnis folgende generierte Klassen:

- `IGpsRemoteService$Stub$Proxy.class`
- `IGpsRemoteService$Stub.class`
- `IGpsRemoteService.class`

Implementierung des AIDL-Interface

Von diesen automatisch generierten Klassen ist besonders die innere Klasse `IGpsRemoteService.Stub` interessant. Unsere Aufgabe ist es gleich, diese Klasse mit der in der Datei `IGpsRemoteService.aidl` definierten Methode zu erweitern. Die Klasse `Stub` hat wiederum eine innere Klasse mit Namen `Proxy`. Sie ist für die Konvertierung der Variablen zwischen der Android-Anwendung und dem Remote Service zuständig.

Listing 8.5 *Remote Service mit IBinder*

```
public class GpsLocationServiceRemote extends Service {

  @Override
  public void onCreate() { }

  @Override
  public void onDestroy() { }

  @Override
  public IBinder onBind(Intent intent) { // (1)
    return gpsBinder;
  }

  private final IGpsRemoteService.Stub gpsBinder = new
      IGpsRemoteService.Stub() { // (2)
    public String getGpsDataAlsString() throws RemoteException {
      String gpsData = "50.7066272, 7.1152637, 69.746456";
      return gpsData;
    }
  };
}
```

Die Stub-Klasse ist ein IBinder.

Listing 8.5 zeigt die Implementierung des Remote Service. Wenn wir den Quelltext mit Listing 8.1 vergleichen, fallen schnell einige Parallelen auf. Bei der Klasse `IGpsRemoteService.Stub` handelt es sich um ein Objekt vom Typ `IBinder` (1). Der wesentliche Unterschied zu einem Local Service ist, dass hier unser automatisch durch das Programm `aidl.exe` generierter `IBinder` namens `IGpsRemoteService.Stub` zum Einsatz kommt. Wir erzeugen ein Exemplar der Klasse, welches die Methoden beinhaltet, die für IPC notwendig sind. Alles, was wir tun müssen ist, die im `.aidl`-Interface deklarierten Methoden zu implementieren. Wenn wir dies getan haben, kann die Methode `onBind` aufgerufen werden und gibt unseren `IBinder` zurück.

Nun muss der Service noch als Remote Service im Android-Manifest bekannt gemacht werden. Dazu fügen wir einen neuen Serviceeintrag hinzu:

Listing 8.6 *Remote Service im Android-Manifest bekannt machen*

```
<service android:name=".location.GpsLocationServiceRemote"
    android:process=":remote">
  <intent-filter>
    <action android:name="de.androidbuch.staumelder.
        location.IGpsRemoteService" />
  </intent-filter>
</service>
```

Im Service-Tag sehen wir, dass ein Remote Service explizit als solcher deklariert werden muss:

```
android:process=":remote"
```

Zusätzlich muss ein Intent-Filter deklariert werden, damit der Service bereit ist, Intents von anderen Komponenten entgegenzunehmen. Nun wollen wir unseren Service nutzen und die GPS-Daten vom Service abrufen. Listing 8.7 zeigt die Implementierung der Activity `GpsDatenAnzeigen`, die den Remote Service verwendet.

Listing 8.7 *Activity mit Verbindung zum Remote Service*

```
public class GpsDatenAnzeigen extends Activity {

  private IGpsRemoteService mGpsRemoteService;

  private OnClickListener mButtonGetGpsDataListener =
      new OnClickListener() {
    public void onClick(View v) {
      final TextView fldOrtsposition =
          (TextView) findViewById(R.id.gpsDatenAnzeigen);
      fldOrtsposition.setText(String.valueOf(
          getOrtsposition())); // (1)
    }
  };

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.gps_daten_anzeigen);
    setTitle(R.string.title_gps_gpsActivityTitle);

    Intent intent =
      new Intent(IGpsRemoteService.class.getName());
    bindService(intent, gpsRemoteServiceVerbindung,
      Context.BIND_AUTO_CREATE);

    Button buttonGetGpsData =
        (Button)findViewById(R.id.opt_gpsDatenAbrufen);
    buttonGetGpsData.
        setOnClickListener(mButtonGetGpsDataListener);
  }

  private ServiceConnection gpsRemoteServiceVerbindung =
      new ServiceConnection() { // (2)
```

```
    @Override
    public void onServiceConnected(ComponentName
        className, IBinder binder) {
      Log.d(TAG, "onServiceConnected(): entered...");
      mGpsRemoteService =                          // (3)
          IGpsRemoteService.Stub.asInterface(binder);
    }

    @Override
    public void onServiceDisconnected(ComponentName
        className) {
      // ...
    }
  };

  private String getOrtsposition() {
    if (mGpsRemoteService != null) {
      try {
        return mGpsRemoteService.
            getGpsDataAlsString(); // (4)
      }
      catch (RemoteException e) {}
    }
    return null;
  }
}
```

Die Activity in Listing 8.7 stellt die vom Remote Service angerufene Ortsposition im einer `TextView` dar (1).

IBinder vom Service holen

An (2) wird die `gpsRemoteServiceVerbindung` definiert. Hier überschreiben wir wieder, wie beim Local Service in Listing 8.1, die beiden Methoden `onServiceConnected` und `onServiceDisconnected`. Wenn wir uns mit dem Service verbinden, bekommen wir in der Methode `onServiceConnected` das Binder-Objekt zu unserem Service. Mittels

```
IGpsRemoteService.Stub.asInterface(binder);
```

Zugriff auf Methoden des Binders über das automatisch generierte Interface

können wir uns den Binder als Java-Interface holen (3), welches das AIDL-Programm `aidl.exe` für uns automatisch aus der `.aidl`-Datei erstellt hat. Über die im Interface deklarierten Methoden haben wir Zugriff auf den Remote Service. Der Zugriff auf den Remote Service erfolgt in der Methode `getOrtsposition` (4). Dort fangen wir eine `RemoteException` einer Oberklasse von `DeadObjectException`. `DeadObjectException` werden ausgelöst, wenn der Prozess nicht mehr existiert, in dem der Remote Service läuft. Mit der `RemoteException` sind wir auf der sicheren Seite, frei nach dem Motto: Wer weiß, welche Exceptions noch ausgelöst werden.

Die vier Listings dieses Abschnitts demonstrieren, wie man einen einfachen Remote Service samt IPC programmiert. Fassen wir die einzelnen Schritte als »Kochrezept« zusammen:

Die einzelnen Schritte

1. Erzeuge ein AIDL-Interface mit den gewünschten Methoden, um mit dem Service zu kommunizieren.
2. Erstelle eine Service-Klasse.
3. Sorge dafür, dass die Stub-Klasse (der IBinder) um die Methoden aus unserem `.aidl`-Interface erweitert wird.
4. Füge den Service als Remote Service samt Intent-Filter dem Android-Manifest hinzu.

Nun noch einige Anmerkungen zu IPC und dem `.aidl`-Interface. Per IPC können nur bestimmte Datentypen übertragen werden. Dies sind:

Zur Verfügung stehende Datentypen

- primitive Datentypen (boolean, int, float etc.)
- String
- CharSequence
- List
- Map
- AIDL-generierte Interfaces
- Klassen, die das Parcelable Interface implementieren

Wir werden im nächsten Abschnitt erklären, was das *Parcelable Interface* ist. `List`-Objekte werden als `ArrayList` übertragen und können als Generic deklariert werden (z.B. `List<CharSequence>`). Die Elemente in der `List` müssen von einem Typ der obigen Auflistung sein.

Verwendet man als Parameter einer Methode eine `Map`, darf diese ebenfalls nur Elemente enthalten, die einem der obigen Typen entsprechen. Maps dürfen jedoch nicht als Generic (z.B. `Map<String, List<String>`) deklariert werden. Tatsächlich verbirgt sich hinter dem `Map`-Interface als konkrete Klasse eine `HashMap`, die zum Empfänger übertragen wird und dort zur Verfügung steht.

Will man eigene Datentypen als Parameter einer Methode per IPC übertragen, so müssen diese ein AIDL-generiertes Interface implementieren.

Eigene Datentypen per IPC übertragen: Es ist weiterhin möglich, beliebige Datenobjekte zu erstellen, die per IPC übertragen werden können. Sie müssen das `Parcelable`-Interface implementieren, was im Grunde nur eine spezielle Android-Variante einer Serialisierung aller in der Klasse enthaltenen Attribute ist. J2ME-Programmierer werden diese Technik des »Serialisierens zu Fuß« sofort wiedererkennen, da in dieser abgespeckten Java-Variante das Interface `Serializable` nicht zur

Verfügung steht. Bei Android gibt es dieses Interface zwar, aber wir verwenden ja AIDL und wollen potenziell auch mit in C++ geschriebenen Prozessen kommunizieren können, die keine serialisierten Java-Objekte kennen und sie demnach wahrscheinlich nicht korrekt deserialisieren werden. Durch die Verwendung des `Parcelable`-Interface wird eine sprachneutrale Serialisierung zur Übertragung der Parameter auf Betriebssystemebene angestoßen.

Schauen wir uns nun unsere `GpsData`-Klasse aus Listing 8.2 auf Seite 112 an. Wir haben sie umgebaut, so dass sie nun als Parameter per IPC übertragen werden kann.

Listing 8.8 *GpsData-Klasse als Parcelable*

```
public class GpsData implements Parcelable {

  public long zeitstempel;
  public float geoLaenge;
  public float geoBreite;
  public float hoehe;

  public static final Parcelable.
      Creator<GpsData> CREATOR = // (1)
    new Parcelable.Creator<GpsData>() {
      public GpsData createFromParcel( // (2)
          Parcel in) {
        return new GpsData(in);
      }
  };

  private GpsData(Parcel in) {
    readFromParcel(in); // (3)
  }

  @Override
  public void writeToParcel(Parcel out, int flags) {
    out.writeLong(zeitstempel);
    out.writeFloat(geoLaenge);
    out.writeFloat(geoBreite);
    out.writeFloat(hoehe);
  }

  public void readFromParcel(Parcel in) { // (4)
    zeitstempel = in.readLong();
    geoLaenge = in.readFloat();
    geoBreite = in.readFloat();
    hoehe = in.readFloat();
  }
```

```
  @Override
  public int describeContents() {
    return 0;
  }
}
```

Als Erstes wenden wir uns der statischen Variable `CREATOR` zu, die das Interface `Parcelable.Creator` implementiert (1). `CREATOR` muss mindestens die Methode `createFromParcel` implementieren (2). Mit dieser Methode wird dafür gesorgt, dass unser Objekt korrekt deserialisiert wird. Dazu verwenden wir mit einem Umweg über den Konstruktor von `GpsData` (3) die Methode `readFromParcel(Parcel in)` (4). Hier werden die Werte in unsere Klasse `GpsData` geschrieben. Das Objekt `Parcel`, welches wir in die Methode übergeben bekommen, können wir uns als einen Wrapper um einen `DataOutputStream` vorstellen. Es speichert unsere Attribute in Form eines `byte`-Arrays und gibt uns wie bei einem `DataOutputStream` die Möglichkeit, jedes einzelne Attribut der Klasse `GpsData` zu deserialisieren bzw. zu serialisieren. Den Prozess des Serialisierens und Deserialisierens bezeichnet man auch als »Marshalling«.

Serialisieren und Deserialisieren von Parcelable-Objekten

`Parcel` und `Parcelable` liegt eine C-Implementierung zugrunde, die diesen Prozess besonders schnell abarbeitet. Die C-Bibliotheken, die diese Implementierung beinhalten, sind nicht Bestandteil von Linux. Google hat sie der Android-Plattform hinzugefügt, und nun sind sie Teil des Betriebssystems. Darüber hinaus hat Google zahlreiche C-Bibliotheken, die normaler Bestandteil des Linux-Betriebssystems sind, für die Android-Plattform geändert und für die geringen Ressourcen der mobilen Endgeräte optimiert. Folglich ist es nicht ohne Weiteres möglich, den Linux-Kernel zu aktualisieren.

Kernel-Update schwierig

Um die Klasse `GpsDataParcelable` zu serialisieren, muss die Methode `writeToParcel(Parcel out, int flags)` implementiert werden. Wir serialisieren alle Attribute in ein Objekt vom Typ `Parcel`. Der Übergabeparameter `flags` gibt an, ob wir unser `Parcel`-Objekt nur an einen anderen Prozess übergeben und nicht erwarten, dass die aufgerufene Methode es verändert, oder ob wir das Objekt verändert zurückbekommen. Somit kann `flags` zwei Werte annehmen:

Attribute werden einzeln (de-)serialisiert.

- 0: das `Parcel`-Objekt wird im aufgerufenen Prozess nicht verändert
- `PARCELABLE_WRITE_RETURN_VALUE`: das `Parcel`-Objekt wird verändert und muss zurückübertragen werden.

Das führt uns zu einem weiteren wichtigen Thema in AIDL. Wir wissen nun, wie wir eigene Objekte für IPC-Aufrufe erstellen können, die das Interface `Parcelable` implementieren müssen. Mit Hilfe aller genannten Datentypen können wir komplexe Datenstrukturen an einen anderen

Prozess übertragen. Dabei macht es einen Unterschied, ob wir nur mit dem Methodenaufruf Parameter an den Prozess übergeben oder ob wir auf einen Rückgabewert warten müssen. Im ersten Fall können wir den Methodenaufruf absetzen und sofort mit dem Programmablauf fortfahren. Im zweiten Fall müssen wir warten, bis der andere Prozess seine Methode abgearbeitet hat und den Rückgabewert an den Aufrufer zurückübermittelt hat.

IPC-Aufrufe sind teuer!

Dies ist ein Unterschied, den wir bei der Implementierung berücksichtigen sollten. IPC-Aufrufe sind sehr teure Operationen, da die übertragenen Parameter serialisiert und deserialisiert werden und über das Netzwerk übertragen werden müssen. Denn die Kommunikation zwischen Prozessen läuft außerhalb der DVM auf Betriebsystemebene über das Netzwerkprotokoll. AIDL bietet uns daher die Möglichkeit, für jeden Parameter eines Methodenaufrufs anzugeben, ob es sich um einen Hingabeparameter, einen Rückgabeparameter oder beides handelt.

Definieren wir nun ein AIDL-Interface, an dem wir die Möglichkeiten demonstrieren. Sagen wir, unsere GPS-Datenpunkte sollen beim Remote Service abgerufen werden können. Einmal übergeben wir dem Service ein `GpsData`-Objekt, welches aktualisiert werden soll. In einer zweiten Methode wollen wir einen neuen Ortspunkt vom Service erhalten. Die dritte Methode liefert einen Ortspunkt an den Server. Listing 8.9 zeigt die Definition des Interface.

Listing 8.9 *Erweitertes Interface für den GPS-Remote-Service*

```
package de.androidbuch.staumelder.location;

import de.androidbuch.staumelder.commons.GpsData;

interface IGpsRemoteService {
  void updateGpsData(inout GpsData gpsData);
  GpsData getGpsData();
  void setGpsData(in GpsData gpsData);
}
```

Performance-Steigerung mittels Schlüsselworten

Die folgenden AIDL-Schlüsselworte dienen der Performanzsteigerung und reduzieren den Aufwand bei (De-)Serialisieren bei IPC-Aufrufen. Die Schlüsselworte `in`, `out` und `inout` können den Übergabeparametern vorangestellt werden. Sie werden bei primitiven Datentypen (`int`, `boolean` etc.) weggelassen. Ihre Bedeutung ist folgende:

in: Der Parameter ist ein reiner Hingabeparameter, der an den Zielprozess übergeben wird. Änderungen an dem Parameter sollen nicht zurück in die aufrufende Methode übertragen werden.

out: Der Parameter wird im Zielprozess neu erzeugt, und es findet keine Übertragung des Parameters aus dem aufrufenden Prozess an den Zielprozess statt. Allerdings wird der Parameter zurück zum aufrufenden Prozess übertragen, wenn die Methode des Zielprozesses beendet ist. Es handelt sich also um einen reinen Rückgabeparameter.

inout: Kombination von beidem. Der Parameter wird an den Zielprozess übertragen, kann dort verwendet und verändert werden und die Änderungen werden zurück an den aufrufenden Prozess übertragen. Das Schlüsselwort `inout` sollte nur verwendet werden, wenn es nötig ist, da der Aufwand für das Serialisieren und Deserialisieren in beiden Richtungen anfällt.

Synchrone vs. asynchrone Methodenaufrufe: IPC-Aufrufe sind standardmäßig synchron, d.h., der IPC-Aufruf einer Remote-Service-Methode dauert so lange, bis die Methode abgearbeitet ist. Das hat einen gravierenden Nachteil. Braucht das Abarbeiten einer Methode in einem Remote Service zu lange, so riskiert man einen ANR. Wenn nämlich der Aufruf einer Methode im Remote Service aus einer Activity heraus stattfindet, reagiert die Activity während der Aufrufdauer nicht auf Eingaben. Glücklicherweise hat man jedoch in AIDL-Interfaces die Möglichkeit, Methoden als asynchron zu kennzeichnen.

ANR vermeiden

Synchron: Die Abarbeitung des Programms wird erst fortgesetzt, wenn die Methode der Zielkomponente vollständig abgearbeitet wurde und die Parameter und Rückgabeparameter an die aufrufende Komponente zurückübertragen wurden. ANR möglich!

Asynchron: Der Programmablauf wird durch den Methodenaufruf nicht unterbrochen. Die Zielkomponente arbeitet die Methode parallel zur aufrufenden Komponente ab. Kein ANR möglich.

Wenn man einen Remote Service aus dem UI-Thread mittels eines synchronen Methodenaufrufs durchführt, riskiert man einen ANR. Es wird empfohlen, die Laufzeit der Methode im Zielprozess bei wenigen hundert Millisekunden zu halten. Langwierige Aufgaben sollte ein Thread im Zielprozess übernehmen.

Verwenden wir nun statt eines synchronen Aufrufs einen asynchronen Aufruf, bleibt die Frage, wie wir an den Rückgabewert der aufgerufenen Methode kommen, falls sie einen hat.

Wie kommt man ans Ergebnis?

Asynchrone Methodenaufrufe mit Rückgabewert: Nehmen wir als Beispiel folgenden Anwendungsfall. Wir möchten Positionsdaten von unserem Remote Service (siehe Listing 8.5 auf Seite 116) mit Hilfe eines asynchronen Methodenaufrufs abrufen. Der Grund für einen asynchronen Aufruf ist, dass wir nur Positionsdaten erhalten möchten, wenn auch wirklich welche durch das GPS-Modul des Android-Geräts ermittelt werden können. Fahren wir durch einen Tunnel, sollen keine Daten durch den Remote Service geliefert werden. Der Service soll so lange warten, bis er wieder ein GPS-Signal hat, und die Position dann erst zurückgeben.

Warten auf Positionsdaten...

Wir werden in den folgenden Quelltexten auf die Ermittlung der Ortsposition verzichten, da wir für dieses Thema Kapitel 15 vorgesehen haben. Stattdessen werden feste Werte für eine Ortsposition verwendet.

Die Übermittlung von Rückgabewerten einer asynchron aufgerufenen Methode an den Aufrufer erfolgt mit Hilfe von Callbacks. Wir belassen es bei der gängigen englischen Bezeichnung. Wenn wir in diesem Abschnitt von Callback sprechen, ist die Rückmeldung des Remote Service an die aufrufende Komponente, die die asynchrone Methode im Service aufgerufen hat, gemeint. Der Callback wird *vor* dem Methodenaufruf an den Service übergeben. Nach Beendigung der asynchronen Methode durch den Service verwendet dieser den Callback, um der Komponente mitzuteilen, dass die Arbeit erledigt ist. Rückgabewerte können im Callback an die Komponente übermittelt werden. Unser Beispiel wird die folgenden Schritte implementieren:

1. Definition eines Callback-Objekts in AIDL
2. Erweitern des AIDL-Interfaces des Remote Service um Methoden zum Registrieren und Entfernen von Callback-Objekten
3. Erweitern des AIDL-Interfaces des Remote Service um die asynchrone Methode `getGpsDataAsynchron`
4. Erweitern des IBinder `IGpsRemoteService.Stub` im Remote Service um die oben genannten Methoden
5. Anpassen der Activity zur Verwendung der asynchronen Methode

In Listing 8.10 implementieren wir zunächst das AIDL-Interface des Callback-Objekts.

Listing 8.10 *AIDL-Interface des Callback-Objekts*

```
package de.androidbuch.staumelder.services;

import de.androidbuch.staumelder.commons.GpsData;

interface IServiceCallback {
  void aktuellePosition(in GpsData gpsData);
}
```

Da das Callback-Objekt an den Service übergeben wird und dafür verwendet wird, die im Remote Service ermittelte Ortsposition an die aufrufende Komponente zurückzugeben, müssen wir das Schlüsselwort `in` verwenden.

Nun erweitern wir das aus Listing 8.9 auf Seite 122 bekannte AIDL-Interface des Remote Service um zusätzliche Methoden:

Listing 8.11 *Erweitertes AIDL-Interface des Remote Service*

```
package de.androidbuch.staumelder.services;

import de.androidbuch.staumelder.commons.GpsData;
import de.androidbuch.staumelder.services.
    IServiceCallback;

interface IGpsRemoteService {
  void updateGpsData(out GpsData gpsData);
  GpsData getGpsData();
  void setGpsData(in GpsData gpsData);

  oneway void getGpsDataAsynchron();
  void registriereCallback(IServiceCallback callback);
  void entferneCallback(IServiceCallback callback);
}
```

Synchrone oder asynchrone Methoden: oneway

In Listing 8.11 haben wir die asynchrone Methode `getGpsDataAsynchron` zum Abrufen der aktuellen Ortsposition hinzugefügt. Stellt man einer Methode das Schlüsselwort `oneway` voran, so wird sie asynchron ausgeführt. Auch kann man das ganze Interface asynchron machen, wenn es nur asynchron auszuführende Methoden enthält. Dann stellt man einfach das Schlüsselwort `oneway` vor das Wort `interface`.

Bei asynchronen Methodenaufrufen müssen alle übergebenen Parameter (außer den primitiven Typen) das Schlüsselwort `in` vorangestellt bekommen, da nicht darauf gewartet wird, ob die Methode den Wert zurückgibt. `out` oder `inout` wird zwar akzeptiert, hat aber keinen Sinn, da beide Schlüsselwörter implizieren, dass es einen Rückgabewert gibt. Ein asynchroner Methodenaufruf wartet aber nicht, bis die aufgerufene Methode abgearbeitet ist und einen Wert zurückgibt.

Die beiden Methoden `registriereCallback` und `entferneCallback` dienen dazu, das Callback-Objekt beim Service bekannt zu machen bzw. wieder zu entfernen.

Kommen wir zum Remote Service. Hier ändert sich nur die Erweiterung des IBinders `IGpsRemoteService.Stub`. Dieses wird um die drei zusätzlichen Methoden aus dem AIDL-Interface 8.11 erweitert.

Listing 8.12 *Erweiterung des Remote Service*

```
...
import android.os.RemoteCallbackList;
...
public class GpsLocationServiceRemote extends Service {
  private final RemoteCallbackList<IServiceCallback>
      callbacks = // (1)
          new RemoteCallbackList<IServiceCallback>();

  private final IGpsRemoteService.Stub gpsBinder = new
      IGpsRemoteService.Stub() {
    @Override
    public void updateGpsData(GpsDataParcelable gpsData)
        throws RemoteException {
      // TODO: veraendere den Datensatz 'gpsData'
    }

    @Override
    public GpsDataParcelable getGpsData()
        throws RemoteException {
      // TODO: Hole Ortsposition vom Location Manager
      // und gebe sie als 'GpsDataParcelable' zurueck
    }

    @Override
    public void setGpsData(GpsDataParcelable gpsData)
        throws RemoteException {
      // TODO: Speichere 'gpsData'
    }

    @Override
    public void getGpsDataAsynchron()
        throws RemoteException {
      // TODO: aktuelle Ortsposition ueber den Location-
      // Manager ermitteln
      GpsDataParcelable gpsData = new
          GpsDataParcelable(System.currentTimeMillis(),
              7.1152637f, 50.7066272f, 69.746456f);

      int anzCallbacks = callbacks.beginBroadcast();
      for (int i = 0; i < anzCallbacks; i++) {
        try {
          callbacks.getBroadcastItem(i).
              aktuellePosition(gpsData);
        }
        catch (RemoteException e) { }
      }
```

```
      callbacks.finishBroadcast();
    }

    @Override
    public void registriereCallback(IServiceCallback
        callback) throws RemoteException {
      if (callback != null) {
        callbacks.register(callback);
      }
    }

    @Override
    public void entferneCallback(IServiceCallback
        callback) throws RemoteException {
      if (callback != null) {
        callbacks.unregister(callback);
      }
    }
  }
  ...
}
```

Am Anfang von Listing 8.12 haben wir eine `android.os.RemoteCallbackList` namens `callbacks` erschaffen. Diese Klasse des Android-SDK verwaltet die Callback-Objekte in einer Liste. Da sich mehrere andere Prozesse gleichzeitig mit diesem Remote Service verbinden können, berücksichtigt die `RemoteCallbackList` Aspekte der Nebenläufigkeit (die Zugriffe sind »thread-safe«). Denn jederzeit kann ein aufrufender Prozess über die Methode `entferneCallback` ein Callback-Objekt aus der Liste entfernen. Also auch, wenn gerade eine der Methoden des Callback ausgeführt wird.

thread-safe

Eine weitere Aufgabe der `RemoteCallbackList` ist das automatische Entfernen von Callbacks, wenn der aufrufende Prozess beendet wird. Dies ist möglich, da man zu einem Service eine dauerhafte Verbindung mittels der Methode `bindService` aufbaut (siehe Listing 8.7). Die `RemoteCallbackList` merkt, wenn die Verbindung zu einem verbundenen Prozess unterbrochen wird, und entfernt automatisch den Callback dieses Prozesses.

Asynchrone Methodenaufrufe verwenden: Nun wissen wir, wie wir einen Remote Service mit asynchronen Methoden implementieren. Um zu zeigen, wie man aus einer Activity heraus eine asynchrone Methode mit Rückgabewerten aufruft, ändern wir Listing 8.7. Dort hatten wir die Ortsposition per synchronem Methodenaufruf ermittelt. Nun verwenden wir die asynchrone Methode `getGpsDataAsynchron`.

Listing 8.13 *Asynchroner Methodenaufruf mittels Callback*

```
public class GpsDatenAnzeigen extends Activity {

  private IGpsRemoteService mGpsRemoteService;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.gps_daten_anzeigen);
    setTitle(R.string.title_gps_gpsActivityTitle);

    Intent intent =
      new Intent(IGpsRemoteService.class.getName());
    bindService(intent, gpsRemoteServiceVerbindung,
      Context.BIND_AUTO_CREATE);

    Button buttonGetGpsData =
        (Button)findViewById(R.id.opt_gpsDatenAbrufen);
    buttonGetGpsData.
        setOnClickListener(mButtonGetGpsDataListener);
  }

  private ServiceConnection gpsRemoteServiceVerbindung =
      new ServiceConnection() {
    @Override
    public void onServiceConnected(ComponentName
        className, IBinder binder) {
      mGpsRemoteService =
          IGpsRemoteService.Stub.asInterface(binder);

      try {
        mGpsRemoteService.registriereCallback( // (1)
            serviceCallback);
      }
      catch (RemoteException e) { }
    }

    @Override
    public void onServiceDisconnected(ComponentName
        className) {
      try { // (2)
        mGpsRemoteService.entferneCallback(serviceCallback);
      }
      catch (RemoteException e) { }
    }
  };
```

```
  private final IServiceCallback serviceCallback = // (3)
      new IServiceCallback.Stub() {
    public void aktuellePosition(GpsDataParcelable gpsData)
        throws RemoteException {
      // TODO: gpsData in TextView anzeigen
    }
  };

  private OnClickListener mButtonGetGpsDataListener =
      new OnClickListener() {
    public void onClick(View v) {
      try {
        mGpsRemoteService.getGpsDataAsynchron();  // (4)
      }
      catch (RemoteException e) { }
    }
  };
}
```

Wir haben die Stellen, in denen sich die Activity in Listing 8.13 durch die Verwendung des asynchronen Methodenaufrufs gegenüber Listing 8.7 auf Seite 117 verändert hat, mit Zahlen gekennzeichnet. In der `onServiceConnected`-Methode registrieren wir nun zusätzlich unseren Callback (1). In der `onServiceDisconnected`-Methode entfernen wir diesen wieder (2), wenn wir die Verbindung zum Service beenden. Das Erweitern des Callback um die asynchrone Methode `aktuellePosition` erfolgt an Stelle (3) im Quelltext. Diese Methode wird vom Remote Service aufgerufen, wenn eine Ortsposition vorliegt. Hier haben wir nur angedeutet, dass die Ortsposition in der `TextView` der Activity dargestellt werden soll. Der Grund ist, dass der Callback im Remote Service ausgeführt wird, also in einem anderen Prozess. Aus der Methode `aktuellePosition` heraus besteht zwar Zugriff auf die Methoden der Activity, jedoch würde folgender Quellcode innerhalb der Methode zu einer Exception (`android.view.ViewRoot$CalledFromWrongThreadException`) zur Laufzeit führen:

```
final TextView fldOrtsposition = (TextView)
    findViewById(R.id.gpsDatenAnzeigen);
fldOrtsposition.setText(String.valueOf(
   gpsData2String(gpsData));
```

Wir werden uns im folgenden Abschnitt über Threads näher mit dieser Problematik beschäftigen und eine Lösung zur Vermeidung der Exception kennenlernen.

Restarbeiten: Bevor wir den Remote Service im Emulator ausprobieren können, bleibt noch ein Schritt zu tun. Wenn wir in einem Android-Projekt `Parcelable`-Objekte per AIDL-Datei definiert haben, muss man die Objekte noch in einer weiteren `.aidl`-Datei bekannt machen. Eclipse stellt mit dem Android-Plug-in dazu einen einfachen Mechanismus zur Verfügung. Man geht im Projektbrowser einfach auf das Projekt, welches das neue `Parcelable`-Objekt enthält, drückt die rechte Maustaste und findet unter dem Menüpunkt *Android Tools* den Eintrag *Create Aidl preprocess file for Parcelable classes*. Beim Anklicken des Menüpunkts wird automatisch eine Datei namens `project.aidl` auf der obersten Projektebene erzeugt. Nach dem Erstellen eines neuen `Parcelable`-Objekts sollte diese Datei aktualisiert werden.

Wir schließen diesen Abschnitt mit einigen zusätzlichen Informationen zum Thema IPC und AIDL.

- Prozess und Zielprozess müssen die gleichen `Parcelable`-Interfaces implementieren, wenn man solche selbstdefinierten Datentypen als Übergabeparameter verwendet. Das hat oft zur Folge, dass Änderungen an den Übergabeparametern Änderungen in mehreren Anwendungen nach sich ziehen. Der Remote Service selbst läuft als Prozess in einer .apk-Datei, und jede Anwendung, die diesen Service verwendet, muss evtl. angepasst werden, wenn sich ein Übergabeparameter ändert. Dies sollte man bei der Implementierung berücksichtigen, indem man z.B. Parameterlisten verwendet, da diese beliebig erweiterbar sind, ohne die Schnittstelle zu ändern.
- `Parcelable`-Objekte eignen sich nicht dafür, persistiert zu werden! Diese Objekte enthalten ihre eigene Serialisierungsvorschrift für IPC-Aufrufe. Ein früher mal gespeichertes `Parcelable`-Objekt ist vielleicht längst überholt, weil ein weiterer Parameter hinzugekommen ist. Verwendet man es in einem IPC-Call, kann das schwer zu findende Fehler in den übertragenen Daten hervorrufen, ohne dass ein Laufzeitfehler das Problem deutlich gemacht hat.
- In AIDL werden derzeit noch keine Exceptions über Prozessgrenzen hinweg unterstützt. Wir haben zwar in den obigen Beispielen schon `RemoteExceptions` gefangen, jedoch werden diese nicht ausgelöst. Fehlerbehandlung muss über Rückgabewerte implementiert werden.
- AIDL-Interfaces kennen keine `public`-Methoden. Sie sind »package-private« zu deklarieren.
- Alle verwendeten Datentypen müssen per `import`-Statement bekannt gemacht werden, auch wenn sie im selben Paket liegen.

- Die Methodennamen müssen alle unterschiedlich sein. Hier verhält es sich wie bei Webservices, Corba oder COM. Es reicht nicht, dass die Übergabeparameter unterschiedlich sind.
- In AIDL-Interfaces sind nur Methodendeklarationen erlaubt, keine statischen Variablen.

8.3.2 Threads

Am Anfang des Kapitels haben wir schon einiges über Threads erfahren. Da Threads nichts Android-spezifisches sind, setzen wir hier Kenntnisse über das Programmieren von Threads voraus und konzentrieren uns auf die Besonderheiten bei Android.

Threads für langlaufende Programmteile einsetzen

Als Programmierer setzt man Threads dort ein, wo langlaufende Programmteile das Verhalten unserer Anwendung nicht negativ beeinflussen sollen. Kandidaten für Threads sind z.B. Netzwerkkommunikation, Laden und Speichern von Daten oder (synchrone) Kommunikation mit anderen Services.

Da wir den Thread für eine klar umrissene Aufgabe verwenden wollen, müssen wir oft auch wissen, wann er diese Aufgabe erledigt hat. Dazu dient unter anderem ein Callback-Mechanismus, der auf einem Handler (`android.os.Handler`) beruht. Der Handler ermöglicht es dem Thread, mit unserem eigentlichen Programmcode (meist Code innerhalb des UI-Threads) zu kommunizieren.

Threads besitzen eine Message Queue.

Threads besitzen eine Message Queue, um an sie gerichtete Nachrichten zu speichern. Für die Kommunikation zwischen dem Thread und unserem eigentlichen Programm ist diese Message Queue von großer Bedeutung. Kommuniziert wird über Message- oder Runnable-Objekte, die wir in die Message Queue des Threads stellen. Message-Objekte (`android.os.Message`) gehören zum Android-SDK. Runnable-Objekte zum Standardsprachumfang von Java (`java.lang.Runnable`).

Ein Handler für die Message Queue

Handler werden in einem Thread erzeugt und können an einen anderen Thread als Parameter übergeben werden. Sie erlauben es, Message-Objekte an die Message Queue des ursprünglichen Threads zu senden oder Runnable-Objekte in dem anderen Thread auszuführen.

Callbacks via Handler

Indem man einen Handler für die Message Queue des aktuellen Threads (des UI-Threads) an den Thread für den Hintergrundprozess übergibt, schafft man sich einen Callback-Mechanismus. Runnable-Objekte enthalten selbst den Code, der beim Callback im UI-Thread ausgeführt werden soll. Message-Objekte werden vom Thread erzeugt und über den Handler in die Message Queue des UI-Threads gestellt, woraufhin der Handler seine `handleMessage`-Methode aufruft. Sie dienen nur als Trigger, um den Handler zu veranlassen, seine `handleMessage`-

Methode auszuführen. Dort kann das Message-Objekt ausgewertet werden.

Beide Möglichkeiten, Callbacks mittels `Message`-Objekt oder `Runnable`-Objekt, werden wir gleich in der Praxis betrachten. Fassen wir zusammen:

- Handler stellen eine Kommunikationsschnittstelle zwischen Threads zur Verfügung.
- Runnable-Objekte erlauben die Ausführung von Programmteilen eines Threads innerhalb eines anderen Threads.
- Message-Objekte dienen zum Transportieren von Daten aus einem Thread in einen anderen.

Callbacks mittels Runnable

Listing 8.14 *Beispiel mit Runnable*

```
public class ActivityMitThread extends Activity {
  private Handler handler = new Handler(); // (1)
  private long ergebnis = 0;

  public void onCreate(Bundle bundle) {
    ...
  }

  private void langlaufendeOperation() { // (2)
    Thread thread = new Thread() {
      public void run() {
        ergebnis = berechneErgebnis();
        handler.post(aktualisiereActivity); // (3)
      }
    };
    thread.start();
  }

  private Runnable aktualisiereActivity = // (4)
      new Runnable() {
    public void run() {
      final TextView fldErgebnis = (TextView)
          findViewById(R.id.txErgebnis);
      fldErgebnis.setText(String.valueOf(ergebnis));
    }
  };

  private long berechneErgebnis() {
    // dauert sehr lange...
```

```
    return ergebnis;
  }
}
```

In Listing 8.14 haben wir zunächst einen Handler erzeugt, der uns den Zugriff auf die Message Queue des UI-Threads zur Verfügung stellt. Wenn die Methode `langlaufendeOperation` aufgerufen wird (2), wird ein Thread gestartet, der erst eine langwierige Berechnung durchführt. Der Thread hat keine Möglichkeit, auf GUI-Komponenten der Activity zuzugreifen. Daher wird die `post`-Methode des Handlers `handler` aufgerufen (3). Dadurch wird das Runnable-Objekt `aktualisiereActivity` (4) ausgeführt. Es überschreibt die Methode `run`. Hier wird der Code eingefügt, der ausgeführt werden soll, nachdem der Thread seine Aufgabe (Methode `berechneErgebnis`) erledigt hat.

Ausführbaren Code auslagern: das Runnable-Objekt

Da der Handler Teil der Activity (des UI-Threads) ist, sorgt er dafür, dass das Runnable `aktualisiereActivity` auch im Thread der Activity ausgeführt wird. `aktualisiereActivity` selbst ist Teil der Activity und nicht des Threads und darf daher auf die UI-Komponenten der Activity zugreifen. Damit führt der Thread einen Callback zurück in den UI Thread aus.

Callback mittels Handler und Runnable

Alternativ zur `post`-Methode des Handlers gibt es noch weitere Methoden, um Runnables in die Message Queue zu stellen. Hier nur zwei weitere wichtige Methoden:

postAtTime(Runnable r, long uptimeMillis) Ausführung des Runnables zu einem bestimmten Zeitpunkt in der Zukunft. `uptimeMillis` ist die Zeit seit dem letzten Booten des Betriebssystems und kann mittels `android.os.SystemClock.uptimeMillis()` ermittelt werden.

postDelayed(Runnable r, long delayMillis) Ausführung des Runnables in delayMillis Millisekunden nach Aufruf dieser Methode

Callbacks mittels Message-Objekt

Listing 8.15 *Beispiel mit Message-Objekt*

```
public class ActivityMitThread extends Activity {
 private Handler handler = new Handler() {
    public void handleMessage(Message msg) { // (1)
      Bundle bundle = msg.getData();
      long ergebnis = bundle.getLong("ergebnis");
      final TextView fldErgebnis =
          (TextView) findViewById(R.id.txErgebnis);
      fldErgebnis.setText(String.valueOf(ergebnis));
      super.handleMessage(msg);
    }
  };
```

```
  public void onCreate(Bundle bundle) {
    ...
  }

  private void langlaufendeOperation() {
    Thread thread = new Thread() {
      public void run() {
        long ergebnis = berechneErgebnis();

        Message msg = new Message();
        Bundle bundle = new Bundle();
        bundle.putLong("ergebnis", ergebnis);
        msg.setData(bundle);
        handler.sendMessage(msg); // (2)
      }
    };
    thread.start();
  }

  private long berechneErgebnis() {
    // dauert sehr lange...
    return ergebnis;
  }
}
```

Listing 8.15 zeigt, wie man Daten aus einem Thread an den aufrufenden Thread übergibt. Unser Handler `handler` implementiert diesmal die Methode `handleMessage` (1). Sie wird mittels `handler.sendMessage(msg)` aufgerufen (2), wenn der Thread seine Aufgabe erledigt hat. Die Methode `sendMessage` nimmt ein Message-Objekt entgegen. Das Message-Objekt ist vielseitig verwendbar. Wir haben es uns ein wenig kompliziert gemacht und ein `android.os.Bundle` verwendet, um darin einen einzigen Wert vom Typ `long` zu speichern. Ein `Bundle` ist ein Container für Schlüssel-Wert-Paare.

Bundle als Datencontainer für Rückgabewerte

Der Aufruf der Methode `sendMessage` des Handlers bewirkt, dass das `Message`-Objekt `msg` in die Message Queue des Threads gestellt wird, der den Handler erzeugt hat. In unserem Fall handelt es sich um den UI-Thread, also den Thread der `ActivityMitThread`. Die Abarbeitung der Elemente in der Message Queue erfolgt sequentiell. Jedes Message-Objekt löst den Aufruf der Methode `handleMessage` des Handlers aus (1). Dort extrahieren wir aus der Message das Bundle `bundle` und daraus das Ergebnis unserer Berechnung.

Aufruf von handleMessage

Der Handler kennt auch im Fall von Message-Objekten wieder verschiedene Methoden, um eine Message in die Message Queue zu stel-

len. Hier auch wieder nur eine Teilmenge. Die weiteren Methoden kann man der API-Dokumentation entnehmen:

sendEmptyMessage(int what) Stellt ein leeres Message-Objekt in die Message Queue. Mit Hilfe des `what`-Attributs kann man z.B. einen Statuswert (OK, Error etc.) des Threads an den Aufrufer übergeben. Dies ist eine schlanke und performante Möglichkeit eines Callbacks.

sendMessageAtTime(Message msg, long uptimeMillis) Die `handleMessage`-Methode des Handlers wird zu einem bestimmten Zeitpunkt in der Zukunft ausgeführt.

sendMessageDelayed(Message msg, long delayMillis) Ausführung der `handleMessage`-Methode des Handlers in `delayMillis` Millisekunden

Abschließend noch der Hinweis, dass man oft auf das Bundle-Objekt verzichten kann. Das Message-Objekt bietet ein `public`-Attribut namens `obj`. Diesem können wir beliebige Objekte zuweisen und sparen uns die Erzeugung eines komplexen Bundle-Objekts:

```
Message message = new Message();
message.obj = new String("Hallo vom Thread!");
```

Threads mit Schleifen

Thread in Lauerstellung

Wer schon Erfahrung mit Threads gesammelt hat, hat wahrscheinlich auch schon öfter eine unendliche while-Schleife in der `run`-Methode eines Threads verwendet. Immer wenn wir wollen, dass unser Thread potenziell unendlich lange aktiv bleibt und auf Anfrage eine bestimmte (langwierige) Aufgabe verrichtet, müssen wir in der `run`-Methode bleiben und verwenden eine solche unendliche Schleife à la `while (true)`:

Listing 8.16
Klassisch: Thread mit Endlosschleife

```
public class KlassischerThread extends Thread {
  private volatile int threadStatus = THREAD_STATUS_WAIT;

  public void run() {
    while (true) {
      synchronized(getClass()) {
        if (threadStatus == THREAD_STATUS_WAIT) {
          wait(); // (3)
        }
```

```
        else if (threadStatus == THREAD_STATUS_WORK) {
          macheIrgendwas(); // (2)
        }
        else if (threadStatus == THREAD_STATUS_STOP) {
          break;
        }
      }
    }
  }

  public void beginneAufgabe() { // (1)
    synchronize(getClass()) {
      threadStatus = THREAD_STATUS_WORK;
      notify();
    }
  }
}
```

Damals... Thread mit `wait` und `notify`

Listing 8.16 zeigt die Implementierung eines Threads, der darauf wartet, dass seine Methode `beginneAufgabe` aufgerufen wird (1). Wenn dies erfolgt, verrichtet er seine Aufgabe (2) und verfällt anschließend wieder in den Wartezustand (3). Der Mechanismus beruht auf den Methoden `wait` und `notify`, die Bestandteil von `java.lang.Object` sind. Entsprechende Teile der Thread-Klasse haben wir synchronisiert, um Probleme mit der Nebenläufigkeit zu verhindern. Wir brauchen auf diese Weise den Thread nicht jedesmal neu zu erzeugen und zu starten, wenn wir ihn verwenden wollen, denn das sind recht teure Operationen. Darüber hinaus wäre auch denkbar, dass der Thread eine dauerhafte Internetverbindung zu einem Server offenhält, über die er bei Bedarf Daten abruft. Diese müsste dann jedesmal neu aufgebaut werden, und in vielen Mobilfunkverträgen entstehen dadurch sogar zusätzliche Kosten.

Threads haben keine Message Queue.

Das Android-SDK bietet uns eine elegantere Art, einen Thread dauerhaft im Hintergrund aktiv zu halten. Allerdings beruht diese Methode auf der Verwendung eines Handlers. Threads haben jedoch keine Message Queue, worauf der Handler operieren könnte. Das Android-SDK besitzt eine Klasse namens `android.os.Looper`. Diese Klasse verwaltet eine Message Queue für einen Thread. Wir bauen das Beispiel aus Listing 8.16 so um, dass es die Vorteile des Android-SDK nutzt.

Listing 8.17 *Android-Version des Threads mit Endlosschleife*

```
public class ThreadMitLooper extends Thread {
  private Handler handler;

  public void run() {
    Looper.prepare(); // (1)
```

```
    handler = new Handler() { // (2)
      public void handleMessage(Message msg) {
        macheIrgendwas();
      }
    };

    Looper.loop(); // (3)
  }

  public void beginneAufgabe() {
    handler.sendEmptyMessage(0); // (4)
  }

  public void beendeThread() {
    handler.getLooper().quit(); // (5)
  }
}
```

Listing 8.17 implementiert einen Thread mit Message Queue. Die Verwendung der Klasse `Looper` und die Erzeugung einer Instanz von `Handler` muss in der `run`-Methode durchgeführt werden, da nur der Quellcode, der innerhalb dieser Methode steht, im Thread ausgeführt wird.

Mittels `Looper.prepare()` (1) wird der Thread darauf vorbereitet, die `run`-Methode nicht zu verlassen und dort zu verweilen. Gleichzeitig wird der Looper in die Lage versetzt, einen Handler aufzunehmen, der den Looper referenziert und auf dessen Message Queue operiert.

Erst nach dem Aufruf der Methode `prepare` darf eine Instanz der Klasse `Handler` erzeugt werden (2). Sobald wir nun `Looper.loop()` (3) aufrufen, verbleibt der Programmablauf innerhalb des Loopers in einer Endlosschleife und damit innerhalb der `run`-Methode.

Sobald nun der Thread verwendet werden soll, ruft man seine `beginneAufgabe`-Methode auf. Dort wird eine Nachricht an die Message Queue im Looper geschickt und die Methode `handleMessage` im Handler aufgerufen, wo die eigentliche Aufgabe des Threads erledigt wird.

Wichtig ist, dass der Looper gestoppt wird (5), bevor man den Thread beendet, um seine Schleife zu beenden und die Ressourcen freizugeben.

Die Verwendung von Looper und Handler bei der Implementierung dieser Art von Threads hat noch einen gewichtigen Vorteil: Der Programmierer muss sich keine Gedanken wegen der Nebenläufigkeit machen, wenn der Thread von mehreren parallel laufenden Prozessen oder Threads verwendet wird. Handler und Message Queue sorgen für eine sequentielle Abarbeitung der Message-Objekte in der Message Queue.

8.4 Fazit

Wir haben uns mit Problemen beschäftigt, die in der klassischen Java-Welt nicht jeden Tag vorkommen. Android-Programme nutzen Activities, und diese müssen »responsive« bleiben. Braucht die Activity zu lange, erfolgt eine ANR-Meldung (Application Not Responding). Langsame Internetverbindungen oder langlaufende Operationen zwingen uns bei Android, häufiger als von Client-Server-Anwendungen gewohnt, auf Hintergrundprozesse zurückzugreifen.

Das Android-SDK bietet uns neben den aus dem Java-SDK bekannten Threads eine neue Klasse namens Service an. Wir haben uns angeschaut, wie sich Services erzeugen lassen und wie man mit ihnen kommuniziert. Auch haben wir gesehen, dass ein Service im gleichen Betriebssystemprozess wie unsere Anwendung laufen kann oder in einem eigenen Prozess. Wir haben zwischen Local Services und Remote Services unterschieden und uns am Beispiel angeschaut.

Fassen wir die Kernbegriffe dieses Kapitels zusammen.

Betriebssystemprozesse: Jede Android-Anwendung läuft in einem eigenen Prozess und sogar in einer eigenen DVM. Um diesen Prozess müssen wir uns als Programmierer nicht weiter kümmern.

UI-Thread: Der UI-Thread ist der Thread, in dem unsere Start-Activity läuft, die wir als solche im Android-Manifest deklariert haben. Der UI-Thread läuft im Prozess unserer Anwendung.

Threads: Threads können überall in Android-Programmen eingesetzt werden, um im Hintergrund zeitaufwendige Aufgaben zu erledigen. Solange wir den Thread nicht in einem Remote Service starten, läuft er innerhalb des UI-Threads. Wir können Threads programmieren, die einmalig eine Aufgabe erledigen oder die in ihrer `run`-Methode verweilen und auf den nächsten Einsatz warten. Wir haben gesehen, dass das Android-SDK hierfür die Klasse `Looper` zur Verfügung stellt, die einige Vorteile gegenüber der klassischen Porgrammierung hat.

Services: Services erledigen Aufgaben im Hintergrund, für die keine Oberfläche gebraucht wird. Wir unterscheiden Local Services von Remote Services. Local Services sind an den UI-Thread und damit an den Prozess einer Anwendung gekoppelt. Remote Services laufen in einem eigenen Prozess und laufen weiter, auch wenn die Anwendung, die den Service gestartet hat, beendet wurde.

Nachdem wir im Theorieteil Threads und Services erklärt haben, haben wir uns an die Praxis gewagt und anhand ausführlicher Codebeispiele erst einen Local Service und dann einen Remote Service implementiert. Bei den Remote Services haben wir IPC mit AIDL kennengelernt. Da die Kommunikation zwischen einer Anwendung und einem Remote Service auf Betriebssystemebene erfolgt, müssen Objekte, die als Parameter einer Methode übergeben werden, serialisiert und später wieder deserialisiert werden. In diesem Zusammenhang haben wir das `Parcelable`-Interface kennengelernt, welches praktisch das AIDL-Pendant zum `Serializable`-Interface in Java ist.

Wir haben das Kapitel mit einem kleinen Ausflug in die Welt der Threads beendet. Dabei war uns natürlich wichtig, sie im Android-Kontext zu betrachten. Hier haben wir die Klasse `android.os.Handler` kennengelernt, mit deren Hilfe wir Zugriff auf die Message Queue eines Threads erhalten. Mit Hilfe dieser Klasse können wir einen Callback-Mechanismus implementieren, über den unser Thread mit dem ihn aufrufenden Thread kommunizieren kann. Wir haben dann zwei Callback-Mechanismen ausprobiert, einmal über Runnable-Objekte und einmal über Message-Objekte.

9 Exkurs: Systemnachrichten

Hinweis: Die 2. Auflage dieses Buchs erscheint unter dem Titel
»Android 2. Grundlagen und Programmierung«
Ende Mai 2010 im dpunkt.verlag, ISBN 978-3-89864-677-2

Wir haben in den vergangenen beiden Kapiteln gelernt, dass Android-Anwendungen aus lose gekoppelten Komponenten bestehen. Bisher haben wir Activities als sichtbare Schnittstelle zum Anwender und Services als Komponente zur Behandlung von Hintergrundprozessen kennengelernt. Der »Leim« (engl. *glue*), der diese Komponenten zusammenhält und zu einer Anwendung verbindet, sind die Intents, wie wir sie aus Kapitel 7 kennen. Wir werden in diesem Exkurs eine weitere Android-Komponente kennenlernen, den *Broadcast Receiver*. Er reagiert auf bestimmte Intents, die *Broadcast Intents*, die Informationen über Änderungen des Zustands der Android-Plattform enthalten. Daher bezeichnen wir hier die Broadcast Intents als Systemnachrichten.

Eine zweite Form von Systemnachrichten sind die *Notifications*. Mit ihrer Hilfe kann man den Anwender über ein Ereignis informieren, ohne eine Activity dafür zu verwenden.

Wir werden in den folgenden Abschnitten die beiden Arten von Systemnachrichten vorstellen.

9.1 Broadcast Intents

Broadcast Intents sind eine spezielle Art von Intents, die auf Betriebssystemebene verschickt werden und nicht auf Anwendungsebene. Broadcast Intents werden von der Android-Plattform in großen Mengen verschickt und informieren Anwendungen über Systemereignisse. Um zu verdeutlichen, was mit Systemereignissen gemeint ist, haben wir uns in Tabelle 9-1 exemplarisch einige Broadcast Intents herausgepickt und erklärt, worüber sie informieren.

Eine vollständige Auflistung der Broadcast Intents kann man der API-Dokumentation zu der Klasse `Intent` des Android-SDKs entnehmen. Dort sind beide Arten von Intents erklärt. In der Dokumentation wird zwischen »Broadcast Action« und »Activity Action« unterschieden, damit man beide Intents nicht verwechselt.

Nicht verwechseln!

Der Unterschied zwischen normalen Intents und Broadcast Intents ist, dass sie auf verschiedenen Ebenen verschickt werden. Broadcast In-

Tab. 9-1
Einige Broadcast Intents...

Broadcast Intent	Systemereignis
`ACTION_BOOT_COMPLETED`	Wird einmal verschickt, wenn die Boot-Sequenz der Android-Plattform nach einem Neustart des Android-Geräts abgeschlossen ist. Erfordert die Vergabe der Berechtigung `RECEIVE_BOOT_COMPLETED` im Android-Manifest.
`ACTION_BATTERY_LOW`	Wird verschickt, wenn der Akku des Android-Geräts fast leer ist.
`ACTION_PACKAGE_ADDED`	Wird verschickt, wenn eine neue Android-Anwendung auf dem Gerät installiert wurde.
`ACTION_SCREEN_OFF`	Wird verschickt, wenn sich der Bildschirm nach längerer Inaktivität abschaltet.

Getrennte Ebenen

tents werden auf der Ebene des Anwendungsrahmens (vgl. Abbildung 2-1), auch *Application Framework* genannt, verschickt. Intents werden auf Ebene der Anwendungsschicht verschickt. Somit kann man mit einem Intent-Filter einer Activity keinen Broadcast Intent fangen. Er bleibt für die Activity unsichtbar. Gleiches gilt umgekehrt, was aber die Frage aufwirft, wie man Broadcast Intents fängt.

9.2 Broadcast Receiver

Systemnachrichten abfangen

Broadcast Intent Receiver oder kurz *Broadcast Receiver* (`android.os.BroadcastReceiver`) dienen dazu, auf der Lauer zu liegen und auf bestimmte Intents zu warten.

Dynamische Deklaration von Broadcast Receivern

Um einen Broadcast Receiver zu verwenden, haben wir wieder zwei Möglichkeiten. Die eine ist, ihn wie eine Activity oder einen Service im Android-Manifest zu deklarieren. Dann wird der Broadcast Receiver zum Zeitpunkt der Installation der Anwendung, die das Android-Manifest enthält, in der Android-Plattform registriert. Jeder Broadcast Intent, auf den der Broadcast Receiver lauscht, startet diesen und er verrichtet seine Arbeit.

Die zweite Möglichkeit ist, einen Broadcast Receiver dynamisch zu verwenden, indem man ihn in einer Activity oder in einem Service implementiert. Wir schauen uns an einem Beispiel zunächst diese zweite Variante an.

9.3 Dynamische Broadcast Receiver

Listing 9.1
Einen Broadcast Receiver dynamisch verwenden

```
public class DatenEingabeActivity extends Activity {

  private EnergiekrisenReceiver mBroadcastReceiver;
  private Context context = null;

  private final IntentFilter intentFilter =
      new IntentFilter(
      "android.intent.action.BATTERY_LOW"); // (1)

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    ...

    Button buttonSendeBroadcastIntent = // (2)
        (Button)findViewById(
         R.id.opt_sendeBroadcastIntent);
    buttonSendeBroadcastIntent.setOnClickListener(
        mButtonSendeBroadcastIntent);

    Button buttonBeenden =
        (Button)findViewById(R.id.opt_demoBeenden);
    buttonBeenden.setOnClickListener(mButtonQuitListener);

    context = this;
  }

  private class EnergiekrisenReceiver // (3)
      extends BroadcastReceiver {
    @Override
    public void onReceive(Context ctxt, Intent intent) {
      Toast.makeText(DatenEingabeActivity.this,
          "Akku fast leer!", Toast.LENGTH_SHORT).show();
    }
  };

  private OnClickListener mButtonSendeBroadcastIntent =
      new OnClickListener() {
    public void onClick(View v) {
      Intent broadcastIntent =
          new Intent(
          "android.intent.action.BATTERY_LOW");
      context.sendBroadcast(broadcastIntent); // (4)
    }
  };
```

```
  private OnClickListener mButtonBeendenListener =
      new OnClickListener() {
    public void onClick(View v) {
      finish();
    }
  };

  @Override
  protected void onResume() { // (5)
    super.onResume();
    mBroadcastReceiver = new EnergiekrisenReceiver();
    context.registerReceiver(mBroadcastReceiver,
        intentFilter);
  }

  @Override
  protected void onPause() { // (6)
    super.onPause();
    context.unregisterReceiver(mBroadcastReceiver);
  }
}
```

Hinweis mittels Toast

Die Activity in Listing 9.1 hat die Aufgabe, dem Anwender eine Meldung (ein `android.widget.Toast`) anzuzeigen, wenn der Akku fast leer ist. Dies hätte zum Beispiel in einer Activity Sinn, in der der Anwender einen längeren Text eingibt. Es bliebe dann genug Zeit, die Eingabe abzuspeichern. Abbildung 9-1 zeigt die Activity, nachdem sie einen Broadcast Intent empfangen hat, der über den leer werdenden Akku informiert.

Damit der Broadcast Receiver auf den vom System verschickten Broadcast Intent reagieren kann, definieren wir zuerst einen IntentFilter, der den `android.intent.action.BATTERY_LOW`-Intent passieren lässt (1).

Schwachen Akku simulieren

In der `onCreate`-Methode haben wir eine Schaltfläche zum Abschicken des Broadcast Intents deklariert (2). Damit können wir den Intent auf Knopfdruck auslösen (4) und den Broadcast Receiver testen, auch wenn der Akku noch voll ist oder wir die Anwendung im Emulator laufen lassen.

Was nun noch fehlt, ist der Broadcast Receiver (3). Wenn wir einen Broadcast Receiver verwenden, überschreiben wir nur die Methode `onReceive`. Damit der Broadcast Receiver der Android-Plattform bekannt ist, wird er in der `onResume`-Methode mittels `Context.registerReceiver(BroadcastReceiver,IntentFilter)` registriert (5). Diese Methode wird automatisch aufgerufen, wenn die Activity

Abb. 9-1
Activity mit Warnung

angezeigt wird (mehr zu den Methoden des Lebenszyklus einer Activity finden Sie im Exkurs 13).

Wir haben so erreicht, dass der Broadcast Intent `android.intent.action.BATTERY_LOW` durch den Broadcast Receiver `EnergiekrisenReceiver` abgefangen wird, während die Activity angezeigt wird. Wird ein solcher Intent ausgelöst, wird die `onReceive`-Methode aufgerufen und die Meldung (»Akku fast leer«) angezeigt.

Da der Broadcast Receiver innerhalb des Threads der Activity läuft, ist die Activity während der Ausführung der `onReceive`-Methode nicht ansprechbar.

ANR möglich

Achtung!

Die `onReceive`-Methode muss in weniger als 5 Sekunden abgearbeitet sein, sonst riskieren wir einen ANR.

9.4 Statische Broadcast Receiver

Unter statischen Broadcast Receivern verstehen wir solche, die im Android-Manifest deklariert werden. Wir implementieren diesmal den Broadcast Receiver in einer eigenen Klasse namens `SystemstartReceiver`.

Seine Aufgabe ist es, nach jedem Neustart des Android-Geräts automatisch die Start-Activity einer Anwendung aufzurufen.

Listing 9.2 *Einen Broadcast Receiver statisch verwenden*

```
public class SystemstartReceiver extends
    BroadcastReceiver {

  public static final String MEIN_INTENT =
      "de.androidbuch.demo.MEIN_INTENT";

  @Override
  public void onReceive(Context context, Intent intent) {
    Intent i = new Intent(MEIN_INTENT, intent.getData());
    i.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);

    context.startActivity(i);
  }
}
```

Keine expliziten Intents verwenden

In Listing 9.2 verwenden wir einen impliziten Intent, wie wir ihn aus Abschnitt 7.3 kennen. Ein expliziter Intent lässt sich in der `onReceive`-Methode eines statischen Broadcast Receivers nicht nutzen. Der Intent muss den Umweg über die Anwendungsschicht nehmen, damit der Activity Manager ihn in einem UI-Thread starten kann. Aus diesem Grund haben wir im Listing ein Flag für den Intent setzen müssen (`Intent.FLAG_ACTIVITY_NEW_TASK`). Dies verhindert, dass wir zur Laufzeit eine Fehlermeldung erhalten. Das Flag teilt dem Activity Manager mit, dass die Activity in einem neuen Thread gestartet werden muss.

Wir müssen nun für den impliziten Intent eine Ziel-Activity und einen Intent-Filter deklarieren.

Listing 9.3 *Android-Manifest für den statischen Broadcast Receiver*

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

<application android:icon="@drawable/icon">
  ...
  <activity android:name=".AutostartActivity">
    <intent-filter>
      <action android:name=
          "android.intent.action.MAIN" />
      <category android:name=
          "android.intent.category.LAUNCHER" />
    </intent-filter>
    <intent-filter>
      <action android:name=
          "de.androidbuch.demo.MEIN_INTENT" />
```

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

  <receiver android:name=".SystemstartReceiver">
    <intent-filter>
      <action android:name=
          "android.intent.action.BOOT_COMPLETED" />
    </intent-filter>
  </receiver>
...
```

Die Android-Plattform verschickt nach dem erfolgreichen Bootvorgang einen Broadcast Intent namens `Intent.ACTION_BOOT_COMPLETED`. Dies machen wir uns in Listing 9.3 zunutze. Der Intent-Filter des Broadcast Receivers sorgt dafür, dass der `SystemstartReceiver` aufgerufen wird. Dieser sorgt für den Aufruf der `AutostartActivity`.

Die Anwendung muss nicht laufen.

Das Besondere an dem statischen Broadcast Receiver ist, dass keine Anwendung laufen muss. Mit dem oben gezeigten Muster können wir eigene Anwendungen oder Services automatisch starten, wenn das Android-Gerät gestartet wird. Statische Broadcast Receiver werden in der Android-Plattform registriert, sobald die Anwendung installiert wird, zu der sie gehören. Ab dem Zeitpunkt lauschen sie auf Broadcast Intents, egal ob die Anwendung läuft oder nicht.

Allerdings haben auch Broadcast Receiver einen kleinen Haken. Sie dienen nur als reine Empfänger für Intents, die Arbeit machen andere. Das erfordert etwas mehr Aufwand bei der Implementierung und eine gewisse Vorsicht. Fangen wir mit Letzterem an.

Ein Broadcast Receiver, egal ob statisch oder dynamisch, ist Teil einer Anwendung. Startet ein Broadcast Receiver eine Activity, so läuft diese im gleichen Prozess wie der Broadcast Receiver.

Nehmen wir nun folgendes Szenario an: In der `onReceive`-Methode des Broadcast Receivers starten wir einen Thread, bevor wir die Activity aufrufen.

```
public void onReceive(Context context, Intent intent) {
  starteThread();

  Intent i = new Intent(MEIN_INTENT, intent.getData());
  i.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);

  context.startActivity(i);
}
```

Die Activity wird gestartet und die `onReceive`-Methode des Broadcast Receivers verlassen. Wird nun die Activity beendet, beendet das Be-

triebssystem den gemeinsamen Prozess. Dadurch wird auch der Thread beendet, was vermutlich nicht gewollt war.

Achtung!

Keine Threads in der `onReceive`-Methode eines Broadcast Receivers starten.

Wir können aber einen Service verwenden, um eine komplexe Aufgabe im Hintergrund zu erledigen, mit der wir nicht die `onReceive`-Methode belasten möchten. In dieser können wir nämlich nicht beliebig langwierige Aufgaben erledigen, weil wir sonst einen ANR riskieren.

Achtung!

Wenn die Ausführung der `onReceive`-Methode länger als 10 Sekunden dauert, wird ein ANR (*Application Not Responding*) ausgelöst.

Vorsicht: Nicht mit dem Service verbinden

Wenn wir aber den langlaufenden Programmteil in einen Service auslagern, dürfen wir uns allerdings nicht mittels `Context.bindService(Intent,ServiceConnection,int)` an diesen Service binden, sondern dürfen ihn nur mittels `Context.startService(Intent service)` starten. Sonst hätten wir das gleiche Problem wie bei Threads: Sobald wir die `onReceive`-Methode verlassen, ist der Broadcast Receiver nicht mehr aktiv, und das System könnte unseren Prozess beenden. Zwar haben wir eine Verbindung zum Service, über die wir über die Fortschritte der Arbeit informiert werden. Diese Verbindung wird aber mit dem Broadcast Receiver abgeräumt. Der Service muss selbst wissen, was er mit dem Ergebnis seiner Arbeit zu tun hat. Aber dafür gibt es ja zum Beispiel Intents. Eine weitere Möglichkeit sind *Notifications*. Mit dieser Art von Systemnachricht beschäftigen wir uns im folgenden Abschnitt.

9.5 Meldungen an den Notification Manager

Es gibt eine Möglichkeit, den Anwender über ein Ereignis zu informieren, ohne eine Activity zu verwenden. Eine *Notification* (`android.app.Notification`) ist eine Nachricht auf Systemebene, die vom *Notification Manager* empfangen wird.

Der Notification Manager (`android.os.NotificationManager`) informiert den Anwender über ein Ereignis, ohne die aktuell angezeigte Ac-

tivity zu beinflussen. Denn dies wird oft als störend empfunden, weil es den Anwender bei seiner Tätigkeit unterbricht. Ein dezenter Hinweis, dass eine neue Information vorliegt, ist völlig ausreichend. Einen MC, einen mobilen Computer, verwendet man anders als einen PC. Oft wird das Gerät gar nicht beachtet, und man macht etwas anders. Von SMS sind wir gewöhnt, dass ihr Empfang einen kurzen Ton oder nur ein Vibrieren des Mobiltelefons auslösen. Um solche dezenten Hinweise geht es bei Notifications, die der Notification Manager empfängt und umsetzt.

Dezenter Hinweis mittels Notification

Notifications können sich auf folgende Arten für den Anwender bemerkbar machen:

- Die Geräte-LED kann angeschaltet oder zum Blinken gebracht werden.
- In der Statuszeile kann ein Piktogramm angezeigt werden.
- Der Bildschirmhintergrund kann flackern, das Gerät vibrieren oder ein Sound ertönen.

Notifications werden meist bei Hintergrundoperationen eingesetzt, die nicht sofort eine Activity starten sollen, um ihr Ergebnis anzuzeigen. Typische Komponenten, die über Notifications mit dem Anwender kommunizieren, sind Broadcast Receiver, Services und nicht aktive Activities, also solche, die zwar gestartet, aber gerade nicht angezeigt werden. Zur Lebensdauer von Activities werden wir ausführlich in Kapitel 13 etwas sagen.

Hintergrundprozesse verwenden Notifications.

Wir schauen uns nun an, wie man Notifications verwendet. Nehmen wir mal an, wir hätten einen Service programmiert, der in regelmäßigen Abständen den Versandstatus einer von uns in einem Online-Shop bestellten Ware prüft. Wenn die Ware versendet wird, wollen wir darüber informiert werden. Der Service ruft in diesem Fall die Methode `benachrichtigeAnwender` auf. Dort versenden wir die Notification. Die Notification soll in der Lage sein, die Activity `BestellstatusActivity` zu starten, die uns Detailinformationen zur Bestellung anzeigt.

Listing 9.4
Versenden einer Notification

```
...
private static final int NOTIFICATION_ID = 12345;
...
private void benachrichtigeAnwender(String bestellNr) {
  Context context = getApplicationContext();

  NotificationManager notificationManager = // (1)
      (NotificationManager)getSystemService(
      Context.NOTIFICATION_SERVICE);
```

```
    String nText = "Bestellung Nr. " + bestellNr +
        " wurde versendet.";

    Notification notification = new Notification( // (2)
        R.drawable.icon, nText,
        System.currentTimeMillis());

    Intent activityIntent = new Intent(context, // (3)
        BestellstatusActivity.class);
    activityIntent.putExtra("bestellNr", bestellNr);
    activityIntent.putExtra("notificationNr",
        NOTIFICATION_ID);
    activityIntent.addFlags(
        Intent.FLAG_ACTIVITY_NEW_TASK); // (4)

    PendingIntent startIntent = PendingIntent.getActivity(
        context, 0, activityIntent, 0); // (5)

    notification.setLatestEventInfo(context,
        "Bestellstatus ansehen",
        "Best.-Nr.: " + bestellNr, startIntent);
    notification.vibrate = new long[] {100, 250}; // (6)

    notificationManager.notify(NOTIFICATION_ID, // (7)
        notification);
}
```

Notifications werden an den Notification Manager geschickt. In Listing 9.4 holen wir daher erst mal einen Notification Manager von der Android-Plattform (1).

Die Notification selbst besteht aus einem Piktogramm, welches in der Statuszeile des Android-Geräts angezeigt wird. Als zweiten Parameter übergibt man einen Anzeigetext. Der Anzeigetext wird neben dem Piktogramm in der Statuszeile angezeigt. Ist der Text zu lang, wird er Zeile für Zeile angezeigt. Abbildung 9-2 (a) zeigt den Emulator, der die Notification empfangen hat (2).

In der Methode `benachrichtigeAnwender` definieren wir als Nächstes einen Intent zum Starten einer Activity. Dadurch lässt sich durch Anklicken der Notification eine Android-Komponente starten. In unserem Fall wollen wir die Activity `BestellstatusActivity` starten (3).

Notifications müssen gelöscht werden.

Wir geben dem Intent die Bestellnummer und eine eindeutige Id für die Notification mit. Diese Id brauchen wir in der Activity, um die Notification wieder zu löschen, da dies nicht automatisch geschieht.

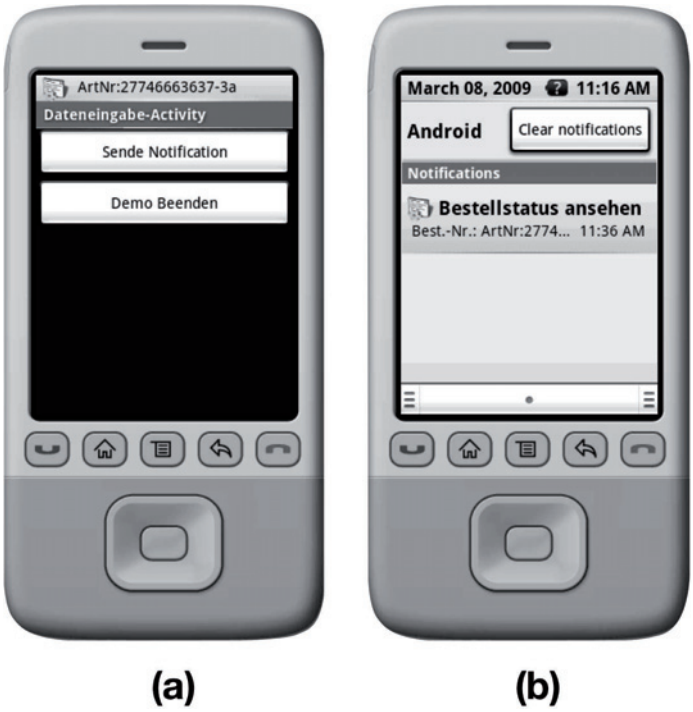


Abb. 9-2
Empfang und Anzeige einer Notification

Auch hier muss man darauf achten, dem Intent das Flag `Intent.FLAG_ACTIVITY_NEW_TASK` hinzuzufügen, wenn die Komponente nicht im UI-Thread läuft (4).

Als Nächstes definiert man einen `android.app.PendingIntent`. Intents werden nur abgeschickt. Gibt es keinen Empfänger für den Intent, geht er ins Leere. Ein `PendingIntent` wird von der Android-Plattform gespeichert. In ihm steckt ein normaler Intent, der auf diese Weise »konserviert« wird. Irgendwann ruft der Anwender den Notification Manager auf und klickt auf die Notification. Dadurch wird der darin enthaltene Intent ausgelöst und die vereinbarte Activity gestartet. Die Methode

Langlebige Intents

```
Notification.setLatestEventInfo(Context context,
    CharSequence contentTitle, CharSequence contentText,
    PendingIntent contentIntent)
```

dient dazu, die Anzeige der Notification zu definieren. Eine Notification lässt sich anzeigen, indem man die Statuszeile anfasst und ganz herunter bis zum unteren Rand des Bildschirms zieht. Im Emulator kann man dies mit der Maus machen. Abbildung 9-2 (b) zeigt die Anzeige der Notification. Hier finden sich die mit den Parametern `contentTitle` und `contentText` übergebenen Parameterwerte wieder.

Notifications anzeigen

Wir haben der Notification noch die Information mitgegeben, dass sie einen Vibrationsalarm auslösen soll. Dauer und Pause zwischen den Vibrationen werden durch die Parameter bestimmt (6). Um den Vibrationsalarm in einer Anwendung verwenden zu können, müssen die notwendigen Berechtigungen im Android-Manifest mittels

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

gesetzt werden. Abschließend weisen wir den Notification Manager an, die Notification zu verschicken (7).

Wir schauen uns nun noch an, wie die Activity den Intent empfängt und wie sie die Notification aus dem Notification Manager löscht. Dies erfolgt nicht automatisch. Damit wir die richtige Notification löschen, hatten wir dem Intent eine eindeutige Id hinzugefügt, die wir auch beim Abschicken der Notification (7) verwendet haben.

Listing 9.5 *Löschen einer Notification in der Ziel-Activity*

```
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  ...

  Intent intent = this.getIntent();

  String bestellNr = intent.getStringExtra("bestellNr");
  int notificationNr =
      intent.getIntExtra("notificationNr", 0);

  NotificationManager nm =
      (NotificationManager)getSystemService(
      Context.NOTIFICATION_SERVICE);
  nm.cancel(nNr);
  ...
}
```

Die Methode `cancel` des Notification Managers löscht anhand der selbst vergebenen Id `nNr` die Notification aus der Statuszeile des Android-Geräts.

9.6 Fazit

Wir haben in diesem Kapitel zwei Arten von Systemnachrichten kennengelernt. Die *Broadcast Intents* und die *Notifications*. Wir wissen nun, wie wir sie versenden und empfangen. Natürlich gibt es noch

viele Feinheiten, die wir verschwiegen haben. Ein Blick in die API-Dokumentation ist sinnvoll, um in eigenen Projekten alle Möglichkeiten auszuschöpfen.

Wir haben aber gesehen, dass Systemnachrichten eine besondere Form der Kopplung eigenständiger Komponenten sind. Neben Intents bilden sie den »Leim«, der die Komponenten verbindet. Dies allerdings auf einer anderen Schicht der Android-Plattform. Systemnachrichten werden über den Anwendungsrahmen verschickt und nicht zwischen Anwendungskomponenten direkt.

Dafür liefern sie aber Informationen über den Systemzustand (Broadcast Intents) oder informieren diskret über Ergebnisse (Notifications). Die richtige Wahl der Mittel macht wie immer eine gute Anwendung aus.

10 Exkurs: Dateisystem

Hinweis: Die 2. Auflage dieses Buchs erscheint unter dem Titel
»Android 2. Grundlagen und Programmierung«
Ende Mai 2010 im dpunkt.verlag, ISBN 978-3-89864-677-2

Dieser Exkurs gibt einen kurzen Einblick in das Dateisystem von Android-Geräten. Wir setzen dabei grundlegende Kenntnisse der Dateiverwaltung in Java sowie des Linux-Dateisystems voraus.

10.1 Aufbau des Dateisystems

Da dem Android-System ein Linux-Kernel zugrunde liegt, überrascht es nicht, dass wir es auch mit einem Linux-Dateisystem zu tun haben.

Basis-Dateisystem

Als Basis-Dateisystem bezeichnen wir das fest installierte Datei-Speichermedium des Geräts.

Externe Dateisysteme

Das Basis-Dateisystem lässt sich um Speicherkarten (SD-Karten) erweitern, deren Speicherplatz nicht an das Gerät gebunden ist. Bevor ein solches *externes Dateisystem* genutzt werden kann, muss es dem Betriebssystem mit Hilfe des Linux-Kommandos `mount` bekannt gemacht werden.

10.1.1 SD-Karten

Auf einem Android-Gerät muss eine SD-Karte mit `mount` angebunden werden. Teilweise erkennen die Geräte ihre externen Speichermedien automatisch. Ansonsten wird eine Notification verschickt, die man sich über den Statusbalken des Android-Geräts anzeigen lassen kann. Durch Anklicken der Notification kann man die SD-Karte mounten.

Sonderfall Emulator

Um die Verwendung der Speicherkarten auf dem Emulator simulieren zu können, sind ein paar vorbereitende Maßnahmen erforderlich.

Zunächst muss mit Hilfe des Kommandos `mksdcard` (Bestandteil des Android-SDK) eine Datei erstellt werden, die als Speicherabbild (engl. *disc image*) der Karte dient. Der folgende Befehl erstellt ein 10 MB großes Speicherabbild in der Datei `c:/eclipse/sdcard.img`.

```
c:\mksdcard -l EMULATOR_SD 10M c:/eclipse/sdcard.img
```

Der Emulator unterstützt Kartengrößen zwischen 8 MB und 128 GB. Das Speicherabbild kann nun mit geeigneten Tools (z.B. `mtools`) beschrieben werden, bevor der Emulator gestartet wird. Soll das Abbild

eine SD-Karte auf dem Emulator simulieren, so muss dieser schon beim Start darüber informiert werden.

```
emulator -sdcard c:/ein-verzeichnis/sdcard-10mb.iso
```

Verzeichnis /sdcard

Dieser Startparameter bewirkt, dass die SD-Karte als externes Dateisystem angesehen wird und allen Anwendungen unter dem Verzeichnisnamen `/sdcard` zur Verfügung steht. Auf dieses Verzeichnis sollte generell nie mit dem absoluten Verzeichnisnamen, sondern immer nur mittels `android.os.Environment.getExternalStorageDirectory` zugegriffen werden. Die Information darüber, ob ein externes Dateisystem verfügbar ist, erhält man über `Environment.getExternalStorageState`.

> Private Anwendungsdaten (Dateien, Datenbanken) sowie Ressourcen einer Anwendung können *nicht* von einer SD-Karte aus genutzt werden. Die Anwendung benötigt diese Daten in ihrem »privaten« Dateisystem auf dem Basis-Speichermedium.

10.1.2 Berechtigungen

Berechtigungen

Abgeleitet vom Berechtigungskonzept des Linux-Dateisystems haben wir es bei Android-Dateien und -Verzeichnissen mit den Rechte-Ebenen

- Anwendung (*user*) und
- Allgemein (*world*)

zu tun. Die aus Unix bekannte Ebene der Gruppenrechte ist hier synonym zu den Anwendungsrechten vergeben. Jede Anwendung wird durch ihren eindeutigen, vom System vergebenen Schlüssel identifiziert (z.B. »*app_1*«, »*app_2*«, »*system*«). Wird aus einer Anwendung heraus eine Datei oder ein Verzeichnis erzeugt, so erhält es standardmäßig nur Lese- und Schreibrechte für die Anwendung selbst. Wir werden gleich noch lernen, die Rechte individuell zu vergeben.

Privatsphäre einer Anwendung

Mit der Installation einer Anwendung auf dem Android-Gerät wird automatisch ein *privates Anwendungsverzeichnis*

```
/data/data/packagename.der.anwendung/
```

erstellt. In dieses Verzeichnis darf nur die Anwendung selbst Schreiboperationen ausführen. Unterhalb des privaten Anwendungsverzeichnisses werden z.B. alle Datenbanken der Anwendung abgelegt.

10.2 Dateizugriffe

Der Zugriff auf Dateien und Verzeichnisse kann entweder aus dem Programmcode heraus oder über die `adb`-Konsole erfolgen. Bei laufendem Emulator bzw. Android-Gerät wird mittels

Zugriff per Konsole

```
> adb shell
#
```

eine Konsole auf dem aktuellen Gerät geöffnet. Dort sind alle wesentlichen Linux-Kommandos zur Anzeige von Dateien und Verzeichnissen verfügbar.

Zugriff per Eclipse

Das Eclipse-Plug-in bietet eine eigene Ansicht, den »File Explorer«, zur Darstellung des Emulator-Dateisystems. Diese ist komfortabler zu erreichen und zu bedienen als die `adb`-Konsole.

Zugriff per Code

Der Zugriff auf das Dateisystem einer Anwendung gestaltet sich ähnlich einfach. Es ist Anwendungen nicht erlaubt, direkt lesend oder schreibend auf Dateien anderer Anwendungen zuzugreifen. Sollte dies gefordert sein, so muss ein Content Provider wie in Abschnitt 12.8.3 auf Seite 216 beschrieben implementiert werden.

Die Zugriffsmethoden auf das Dateisystem werden vom `android.content.Context` bereitgestellt. Diese sind in Abbildung 10-1 zusammengefasst.

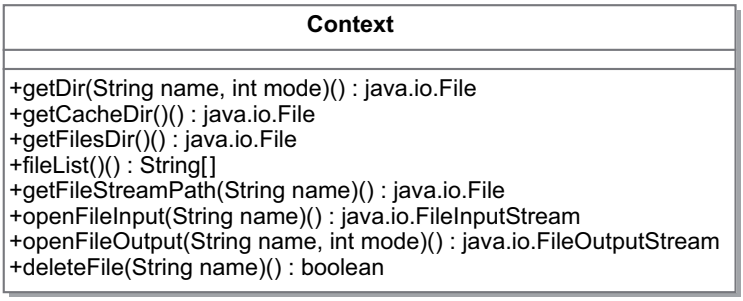


Abb. 10-1 *Dateisystem-Schnittstelle von Context*

Wir gruppieren diese Zugriffsmethoden in Verzeichnis- und Datei-Methoden.

10.2.1 Verzeichnisverwaltung

Mit Hilfe der Methoden `getDir`, `getCacheDir` und `getFilesDir` erhält man eine Referenz auf Verzeichnisse innerhalb des privaten Anwendungsverzeichnisses.

Cache-Ve

Im Cache-Verzeichnis sollten Dateien nur vorübergehend gespeichert werden, da sie jederzeit vom Betriebssystem gelöscht werden können.

Mittels `getDir` wird eine Referenz auf ein Unterverzeichnis des privaten Anwendungsverzeichnisses zurückgegeben. So liefert

```
File verzeichnis =
  context.getDir("routen", Context.MODE_PRIVATE);
```

eine Referenz auf das Verzeichnis

```
/data/data/packagename.der.anwendung/app_routen
```

zurück. Das Präfix `app_` wird vom Betriebssystem automatisch ergänzt, ist aber für die Programmierung nicht relevant. Falls kein Verzeichnis mit dem übergebenen Namen existiert, wird ein neues erstellt.

Organisation per Datenbank

Die Dateiverwaltungsklassen der Android-API reichen für viele Anwendungsfälle aus. Werden komplexere Anforderungen an die Organisation von Dateien gestellt, empfehlen wir den Einsatz einer Datenbank. Dort werden alle Metadaten einer Datei wie Name, Berechtigungen, Größe, Erstellungsdatum etc. abgelegt. Die Verwaltung der Dateien kann über Datenbankzugriffe flexibel gestaltet werden.

Berechtigungen

Der als zweiter Parameter an `getDir` übergebene `mode` sollte im Allgemeinen auf den Standardwert `Context.MODE_PRIVATE` gesetzt werden. Dadurch verdeutlicht man, dass es sich um ein privates Verzeichnis handelt. Alternativ dazu kann das Verzeichnis auch für andere Anwendungen lesbar (`Context.MODE_WORLD_READABLE`) oder schreibbar (`Context.MODE_WORLD_WRITABLE`) gemacht werden. Davon wird aber aus Sicherheitsgründen dringend abgeraten!

Ein Verzeichnis oder eine Datei kann auch über die Methoden der `java.io` Packages verwaltet werden. Wir haben dann aber *keinen* Einfluss auf die Berechtigungen. Wird die Datei unterhalb des Packages einer Anwendung erstellt, so ist sie nur für diese lesbar. Wird sie in einem allgemein schreibbaren Bereich (z.B. SD-Karte) erstellt, so ist sie allgemein lesbar und schreibbar.

> In `java.io` enthaltene Methoden zur Kontrolle der Berechtigungen, z.B. `File.setReadOnly`, werden von Android ignoriert.

getFilesDir

Die Methode `getFilesDir` liefert das Verzeichnis aller Dateien, die mittels `openFileOutput` erstellt wurden, zurück. Der Verzeichnisname (derzeit `/data/package-name/files`) wird von Android verwaltet.

SD-Karten

Alle von `Context` einer Anwendung bereitgestellten Methoden beziehen sich auf deren privates Anwendungsverzeichnis. Es ist dem Entwickler jedoch freigestellt, über die `java.io`-Schnittstellen auch in die anderen Bereiche des Dateisystems vorzudringen. Dies ist beispielsweise notwendig, wenn auf eine SD-Karte zugegriffen werden soll.

10.2.2 Dateiverwaltung

Für die Verwaltung von Dateien gelten die gleichen Regeln wie für die Verzeichnisverwaltung. Es gibt einen »privaten« Bereich, der von der Android-API unterstützt wird, und einen »allgemeinen« Bereich, der nur über die `java.io`-Schnittstellen erreicht werden kann. Wir wollen uns hier überwiegend dem Android-typischen Einsatz von Dateien, die an eine Anwendung gebunden sind, widmen.

Allen privaten Dateien ist gemein, dass sie nur über ihren Dateinamen identifiziert werden. Das Zielverzeichnis wird von Android ergänzt. Zu beachten ist daher, dass die Methoden `openFileInput`, `openFileOutput` und `deleteFile` *keine Verzeichnisangabe* im Dateinamen haben dürfen. Sie beziehen sich immer auf das private Dateiverzeichnis der Anwendung, das mittels `getFilesDir` ermittelt werden kann.

Der Codeausschnitt in Listing 10.1 demonstriert den Lebenszyklus einer privaten Datei.

Listing 10.1
Lebenszyklus einer Datei

```
FileOutputStream outputStream = null;
try {
  // Datei erzeugen...
  final String dateiName = "eineRoute.txt";
  outputStream = openFileOutput(dateiName, MODE_PRIVATE);
  String dateiInhalt = "Dies ist eine Routenbeschreibung.";
  outputStream.write(dateiInhalt.getBytes());
  outputStream.flush();

  // Datei erweitern...
  outputStream = openFileOutput(dateiName, MODE_APPEND);
  String dateiAnhang =
    "Dies ist die Fortsetzung der Routenbeschreibung.";
  outputStream.write(dateiAnhang.getBytes());
  outputStream.flush();

  // Verzeichnis auslesen...
  String[] privateDateien = fileList();
  int anzahlDateien =
    (privateDateien != null ? privateDateien.length : 0);
  for( int i = 0 ; i < anzahlDateien ; i++ ) {
    Log.i(TAG,privateDateien[i]);
  }
```

```
    // Datei auslesen...
    FileInputStream inputStream = openFileInput(dateiName);
    try {
      Log.i(TAG,"Inhalt der Datei:");
      Log.i(TAG,
          org.apache.commons.io.IOUtils.toString(inputStream));
    } finally {
      if( inputStream != null ) {
        inputStream.close();
      }
    }

    // Datei löschen...
    deleteFile(dateiName);

  } catch (IOException e) {
    Log.e(TAG,e.getMessage(),e);
  } finally {
    if ( outputStream != null ) {
      try {
        outputStream.close();
      } catch (IOException e) {
        Log.e(TAG,e.getMessage(),e);
      }
    }
  }
```

11 Iteration 4 – Datenbanken

Hinweis: Die 2. Auflage dieses Buchs erscheint unter dem Titel
»Android 2. Grundlagen und Programmierung«
Ende Mai 2010 im dpunkt.verlag, ISBN 978-3-89864-677-2

Wir lernen in diesem Kapitel, wie man über die Android-Programmierschnittstelle, trotz eingeschränkter Hardware-Ressourcen, effizient auf Datenbanken zugreift. Wir nutzen dabei das Datenbanksystem *SQLite*, das im Lieferumfang von Android enthalten ist.

Neben seiner Programmierschnittstelle bietet Android noch ein kommandozeilenorientiertes Programm für Datenbankzugriffe an, welches wir ebenfalls vorstellen werden.

11.1 Iterationsziel

Routen speichern

Wir wollen eine Datenzugriffsschicht für die Speicherung von Routen implementieren. Jede Route besteht aus einem Start- und einem Zielort sowie einem Schlüsselwert. Routen werden über eine Weboberfläche auf dem Staumelder-Server gepflegt. Die Datenbank dient also hier als Zwischenspeicher für Daten, die regelmäßig vom Staumelder-Server geladen werden.

Kommandozeile zur Kontrolle

Wir wollen die dabei eingesetzten SQL-Operationen auf der Kommandozeile auszuprobieren, bevor wir sie in Java implementieren.

11.2 Wozu Datenbanken?

Neue Anforderungen

Normalerweise dienen Datenbanken zur Speicherung und zum effizienten Zugriff auf große, strukturierte Datenmengen, die mehreren Anwendungen zeitgleich zur Verfügung gestellt werden. Diese Rahmenbedingungen sind auf einem Android-Gerät *nicht* vorhanden. In der Regel werden dort aus Platzgründen kleinere Datenmengen verwaltet. Auf die Inhalte einer Datenbank greift meist nur eine Anwendung zu.

Einsatzbereiche

Für die folgenden Aufgaben sind Datenbanken dennoch erforderlich:

Dauerhafte Datenhaltung Strukturierte Daten, die über die Lebensdauer einer Anwendung hinaus gespeichert werden müssen und auf die

die Anwendung häufig zugreift, werden in einer Datenbank vorgehalten.

Zwischenspeicher Mobile Anwendungen können via Internet-Protokoll Datensätze von externen Datenbankanwendungen anfordern. Wenn diese Daten während einer Anwendungssitzung mehrfach benötigt werden, sollten sie in einer Datenbank auf dem Android-Gerät zwischengespeichert werden. Das reduziert die Anzahl zeitraubender und fehleranfälliger Netzwerkverbindungen.

Es scheint also gerechtfertigt, dass ein Datenbanksystem zum Lieferumfang von Android gehört. Der folgende Abschnitt stellt dieses System vor.

11.3 Das Datenbanksystem SQLite

Kraftzwerg SQLite

Bei SQLite (`www.sqlite.org`) handelt es sich um ein kompaktes, für den Einsatz in mobilen Plattformen optimiertes, quelloffenes, relationales Datenbanksystem. Es unterstützt fast alle Funktionen seiner Verwandten auf komplexen Serversystemen (umfangreiche SQL-Syntax, Transaktionen etc.), benötigt aber selbst nur ein Minimum an Speicherplatz (175kB bis 250kB). Wir werden es daher auch als *leichtgewichtiges* Datenbanksystem bezeichnen.

Nur der Einstieg

Wir wollen in diesem Buch bewusst nur die für die simple Datenbankprogrammierung auf einem Android-Gerät erforderlichen Kenntnisse über SQLite vermitteln. Für tiefergehende Details verweisen wir gerne auf die sehr gute Online-Dokumentation unter [7].

SQLite grenzt sich durch die folgenden charakteristischen Merkmale von vergleichbaren Produkten ab:

Kein Server

Keine Installation, keine zentrale Konfiguration SQLite ist ein serverloses (»standalone«) Datenbanksystem. Es wird kein separater Serverprozess benötigt. Eine Datenbank wird für eine Anwendung definiert und steht dieser ohne explizite Startbefehle zur Verfügung.

Zugriffe aus mehreren Anwendungen erlaubt SQLite erlaubt die zeitgleiche Nutzung *einer* Datenbank durch *mehrere* Anwendungen. Dieses Merkmal haben die wenigsten leichtgewichtigen Datenbanksysteme.

Datenbankverzeichnis

Eine Datenbank = eine Datei Pro Datenbank wird *eine* Datei im Dateisystem von Android angelegt. Die Datenbanken einer Anwendung werden im Verzeichnis

```
/data/data/package.name/databases
```

abgelegt. Als `package.name` setzt man den Paketnamen der Anwendung ein. In unserem Fall wäre dies `de.androidbuch.staumelder`. Der Datenbankname muss innerhalb einer Anwendung eindeutig sein.

»Nimm, was Du brauchst.«

Dynamische Feldlängen SQLite nutzt bei Textfeldern immer die tatsächliche Länge des Feldes zum Speichern aus. Die definierte Feldlänge wird dabei ignoriert. Sei z.B. eine Spalte vom Typ `VARCHAR(5)` definiert, so erlaubt SQLite auch Felder der Länge 100, ohne den Attributwert beim Speichern zu kürzen. Andererseits wird auch für einen Wert der Länge 1 nur der Speicherplatz für ein Zeichen reserviert. Dieses Verhalten führt zu einer optimalen Nutzung des vorhandenen Speichers auf Kosten einer leicht ineffizienteren Datenhaltung. Da sich das System ohnehin nicht an Längenvorgaben hält, definieren wir in diesem Buch Textattribute als `TEXT` statt als `VARCHAR(xy)`.

Eine vollständige Liste der Merkmale von SQLite ist der Dokumentation unter [8] zu entnehmen. In den nächsten Abschnitten erzeugen wir eine SQLite-Datenbank und lernen, wie man auf sie zugreift.

11.4 Eine Datenbank erstellen

Bevor wir die erste Datenbank erstellen, werfen wir noch einen kurzen Blick auf die Theorie. Dazu gehören die Themen »Berechtigungen« und »Datensicherheit«. Anschließend stellen wir die Teile der Android-API vor, die wir zum Erzeugen einer Datenbank benötigen.

11.4.1 Berechtigungen

Ganz oder gar nicht

Die Lese- und Schreibberechtigungen für eine Datenbank ergeben sich aus den Berechtigungen ihrer Datendatei. Eine Anwendung hat also entweder Zugriff auf die komplette Datenbank oder gar keinen Zugriff. Individuelle Lese- und Schreibrechte pro Tabelle sind nicht vorgesehen. Es gibt auch keine Datenbank-Nutzerkonten (engl. *accounts*) oder vergleichbare Sicherheitsmaßnahmen.

Eigentumsfrage

Die Anwendung, die eine Datenbank *erzeugt*, bezeichnen wir als deren *Eigentümer*. Zugriffe auf eine Datenbank sind nicht ohne Einverständnis der Eigentümer-Anwendung erlaubt.

Zugriff nur für Eigentümer!

Da sich eine Datenbank im *privaten Anwendungsverzeichnis* des Eigentümers befindet (s. Abschnitt 10.1.2 auf Seite 156), ist sie nur für diese Anwendung erreichbar. Um die Daten anderen Anwendungen

zugänglich zu machen, muss der Eigentümer *Content Provider* bereitstellen. Doch das ist Thema eines eigenen Kapitels (12 ab Seite 189).

Fazit

Datenbankinhalte sind also vor unerlaubten Zugriffen durch andere Anwendungen geschützt. Es reicht aus, Lese- und Schreibrechte auf Datenbankebene anzubieten. Die komplexe Rechteverwaltung anderer Datenbanksysteme wird ja meist nur dann erforderlich, wenn mehrere Anwendungen unabhängig voneinander auf einer Datenbank arbeiten. Dieses Szenario ist bei Android-Anwendungen ausgeschlossen, da der Eigentümer Fremdanwendungen explizit Zugriff auf die Daten gewähren muss. Details dazu geben wir im Kapitel über Content Provider.

11.4.2 Schemaverwaltung

Datenbanken stehen unter der Kontrolle ihres Eigentümers. Daher ist dieser auch für die Erstellung und Pflege des Datenbankschemas verantwortlich. Doch wie erkennt die Eigentümer-Anwendung, dass sie für eine oder mehrere Datenbanken zuständig ist? Und wie stellt sie sicher, dass die Datenbanken bereits zu Anwendungsbeginn nutzbar sind?

Machen wir einen Schritt nach dem anderen. Zunächst überlegen wir uns, wie wir das Konstrukt »Datenbank« konkret im Programmcode abbilden. Anschließend definieren wir das Schema unserer Datenbank. Danach verbinden wir sie mit ihrem Eigentümer. Zum Schluss befassen wir uns mit Schemaänderungen.

Der Datenbank-Manager

Datenbank-Manager

Um eine Datenbank »greifbar« zu machen, definieren wir für sie eine eigene Verwaltungsklasse, die wir *Datenbank-Manager* nennen. Wir geben dem Datenbank-Manager als Präfix den Namen der Eigentümer-Anwendung, um ihren Wirkungsbereich zu verdeutlichen. In unserem Fall würden wir die Klasse `de.androidbuch.staumelder.StaumelderDatenbank` erstellen.

Name, Version, Eigentümer

Der Datenbank-Manager muss mindestens drei Parameter kennen: den Namen der Datenbank, die aktuelle Version des Datenbankschemas und den Anwendungskontext des Eigentümers. Die ersten beiden Werte können wir als Konstanten definieren. Der aktuelle Anwendungskontext muss bei Erzeugung des Datenbank-Managers mitgegeben werden.

Das Schema erstellen

Glücklicherweise müssen wir nicht bei jedem Zugriff auf den Datenbank-Manager prüfen, ob die Datenbank bereits existiert oder ob sich ihr Schema gegenüber dem letzten Aufruf geändert hat. Das

erledigen das Betriebssystem, der Anwendungskontext und die Klasse `android.database.sqlite.SQLiteOpenHelper` für uns. Der abstrakte `SQLiteOpenHelper` fordert von seinen Unterklassen, dass diese wissen, wie sie auf die oben genannten Ereignisse reagieren müssen. Konkret gibt er dafür zwei Methoden zur Ereignisbehandlung vor: `onCreate` und `onUpdate`.

SQLiteOpenHelper

Der Datenbank-Manager muss sich um genau diese Ereignisse kümmern. Wir leiten ihn also von `SQLiteOpenHelper` ab und erhalten die in Listing 11.1 abgebildete Rumpf-Implementierung.

Listing 11.1 *Skizze Datenbank-Manager*

```
public class StaumelderDatenbank
                 extends SQLiteOpenHelper {
  private static
      final String DATENBANK_NAME = "staumelder.db";
  private static
      final int DATENBANK_VERSION = 1;

  public StaumelderDatenbank(Context context) {
    super(
        context,
        DATENBANK_NAME,
        null,
        DATENBANK_VERSION);
  }

  public void onCreate(SQLiteDatabase db) {
    db.execSQL("CREATE TABLE ...");
  }

  public void onUpgrade(SQLiteDatabase db,
      int oldVersion,
      int newVersion) {
    db.execSQL("DROP TABLE ...");
    onCreate(db);
  }
}
```

Die SQL-Befehle zum Erzeugen bzw. Löschen der Tabellen werden über die Methode `execSQL` abgeschickt. Mit dieser werden wir uns in Abschnitt 11.5.4 auf Seite 175 befassen.

Wir wissen jetzt, wie wir ein Datenbankschema definieren. Doch wie greifen wir auf die Datenbank zu? Dazu liefert `SQLiteOpenHelper` eine Referenz auf die von ihr verwaltete Datenbank zurück. Wir haben die Wahl, ob wir mittels `getReadableDatabase` nur lesend oder mittels `getWriteableDatabase` auch schreibend auf die Daten zugreifen wollen.

Datenbankreferenz

Die Referenz auf die Datenbank wird innerhalb von `SQLiteOpenHelper` zwischengespeichert, so dass die Methoden ohne Geschwindigkeitsverlust für jeden Datenbankzugriff verwendet werden können.

Erst prüfen, dann verbinden

Bei jeder Anforderung einer Datenbankreferenz prüft Android, ob die Datenbank schon existiert oder eine neue Version des Schemas vorliegt. Bei Bedarf werden die Ereignisse zur Erzeugung oder Aktualisierung der Datenbank ausgelöst. Die Datenbank wird also erst erstellt, wenn auf sie zugegriffen wird.

Mit dem Eigentümer bekannt machen

Anwendungsstart = Datenbankstart

Der Datenbank-Manager sollte so früh wie möglich dem Eigentümer der Datenbank bekannt gemacht werden. Idealerweise findet daher die Erstellung der Manager-Instanz und der erste Zugriff über `getReadableDatabase` bei der Erzeugung der ersten Activity bzw. des ersten Service der Anwendung statt. Auf diese Weise erkennt man Fehler in der Datenbankdefinition frühzeitig.

Das Schema ändern

Zum Schluss bleibt noch die Frage, wie der Datenbank-Manager Änderungen am Schema erkennt. Er orientiert sich an der Versionsnummer, die bei der Initialisierung des `SQLiteOpenHelper` mitgegeben wird.

Versionsnummer

Wollen wir also das Datenbankschema ändern, so müssen wir die Versionsnummer erhöhen und die `onUpdate`-Methode im Datenbank-Manager anpassen. Beim nächsten Zugriff auf die Datenbank wird das Schema aktualisiert.

Fazit

Fassen wir diesen Abschnitt kurz zusammen. Wir haben einen Datenbank-Manager erstellt, der für die Erzeugung und Aktualisierung des Datenbankschemas zuständig ist. Er liefert eine Verbindung zur Datenbank, mit deren Hilfe wir auf dieser Anfragen und Operationen ausführen können.

11.5 Datenzugriff programmieren

Der grobe Überblick

In diesem Abschnitt zeigen wir, wie SQL-Anfragen über die Programmierschnittstelle von Android ausgeführt werden. SQL-Kenntnisse sind zum Verständnis des Kapitels hilfreich.

Routentabelle

Wir wollen in dieser Iteration Routendaten in einer Datenbank speichern. Wir definieren die Datenbanktabelle `routen` wie in Listing 11.2 beschrieben.

Konvention

Tabellen- und Feldnamen schreiben wir durchgängig klein (z.B. `routen`), SQL-Befehle durchgängig groß (z.B. `SELECT`).

```
CREATE TABLE routen (
  _id INTEGER PRIMARY KEY AUTOINCREMENT,
  start TEXT NOT NULL,
  ziel TEXT NOT NULL
);
```

Listing 11.2 *Schema der Routentabelle*

11.5.1 SQLiteDatabase – Verbindung zur Datenbank

Die Klasse `android.database.sqlite.SQLiteDatabase` stellt Methoden für Zugriffe auf eine SQLite-Datenbank bereit. Sie kann als Zeiger auf eine aktive Datenbankverbindung verstanden werden.

Konvention

Wir legen für dieses Kapitel fest, dass die Variable `db` immer vom Typ `SQLiteDatabase` ist.

Tabelle 11-1 fasst die wichtigsten Zugriffsoperationen zusammen. Wir verwenden dabei die Bezeichnung *Datensatz* für den Inhalt einer Tabellenzeile. Jeder Datensatz verfügt über einen in der Tabelle eindeutigen *Schlüsselwert*.

Methode	Beschreibung
query	Führt eine SQL-Anfrage aus. Als Parameter werden die Bestandteile der Anfrage übergeben.
rawQuery	Führt eine SQL-Anfrage aus. Als Parameter wird der SQL-Befehl als String übergeben.
insert	Fügt einen neuen Datensatz in eine Tabelle ein.
update	Ändert Attribute eines vorhandenen Datensatzes.
delete	Löscht einen Datensatz anhand seines Schlüsselwertes.
execSQL	Führt ein SQL-Änderungskommando aus. Als Parameter wird der SQL-Befehl als String übergeben.

Tab. 11-1 *Zugriff mit SQLiteDatabase*

Anfragen und Änderungen

Mit diesem Befehlssatz lassen sich alle Datenbankoperationen durchführen. Wir unterscheiden dabei zwischen Datenbankanfragen und Änderungsoperationen. Eine *Datenbankanfrage* liefert durch Suchkriterien eingeschränkte Inhalte von Datenbanktabellen zurück. Eine *Änderungsoperation* kann hingegen die Inhalte der Datenbank verändern.

Schließen nicht vergessen

Bevor wir uns Anfragen und Änderungsoperationen im Detail anschauen, wollen wir noch ein wichtiges Thema ansprechen. Eine Datenbankverbindung sollte *geschlossen* werden, wenn sie nicht mehr gebraucht wird (`SQLiteDatabase::close`). Android sorgt dafür, dass bei Beendigung einer Anwendung alle Verbindungen zu Datenbanken, deren Eigentümer sie ist, ordnungsgemäß beendet werden. Wir empfehlen den expliziten Aufruf von `close` nur dann, wenn eine Datenbank ausschließlich in einem klar definierten Teil der Anwendung, z.B. nur von *einer* Activity, genutzt wird.

11.5.2 Datenbankanfragen

Anfragestruktur

In diesem Abschnitt lernen wir, Datenbankanfragen in Android-Anwendungen einzusetzen. Über die Android-API werden Anfragen immer nach dem Schema

```
Cursor ergebnis = db.query(anfrageparameter);
```

Cursor

gestellt. Das Ergebnis einer Anfrage ist also immer ein sogenannter *Cursor*, der vorerst einfach nur als Zeiger auf die Ergebnismenge angesehen werden sollte. Dem Ergebnistyp Cursor wenden wir uns ab Seite 173 genauer zu.

Einfache Anfragen

Als »einfach« bezeichen wir eine Anfrage, die auf nur eine Tabelle zugreift. Wenn wir beispielsweise alle Routen finden wollen, deren Start Bielefeld ist, würden wir dies in SQL so formulieren:

```
SELECT _id, start, ziel
FROM routen
WHERE start = 'Bielefeld';
```

Ergebnisstruktur klar definieren

Die Schreibweise `SELECT * FROM...` sollte man vermeiden. Sie führt leicht dazu, dass mehr Attribute als benötigt übertragen werden. Dies wirkt sich negativ auf Geschwindigkeit und Speicherverbrauch der Anwendung aus.

Außerdem wird so die Struktur der Ergebnisdatensätze für den Anfragenden nicht exakt beschrieben. Es ist nicht ersichtlich, unter welchem Namen und in welcher Position eine Spalte im Ergebnisdatensatz

auftaucht. Diese Informationen sind zur Auswertung von Anfrageergebnissen wichtig.

`SQLiteDatabase` bietet zwei Methoden zur Formulierung von Anfragen an: `query` und `rawQuery`. Die Listings 11.3 und 11.4 zeigen, wie die Suche nach Bielefeld-Routen mittels beider Methoden formuliert wird.

Listing 11.3 *SQLiteDatabase query*

```
Cursor bielefeldRouten = db.query(
  false, // distinct?
  "routen",
  new String[] { // SELECT
    "_id",
    "start",
    "ziel"
    },
  "start = ?", // WHERE-Bedingung
  new String[] { // Parameter für WHERE
       "Bielefeld"
  },
  null,      // GROUP BY
  null,      // HAVING
  null, // ORDER BY
  null      // LIMIT
);
```

Listing 11.4 *SQLiteDatabase rawQuery*

```
Cursor bielefeldRouten = db.rawQuery(
  "SELECT _id,start,ziel "+
  "FROM routen "+
  "WHERE start = ? ",
  new String[] {
       "Bielefeld"
  }
);
```

query bietet die strukturiertere Schnittstelle, die kaum Kenntnisse von SQL voraussetzt. Sie mag auf den ersten Blick etwas umständlich wirken, ist aber in der Praxis häufig einfacher zu nutzen als die `rawQuery`. *Strukturierte query*

rawQuery erinnert an die Syntax von JDBC-Anfragen und ist gut für komplexere Anfragen geeignet. Nach unseren Messungen wird die `rawQuery` etwas schneller verarbeitet als eine vergleichbare `query`. Die Unterschiede sind aber nicht so groß, als dass wir deswegen eine Empfehlung für eine der beiden Anfragemethoden aussprechen würden. *Flexible rawQuery*

Achtung!

Die an eine `rawQuery` übergebene SQL-Anweisung darf *kein abschließendes Semikolon* enthalten.

Neben den beiden Anfragemethoden bietet die Android-API noch eine Hilfsklasse an: den `android.database.sqlite.SQLiteQueryBuilder`. Mit seiner Hilfe kann man eine Datenbankanfrage Schritt für Schritt »zusammenbauen« (Listing 11.5).

Listing 11.5 *SQLiteQueryBuilder*

```
Map<String,String> aliasZuordnung =
    new HashMap<String, String>();
aliasZuordnung.put("_id", "routen._id");
aliasZuordnung.put("start", "routen.start");
aliasZuordnung.put("ziel", "routen.ziel");

SQLiteQueryBuilder qb = new SQLiteQueryBuilder();

qb.setTables("routen");
qb.setProjectionMap(aliasZuordnung);
qb.appendWhere("start = ?");
Cursor bielefeldRouten = qb.query(
    db,
    new String[] {
      "_id",
      "start",
       "ziel"
    },
    null, // weiteres WHERE
    new String[] {
        "Bielefeld"
    },
    null,      // GROUP BY
    null,      // HAVING
    null,      // ORDER BY
    null      // LIMIT
);
```

Die Verwendung eines `SQLiteQueryBuilder` ist besonders dann sinnvoll, wenn der Aufbau der Anfrage von externen Faktoren beeinflusst wird und nicht ohne Fallunterscheidungen durchgeführt werden kann. Bei der Erstellung von Content Providern in Kapitel 12 wird uns dieses Verfahren wieder begegnen. Dort werden wir die Elemente des `SQLiteQueryBuilder` im Detail beschreiben.

Wie geht es weiter?

Da die Erstellung einer `rawQuery` SQL-erfahrenen Entwicklern keine Probleme bereiten dürfte, wollen wir die Anfragen mit Hilfe der `query`-Methode formulieren.

Sortieren

Wir wollen die Suchergebnisse nach Zielort sortiert ausgeben. In Tabelle 11-2 sind die Anfragen in SQL und als `query` formuliert.

Tab. 11-2 *Sortieren der Ergebnisse*

SQL	SQLiteDatabase.query

`SELECT _id, start, ziel`
`FROM routen`
`WHERE start = 'Bielefeld'`
`ORDER BY ziel ASC;`
	`db.query(`
	`  false, "routen",`
	`  new String[] {`
	`  "_id",`
	`  "start",`
	`  "ziel"`
	`  },`
	`  "start = ?",`
	`  new String[] {`
	`  "Bielefeld"`
	`  },`
	`  null, null,`
	`  "ziel ASC",`
	`  null`
	`  );`

Joins

Mehrere Tabellen

Wenn sich eine Suchanfrage über mehrere Tabellen erstreckt, benötigen wir *Joins*. Wollen wir beispielsweise alle Staumeldungen zusammen mit ihren Routeninformationen auflisten, müssen wir sowohl auf die Tabelle `routen` als auch auf die in Listing 11.6 definierte Tabelle `staus` zugreifen. Das Bindeglied zwischen beiden Tabellen stellt der Routenschlüssel (`routen._id` bzw. `staus.route_id`) dar.

Listing 11.6 *Schema der Stau-Tabelle*

```
CREATE TABLE staus (
  _id INTEGER PRIMARY KEY AUTOINCREMENT,
  route_id INTEGER NOT NULL,
  ursache TEXT,
  anfang TEXT NOT NULL,
  ende TEXT NOT NULL,
  laenge INTEGER,
  letzteaenderung INTEGER
);
```

Tabelle 11-3 zeigt, wie Joins in Android formuliert werden können.

Tab. 11-3
Datenbank-Joins

SQL	SQLiteDatabase.query

`SELECT`
`  routen._id, routen.start,`
`  routen.ziel, staus._id,`
`  staus.anfang,`
`  staus.laenge`
`FROM`
`  routen INNER JOIN staus`
`  ON`
`  routen._id = staus.route_id`
`ORDER BY staus.laenge DESC;`
	`db.query(`
	`  false,`
	`  "routen INNER JOIN staus`
	`  ON`
	`  routen._id =`
	`  staus.route_id",`
	`  new String[] {`
	`  "routen._id",`
	`  "routen.start",`
	`  "routen.ziel",`
	`  "staus._id",`
	`  "staus.anfang",`
	`  "staus.laenge"`
	`  },`
	`  null, null, null, null,`
	`  "staus.laenge DESC",`
	`  null`
	`  );`

Nach dem gleichen Prinzip werden auch *outer joins* gebildet. Die Syntax ist in der Dokumentation zu SQLite beschrieben.

Aggregationen

Gesucht sei die Anzahl der Routen pro Startort. Es sollen nur Startorte angezeigt werden, von denen mindestens zwei Routen starten.

GROUP BY... HAVING...

Zur Lösung des Problems verwendet man den `GROUP BY`-Befehl gefolgt von einem `HAVING`. Die Umsetzung der Anfrage als `query` ist in Tabelle 11-4 beschrieben.

Optimierungshinweis

Falls maximal *ein* Wert als Ergebnis der Aggregation erwartet wird, z.B. bei `SELECT count(*) FROM...`, kann man auch ein *Prepared Statement* verwenden, um die Anfrage zu beschleunigen (s. Seite 177).

Begrenzung der Ergebnismenge

LIMIT

Mit Hilfe der `LIMIT`-Anweisung kann die maximale Anzahl der Ergebnisse eingeschränkt werden. Die in Tabelle 11-5 formulierten Anfragen begrenzen die Ergebnismenge auf zehn Datensätze.

SQL	SQLiteDatabase.query

`SELECT start, count(*)`
`FROM routen`
`GROUP BY start`
`HAVING count(*) > 1;`
	`db.query(`
	`  false, "routen",`
	`  new String[] {`
	`  "start",`
	`  "count(*)"`
	`  },`
	`  null,null,`
	`  "start",`
	`  "count(*) > 1",`
	`  null,null`
	`  );`

Tab. 11-4
GROUP BY und count(*)

SQL	SQLiteDatabase.query

`SELECT _id, start, ziel`
`FROM routen`
`LIMIT 10;`
	`db.query(`
	`  false, "routen",`
	`  new String[] {`
	`  "_id",`
	`  "start",`
	`  "ziel"`
	`  },`
	`  null,null,`
	`  null, null,`
	`  null,`
	`  "10"`
	`  );`

Tab. 11-5
Ergebnismenge begrenzen

Fazit

Mit der `query`-Methode der Klasse `SQLiteDatabase` lassen sich alle Datenbankanfragen an die SQLite-Datenbank von Android programmieren. Wir wollen nun einen Blick auf den Ergebnistyp aller Anfragemethoden werfen: den `android.database.Cursor`.

11.5.3 Ergebnistyp Cursor

Cursor als Zeiger

Wann immer wir auf das Ergebnis einer Datenbankanfrage zugreifen wollen, werden wir einen `Cursor` nutzen. Durch einen Cursor wird nur der Zeiger auf einen Ergebnisdatensatz und nicht die komplette Datenmenge übertragen.

Anzahl Ergebnisse

Die Anzahl der Ergebnisdatensätze einer Anfrage ist dem Cursor bekannt und kann über seine Methode `getCount` herausgefunden werden.

Mit einem `Cursor` kann man

- innerhalb der Ergebnismenge vor und zurück navigieren,
- die von einem `SELECT`-Befehl angefragten Ergebnisdatensätze auslesen,
- die Anzahl der Ergebnisdatensätze ermitteln,
- große Ergebnismengen effizient laden.

Cursor-Navigation

Cursor als Iterator

Nach einer erfolgreichen Datenbankanfrage wird der Cursor *vor* den ersten Ergebnisdatensatz positioniert. Von dort aus kann man durch Aufruf der `moveToNext`-Methode des Cursors über die Ergebnismenge iterieren.

Beliebig positionierbar

Ein Cursor kann aber auch beliebig in der Ergebnismenge positioniert werden. So ist auch die Navigation von hinten nach vorne durchaus möglich. Die dazu angebotenen Methoden sind im JavaDoc zum Interface `Cursor` beschrieben.

Ergebniswerte auslesen

Für jeden Cursor kann der Wert einer Spalte des Ergebnisdatensatzes der aktuellen Cursorposition ausgelesen werden. Es müssen dazu die Spaltennummer (beginnend bei 0) und der Datentyp des Spaltenattributs bekannt sein. Bei letzterem ist SQLite sehr flexibel und wandelt den Wert der Spalte, wenn möglich, in den geforderten Datentyp um. Ein kurzes Beispiel verdeutlicht das Auslesen der Ergebniswerte.

```
Cursor cAlleRouten = db.query(
  false, "routen",
  new String[] {
    "_id",
    "start",
    "ziel"
    },
  null,null,null,null,null,null
  );

while( cAlleRouten.moveToNext() ) {
  cAlleRouten.getLong(0);   // _id
  cAlleRouten.getString(1); // start
}
```

Spaltennummer ermitteln

Sollte die Spaltennummer nicht bekannt sein, so kann man diese über die Cursor-Methode `getColumnIndexOrThrow(String columnName)` herausfinden. Dieser Weg sollte aber aus Zeitgründen nur dann gewählt werden, wenn die Spaltennummer anders nicht ermittelt werden kann.

Umgang mit großen Datenmengen

windowing

Wenn nicht anders definiert, wird das Java-Interface `android.database.Cursor` als `android.database.SQLiteCuror` implementiert. Diese Implementierung ist dahingehend optimiert, dass sie große Ergebnismengen in Teilmengen, sogenannten Datenfenstern, Schritt für Schritt einliest. Dieses Vorgehen wird auch als *windowing* bezeichnet.

Vorsicht bei `getCount`

Ein Aufruf von `Cursor::getCount` zur Abfrage der Größe der Ergebnismenge würde diese Optimierung jedoch ad absurdum führen. Daher sollte man darauf nur zurückgreifen, wenn man kleine Ergebnismengen erwartet. Besser wäre es, zur Ermittlung der Größe der Ergebnismenge ein *Prepared Statement* (siehe Abschnitt 11.5.4) zu verwenden.

Cursor schließen!

Um die Anzahl aktiver Datenbankverbindungen so gering wie nötig zu halten, sollte jeder Cursor nach Benutzung geschlossen werden. Dies geschieht über seine Methode `close`, die am besten innerhalb eines `finally` Blocks aufgerufen wird.

Managing Cursor

Eine Sonderrolle spielt der sogenannte *Managing Cursor* einer `Activity`. Die `Activity` sorgt dafür, dass ihr Managing Cursor korrekt geschlossen und bei Bedarf wieder geöffnet wird. Nach dem Öffnen wird der Cursor automatisch auf den aktuellen Stand gebracht. Dies geschieht durch erneute Ausführung der mit dem Cursor verbundenen Datenbankanfrage. Mit Hilfe der Methode `startManagingCursor` wird der Activity ein Cursor zur Verwaltung übergeben.

Cursor oder Datentyp

Wird *ein* konkreter Datensatz gesucht, z.B. Suche anhand des Primärschlüssels, und sollen dessen Attribute weitgehend auf der Oberfläche angezeigt oder von mehreren Komponenten genutzt werden, so empfehlen wir die Erzeugung eigener Datenobjekte mit Hilfe des Cursors. Datenobjekte erhöhen die Lesbarkeit und Wartbarkeit des Codes und sind nach unseren Messungen *in diesen Fällen* nicht langsamer als Direktzugriffe über den Cursor.

11.5.4 Änderungsoperationen

Freie Wahl

In Tabelle 11-1 auf Seite 167 haben wir gezeigt, dass auch Änderungsoperationen wie `INSERT`, `UPDATE` und `DELETE` zum Portfolio von `SQLiteDatabase` gehören. Diese wollen wir zunächst vorstellen.

Danach wenden wir uns einer Alternative zu, die auch schon in JDBC bekannt ist: vorkompilierten Änderungsanweisungen, sogenannten »Prepared Statements«.

Änderungsoperationen von SQLiteDatabase

Rückgabewerte

`SQLiteDatabase` bietet für alle SQL-Änderungsoperationen entsprechende Methoden an. `insert` (Tabelle 11-6, Seite 176) liefert im Erfolgsfall den Schlüsselwert des neu angelegten Datensatzes zurück. Die Methoden `update` (Tabelle 11-7) und `delete` (Tabelle 11-8) liefern die Anzahl der geänderten Datensätze zurück.

Optimierungshinweis!

Wir empfehlen, diese Methoden nur dann einzusetzen, wenn die Änderungsoperation *selten* aufgerufen wird. Für alle anderen Fälle lohnt sich der Mehraufwand für das Programmieren eines Prepared Statements.

Tab. 11-6
Datensatz einfügen

SQL	SQLiteDatabase

`INSERT INTO routen`
`  (start, ziel)`
`VALUES`
`  ('Bochum','Bielefeld');`
	`ContentValues werte =`
	`  new ContentValues();`
	`werte.put(`
	`  "start", "Bochum");`
	`werte.put(`
	`  "ziel", "Bielefeld");`
	`db.insert(`
	`  "routen", null, werte);`

Tab. 11-7
Datensatz ändern

SQL	SQLiteDatabase

`UPDATE routen`
`SET start = 'Berlin'`
`WHERE _id = 123;`
	`ContentValues werte =`
	`  new ContentValues();`
	`werte.put(`
	`  "start", "Berlin");`
	`db.update(`
	`  "routen", werte,`
	`  "_id=?",`
	`  new String[] {"123"});`

SQL	SQLiteDatabase

`DELETE FROM routen`
`WHERE _id = 123;`
	`db.delete(`
	`  "routen",`
	`  "_id=?",`
	`  new String[] {"123"});`

Tab. 11-8
Datensatz löschen

Prepared Statements

Spürbar effizienter gestalten sich Datenänderungen durch den Einsatz von Prepared Statements, die in der Android-API als `android.database.sqlite.SQLiteStatement` formuliert und über die Methode `compileStatement` von `SQLiteDatabase` erzeugt werden. Das folgende Beispiel soll das verdeutlichen.

```
SQLiteStatement stmtInsert =
  db.compileStatement(
    "insert into routen "+
    "(start,ziel) "+
    "values (?,?)"
  );
stmtInsert.bindString(1,"Bochum");
stmtInsert.bindString(2,"Bielefeld");
long id = stmtInsert.executeInsert();
```

Hier haben wir die Einfügeoperation aus Tabelle 11-6 als Prepared Statement umgesetzt.

Deutlich schneller!

Nach unseren Messungen ließen sich auf diese Weise Einfügeoperationen mit bis zu vierfacher Geschwindigkeit gegenüber den `insert`-Methodenaufrufen von `SQLiteDatabase` ausführen. Die Programmierung ist zwar ein wenig umständlicher, doch der Zweck heiligt hier die Mittel.

Auch für Datenbankanfragen?

Für Datenbankanfragen lassen sich diese optimierten Anweisungen nur bedingt einsetzen. Die Ausführung eines `SQLiteStatement` liefert nämlich immer nur *einen* numerischen `SQLiteStatement::simpleQueryForLong` oder Text-Wert `SQLiteStatement::simpleQueryForString` zurück. Häufig aufgerufene Aggregationen, z.B. `SELECT count(*) FROM ...`, kann man aber durchaus als Prepared Statements formulieren.

Transaktionen

Transaktionen

Eine Änderungsoperation verändert den Datenbankinhalt. Man muss sichergestellen, dass der Geschäftsprozess, dessen Teil die Änderungsoperation ist, korrekt ausgeführt wurde, bevor die Änderungen in der Datenbank gültig werden. Daher müssen Änderungsoperationen innerhalb von *Transaktionen* stattfinden. Wikipedia definiert eine Transaktion als *»... eine feste Folge von Operationen, die als eine logische Einheit betrachtet werden. Insbesondere wird für Transaktionen gefordert, dass sie entweder vollständig oder überhaupt nicht ausgeführt werden (Atomizität).«*

Transaktionsende

Eine Transaktion wird im Erfolgsfall *abgeschlossen* (engl. *commit*). Im Fehlerfall wird die Datenbank wieder auf den Zustand vor Transaktionsbeginn zurückgesetzt (engl. *rollback*).

Implementierung

SQLite unterstützt geschachtelte Transaktionen. `SQLiteDatabase` stellt die zur Definition einer Transaktion erforderlichen Methoden `beginTransaction` und `endTransaction` bereit.

commit und rollback

Der erfolgreiche Abschluss einer Transaktion wird vor deren Ende durch Aufruf von `setTransactionSuccessful` signalisiert. Ist dieser Aufruf nicht bei Erreichen von `endTransaction` erfolgt, so wird ein *rollback* der Transaktion und aller ihrer geschachtelten Unter-Transaktionen ausgeführt. Anderenfalls wird ein *commit* ausgeführt und die Änderungen werden festgeschrieben. Explizite `commit`- und `rollback`-Befehle bietet Android nicht an.

In Listing 11.7 wird der Aufbau einer Transaktion deutlich.

Listing 11.7 *Eine Datenbank-Transaktion*

```
db.beginTransaction();
try {
 ...
 aendereDatenbestand();
 ...
 db.setTransactionSuccessful(); // commit
} finally {
 db.endTransaction(); // Transaktion immer beenden
}
```

11.6 Datenzugriff per Kommandozeile

Oft möchte man Ergebnisse von Datenbankoperationen auch außerhalb des Java-Quellcodes überprüfen und nachvollziehen. Bei komplexeren SQL-Anfragen ist es vor der Programmierung ratsam, die Ausführungszeit verschiedener Formulierungsalternativen zu vergleichen.

sqlite3: die Kommandozeile

Mit dem Kommandozeilen-Programm `sqlite3` kann auf SQLite-Datenbanken zugegriffen werden. Dieses Programm ist im Lieferumfang von Android enthalten. Der Zugriff auf `sqlite3` ist nur über eine Android-Shell möglich. Daher muss man über den Android-Debugger `adb` zunächst eine Shell-Konsole auf dem Gerät oder Emulator öffnen. Von dort aus kann man `sqlite3` starten.

sqlite3 starten

Nehmen wir an, dass der Emulator als Gerät `emulator-5554` gestartet wurde[1]. Dann öffnet man die Datenbank `staumelder.db`, deren Eigentümer-Anwendung im Package `de.androidbuch.staumelder` definiert ist, wie folgt:

```
$ adb -s emulator-5554 shell
# cd /data/data/de.androidbuch.staumelder/databases
# sqlite3 staumelder.db
sqlite3 staumelder.db
SQLite version 3.5.9
Enter ".help" for instructions
sqlite>
```

Dem `sqlite3`-Aufruf gibt man also den Namen der Datenbankdatei mit.

Die `sqlite3`-Oberfläche ist unspektakulär. Man kann semikolonterminierte SQL-Befehle gegen eine geöffnete Datenbank ausführen. Darüber hinaus bietet `sqlite3` noch einige Kommandos, deren für den Einstieg wichtigste wir in Tabelle 11-9 zusammengefasst haben.

***Tab. 11-9** Wichtige sqlite3-Befehle*

Befehl	Anzeige
.help	Liste aller Befehle
.tables	Liste aller Tabellennamen
.dump ?table?	Tabelleninhalt als INSERT-Anweisungen
.schema ?table?	Schema der Tabelle
.schema	Schema der Datenbank
.exit	Programmende

Die Liste aller Tabellennamen lässt sich übrigens auch mit der Anfrage `SELECT name FROM sqlite_master WHERE type = 'table';` herausfinden. Die Tabelle `sqlite_master` speichert wichtige Metainformationen zum Datenbankschema und ist zu Analysezwecken oder bei der Fehlersuche hilfreich.

[1] Der Parameter -s ist nur notwendig, wenn mehrere Emulatoren oder Geräte parallel laufen.

EXPLAIN... Manchmal möchte man vor der Ausführung einer Datenbankanfrage wissen, welche Indizes genutzt und wie Joins verarbeitet werden. Um diese Information zu erhalten, kann der `EXPLAIN QUERY PLAN`-Befehl vor das zu testende `SELECT` gesetzt werden.

```
sqlite> explain query plan select *
from stau
where _id = 7;
explain query plan select * from stau where _id = 7;

0|0|TABLE stau USING PRIMARY KEY
```

Mehr Informationen... Damit enden unsere Ausführungen zum Umgang mit dem Kommandozeilen-Werkzeug `sqlite3`. An weiteren Details interessierte Leser verweisen wir auf die Online-Dokumentation [4].

11.7 Alternative zu SQLite

SQLite ist nur *ein* mögliches Datenbanksystem für Android. Wir haben uns `db4objects` [5] etwas näher angeschaut. Es handelt sich dabei um ein Datenbanksystem, dessen Programmierschnittstelle an objektrelationale Frameworks wie Hibernate oder Toplink erinnert.

Dafür zahlt man aber einen höheren Preis im Bereich Geschwindigkeit. SQLite war bei unseren Vergleichstest in allen Bereichen signifikant schneller als `db4objects`. Dennoch ist der »Verlierer« eine elegante Alternative zum sehr technisch orientierten SQLite, wenn die Datenmengen gering und die zu speichernden Strukturen komplex sind.

11.8 Implementierung

Die Routenliste Wenden wir nun das Gelernte auf unseren Staumelder an. Wir wollen Routendaten, die regelmäßig vom Staumelder-Server geladen werden, auf dem Gerät speichern. Diese Daten wollen wir auf einer Bildschirmseite in Listenform anzeigen.

Achtung »Architektur«! Als Erstes stellen wir einen Architekturentwurf für Datenbankzugriffsschichten vor. Wir haben in unseren bisherigen Android-Projekten gute Erfahrungen damit gemacht.

Auf dieser Basis setzen wir dann das Iterationsziel in die Praxis um.

11.8.1 Ein Architekturvorschlag

Schön langsam...

An dieser Stelle wollen wir uns kurz Gedanken über die Architektur einer Datenbankzugriffsschicht für Android-Anwendungen machen. In unserem ersten Android-Projekt haben wir die jahrelang praktizierten Konzepte aus der Java-EE-Welt übernommen und umfangreiche Data Access Objects (DAO), Speicher-Schnittstellen (*Repositories*) und vieles mehr entwickelt. Leider mussten wir feststellen:

- Die Anwendung ist zu groß und zu langsam.
- Änderungen an der Datenbank sind wegen fehlender Frameworks (Spring, Hibernate) aufwendig.
- Wir bekommen die Datenbankschicht nicht von der Oberfläche getrennt, ohne massive Laufzeiteinbußen in Kauf zu nehmen.

Keine Schichtentrennung

Beim nächsten Projekt sind wir anders an das Thema Datenbankzugriff herangegangen. Wir haben uns damit abgefunden, dass eine »reine« Schichtentrennung nicht möglich ist, ohne auf Cursors zu verzichten. Der Cursor ist für den Umgang mit großen Datenmengen optimiert. Also mussten wir ihn auch bis an die Activity »hochreichen«, die mit seiner Unterstützung die Anzeige der Daten auf dem Bildschirm optimal gestalten konnte.

Keine Datenobjekte

Wenn wir einen Cursor für die Übertragung von Ergebnisdatensätzen verwenden, brauchen wir normalerweise auch keine Datenobjekte mehr. Daher konnten wir uns auch die Definition von Datentransfer-Objektklassen sparen.

Interface beschreibt Tabelle.

Wir benötigen für viele `SQLiteDatabase`-Methoden die Angabe von Tabellenspalten. Daher haben wir sogenannte *Schema-Interfaces* für Tabellen eingeführt. Ein Schema-Interface ist einer Datenbanktabelle zugeordnet und enthält Informationen über Spaltennamen, Prepared Statements und den SQL-Befehl zur Erzeugung der Tabelle. Listing 11.8 zeigt das Schema-Interface für unsere Tabelle `routen`.

Listing 11.8
Schema-Interface

```
public interface RoutenTbl {

  String ID = "_id";
  String START = "start";
  String ZIEL = "ziel";

  String TABLE_NAME = "routen";

  String SQL_CREATE = "CREATE TABLE routen ("
      + "_id INTEGER PRIMARY KEY AUTOINCREMENT,"
      + "start TEXT NOT NULL,"
      + "ziel TEXT NOT NULL"
      + ");";
```

```
  String STMT_FULL_INSERT = "INSERT INTO routen "
      + "(start,ziel) VALUES (?,?)";
}
```

Der *Primärschlüssel* der Tabelle wird durch den Ausdruck `PRIMARY KEY` definiert. Wir empfehlen, für technische Primärschlüssel einen automatischen Zähler mittels `AUTOINCREMENT` zu definieren.

Datenbank-Manager

In Abschnitt 11.4.2 ab Seite 164 haben wir den Begriff des *Datenbank-Managers* als Repräsentant einer Datenbank bereits eingeführt. Dieser Manager ist die zweite und letzte Komponente unserer Datenzugriffsschicht. Das Datenbankschema ist durch die Schema-Interfaces eindeutig beschrieben. Daher kann der Datenbank-Manager die Befehle zur Schemaverwaltung von dort abrufen. Listing 11.9 zeigt den auf unsere Bedürfnisse zugeschnittenen Datenbank-Manager, der die Tabellen `routen` und `staus` verwaltet.

Datenzugriffe

Die Datenbankoperationen werden direkt auf der vom Datenbank-Manager gelieferten Verbindung ausgeführt. Eine Kapselung ist aus oben genannten Gründen in den meisten Fällen nicht notwendig.

Listing 11.9 *Staumelder Datenbank-Manager*

```
public class StaumelderDatenbank
                 extends SQLiteOpenHelper {
  private static
    final String DATENBANK_NAME = "staumelder.db";
  private static
    final int DATENBANK_VERSION = 1;

  public StaumelderDatenbank(Context context) {
    super(
      context,
      DATENBANK_NAME,
      null,
      DATENBANK_VERSION);
  }

  public void onCreate(SQLiteDatabase db) {
    db.execSQL(RoutenTbl.SQL_CREATE);
  }

  public void onUpgrade(SQLiteDatabase db,
      int oldVersion,
      int newVersion) {
```

```
    db.execSQL(
      "DROP TABLE IF EXISTS " +
      RoutenTbl.TABLE_NAME);
    onCreate(db);
  }
}
```

Abhängigkeiten berücksichtigen

Ein weiterer Vorteil einer einzigen Klasse zur Verwaltung des kompletten Schemas ist, dass dort die Abhängigkeiten der Tabellen untereinander bekannt sind. Man kann also `CREATE`- und `DROP`-Operationen in korrekter Reihenfolge aufrufen.

Was ist zu tun?

Um auf die in der Staumelder-Datenbank abgelegten Routendaten zuzugreifen, wollen wir also nach folgendem Schema vorgehen:

1. Erstellung eines Schema-Interface für die Tabelle `routen`
2. Erstellung des Datenbank-Managers
3. Implementierung der Datenbankanfragen
4. Darstellung der Ergebnisdatensätze mit Hilfe einer neuen Activity `RoutenlisteAnzeigen`

Die ersten beiden Punkte sind bereits erledigt. Als Nächstes wird eine Activity erzeugt und der Datenbank-Manager mit dieser verbunden.

11.8.2 Das Schema erstellen

Das Datenbankschema ist im Schema-Interface definiert und wird vom Datenbank-Manager erzeugt, sobald dieser mit einer Anwendung verbunden wird und der erste Zugriff auf die Datenbank erfolgt. Diese Schritte wollen wir nun durchführen.

Datenbank bekannt machen

Wir werden die Staumelder-Datenbank nicht nur zum Anzeigen von Routen verwenden. Sie sollte schon zum Start der Anwendung verfügbar sein. Daher erweitern wir die Activity `StartseiteAnzeigen` (aus Kapitel 5) und binden den Datenbank-Manager an diese an (Listing 11.10). So erkennen wir Fehler bei der Schemadefinition frühzeitig.

Listing 11.10 *Datenbank mit Anwendung verbinden*

```
public class StartseiteAnzeigen extends Activity {
  @Override
  public void onCreate(Bundle icicle) {
    ...
    new StaumelderDatenbank(this).
      getReadableDatabase();
    ...
  }
}
```

Beim ersten Aufruf von `getReadableDatabase` bzw. `getWriteableDatabase` wird geprüft, ob

- die Datenbank bereits existiert,
- ihre vorgegebene Version der existierenden entspricht und
- eine Verbindung hergestellt werden kann (Berechtigungsprüfung, Sperren etc.).

Beim nächsten Start der Anwendung wird die Datenbank `staumelder.db` also erzeugt, da sie dem Anwendungskontext noch nicht bekannt ist. Das Datenbankschema ist erstellt.

Datenbank verwenden

Die Routenliste soll über einen eigenen Bildschirmdialog `RoutenlisteAnzeigen` dargestellt werden. Da wir in dieser Activity den Zugriff auf die Datenbank mehrmals benötigen, definieren wir den Datenbank-Manager als Attribut der Klasse. Listing 11.11 zeigt die erste Definition der neuen Activity.

Listing 11.11 *Activity RoutenlisteAnzeigen*

```
public class RoutenlisteAnzeigen extends ListActivity {
  private StaumelderDatenbank stauDb;

  public void onCreate(Bundle savedInstanceState) {
    ...
    stauDb = new StaumelderDatenbank(this);
    routenAnzeigen();
  }

  private void routenAnzeigen() {
  }
}
```

11.8.3 Anfrageergebnisse an der Oberfläche darstellen

In der Methode `routenAnzeigen` wollen wir die aktuellen Routendaten über die `ListView` der Activity anzeigen lassen.

Als Erstes nutzen wir die Datenbankverbindung und suchen alle gespeicherten Routendaten. Die Liste der Ergebnisspalten speichern wir als Konstante, um sie nicht jedesmal neu aufbauen zu müssen.

```
private static final String[] DB_SUCHSPALTEN =
    new String[] {
      RoutenTbl.ID, RoutenTbl.START,
      RoutenTbl.ZIEL };

private void routenAnzeigen() {
  Cursor mcRouten = staumelderDb
```

```
      .getReadableDatabase().query(RoutenTbl.TABLE_NAME,
          DB_SUCHSPALTEN, null, null, RoutenTbl.START,
          null, null);
  startManagingCursor(mcRouten);
}
```

mc = Managing Cursor

Der Cursor wird hier als Managing Cursor verwendet (s. Abschnitt 11.5.3 auf Seite 175). Wir müssen uns also nicht um das Schließen des Cursors kümmern. Damit der Überblick in komplexeren Anwendungen nicht verloren geht, setzen wir das Präfix `mc` vor den Variablennamen eines Managing Cursors.

SimpleCursorAdapter

Das Suchergebnis wird über die `ListView` der Activity `RoutenlisteAnzeigen` auf dem Bildschirm dargestellt. Die Verbindung zwischen View und Daten stellt ein `SimpleCursorAdapter` her. Mit dem Konzept der Adapter haben wir uns in Abschnitt 6.2.2 auf Seite 78 befasst. Der folgende Codeabschnitt zeigt, wie die Ergebnismenge an die `ListView` angebunden wird.

```
private static final String[] ANZEIGE_SPALTEN =
    new String[] {
      RoutenTbl.START, RoutenTbl.ZIEL };

private void routenAnzeigen() {
  ...
  int[] widgetSchluessel = new int[] {
    android.R.id.text1,
    android.R.id.text2 };

  SimpleCursorAdapter routenAdapter =
      new SimpleCursorAdapter(
        this, android.R.layout.simple_list_item_2,
        mcRouten,
        ANZEIGE_SPALTEN,
        widgetSchluessel );

  setListAdapter(routenAdapter);
}
```

Vorsicht, Falle!

Damit die `ListView` immer einen eindeutigen Bezug zu einem Ergebnisdatensatz hat, muss in diesem eine Spalte als `_id` deklariert worden sein. Entweder wird also immer ein Schlüsselwert der Tabellen mitgegeben oder man definiert sich den für die Anzeige eindeutigen Spaltennamen mit dem Spalten-Alias `_id`.

Jetzt muss man nur noch die neue Activity `RoutenlisteAnzeigen` im Android-Manifest bekannt machen und schon ist die Anzeige der Routenliste fertig (Abb. 11-1).

Abb. 11-1
Routendaten aus der Datenbank

11.9 Fazit

Das Thema »Datenbanken« kann man noch sehr viel intensiver diskutieren, als wir es hier tun. Die Leistungsfähigkeit von Datenbanken spielt auf Mobilgeräten jedoch nicht die gleiche Rolle wie auf großen Serversystemen. Daher sollten wir das Thema auch nicht überstrapazieren.

Spielprojekt

Wir haben ein eigenes Eclipse-Projekt *DBsimple* erstellt und auf der Webseite des Buches zum Download bereitgestellt. Importieren Sie sich dieses Projekt und experimentieren Sie damit. Nach dem ersten Start können Sie eine Datenbank mit Testdaten füllen und sich die Ergebnisse verschiedener Anfragen anschauen. Zu nahezu allen Operationen haben wir Zeitmessungen eingebaut, die im Log ausgegeben werden.

Auf diese Weise können Sie sich ein Bild von den im Theorieteil dieser Iteration beschriebenen Implementierungsalternativen machen. Probieren Sie dabei auch die `sqlite3`-Konsole aus.

Mit diesem Aufruf zum Selbstversuch wollen wir die Iteration abschließen. Wir können jetzt elegant und performant die Datenbankfunktionalitäten von SQLite auf der Android-Plattform einsetzen.

12 Iteration 5 – Content Provider

Hinweis: Die 2. Auflage dieses Buchs erscheint unter dem Titel
»Android 2. Grundlagen und Programmierung«
Ende Mai 2010 im dpunkt.verlag, ISBN 978-3-89864-677-2

Private Datenquellen

Wir haben uns bereits in früheren Kapiteln mit Datenspeicherung und -zugriffen innerhalb einer Android-Anwendung befasst. Es wurde darauf hingewiesen, dass sowohl Datenbanken als auch Teile des Dateisystems ausschließlich für Zugriffe »ihrer« Anwendung ausgelegt sind.

Öffnung erwünscht

Was aber nun, wenn eine Anwendung Daten für eine oder mehrere andere Anwendungen bereitstellen will? So könnte man sich zum Beispiel eine Multimedia-Verwaltung vorstellen, die ein zentrales »Album« für Audio-, Video- und Bilddaten bereitstellt. Oder man möchte auf Inhalte des Android-Adressbuches zugreifen und diese in einer eigenen Anwendung weiterverarbeiten.

Content Provider = Datenschnittstelle

Für solche Anwendungsfälle sind sogenannte *Content Provider* vorgesehen. Diese Android-Komponenten erlauben die Implementierung von Schnittstellen, über die eine Anwendung ihre privaten Daten ganz oder teilweise für andere Anwendungen des gleichen Systems zugänglich machen kann.

12.1 Iterationsziel

Unser Staumelder benötigt auf den ersten Blick keine Daten aus anderen Anwendungen. Er stellt im Gegenzug auch keine besonderen Informationen für die Allgemeinheit bereit. Daher wollen wir hier unsere Beispielanwendung nicht künstlich aufblähen, sondern erweitern unsere Anwendungspalette um ein nützliches Werkzeug: eine Verbindungsverwaltung für Netzwerkeinstellungen. Wie wollen also einen `ConnectionSettingsManager` entwickeln, der folgende Funktionalität bereitstellt:

- Anzeige einer Liste aller Netzbetreiber, deren Datentarife und die damit verbundenen Geräteeinstellungen für Netzwerkverbindungen (Internet und MMS),
- Übernahme der vorgeschlagenen Netzwerkverbindungsdaten in die Systemeinstellungen des Android-Geräts,

- Bereitstellung der Verbindungsdaten aller Netzbetreiber für andere Anwendungen.

Aus technischer Sicht bedeutet dies, dass wir

- auf externe Datenquellen (Systemeinstellungen) zugreifen müssen,
- eine eigene Datenbank mit Verbindungsdaten aufbauen werden,
- die Inhalte dieser Datenbank für andere Anwendungen zugänglich machen.

Da es sich bei dem geplanten Content Provider um eine sehr technische Komponente handelt, werden wir kurz auf die englische Sprache als Quellcodesprache wechseln.

Erst Theorie

Wir werden uns in den nächsten Abschnitten Zugriffe auf Content Provider anschauen und dabei die Adressierungssyntax für Content Provider kennenlernen. Anschließend beschreiben wir, wie ein Content Provider implementiert wird. Zum Schluss des theoretischen Teils vergleichen wir das Konzept des Content Providers mit dem von AIDL-Services.

Dann Praxis

Im Praxisteil implementieren wir je einen Content Provider für Datenbankdaten und für Dateien.

12.2 Grundbegriffe

In diesem Abschnitt werden die Begriffe *Content* und *Content Provider* eingeführt.

Content Provider

Ein *Content Provider* ist eine Android-Komponente, die eine von `android.content.ContentProvider` vorgegebene Schnittstelle für den Zugriff auf private Daten der eigenen Anwendung implementiert. Ein Content Provider kann von der eigenen Anwendung ohne Einschränkung genutzt werden. Externe Anwendungen, die einen Content Provider ansprechen wollen, müssen von der datenhaltenden Anwendung erst dazu berechtigt werden.

Datenbank- oder Dateiinhalte

Vor der Nutzung oder Implementierung eines Content Providers muss geklärt werden, welche Inhalte (engl. *content*) von diesem geliefert werden sollen. Android bietet zwei mögliche Arten von Inhalten an: strukturierte Datenbankinhalte oder Binärdaten. Die Schnittstelle zu Datenbankdaten wird über einen Cursor hergestellt. Auf Binärdaten wird über Streams zugegriffen.

Eine Typfrage...

Ein Content Provider muss seine Umwelt darüber informieren, für Inhalte welcher Datentypen er sich zuständig fühlt. Daher ist die Implementierung der Methode `getType` mandatorisch. Über diese liefert er für jede Anfrage an den Provider den Mime-Type des zu erwartenden

Content zurück. Hierbei handelt es sich um Varianten der klassischen Mime-Types wie z.B. `application-content/text`, deren genaue Definition wir zu einem späteren Zeitpunkt kennenlernen werden.

12.3 Auf Content Provider zugreifen

Wir lernen als Erstes, Zugriffe auf bekannte Content Provider durchzuführen. Auf diese Weise wird der grundlegende Umgang mit diesen Datenlieferanten vermittelt. Danach können wir die Content Provider des Android-SDK nutzen (z.B. für Zugriffe auf das Adressbuch). Starten wollen wir mit der allgemeinen Adressierung eines Providers. Im Anschluss daran werden wir gezielt Anfragen an einen Content Provider stellen. Auch dies wird über die Adresse des Providers geregelt.

12.3.1 Content-URIs

Vergleich mit HTTP

Der Zugriff auf einen Content Provider ist dem Zugriff auf eine Webseite nicht unähnlich. Wenn man einer Webadresse in den Browser eingibt, bekommt man Daten (die HTML-Seite) zurück. Auch Content Provider sind auf ihrem Gerät unter eindeutigen Adressen ansprechbar.

URIs

Möchte man auf Daten anderer Anwendungen zugreifen, muss man zunächst die Adresse des Content Providers in Erfahrung bringen, der die Daten anbietet. Zur Definition der Adressen nutzt die Android-API das bereits im Umkreis impliziter Intents in Abschnitt 7.3 auf Seite 94 vorgestellte Konstrukt eines »Universal Resource Identifiers«, kurz *URI* genannt. Die URI ist nach folgendem Schema aufgebaut:

```
$scheme://$authority/$dataDescriptor[/$id]
```

Die Bedeutung der einzelnen URI-Elemente soll an einem Beispiel beschrieben werden. Unser geplanter Content Provider soll Verbindungsdaten (engl. *connection settings*) exponieren. Daher wollen wir die URI

```
content://de.androidbuch.settingsmanager.
    settingsprovider/settings
```

als Basis-URI nutzen. Schauen wir uns die Bestandteile der Adresse an:

scheme: Das `scheme` klassifiziert die Datenquelle. Wir unterscheiden zwischen `android.resource://`, `file://` und der allgemeinen Schemabezeichnung `content://`, welche auf Datenbank- oder Binärdateninhalte hinweist.

authority: Namensraum, für den der Content Provider definiert ist. Normalerweise wird hier der vollqualifizierte Name der implementierenden Klasse in Kleinbuchstaben verwendet, um die systemweite Eindeutigkeit sicherzustellen. Anhand dieser Bezeichnung wird der Content Provider innerhalb der Anwendung, d.h. im Android-Manifest, registriert.

dataDescriptor: Fachliche Bezeichnung des von dieser URI zu erwartenden Datentyps. Die Deskriptoren können auch über mehrere Ebenen definiert werden (z.B. `/settings`, `/settings/contracts`). Auf diese Weise lassen sich verschiedene Einstiegspunkte in den Content Provider definieren. Während `/settings` beispielsweise den Zugriff auf alle Konfigurationsdaten ermöglicht, würden unter `/settings/contracts` speziell die Vertragsdaten verfügbar gemacht.

id: Optional kann hier ein Schlüsselparameter z.B. (`_id`) angegeben werden, wenn sich die mit der URI verbundene Anfrage oder Operation auf genau einen Datensatz beziehen soll. Wird der Schlüssel nicht mitgeliefert, so sind immer *alle* Datensätze betroffen. Sollen mehrere Parameter übergeben werden, so hängt man diese einfach an die bestehende URI an (vgl. REST und HTTP). So wäre z.B. ein gültiger Descriptor für einen Vertrag mit dem Schlüsselwert `5` definiert als: `/settings/contracts/5`.

Wer suchet, der findet...

So weit, so gut. Die Adressen unserer selbst erstellten Content Provider sind uns nun bekannt. Doch wie kommen wir an die URIs eines »fremden« Providers? Soll z.B. auf einen Content Provider des Android-SDK zugegriffen werden (Adressbuch, E-Mail-Posteingang etc.), so hilft häufig ein Blick auf die Dokumentation des `android.provider` Package weiter.

Informationspflicht

Jeder Content Provider sollte seine URIs bekannt machen. Dies geschieht über Interfaces oder Inner Classes, über die nicht nur die `CONTENT_URI`, sondern auch weitere Details der gelieferten Datenstrukturen veröffentlicht werden. Tabelle 12-1 gibt einen ersten Überblick über die Liste der vom SDK bereitgestellten Content Provider. Wir verweisen an dieser Stelle auf die Originaldokumentation und empfehlen Ihnen, die ein oder andere URI auszuprobieren.

Datenstrukturen

Wollen wir auf strukturierte Datenbankinhalte zugreifen, so stehen wir nun vor dem Problem, dass uns die gelieferten Datenstrukturen noch unbekannt sind. Wie bereits erwähnt, liefert die Anfrage beim Content Provider einen Cursor auf die Zieldatenmenge zurück. Für die unter `android.provider` gelisteten Datenstrukturen sind dankenswerterweise alle Feldnamen dokumentiert, so dass die aufru-

android.provider.*	Beschreibung
Contacts	Einstiegspunkt für Adressbuch-Daten
MediaStore	Einstiegspunkt für Multimedia-Daten
Browser	Zugriff auf die Bookmarks und Suchergebnisse des WWW-Browsers
CallLog	Zugriff auf Daten der Anruferliste
Settings	Einstiegspunkt für Zugriffe auf Systemeinstellungen

Tab. 12-1
Beispiele für Content Provider

fende Komponente gezielt auf einzelne Attribute zugreifen kann (z.B. `android.provider.Contacts.PhonesColumns.NUMBER`). Diesen Komfort sollten wir auch allen Nutzern unserer selbst erstellten Content Provider anbieten. Aber dazu später mehr.

Probieren geht über...

Was aber, wenn wir lediglich die URI des Datenlieferanten kennen und wissen, dass es sich um strukturierte Daten handelt? In solchen Fällen müssen wir zunächst die erste Anfrage »blind« codieren und uns per Debugger oder im Log die zurückgelieferten Attributnamen ausgeben lassen. Dieses Vorgehen wird für unsere Beispielanwendung erforderlich sein, daher verweisen wir hier auf den Praxisabschnitt dieses Kapitels.

12.3.2 Zugriff über implizite Intents

Mit Hilfe impliziter Intents werden Activities gestartet, die bestimmten Zielbeschreibungen des Intent genügen (s. Abschnitt 7.3 auf Seite 94). Eine solche Activity könnte nun beispielsweise eine Anzeige- oder Bearbeitungsmaske für Datensätze bereitstellen, die unter der Kontrolle eines Content Providers stehen. Auf diese Weise muss der aufrufenden Anwendung der exakte Aufbau der Datenstruktur nicht bekannt sein. Außerdem muss eine an die Maske gekoppelte Geschäftslogik nicht kopiert werden.

Wiederverwendung von Masken

Der Zugriff auf einen Content Provider via implizitem Intent ist dann sinnvoll, wenn die Quellanwendung wiederverwendbare Bildschirmmasken bereitstellt. Es spielt dabei keine Rolle, ob über den Intent eine Activity der gleichen oder einer anderen Anwendung gestartet wird. Entscheidend ist lediglich die Konfiguration des `<intent-filter>` der Zielkomponente.

Durch Aufruf einer anderen Activity wird die Kontrolle an diese übergeben. Daher sollte dieser »Ausflug« in die andere Anwendung gut überlegt sein. Es empfiehlt sich, den Aufruf als Sub-Activity-Aufruf (sie-

he Abschnitt 7.7 auf Seite 102) auszuführen, um nach Verarbeitung oder Anzeige des gewünschten Datensatzes die Kontrolle wieder an die aufrufende Anwendung zurückzugeben.

Sollen die vom Content Provider bereitgestellten Daten allerdings in der Zielanwendung individuell ausgelesen, verarbeitet oder geändert werden, so reicht der Aufruf eines Intent nicht mehr aus. In diesem Fall müssen wir direkt an die Schnittstelle des Datenlieferanten herantreten. Dies geschieht mit Hilfe von `ContentResolver` Implementierungen, die vom Context einer Anwendung zur Verfügung gestellt werden.

12.3.3 Zugriff über Content Resolver

Die Klasse `android.content.ContentResolver` dient zur Implementierung synchroner und asynchroner Zugriffe auf Content Provider. Die Klasse ist verantwortlich für

- Operationen auf strukturierten Datenbankinhalten,
- Operationen zum Auslesen und Schreiben von Binärdaten,
- Unterstützung synchroner und asynchroner Operationen,
- Registrierung von Komponenten, die bei Änderung des vom Content Provider verwalteten Datenbestandes benachrichtigt werden sollen.

Für jede Android-Komponente, die von `android.content.ContextWrapper` erbt, steht ein Content Resolver über `getContentResolver` bereit. Die Auswahl des Content Providers, der von diesem angesprochen werden soll, wird über die mitgelieferte URI in der Signatur getroffen.

Datenbankzugriffe

Zugriff auf Datenbankinhalte Wir wollen uns jetzt die Aufgaben des Content Resolvers im Detail anschauen. Beginnen wir mit Datenbankzugriffen. Die Methoden für synchrone Datenbankoperationen sind in Abbildung 12-1 aufgeführt.

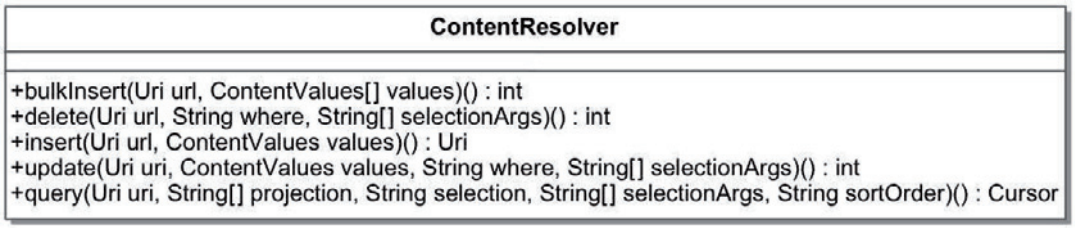


Abb. 12-1
Content Resolver Datenbankschnittstelle

Die Ähnlichkeit mit der Syntax der SQLite-Anfragen über `SQLiteDatabase` ist nicht verwunderlich. Ein Beispiel für eine Anfrage eines entfernten Datenbank-Content-Providers wäre

```
Uri cpCarriers =
    Uri.parse("content://telephony/carriers");
Cursor cCurrentApn =
    contentResolver.query(cpCarriers,
        new String[] {
          SettingsDisplay._ID,
          SettingsDisplay.NAME },
        SettingsDisplay.CURRENT + " is not null",
        null,
        null);
```

Da wir die Syntax von Datenbankoperationen bereits in Kapitel 11 ausführlich diskutiert haben, wollen wir uns hier nicht wiederholen. Man fragt sich an dieser Stelle, warum »nur« die `query`-Methode für Anfragen bereitgestellt wird. Dazu muss man aber beachten, dass der Zweck von Content Providern die Lieferung von »Content« und nicht ausschließlich von »SQLite-Datenbank-Content« ist. Es bleibt dem Entwickler des jeweiligen Providers überlassen, für verschiedene URIs unterschiedliche Datenbankanfragen zu implementieren. Ihnen gemein ist ja nur, dass sie einen Cursor zurückliefern müssen. Auch wenn die Datenbankanfragemethode eines Content Providers `query` heißt, so kann sie durchaus intern als `rawQuery` implementiert werden. Vorausgesetzt, es handelt sich überhaupt um eine SQLite-Datenbank. Man sieht an dieser Stelle, dass die Autoren des Content Providers eine möglichst schmale und damit flexible Schnittstelle zur Bereitstellung von Daten an andere Anwendungen schaffen wollten.

Transaktionen

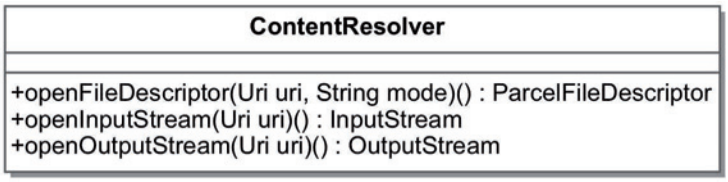
Erwähnenswert ist weiterhin, dass alle Operationen eines Content Providers atomar sind. Es ist also nicht möglich, eine Transaktionsklammer um entfernte Datenbankaufrufe zu definieren. Falls Transaktionssicherheit gefordert ist, muss diese durch Kompensationsoperationen implementiert werden. Nehmen wir als Beispiel das Anlegen eines neuen Adressbucheintrags `x` durch eine Anwendung `A`. Die Operation wird innerhalb eines Try/Catch-Blockes mit Hilfe des Content Resolvers durchgeführt. Tritt dabei ein Fehler auf und wird dieser via Exception vom Content Provider an den Aufrufer weitergegeben, so muss im `catch`-Block von `A` sichergestellt werden, dass `x` nicht im Adressbuch vorhanden ist. Dies geschieht durch den Aufruf der »Kompensationsoperation« `delete`. Dieses Verfahren muss immer dann gewählt werden, wenn mehrere Schreiboperationen als atomare Einheit behandelt werden sollen.

Für den theoretischen Teil soll das genügen. Wenden wir uns nun den Zugriffen auf Binärdaten zu.

Binärdaten => Streams

Zugriff auf Dateien/Ressourcen Für Lese- und Schreiboperationen auf Binärdaten, die von einem Content Provider veröffentlicht werden, werden die in Abbildung 12-2 aufgeführten Methoden des Content Resolvers angeboten.

Abb. 12-2
Content Resolver Binärdatenschnittstelle



Für Lesezugriffe auf Dateien oder Ressourcen weicht die »Schema«-Information (scheme) in der URI von dem für Datenbankzugriffe üblichen Wert »content« ab. Hier einige Beispiele für Lesezugriffe:

```
// android.resource://package_name/id_number
Uri uriRemoteResource =
  Uri.parse(
    ContentResolver.SCHEME_ANDROID_RESOURCE+
    "://com.example.myapp/" +
    R.raw.my_resource");

// android.resource://package_name/type/name
Uri uri =
  Uri.parse(
  "android.resource://com.example.myapp/raw/" +
  "my_resource");

Uri uriFile =
  Uri.parse(
  ContentResolver.SCHEME_FILE+
  "://com.example.myapp/filename");
```

Für Schreibzugriffe wird ausschließlich das Schema `content://` genutzt.

12.4 Content Provider erstellen

Nachdem wir den Zugriff auf Content Provider hinreichend beschrieben haben, wollen wir uns in diesem Abschnitt damit befassen, eigene Datenlieferanten für andere Anwendungen bereitzustellen. Dieser Prozess gestaltet sich wie folgt:

1. Klären, um *was* für »Content« es sich handelt (Daten oder Dateien)
2. Implementierung einer Unterklasse von `android.content.ContentProvider`
3. Erstellung einer »Metadatenklasse« pro strukturiertem Datentyp. Diese Klasse definiert u.a. die `CONTENT_URI` und die Namen aller vom Content Provider exponierten Datenbankattribute.
4. Implementierung der datentypabhängigen Methoden des Content Providers (z.B. `query`, `update`, `openFile`)
5. Registrierung des Content Providers im Android-Manifest der implementierenden Anwendung

Der Implementierungsabschnitt wird zeigen, wie Content Provider für konkrete Rückgabedatentypen implementiert werden. Wir wollen an dieser Stelle daher nur kurz auf die von `android.content.ContentProvider` bereitgestellte Schnittstelle eingehen (Abbildung 12-3).

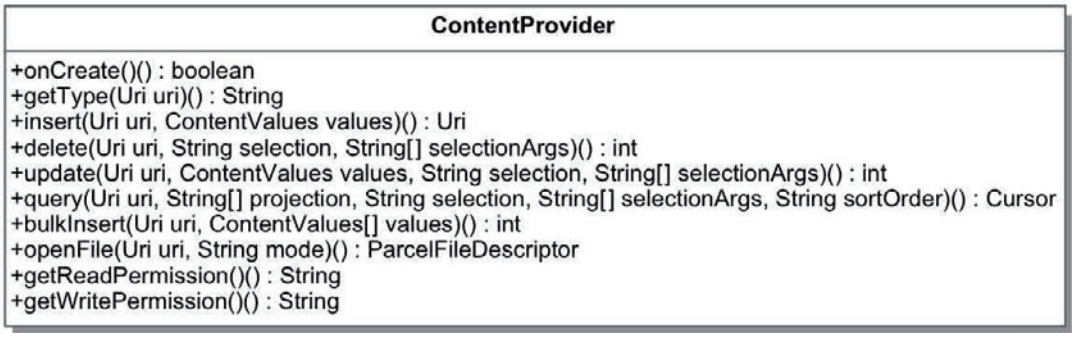


Abb. 12-3
Die Content-Provider-Schnittstelle

Auch hier haben wir nicht alle Methoden angezeigt, sondern beschränken uns auf die zu Beginn wichtigen. Die Methoden lassen sich recht einfach in inhaltlich verwandte Blöcke unterteilen:

- Allgemeines
- Datenbank-Zugriffsmethoden
- Datei-Zugriffsmethoden

12.4.1 Allgemeines

Content Provider stehen genauso wie Activities und Services unter der Kontrolle des Betriebssystems. Bei ihrer Initialisierung wird daher automatisch die `onCreate`-Methode aufgerufen.

Datentyp = Mime-Type

Jeder über einen Content Provider herausgegebene *Datentyp* muss für den Rest des Systems unter einem eindeutigen Namen identifiziert

werden können. Das Android-SDK bedient sich dabei abstrakter *Mime-Types*, die vom Ersteller des Content Providers individuell nach folgendem Schema vergeben werden können: Datenquellen, die einzelne Exemplare eines Datentyps zurückgeben, liefern diese vom Mime-Type

```
vnd.android.cursor.item/mein.package.datentypbezeichnung
```

aus. Wird dagegegen eine Menge von Exemplaren, also quasi ein *Verzeichnis* von Objekten, zurückgeliefert, so hat dieser Rückgabewert den Typ

```
vnd.android.cursor.dir/mein.package.datentypbezeichnung
```

Damit der Content Provider anfragenden Intents stets mitteilen kann, welche Datentypen er ausliefern kann, muss die Methode `getType` für jede erlaubte URI implementiert werden.

Berechtigungen

Der Lese- oder Schreibzugriff kann für einen Content Provider, unabhängig von der aufgerufenen URI, eingeschränkt werden. Die aufrufende Anwendung muss dann über die verlangten Rechte verfügen (`<uses-permission>` im Android-Manifest), damit der Zugriff erlaubt wird. Da es keine zentrale Instanz gibt, die alle Berechtigungen aller aktiven und zukünftigen Android-Anwendungen kennt, erfolgt die Definition der jeweiligen Rechte über simple Strings. Im Package `android.permission` sind alle das Android-System und seine Standardanwendungen betreffenden Rechte (Permissions) als Konstanten hinterlegt. Um den aufrufenden Anwendungen nicht unnötig Steine in den Weg zu legen und die Allgemeinverträglichkeit des Content Providers zu erhöhen, sollte man auf diesen Satz von Berechtigungen auch für eigene Content Provider-Implementierungen zurückgreifen. Falls eine derartige Zugriffsbeschränkung notwendig wird, müssen die Methoden `getReadPermission` bzw. `getWritePermission` implementiert werden.

12.4.2 Datenbank-Zugriffsmethoden

Die Semantik der bereits aus dem Datenbank-Kapitel bekannten Methoden für Zugriffe auf strukturierte Datenbankinhalte muss hier nicht noch einmal erklärt werden. Auch hier wird in jeder Signatur die URI der Zieldatensätze bzw. des Zieldatensatzes mitgegeben. Es muss also für nahezu jede Datenbankmethode eine Fallunterscheidung gemäß der URI durchgeführt werden. Man sollte von Anwendung zu Anwendung unterscheiden, ob man einen Content Provider zur Verwaltung mehrerer Tabellen implementiert oder pro Tabelle einen eigenen Provider schreibt. `if-else`-Kaskaden sind genauso unschön wie `switch-case`-Blöcke, aber der Zweck heiligt für unsere beschränkten Hardware-Ressourcen dann doch letztlich die Mittel. Wichtig ist, den eigenen und

auch potenziellen »fremden« Android-Komponenten eine saubere und wiederverwendbare Schnittstelle zu unseren Daten bereitzustellen.

12.4.3 Datei-Zugriffsmethoden

Während die Datenbank-Zugriffsmethoden weitgehend deckungsgleich mit ihren »Anfragemethoden« des Content Resolvers sind, stellen wir fest, dass für Datei- und Ressourcenzugriffe lediglich eine einzige Methode im Content Provider vorgesehen ist: `openFile`.

Durch Implementierung dieser Methode lassen sich Zugriffe auf Dateien des anwendungseigenen, »privaten« Dateisystems realisieren. Als Rückgabewert wird ein `android.os.ParcelFileDescriptor` verwendet, über den sowohl Lese- als auch Schreibzugriffe über Streams implementiert werden können. Das folgende Beispiel demonstriert den Lesezugriff auf eine entfernte Datei. Der vollständige Code ist auf Seite 216 beschrieben.

```
...
ParcelFileDescriptor routenDateiPD =
  getContentResolver().
    openFileDescriptor(dateiUri, "r");
List<String> routenDatei =
  org.apache.commons.io.IOUtils.readLines(
    new FileInputStream(
          routenDateiPD.getFileDescriptor()));
...
```

Verzeichnis => Datenbank

Da die Kommunikation über Streams erfolgt, können wir die Ausgabe von Verzeichnisinhalten auf diese Weise nicht abbilden. Dateien aus Verzeichnissen, die für die Nutzung in Content Providern vorgesehen sind, müssen daher zunächst in einer Datenbanktabelle erfasst werden. Danach ist der Zugriff sehr einfach möglich, wie unser Beispiel im anschließenden Praxisteil zeigen wird.

In der Methode `ContentProvider.openFileHelper` wird der komplette Dateizugriff gekapselt. Voraussetzung für eine erfolgreiche Nutzung ist allerdings, dass

1. mindestens der Zieldateiname in einer Tabelle erfasst wird, die vom Content Provider verwaltet wird,
2. der Zieldateiname (inklusive komplettem Pfad) in der Tabellenspalte `_data` steht und
3. der Datenbankeintrag zu dieser Datei über die Schlüsselsuche des Content Providers eindeutig geladen werden kann.

Es ist je nach Anwendungsfall aber auch problemlos möglich, direkt auf das Dateisystem zuzugreifen und die Ergebnisdatei für Lese- oder Schreibzugriffe als `ParcelFileDescriptor` zurückzuliefern.

```
...
File datei = ...;
ParcelFileDescriptor descriptor =
  ParcelFileDescriptor(
    datei,
    ParcelFileDescriptor.MODE_READ_ONLY);
...
```

12.5 Asynchrone Operationen

Android-Anwendungen müssen schnell auf Benutzereingaben reagieren. Zugriffe auf große Datenmengen oder große Dateien werden daher gerne im Hintergrund ausgeführt, um die Aktivitäten an der Oberfläche nicht zu behindern.

Asynchrone Aufrufe

Zugriffe auf Content Provider können also auch dazu führen, dass die aufrufende Anwendung von langen Ladezeiten beeinträchtigt wird. Für solche Fälle bietet die Android-API eine Schnittstelle für *asynchrone* Zugriffe auf Content Provider an.

ANR wird vermieden.

Auf diese Weise kann sichergestellt werden, dass die aufrufende Anwendung zwischen Aufruf und Rückruf (engl. *callback*) durch den Content Provider seine Zeit für sinnvolle Aufgaben nutzen kann und kein ANR ausgelöst wird.

Dem Thema »Asynchronität bei Android« haben wir einen eigenen Abschnitt 8.3.2 auf Seite 133 gewidmet, dessen Verständnis im Folgenden vorausgesetzt wird.

Der AsyncQueryHandler

Soll der Aufruf eines Content Providers über einen Content Resolver asynchron erfolgen, so muss ein `android.content.AsyncQueryHandler` hinzugezogen werden. Dieser »wickelt« sich um den Content Resolver und koordiniert die asynchronen Aufrufe. Das folgende Beispiel (Listing 12.1) demonstriert diesen Prozess.

Listing 12.1 *Asynchrone Provider-Aufrufe*

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

  // (1)
  static final int MY_INSERT_TOKEN = 42;
  ...
  protected void zeigeRoutenverzeichnis() {
    ...
  }
  ...
```

```
  // (2)
  private class QueryHandler extends AsyncQueryHandler {
    public QueryHandler(ContentResolver cr) {
      super(cr);
    }

    @Override
    protected void onInsertComplete(int token, Object cookie,
        Uri uri) {
      // wir interessieren uns hier nur fuer INSERTs
      zeigeRoutenverzeichnis();
    }
  }

  private String speichereTestroute() {
    String dateiName = null;
    ...
    // (3)
    QueryHandler queryHandler =
      new QueryHandler(getContentResolver());
    ContentValues insertParams = new ContentValues();
    insertParams.put(
      Dateimanager.RoutenDateiInfo.ROUTEN_ID, routenId);
    insertParams.put(
      Dateimanager.RoutenDateiInfo.DATEINAME, dateiName);
    // (4)
    queryHandler.startInsert(
        MY_INSERT_TOKEN, null,
        Dateimanager.RoutenDateiInfo.CONTENT_URI,
        insertParams);
    Log.i(TAG, "Routendatei-Job gestartet.");
    ...
    return dateiName;
  }
  ...
}
```

Betrachten wir uns den Code einmal Schritt für Schritt.

(1) correlation id: Damit die Callback-Implementierung das Ergebnis eines Aufrufes eindeutig zuordnen kann, muss beim Aufruf ein sogenanntes *Token* übergeben werden. Dieses Token sollte diesen konkreten Aufruf eindeutig identifizieren, wenn er von Ergebnissen anderer Aufrufe unterschieden werden muss. Vielfach wird dieses Token auch als »correlation id« bezeichnet. In unserem Beispiel geht es lediglich darum, *dass* eine INSERT-Operation beendet wurde. Die Frage, *welcher* kon-

krete Aufruf dahintersteckt, ist nicht relevant. Daher kommen wir hier mit einer recht groben Definition des Tokens aus.

(2) Definition des AsyncQueryHandlers: Hier werden die Callbacks für einzelne Content-Provider-Aufrufe implementiert. Im Beispiel werden nur INSERT-Operationen asynchron aufgerufen. Analog dazu könnten auch Callbacks für `onQueryComplete`, `onDeleteComplete` und `onUpdateComplete` implementiert werden.

(3) Nutzung des QueryHandlers: Der Content Resolver des aktuellen Context wird an den `AsyncQueryHandler` übergeben. Um ihn nicht unnötig zu wiederholen, sollte dieser Vorgang in der `onCreate`-Methode stattfinden.

(4) Asynchrone Aufrufe: Der `AsyncQueryHandler` bietet für jede Datenbankoperation eines Content Providers eine passende Schnittstelle für asynchrone Aufrufe an. Analog zum hier verwendeten `startInsert` könnte man auch `startQuery`, `startDelete` und `startUpdate` aufrufen. Die entsprechenden Callback-Methoden müssen dann natürlich auch individuell implementiert werden, wenn nach erfolgreicher Bearbeitung der Operation noch Anschlussaktivitäten stattfinden sollen.

Wir wollen an dieser Stelle auf die Dokumentation der Klasse `AsyncQueryHandler` verweisen. Die Schnittstelle ist intuitiv nutzbar. Probieren Sie es aus.

12.6 Deployment

Um einen Content Provider für die eigene oder andere Anwendungen verfügbar zu machen, muss man ihn ins Android-Manifest aufnehmen. Zu diesem Zweck dient das `<provider>`-Tag, welches unterhalb der betroffenen `<application>` platziert werden muss (Listing 12.2).

Listing 12.2 *Registrierung eines Content Providers*

```
<application
  android:icon="@drawable/icon"
  android:label="@string/app_name">

  <provider
    android:name="ConnectionSettingsProvider"
    android:authorities=
      "de.androidbuch.settingsmanager.settingsprovider"
    android:enabled="true"
  />
```

```
<activity
  android:name=".ConnectionSettingsManager"
  android:label="@string/app_name"
>
...
```

Das <provider>-Tag stellt noch einige andere Attribute bereit, die der Online-Dokumentation zu entnehmen sind. Entscheidend für die simple Bekanntmachung und Nutzung eines Providers sind die hier angegebenen Attribute.

12.7 Alternativen zum Content Provider

Die Schnittstelle eines Content Providers erscheint auf den ersten Blick etwas unflexibel und unhandlich. Es gibt nur eine Methode für Anfragen, man muss immer alle Methoden für Datenbankanfragen implementieren, auch wenn der Provider gar keine derartigen Daten bereitstellt.

AIDL statt Content Provider?

Auf der Suche nach »eleganteren« Alternativen fällt der Blick auf Remote Services, die ausführlich in Kapitel 8 ab Seite 105 beschrieben werden. Diese sind vom Prinzip mit Webservices zu vergleichen. Anhand einer Schnittstellenbeschreibung werden Coderümpfe generiert, die über ein XML-Protokoll den Datentransfer über Anwendungsgrenzen hinweg ermöglichen. Allerdings zahlt man dafür einen gewissen Preis in Form von Mehraufwand der Konfiguration und Laufzeiteinbußen durch XML-Marshalling und -Unmarshalling.

Die Content-Provider-Schnittstelle bietet hier deutlich mehr. Sie liefert, gerade für Operationen, die Verzeichnisstrukturen bzw. Datenmengen zurückliefern, performanten Zugriff per Cursor. Der Konfigurationsaufwand beschränkt sich auf den Eintrag im Android-Manifest und die Definition der über den Cursor bereitgestellten Attributnamen.

Unhandlich wird ein Content Provider lediglich dann, wenn es darum geht, einzelne Exemplare komplexer Datentypen zurückzuliefern. Per Cursor könnten die Attribute zwar übertragen werden, die Empfängerklasse müsste sich aber dann mit dem Aufbau des komplexen Zielobjektes befassen. In solchen Fällen wäre über den Einsatz eines Remote Service nachzudenken. Ansonsten würden wir zur Datenübertragung zwischen Anwendungen dem Content Provider den Vorzug geben.

12.8 Implementierung

Wir wollen nun an zwei Beispielen zeigen, wie die beiden grundsätzlichen Arten von Content Providern implementiert werden können: Datenbank- und Dateiprovider. Beginnen wollen wir mit der Erstellung eines `ConnectionSettingsProvider`, der Internet- und MMS-Systemeinstellungen für verschiedene Netzbetreiber und Mobilfunkverträge bereitstellt. Da Quellcode bekanntlich ein volatiles Medium ist, wird der gesamte Code dieses Kapitels auf unserer Webseite zum Buch unter `www.androidbuch.de/contentprovider` zum Herunterladen angeboten. Wir beschränken uns hier nur auf einige wesentliche Codeauszüge.

Zur Demonstration eines Dateiproviders können wir wieder auf unseren Staumelder zurückgreifen. Dateien, die empfohlene Ausweichrouten für Staus beschreiben, werden innerhalb des Staumelders erstellt. Wir implementieren nun einen Provider, der diese Routendateien im GPX-Format auch für andere Anwendungen zum Lesezugriff bereitstellt.

Wir haben es in den folgenden Abschnitten daher mit zwei Anwendungen zu tun: dem `ConnectionSettingsProvider` für Datenbankzugriffe und dem `RoutenBrowser` für Dateizugriffe.

12.8.1 Ein Datenbank-Content-Provider

Wir starten mit der Definition des Content Providers für die Bereitstellung von Verbindungsdaten für verschiedene Netzbetreiber und deren Mobilfunk-Vertragsoptionen. Als Erstes wird dazu die Datenbank definiert. Wir erstellen, gemäß der Empfehlungen aus Kapitel 11, eine Klasse `ConnectionSettingsDatabase` als Datenbank-Manager sowie ein Schema-Interface für eine Tabelle `APN_CONN_SETTINGS`.

Der Zugriff auf diese Datenbank soll sowohl für die »eigene« als auch für externe Anwendungen ausschließlich über den `ConnectionSettingsProvider` erfolgen.

Test first...

Ein Content Provider wird normalerweise von vielen Anwendungen und Komponenten genutzt. Daher empfehlen wir, vor der Implementierung der Schnittstelle erst die Aufrufe durch einen Konsumenten zu erstellen. Auf diese Weise können wir schnell erkennen, welche Daten vom Content Provider geliefert werden müssen.

Unser Beispiel-Content-Provider soll folgende Ergebnisse liefern:

- eine Liste aller Netzbetreiber (»Provider«) eines Landes,
- alle Verbindungsdaten für einen bestimmten Vertrag eines Netzbetreibers.

Die als Ergebnis der ersten Anfrage gelieferte Provider-Liste soll in einer `ListActivity` dargestellt werden. Diese Darstellung soll sowohl für externe als auch für interne Anwendungen angeboten werden.

Die Verbindungsdaten müssen dagegen auf verschiedene Weise behandelt werden. Unsere `ConnectionSettingsManager`-Anwendung stellt die Parameter zwar in einer eigenen View dar, die aber nicht zwangsläufig für andere Anwendungen sichtbar ist. Externe Anwendungen sollen lediglich den Zieldatensatz über einen Cursor geliefert bekommen.

Beiden Varianten ist gemeinsam, dass die Daten zunächst vom Content Provider geliefert werden müssen. Wir implementieren zunächst den Zugriff aus einer Activity `ProviderBrowser` der Anwendung »ConnectionSettingsProvider«.

```
public class ProviderBrowser extends ListActivity {
  ...
  private static final String[] PROV_COLUMNS =
    { ConnectionSettings.Settings._ID,
      ConnectionSettings.Settings.MNC,
      ConnectionSettings.Settings.PROVIDER };
  private static final String[] UI_COLUMNS =
    { ConnectionSettings.Settings.PROVIDER
    };
  private static final int[] UI_IDS =
    { android.R.id.text1
    };
  ...
  private void fillProviderList() {
    Cursor cursor = managedQuery(
        ConnectionSettings.Settings.CONTENT_URI,
        PROV_COLUMNS,
        ConnectionSettings.Settings.MCC + "=?",
        new String[] { String.valueOf(mcc) },
        ConnectionSettings.Settings.PROVIDER);
    if (cursor.moveToFirst()) {
      SimpleCursorAdapter adapter =
        new SimpleCursorAdapter(
          this, android.R.layout.simple_list_item_1,
          cursor,
          PROV_COLUMNS,
          PROV_UI_IDS);
      setListAdapter(adapter);
    } else {
      Log.w(TAG, "No providers found for mcc " + mcc);
    }
  }
  ...
  }
```

Dieses Codefragment demonstriert einige wichtige Aspekte der Implementierung von Content Providern. In diesem Fall wird die Kontrolle über das Anfrageergebnis an den Managing Cursor der (List)Activity übergeben. Das hat Sinn, wenn es sich um die einzige Ergebnismenge handelt, die über die View der Activity dargestellt werden soll. Falls wir in einer Activity mehrere Cursors verwenden wollen, muss auch ein »unmanaged« Cursor dabei sein. Ein Beispiel dafür ist in der Klasse `de.visionera.settingsmanager.ContractProvider` zu finden.

```
private Uri updateApnSettings(long connSettingId) {

  ContentResolver contentResolver =
      getContentResolver();

  // lade neue APN-Daten:
  Cursor cSelectedApnDataset = contentResolver.query(
      ContentUris.withAppendedId(
          ConnectionSettings.Settings.CONTENT_URI,
          connSettingId)
      ,null, null, null, null);
  if( !cSelectedApnDataset.moveToFirst() ) {
    Log.w(TAG,
      "Keine APN-Einstellungen gefunden. _id = " +
          connSettingId);
    cSelectedApnDataset.close();
    return null;
  }
...
}
```

Dieser Nicht-Managing-Cursor unterliegt der Kontrolle des Implementierers und muss, wie im Beispiel zu sehen, manuell geschlossen werden.

Im ersten Beispiel verwenden wir Konstanten für die Definition der beteiligten Gui-Spalten und Content-Provider-Ergebnisdaten. Dies ist der aus Performanzgründen empfohlene Weg. Wir werden in diesem Buch aber oft davon abweichen, um die Lesbarkeit des Codes zu verbessern.

An beiden Beispielen wird klar, wie via Content Resolver auf einen Content Provider zugegriffen wird und wie dessen Methoden aufgerufen werden. Wir betrachten nun das Umfeld unseres Providers etwas genauer.

Verzeichnis der Datenstruktur

Die Klasse `ConnectionSettings` dient quasi als »Inhaltsverzeichnis« unseres Content Providers. Dort finden wir die konkreten URIs sowie die Attributnamen der über einen Cursor herausgegebenen Werte. Schauen wir uns diese Klasse daher einmal an.

```
package de.visionera.settingsmanager;

public class ConnectionSettings {
  public static final String AUTHORITY =
    "de.androidbuch.settingsmanager.settingsprovider";

  private ConnectionSettings() {
  }

  public static final class Settings
      implements BaseColumns {
    private Settings() {
    }

    /**
     * content:// -URL fuer diese Tabelle
     */
    public static final Uri CONTENT_URI =
      Uri.parse("content://"+ AUTHORITY + "/settings");

    /**
     * Der MIME-Type einer {@link #CONTENT_URI}. Liefert
     * die Tabelle der APN-Einstellungen
     */
    public static final String CONTENT_TYPE =
      "vnd.android.cursor.dir/vnd.androidbuch." +
      "apnsettings";

    /**
     * Der MIME-Type einer {@link #CONTENT_URI}. Liefert
     * einen einzelnen APN-Eintrag
     */
    public static final String CONTENT_ITEM_TYPE =
      "vnd.android.cursor.item/vnd.androidbuch." +
      "apnsettings";

    // alle "Spaltennamen", die nach außen gegeben werden
    public static final String MCC = "mcc";
    public static final String MNC = "mnc";
    public static final String PROVIDER =
        "provider_name";

    // Die Standard-Sortierreihenfolge
    public static final String DEFAULT_SORT_ORDER =
      MCC + ","+ MNC + "," + CONTRACT;
  }
```

```
  public static final class Providers
      implements BaseColumns {
    private Providers() {}
    ...
  }
  ...
}
```

Sämtliche hier aufgelisteten Attribute könnten auch direkt in den Content Provider codiert werden. Dies führt aber bei Verwendung mehrerer URIs (mehr als zwei) sowie mehrerer beteiligter Datenbanktabellen und herausgegebener Datenstrukturen zu einer unübersichtlichen Content-Provider-Implementierung.

_id und _count

Jeder Ergebnistyp eines Content Providers sollte das Interface `android.provider.BaseColumns` implementieren. Dadurch wird festgelegt, dass jede solche Datenstruktur mindestens über die Felder `_id` und `_count` verfügt. Somit kann für jeden von einem Content Provider gelieferten Cursor durch simplen Zugriff

```
int anzahlErgebnisse = cursor.getCount();
```

die Anzahl der Datensätze in der Ergebnismenge ermittelt werden. Der Wert dieses Attributs wird von der Android-Datenbankschicht »automatisch« gesetzt.

> Falls ein »externer« Content Provider mit bekannter URI nicht so bereitwillig Auskunft über seine Datenstrukturen gibt, so hilft folgender Trick weiter. Anhand der URI führt man eine Anfrage auf alle Daten des Providers durch. Dazu reicht ein `query`-Aufruf ohne Angabe von Selektions- und Einschränkungskriterien. Im Debugger verfolgt man nun den Aufruf und schaut sich den (hoffentlich) zurückgelieferten Cursor genau an. Dieser enthält auch alle Metainformationen der ihm zugrunde liegenden Datenstrukturen. Darunter fallen auch die Attributnamen.

Der Content Provider

Nachdem wir unsere Datenstrukturen und die damit verbundenen URIs und Mime-Types definiert haben, wenden wir uns dem Kern der Sache zu: der Implementierung des Content Providers. Betrachten wir als Beispiel unsere Klasse `ConnectionSettingsProvider` in Listing 12.3.

Listing 12.3 *Ein minimaler Content Provider*

```
public class ConnectionSettingsProvider
  extends ContentProvider {

  @Override
  public boolean onCreate() {
```

```
    return false;
  }

  @Override
  public String getType(Uri uri) {
    return null;
  }

  @Override
  public Cursor query(Uri uri, String[] projection,
      String selection, String[] selectionArgs,
      String sortOrder) {
    return null;
  }

  @Override
  public Uri insert(Uri uri, ContentValues values) {
    return null;
  }

  @Override
  public int delete(Uri uri, String selection,
      String[] selectionArgs) {
    return 0;
  }

  @Override
  public int update(Uri uri, ContentValues values,
      String selection, String[] selectionArgs) {
    return 0;
  }
}
```

Aus diesem wirklich minimalen Content Provider können wir noch keinen wirklichen Wert schöpfen. Also binden wir die `ConnectionSettingsDatabase` an und implementieren `getType`, damit der Provider bei der Auflösung anfragender Intents erkannt werden kann.

```
...
public class ConnectionSettingsProvider
  extends ContentProvider {

  // Schluessel fuer internes URI-Matching
  private static final int CONN_SETTINGS_ALL = 1;
  private static final int CONN_SETTINGS_ID = 2;
```

```
  private static final UriMatcher sUriMatcher;

  private ConnectionSettingsDatabase db;

  static {
    sUriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
    sUriMatcher.addURI(ConnectionSettings.AUTHORITY,
        "settings", CONN_SETTINGS_ALL);
    sUriMatcher.addURI(ConnectionSettings.AUTHORITY,
        "settings/#", CONN_SETTINGS_ID);
  }

  @Override
  public boolean onCreate() {
    db = new ConnectionSettingsDatabase(getContext());
    return true;
  }

  @Override
  public String getType(Uri uri) {
    switch (sUriMatcher.match(uri)) {
      case CONN_SETTINGS_ALL:
        return ConnectionSettings.Settings.
            CONTENT_TYPE;

      case CONN_SETTINGS_ID:
        return ConnectionSettings.Settings.
            CONTENT_ITEM_TYPE;

      default:
        throw new IllegalArgumentException(
            "Unknown URI " + uri);
    }
  }
  ...
}
```

UriMatcher

Die Funktion des `android.content.UriMatcher` ist selbsterklärend. Da ein Content Provider häufig als Fassade für mehrere, durch ihre URI unterschiedene Anfragen dient, wird der `UriMatcher` zur Konvertierung der eingehenden URI in einen für Switch-Case-Anweisungen nutzbaren Datentyp eingesetzt.

Bevor wir jetzt die weiteren Operationen implementieren, sollten wir die Verknüpfung zwischen dem vom Content Provider gelieferten Ergebnistyp und der zugrunde liegenden Datenquelle herstellen. Bei dieser Datenquelle sind wir keineswegs auf nur eine Tabelle beschränkt.

Die query-Syntax erlaubt Datenbank-Joins ebenso wie die Erstellung von Vereinigungsmengen (UNIONs) mehrerer Tabellen. Wichtig ist, dass bei der Verknüpfung zwischen Provider und Tabellen eventuelle Mehrdeutigkeiten in den Attributnamen (z.B. _id) durch Voranstellen des Tabellennamens aufgelöst werden müssen. Zur Verknüpfung der virtuellen mit den echten Datenstrukturen bedienen wir uns einer Map wie im folgenden Codeabschnitt gezeigt.

```
...
// Zuweisung CP-Datenstruktur => Datenbanktabelle
private static Map<String, String>
  connSettingsProjectionMap;
...
static {
  ...
  connSettingsProjectionMap =
    new HashMap<String, String>();
  connSettingsProjectionMap.put(
    ConnectionSettings.Settings._ID,
    ConnectionSettingsTbl.TABLE_NAME+"."+
    ConnectionSettingsTbl.COL_ID);
  connSettingsProjectionMap.put(
    ConnectionSettings.Settings.APN_NAME,
    ConnectionSettingsTbl.COL_APN_APN_NAME);
  ...
}
...
```

Die Initialisierung unseres Content Providers ist abgeschlossen und wir können mit der Implementierung der Operationen beginnen. Um möglichst schnell Resultate zu sehen, fangen wir mit der query-Methode an.

```
@Override
public Cursor query(Uri uri, String[] projection,
    String selection, String[] selectionArgs,
    String sortOrder) {
  SQLiteQueryBuilder qb = new SQLiteQueryBuilder();


  String defaultSortOrder = null;

  switch (sUriMatcher.match(uri)) {
  case CONN_SETTINGS_ALL:
    qb.setTables(
        ConnectionSettingsTbl.TABLE_NAME);
    qb.setProjectionMap(connSettingsProjectionMap);
```

```
      defaultSortOrder =
        ConnectionSettings.Settings.DEFAULT_SORT_ORDER;
      break;

    case CONN_SETTINGS_ID:
      qb.setTables(
          ConnectionSettingsTbl.TABLE_NAME);
      qb.setProjectionMap(connSettingsProjectionMap);
      qb.appendWhere(
          ConnectionSettingsTbl.COL_ID
              + "=" +
              uri.getPathSegments().get(1));
      defaultSortOrder =
        ConnectionSettings.Settings.DEFAULT_SORT_ORDER;
      break;

    default:
      throw new IllegalArgumentException(
        "Unknown URI " + uri);
    }

    String orderBy;
    if (TextUtils.isEmpty(sortOrder)) {
      orderBy = defaultSortOrder;
    }
    else {
      orderBy = sortOrder;
    }

    Cursor c = qb.query(
        db.getReadableDatabase(),
        projection,
        selection,
        selectionArgs,
        null, null,
        orderBy);

    return c;
  }
```

Wir sehen hier ein schönes Beispiel für den Einsatz des `SQLiteQueryBuilder`, da der Aufbau einer Query durch die aufrufende URI maßgeblich beeinflusst wird.

Komplexe Pfade

Doch der vorliegende Code macht noch ein wenig mehr deutlich. Das Fragment

```
uri.getPathSegments().get(1));
```

zeigt, wie auf die einzelnen Elemente der URI zugegriffen werden kann. In diesem simplen Beispiel erwarten wir nur maximal einen Parameter (den Schlüsselwert) als Bestandteil der URI. Es wären aber durchaus auch Fälle denkbar, in denen z.B. durch die URI

```
content://de.androidbuch.foobar/settings/262/
    provider/17/contracts
```

die Verbindungseinstellungen aller Verträge des Netzbetreibers mit dem Schlüssel `17`, der über die Nationalitätenkennung `262` verfügt, ermittelt werden sollen. In diesem Fall müsste sich der Content Provider erst die Einzelbestandteile des URI-Pfades zusammensuchen, bevor er die Anfrage an die Datenbank stellen kann.

Joins

Zur Demonstration der Mächtigkeit dieser `query`-Schnittstelle skizzieren wir hier noch kurz die Implementierung eines Joins über zwei Tabellen.

```
import static de.androidbuch.staumelder.
    routenspeicher.ConnectionSettingsTbl.*;
...
  case ROUTENDATEI_VERZ:
    qb
    .setTables(
        T_RoutenDateien.TABLE_NAME+","+
        T_Routen.TABLE_NAME
       );
    qb.setProjectionMap(routenDateienProjectionMap);
    qb.appendWhere(
        T_RoutenDateien.TABLE_NAME+".routen_id = "+
        T_Routen.TABLE_NAME+"._id ");
    break;
...
```

Wir haben hier einen statischen Import der Klasse `ConnectionSettingsTbl` verwendet, um diesen Klassennamen nicht für jedes Tabellenattribut schreiben zu müssen. Diese Vereinfachung macht den Code lesbarer. Wir werden jedoch hier im Buch weiterhin die vollqualifizierte Schreibweise wählen, da diese unserer Meinung nach die Trennung zwischen Content-Provider-Attributen und den Spaltennamen der Datenbanktabellen anschaulicher macht.

Exceptions

Falls trotzdem während der Ausführung der Operation etwas schiefgeht, so sollte dieser Fehler möglichst detailliert an den Aufrufenden weitergereicht werden. Als Medium dient dazu immer eine `RuntimeException`. Dies bedeutet im Umkehrschluss, dass der Client eines Content Providers auch unchecked Exceptions behandeln muss. Es

sollte zumindest ein Eintrag in der Log-Datei implementiert werden. Gegebenenfalls müssen bei Auftreten eines Fehlers geeignete »Umkehrmaßnahmen« eingeleitet werden, um den vom Content Provider verwalteten Datenbestand nicht in einem inkonsistenten Zustand zu hinterlassen.

Schreibzugriffe

Die Umsetzung der Schreiboperationen `update` und `delete` erfolgt nach dem gleichen Schema. Für beide Operationen sollten zumindest Implementierungen für den »Allgemeinfall« (alle Datensätze sind betroffen) oder den »Einzelfall« (maximal ein durch seinen Schlüssel innerhalb der URI identifizierter Datensatz ist betroffen) betrachtet werden. Zu beachten ist außerdem, dass die Content-Provider-Operationen alle atomar sind. Eine Transaktionsklammer außerhalb des Content-Providers wird nicht berücksichtigt. Die Änderungs- und Löschoperationen sollten immer die Anzahl der betroffenen Datensätze zurückliefern, damit der Aufrufer erkennen kann, ob alles korrekt ausgeführt wurde.

Die `insert`-Operation wollen wir uns aber noch einmal genauer anschauen. Diese verdeutlicht das generelle Vorgehen zur Realisierung einer Schreiboperation. Darüber hinaus kann sie nur für »Verzeichnis«-URIs ausgeführt werden, da ja noch kein eindeutiger Schlüsselwert für den neuen Datensatz existiert. Der im Erfolgsfall neu erstellte Schlüssel wird als Teil der zum Zugriff auf den neuen Datensatz erforderlichen URI als Ergebnis zurückgegeben.

```
...
@Override
public Uri insert(Uri uri, ContentValues
    initialValues) {
  // nur 'Verzeichnis-URI' erlauben
  if (sUriMatcher.match(uri) != CONN_SETTINGS_ALL) {
    throw new IllegalArgumentException("Unknown URI " +
        uri);
  }

  ContentValues values;
  if (initialValues != null) {
    values = new ContentValues(initialValues);
  }
  else {
    values = new ContentValues();
  }
```

```
  long rowId = db
    .getWritableDatabase()
    .insert(
      ConnectionSettingsTbl.TABLE_NAME,
      ConnectionSettingsTbl.COL_APN_APN,
      values);
  if (rowId > 0) {
    Uri noteUri = ContentUris.withAppendedId(
        ConnectionSettings.Settings.CONTENT_URI, rowId);

    return noteUri;
  }

  throw new SQLException(
    "Failed to insert row into " + uri);
}
...
```

12.8.2 Fazit

Zusammenfassend kann man also die Schritte zur Erstellung und Verwendung eines Datenbank-Content-Providers als folgenden Prozess zusammenfassen:

1. Implementierung einer Testanwendung zur Verprobung aller URIs und Operationen des zu erstellenden Content Providers
2. Beschreibung der zu exponierenden Datenstrukturen in einer eigenen Klasse
   - Definition der Content-URI
   - Definition der Mime-Types der Datentypen
   - Festlegung der Namen der nach außen sichtbaren Attribute
3. Implementierung einer Unterklasse von `ContentProvider`
4. Erstellung eines `UriMatcher` für alle vom Content Provider verwalteten URIs
5. Zuordnung der Content-Provider-Attribute zu Tabellenspalten
6. Implementierung von `getType`
7. Implementierung der restlichen datenbankrelevanten Methoden von `ContentProvider`
8. Registrierung des Content Providers als `<provider>` im Android-Manifest (s. Abschnitt 12.6 auf Seite 202)

12.8.3 Ein Datei-Content-Provider

Als Nächstes wollen wir einen Provider implementieren, um anderen Anwendungen oder Komponenten Zugriff auf eine konkrete Datei zu erlauben.

Hierzu betrachten wir den Content Provider `RoutenBrowser`. Die Provider-API unterstützt nur Zugriffe auf konkrete Dateien, nicht aber auf Verzeichnisse. Daher kann die Datei nur ausgelesen werden, wenn sie durch eine URI eindeutig identifiziert werden kann.

Wir implementieren also noch die Methode `openFile` unseres Routenbrowsers (Listing 12.4).

Listing 12.4 *Dateizugriff per Content Provider*

```
public ParcelFileDescriptor openFile(Uri uri, String mode)
  throws FileNotFoundException {
  if (uriMatcher.match(uri) != DATEI_ID) {
    throw new IllegalArgumentException(
      "openFile nur für Dateizugriffe erlaubt!");;
  }
  try{
    return this.openFileHelper(uri, mode);
  }
  catch (FileNotFoundException e) {
    Log.e(TAG, "Datei nicht gefunden. "+uri,e);
    throw new FileNotFoundException();
  }
}
```

Die URI einer Datei speichern wir zusammen mit dem Titel und einigen weiteren Attributen in einer eigenen Tabelle der Staumelder-Datenbank. Dieses Verfahren wurde im Theorie-Abschnitt bereits beschrieben und kann dem online verfügbaren Quellcode entnommen werden.

Interessant ist an dieser Stelle noch der Zugriff auf eine konkrete Datei aus einer entfernten Anwendung heraus. Dieser Vorgang ist in Listing 12.5 skizziert.

Listing 12.5 *Lesen einer entfernten Datei*

```
Uri zielDateiUri =
  ContentUris.withAppendedId(
    RoutenBrowser.Dateizugriff.CONTENT_URI, 1);

ParcelFileDescriptor routenDateiPD =
  getContentResolver().openFileDescriptor(zielDateiUri,
      "r");
```

```
if( routenDateiPD != null ) {
  try {
    List<String> routenDatei =
      org.apache.commons.io.IOUtils.readLines(
        new FileInputStream(
          routenDateiPD.getFileDescriptor()));
    for (Iterator<String> it = routenDatei.iterator();
         it.hasNext();) {
      String zeile = it.next();
      Log.i(TAG,zeile);
    }
  }
  catch (IOException e) {
    Log.e(TAG,"Datei kann nicht gelesen werden. "+e,e);
  }
  finally {
    try {
      routenDateiPD.close();
    }
    catch (IOException e) {
      Log.e(TAG,e.getMessage(),e);
    }
  }
}
```

13 Exkurs: Lebenszyklen

Hinweis: Die 2. Auflage dieses Buchs erscheint unter dem Titel
»Android 2. Grundlagen und Programmierung«
Ende Mai 2010 im dpunkt.verlag, ISBN 978-3-89864-677-2

Mobile Geräte sollten klein und leicht sein, damit man sie bequem transportieren kann. Die Hersteller verbauen daher meist Akkus, deren Laufzeit bei intensivem Gebrauch des Geräts recht kurz ist. Auch der eingebaute Speicher ist oft recht knapp dimensioniert und ohne zusätzliche Speicherkarte schnell erschöpft. Ein weiterer Flaschenhals ist der Prozessor, den sich alle Anwendungen teilen müssen.

Die Android-Plattform trägt dem Rechnung, indem sie in die Lebensdauer von Prozessen eingreift. Hier unterscheidet sich Android von den meisten anderen Betriebssystemen. Im Falle knapper Ressourcen können Prozesse beendet werden. Dies hat Auswirkungen auf die darin laufenden Anwendungen und deren Komponenten. Als *Lebenszyklus* bezeichnet man verschiedene Zustände von Komponenten. Der Lebenszyklus einer Komponente beginnt mit deren Erzeugung und endet, wenn die Komponente beendet wird.

Da die Android-Plattform Prozesse eigenständig beenden kann, unterliegen die Lebenszyklen der Komponenten auch der Kontrolle durch das Betriebssystem. Datenverlust ist möglich, wenn der Prozess einer Komponente »von außen«, also durch die Android-Plattform, beendet wird.

In diesem Exkurs widmen wir uns dem Lebenszyklus von Prozessen und Komponenten. Wir lernen die Methoden kennen, die die einzelnen Phasen des Lebenszyklus einer Komponente repräsentieren. Wir zeigen Implementierungsmuster, wie man mit dem plötzlichen Beenden einer Komponente umgeht und wie man Datenverlust vermeidet.

13.1 Prozess-Management

Android beendet Prozesse.

Mit Prozessen haben wir uns schon in Kapitel 8 über Hintergrundprozesse beschäftigt. Die Android-Plattform ordnet den laufenden Prozessen verschiedene Prioritäten zu. Im Falle knapper Ressourcen werden Prozesse der geringsten Priorität beendet. Die Prioritäten werden anhand des Zustands der im Prozess laufenden Komponenten vergeben.

Eine Activity, die gerade angezeigt wird, hat eine hohe Priorität. Ihr Prozess darf daher nur im äußersten Notfall beendet werden.

Zum Abschuss freigegeben

Anders verhält es sich mit Anwendungen, die länger nicht verwendet wurden. Wenn keine Activity angezeigt wird und kein Hintergrundprozess in Form eines Broadcast Receivers oder Service läuft, ist der umschließende Prozess ein guter Abschuss-Kandidat. Die folgende Aufzählung listet die verschiedenen Zustände von Prozessen auf. Die Zustände entsprechen Prioritätsstufen aus Sicht der Android-Plattform. Wir beginnen mit der niedrigsten Priorität. Solche Prozesse werden im Falle knapper Ressourcen als Erstes von Android gelöscht.

Leere Prozesse Die Android-Plattform hält Prozesse am Leben, auch wenn alle ihre Komponenten beendet wurden. Die Komponenten werden in einem Cache gespeichert. Dadurch lassen sich Anwendungen sehr viel schneller wieder starten. Leere Prozesse werden bei knappen Ressourcen sofort beendet.

Hintergrundprozesse Prozesse, die keine Activities enthalten, die gerade angezeigt werden oder die keine laufenden Services enthalten, nennt man Hintergrundprozesse. Die genannten Komponenten wurden zwar nicht beendet, sind aber inaktiv und für den Anwender nicht sichtbar. Die ältesten Hintergrundprozesse werden zuerst beendet.

Serviceprozesse Serviceprozesse beinhalten Remote Services, wie wir sie aus Abschnitt 8.3.1 kennen. Sie erledigen reine Hintergrundarbeiten, wie zum Beispiel das Abspielen eines MP3-Musikstücks, und haben keinerlei Verbindung zu anderen Komponenten.

Sichtbare Prozesse Eine Activity kann durch eine andere Activity, die nicht den ganzen Bildschirm einnimmt, überlagert werden. Zwar ist sie dann teilweise noch sichtbar, gilt aber als *inaktiv*, da sie den Fokus verloren hat und nicht auf Anwendereingaben reagiert. Prozesse, die höchstens solche inaktiven Activities enthalten, werden sichtbare Prozesse genannt.

Aktive Prozesse Prozesse, die aktive Komponenten enthalten. Eine aktive Komponente ist

- eine Activity, die im Vordergrund angezeigt wird,
- ein Service, der eine Verbindung zu einer Activity im Vordergrund hat,
- ein Broadcast Receiver, dessen `onReceive`-Methode ausgeführt wird.

13.2 Lebenszyklus von Komponenten

Wir betrachten hier die Komponenten *Activity*, *Service* und *Broadcast Receiver*. Die verschiedenen Zustände, die die Komponenten von ihrer Erzeugung bis zu ihrer Zerstörung durchlaufen, nennt man *Lebenszyklus*.

13.2.1 Lebenszyklus einer Activity

Wenn wir eine Anwendung starten, wird die Start-Activity aufgerufen, die wir im Android-Manifest deklariert haben. Die Activity wird erzeugt und sichtbar gemacht. Beenden wir die Anwendung, wird die Activity zerstört oder im Cache gespeichert. Dies hängt davon ab, ob wir die Methode `finish` der Activity aufrufen. Von der Erzeugung bis zur Zerstörung durchläuft die Activity verschiedene Stadien des Lebenszyklus. Für jedes Stadium gibt es eine Methode in der Klasse `Activity`, die in der eigenen Implementierung überschrieben werden kann. Abbildung 13-1 zeigt den Lebenszyklus einer Activity.

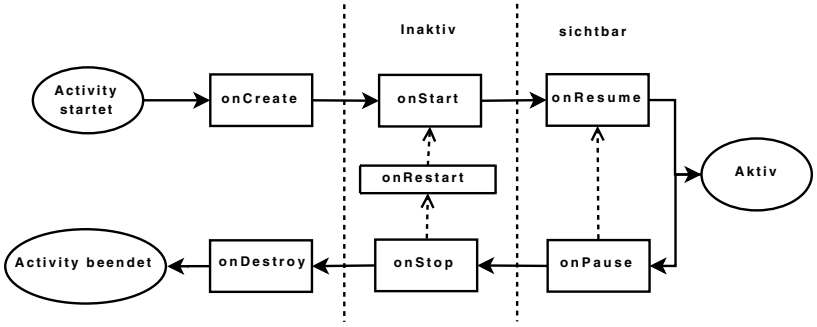


Abb. 13-1
Lebenszyklus einer Activity

Wenn eine Activity nicht existiert, auch nicht im Cache, wird ihre `onCreate`-Methode aufgerufen. Diese Methode haben wir schon kennengelernt und zum Initialisieren der Layouts, der Schaltflächen und des Menüs verwendet. Bis die Activity angezeigt wird, durchläuft sie zusätzlich die Methoden `onStart` und `onResume`. Erst dann ist die Activity vollständig sichtbar und gilt als »aktiv«, reagiert also auf Engaben des Anwenders.

Aktive Activity...

Wird die Activity teilweise durch eine andere Activity überlagert, wird die Methode `onPause` aufgerufen. Die Activity ist zwar noch teilweise sichtbar, aber inaktiv, reagiert also nicht auf Interaktionen des Anwenders. Wird die überlagernde Activity beendet, wird die Methode `onResume` aufgerufen.

... und inaktive Activity

Wird die Activity vollständig durch eine andere überlagert, wird die Methode `onStop` aufgerufen. Die Activity ist inaktiv und nicht mehr

sichtbar. Wird die überlagernde Activity beendet, werden die Methoden `onRestart`, `onStart` und `onResume` aufgerufen.

Activities beenden

Nur wenn die Activity auch aus dem Cache entfernt wird und ihre Ressourcen freigegeben werden, wird die Methode `onDestroy` aufgerufen. Dies kann man mit der Activity-Methode `finish` erzwingen. Dadurch spart man Ressourcen, aber es ist für das Laufzeitverhalten nachteilig, wenn man die Activity doch noch mal anzeigen möchte. Will man die gesamte Anwendung beenden, ruft man auf der Start-Activity, also der Activity, die man im Android-Manifest per Intent-Filter als Start-Activity deklariert hat, die `finish`-Methode auf.

Tabelle 13-1 zeigt die Methoden des Lebenszyklus im Überblick.

Tab. 13-1 *Lebenszyklus-Methoden der Activity*

Methode	Beschreibung
`onCreate(...)`	Die Activity wird erzeugt. Die Methode kann praktisch wie ein Konstruktor verwendet werden. Hier werden alle Initialisierungen der Activity vorgenommen (Menüs, Layout, Vorbelegung von Formularfeldern etc.).
`onStart()`	Wird aufgerufen, wenn die Activity neu erzeugt wird oder wenn sie vorher nicht sichtbar war, aber nun wieder angezeigt wird (wenn man z.B. in der Historie zurückgeht).
`onResume()`	Wird aufgerufen, wenn eine teilweise verdeckte Activity wieder vollständig angezeigt werden soll.
`onPause()`	Die Activity ist nur teilweise sichtbar und teilweise von einer anderen Activity überlagert. Sie ist inaktiv, reagiert also nicht auf Eingaben des Anwenders.
`onStop()`	Die Activity wird beendet und tritt vollständig in den Hintergrund. Sie wird auf den Activity-Stack gelegt, falls sie erneut aufgerufen wird.
`onRestart()`	Die Activity wird wieder aufgerufen, z.B. weil jemand die Zurück-Taste des Android-Geräts gedrückt hat.
`onDestroy()`	Die Activity wird beendet. Entweder durch das System oder weil auf ihr die Methode `finish` aufgerufen wurde. Zur Unterscheidung beider Fälle dient die Methode `isFinishing`. Alle belegten Ressourcen müssen in dieser Methode freigegeben werden.

Achtung! Bei allen hier aufgeführten Methoden des Lebenszyklus muss die jeweilige Implementierung aus der Oberklasse am Anfang der überschriebenen Methode aufgerufen werden, sonst wird eine Exception geworfen.

13.2.2 Lebenszyklus eines Service

Ein Service hat drei Methoden, die zu seinem Lebenszyklus gehören.

Tab. 13-2 Lebenszyklus-Methoden des Service

Methode	Beschreibung
`onCreate()`	Wird beim Erzeugen des Service aufgerufen und kann für Initialisierungen verwendet werden.
`onStart(...)`	Wird aufgerufen, wenn eine Komponente den Service startet (Methode `Context.startService`) oder sich mit ihm verbindet (Methode `Context.bindService`).
`onDestroy()`	Wird aufgerufen, wenn der Service beendet wird. Ein Service kann sich selbst beenden, indem er seine `stopSelf`- oder `stopSelfResult`-Methode aufruft. Alle belegten Ressourcen müssen hier freigegeben werden.

Services werden meistens von außen gesteuert. Sie haben keine Schnittstelle zum Anwender. Andere Komponenten der Anwendung übernehmen die Kommunikation mit dem Service. Oft wird der Service durch eine Komponente gestartet (siehe Abschnitt 8.3.1), kann aber von vielen Komponenten verwendet werden. Solange nur eine dieser Komponenten mittels der Methode `Context.bindService` eine Verbindung zum Service aufgebaut hat, kann der Service nicht gestoppt werden, selbst wenn eine Komponente die Methode `Context.stopService` aufruft.

Nur bei Bedarf verbinden

Es empfiehlt sich, Verbindungen zu einem Service nur dann aufzubauen, wenn man sie auch braucht. So lässt sich bei von mehreren Komponenten parallel genutzten Services der Lebenszyklus besser steuern, da jede Komponente den Service stoppen kann. Dies gibt der Android-Plattform außerdem die Möglichkeit, den inaktiven Service bei Ressourcenknappheit zu beenden. Ein Service ist inaktiv, wenn keine seiner drei Lebenszyklus-Methoden ausgeführt wird und keine Komponente mit ihm verbunden ist. Ansonsten ist er aktiv.

13.2.3 Lebenszyklus eines Broadcast Receivers

Broadcast Receiver kommen mit einer Lebenszyklus-Methode aus.

```
void onReceive(Context context, Intent broadcastIntent)
```

Aktiv nur während onReceive

Die `onReceive`-Methode wird aufgerufen, wenn ein Broadcast Intent eintrifft (siehe Abschnitt 9.2). Der Broadcast Receiver ist nur während der Ausführungsdauer dieser Methode aktiv. Ansonsten ist er inaktiv und kann im Rahmen des Prozess-Managements der Android-Plattform gelöscht werden, um Ressourcen freizugeben.

Kommen wir nun zu einem weiteren Punkt, der eng mit dem Lebenszyklus zu tun hat und nur die Activities betrifft.

13.2.4 Activities: Unterbrechungen und Ereignisse

Für aktive Activities ist so ziemlich alles, was von außen kommt, eine Unterbrechung bzw. ein Ereignis. Beispielsweise gibt der Anwender gerade einen längeren Text ein, was auf der kleinen Tastatur seines Android-Geräts etwas mühselig ist. Und dann passiert's: ein Anruf.

Ein Anruf ist ein Ereignis. Wir geben erst mal einige Beispiele für Ereignisse und Unterbrechnungen:

Unterbrechung:

- Die Zurück-Taste des Android-Geräts wird gedrückt.
- Eine andere Activity wird aufgerufen.
- Eine andere Anwendung wird gestartet.
- Die Anwendung wird beendet.
- Der Bildschirm wird gedreht.
- Ein Systemparameter (z.B. Sprache) wird geändert.
- etc.

Ereignis:

- eingehender Telefonanruf
- Akku ist leer
- Telefon wird ausgeschaltet
- etc.

Lebenszyklus-Methoden zur Vermeidung von Datenverlust

Wir haben eben erfahren, dass die Android-Plattform Prozesse beenden kann. Ein eingehender Telefonanruf wird auf dem Bildschirm angezeigt und führt dazu, dass die gerade angezeigte Activity inaktiv wird. Sind die Ressourcen knapp und die zur Activity gehörende Anwendung hat ansonsten keine aktiven Komponenten, kann es sein, dass die Android-Plattform sie beendet. Dann wären Dateneingaben, die der Anwender

vor dem Telefonanruf gemacht hat, verloren. Glücklicherweise gibt es Mechanismen im Lebenszyklus, die uns dabei helfen, einen ungewollten Datenverlust zu vermeiden.

Merksatz

Eine Activity muss die Methoden ihres Lebenszyklus nutzen, um Anwendereingaben vor einem Datenverlust zu schützen.

onPause() vs. onSaveInstanceState(Bundle outState)

Fehler vermeiden!

Für Activities gibt es noch zwei weitere Methoden, die im Lebenszyklus eine Rolle spielen: die `onSaveInstanceState`-Methode und ihr Gegenstück, die `onRestoreInstanceState`-Methode. `onSaveInstanceState` dient dazu, den Zustand und die Anwendereingaben einer Activity in einem `Bundle` zu speichern, welches einem in der `onCreate`- bzw. in der `onRestoreInstanceState`-Methode wieder zur Verfügung steht.

```
protected void  onCreate(Bundle savedInstanceState)

protected void  onRestoreInstanceState(
    Bundle savedInstanceState)
```

Vorsicht ist dabei geboten, den Zustand einer Activity in der `onSaveInstanceState`-Methode zu speichern. Wie wir uns den Zustand einer Activity und ihre Daten aus den Eingabefeldern merken, zeigen wir in den letzten Abschnitten des Kapitels. Zunächst wollen wir erklären, unter welchen Umständen die `onSaveInstanceState`-Methode aufgerufen wird und unter welchen Umständen nicht.

Wann wird was aufgerufen?

Die `onSaveInstanceState`-Methode wird aufgerufen, wenn eine Activity *unvorhergesehen* inaktiv wird. Dies ist der Fall, wenn Activity »A« durch Activity »B« überlagert wird. Activity A ist ohne eigenes Zutun inaktiv geworden und könnte von der Android-Plattform beendet werden, wenn wenig Ressourcen zur Verfügung stehen.

Würde man Activity A jedoch mittels der Methode `finish` beenden, bevor man Activity B startet, wird die Methode `onSaveInstanceState` nicht aufgerufen. Der Aufruf von `finish` zeigt der Android-Plattform an, dass man sich bewusst war, was man tut, und Android geht davon aus, dass man seine Daten schon in der `onPause`-Methode gesichert hat, falls dies nötig war.

Schauen wir uns den wesentlichen Ausschnitt aus den Lebenszyklen der beiden Activities an. Activity A ruft Activity B auf. Im ersten Fall wird Activity A mit der `finish`-Methode beendet.

Fall 1: mit finish()

```
final Intent i = new Intent(getApplicationContext(),
    StauinfoBearbeiten.class);
startActivity(i);
finish();
```

Fall 2: ohne finish()

```
final Intent i = new Intent(getApplicationContext(),
    StauinfoBearbeiten.class);
startActivity(i);
```

Lebenszyklus mit finish

Der Lebenszyklus beider Activities sieht im Fall 1 wie folgt aus. Wir geben hier die jeweilige Activity (`Activity_A` und `Activity_B`) zusammen mit der durchlaufenen Lebenszyklus-Methode an.

- `Activity_A::onCreate()`
- `Activity_A::onStart()`
- `Activity_A::onResume()`
- `Activity_A::onPause()`
- `Activity_B::onCreate()`
- `Activity_B::onStart()`
- `Activity_B::onResume()`
- `Activity_A::onStop()`

Im Fall 2 haben wir einen kleinen, aber feinen Unterschied:

- `Activity_A::onCreate()`
- `Activity_A::onStart()`
- `Activity_A::onResume()`
- **`Activity_A::onSaveInstanceState()`**
- `Activity_A::onPause()`
- `Activity_B::onCreate()`
- `Activity_B::onStart()`
- `Activity_B::onResume()`
- `Activity_A::onStop()`

Vorsicht mit der Zurück-Taste

Ein weiterer Fall ist der, in dem der Anwender aus Activity A die Activity B aufruft, dort Daten eingibt und dann die Zurück-Taste drückt. Auch hier geht die Android-Plattform davon aus, dass der Anwender weiß, was er tut. Daher wird in Activity B die Methode `onSaveInstanceState` *nicht* aufgerufen.

Würden wir diese Methode nutzen, um die Daten in den Eingabefeldern einer Activity zu speichern, wären sie verloren, sobald der Anwender die Zurück-Taste drückt, da die `onSaveInstanceState`-Methode dann

nicht aufgerufen wird. Die Activity existiert aber derzeit noch mitsamt den eingegebenen Daten. Sobald sich Android aber nun entschließt, aus Mangel an Ressourcen diese nun inaktive Activity zu zerstören, sind die Daten verloren.

Zurück-Taste macht Activity inaktiv!

Navigiert der Anwender nun mittels des Zurück-Knopfs (oder über das Menü) zur Ausgangs-Activity, wird diese Activity neu erzeugt, weshalb Android nicht über `onRestart` die Methode `onStart` aufruft, sondern bei `onCreate` am Anfang des Lebenszyklus startet. Und zwar mit einem leeren Eingabeformular, was für den Anwender recht unverständlich sein dürfte, da er ja den Zurück-Knopf gedrückt hat.

Fassen wir kurz unsere Erkenntnisse in einer Merkliste zusammen:

- `onSaveInstanceState` wird aufgerufen, wenn Activities aufgrund knapper Ressourcen durch die Android-Plattform beendet werden.
- `onSaveInstanceState` wird aufgerufen, wenn die Activity durch eine andere Activity überlagert wird.
- Die Zurück-Taste führt nicht dazu, dass auf der aktuellen Activity die `onSaveInstanceState`-Methode aufgerufen wird.
- Wird eine Activity mit `finish` beendet, wird `onSaveInstanceState` nicht aufgerufen.

Betrachten wir nun folgende Methode:

```
onCreate(Bundle savedInstanceState)
```

Sie wird aufgerufen, wenn die Activity noch nicht existiert. Der Parameter `savedInstanceState` ist null, wenn die Activity während der Laufzeit der Anwendung noch nicht existierte, vorher mit der Methode `finish` beendet wurde oder über die Zurück-Taste verlassen wurde.

Wurde die Activity zuvor ungewollt beendet, enthält der Parameter `savedInstanceState` die Daten, die man in der Methode `onSaveInstanceState(Bundle outState)` im Parameter `outBundle` gespeichert hat. Den Parameter `outState` erhält man in der `onCreate`-Methode samt den darin gespeicherten Daten zurück, wenn die Activity später wieder aufgerufen wird.

Den Zustand speichern, nicht die Daten!

Wir haben also für den Fall, dass die Activity von Android beendet wird, einen Mechanismus, um etwas zu speichern. Gedacht ist dies, um den Zustand der Activity zu speichern, nicht ihre Daten.

»Zustand« bedeutet z.B., dass der Anwender in einer Kalender-Activity den Ansichtsmodus »Wochenansicht« gewählt hat und sich gerade eine Woche im nächsten Jahr anzeigen lässt. Kehrt er zur Activity zurück, erwartet er, dass er wieder die Wochenansicht statt die Monatsansicht hat und die betreffende Woche im nächsten Jahr angezeigt

wird. Oder wir haben einen Texteditor, und als Zustand speichern wir die Schriftart und die Cursorposition im Text.

Anwendungsdaten selbst speichern

Die Daten selbst sollten normalerweise persistiert werden, indem man einen Content Provider (siehe Kapitel 12) verwendet und die vom Anwender in den View-Elementen der Activity eingegebenen Daten in einer Datenbank oder im Dateisystem speichert.

Zum Speichern der Daten eignet sich die `onPause`-Methode. Sie wird immer aufgerufen, wenn die Activity vom aktiven in den inaktiven Zustand wechselt, wodurch die Activity Gefahr läuft, von der Android-Plattform beendet zu werden. Daher ist es sinnvoll, die Daten hier zu speichern. Da die Methode parameterlos ist, haben wir kein Bundle, welches uns Arbeit abnimmt, indem wir es als Datencontainer verwenden können. Wir müssen die Daten persistieren.

Natürlich können wir auch in der `onPause`-Methode den Zustand einer Activity speichern und nicht nur ihre Daten. Meistens wird man das auch tun, da z.B. der oben erwähnte Kalender auch genauso wieder angezeigt werden soll, wenn man die Zurück-Taste am Android-Gerät verwendet.

Tipp

Meist werden wir die Daten einer Activity in der `onPause`-Methode persistieren. Daher ist es sinnvoll, im Menü einen Punkt »Abbrechen« einzubauen, um dem Anwender die Möglichkeit zu geben, die Activity ohne Speichern seiner bisher gemachten Eingaben zu verlassen.

Eine weitere Auswirkung auf Activities hat die Änderung von Systemparametern oder wenn wir die Ausrichtung des Bildschirms ändern (ihn drehen).

Vorsicht beim Ändern der Perspektive

Eine Änderung der Ausrichtung des Bildschirms führt zum Beenden aller aktiven und inaktiven Activities.

Was die Änderung von Systemparametern betrifft, so geht die Android-Platform davon aus, dass grundsätzlich jede Activity Systemparameter verwenden kann und diese dazu verwendet, Daten oder Darstellung der Activity zu beeinflussen. Werden Systemparameter geändert, ist die Activity nicht mehr aktuell und muss neu angezeigt werden. Die Android-Plattform kann nicht wissen, welche Activities von der Änderung von Systemparametern betroffen sind. Daher werden alle aktiven und inaktiven Activities beendet. Die Activities durchlaufen den Rest ihres Lebenszyklus bis zur `onDestroy`-Methode. Ruft der Anwender die Activity wieder auf oder kehrt zu ihr zurück, so wird sie mittels `onCreate` neu erzeugt. Für den Anwender ist dies nicht transpa-

rent, und er erwartet, dass die Activity so angezeigt wird, wie er sie eben gesehen hat. Daher müssen die Daten und Zustände der Activity in der `onPause`-Methode gespeichert werden, um Datenverlust zu vermeiden.

Achtung!

Beim ersten verfügbaren Android-Gerät, dem G1 von HTC, führt das Aufklappen der Tastatur zur Änderung der Bildschirmperspektive. Dies bewirkt, wie oben gesagt, dass alle Activities (aktive und inaktive) beendet werden.

13.3 Beispiele aus der Praxis

Wir schauen uns nun drei Beispiele an, die das Gesagte in der Praxis demonstrieren.

13.3.1 Beispiel: Kalender-Activity

Zustand eines Kalenders speichern

Angenommen, die Activity stellt einen Kalender dar. Über das Menü lässt sich einstellen, ob Wochen- oder Monatsansicht gewünscht ist. Standardeinstellung ist die Monatsansicht. Wir wollen die gewählte Ansichtsweise speichern, falls die Activity inaktiv wird, nehmen aber in Kauf, dass sie in der Standardansicht angezeigt wird, wenn jemand die Zurück-Taste gedrückt hat. Folglich reicht uns die `onSaveInstanceState`-Methode, um im Bundle `outState` den Zustand (Wochen- oder Monatsansicht) speichern zu können.

Listing 13.1 *Einstellungen mittels `onSaveInstanceState` speichern*

```
public class KalenderActivity extends Activity {

  private static final int WOECHENTLICH = 1;
  private static final int MONATLICH = 2;

  private int ansicht = MONATLICH;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
     ...
    if (savedInstanceState != null) {
      ansicht = savedInstanceState.getInt("ansicht");
      // TODO: stelle Ansicht wieder her
    }
    ...
  }
```

```
  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    final boolean result =
        super.onCreateOptionsMenu(menu);
    menu.add(0, WOECHENTLICH, 0, "Wöchentlich");
    menu.add(0, MONATLICH, 1, "Monatlich");
    return result;
  }

  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
      case WOECHENTLICH:
        ansicht = WOECHENTLICH;
        return true;
      case MONATLICH:
        ansicht = MONATLICH;
        return true;
    }
    return super.onOptionsItemSelected(item);
  }

  @Override
  protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    outState.putInt("ansicht", ansicht);
  }
}
```

Wie wir in Listing 13.1 sehen, setzt der Anwender den Zustand (monatliche oder wöchentliche Kalenderansicht) über das Optionsmenü. In der Methode `onSaveInstanceState(Bundle outState)` können wir den Parameter `outState` verwenden, um die Ansicht der Activity wiederherzustellen.

13.3.2 Beispiel: E-Mail-Programm

In einem E-Mail-Programm wird man nicht wollen, dass der gerade eingegebene E-Mail-Text aufgrund der oben genannten Ereignisse oder Unterbrechungen (Telefonanruf, Zurück-Taste gedrückt etc.) verloren geht. Will man als Entwickler eingegebene Daten vor Verlust schützen, muss man sie in der `onPause`-Methode speichern und in der `onResume`-Methode wieder in die View ihrer Activity zurückspielen.

Content Provider zum Speichern verwenden

Für das E-Mail-Programm setzen wir voraus, dass ein Content Provider existiert, der E-Mail-Entwürfe in einer Datenbank speichert.

Wir wollen auf jeden Fall verhindern, dass einmal gemachte Eingaben verloren gehen. Daher nutzen wir die `onPause`-Methode, da diese immer aufgerufen wird, auch wenn die Zurück-Taste gedrückt wird. Die Anwendung soll dann die bisher gemachten Eingaben als Entwurf über den Content Provider in eine Datenbank oder in eine Datei schreiben.

Listing 13.2
Einstellungen mittels `onSaveInstanceState` speichern

```
public class EMailActivity extends Activity {

  public static final Uri CONTENT_URI =
      Uri.parse("content://de.visionera.email/drafts");
  Cursor mCursor;
  final String[] COLUMNS =
      new String[] {"empfaenger", "text", "zeitstempel"};

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
     ...
    mCursor = managedQuery(CONTENT_URI, COLUMNS, null,
          null, "zeitstempel DESC");
  }

  @Override
  protected void onPause() {
    super.onPause();
    ContentValues values = new ContentValues(3);
    // TODO: Eingabefelder auslesen uns setzen:
    // values.put(COLUMNS[0], Empfaengeradresse...);
    // values.put(COLUMNS[1], E-Mail-Text...);
        values.put(COLUMNS[2], System.currentTimeMillis());
    getContentResolver().insert(CONTENT_URI,
              values);
  }

  @Override
  protected void onResume() {
    super.onResume();
    if (mCursor != null) {
      mCursor.moveToFirst();
      String empfaenger = mCursor.getString(1);
      String text = mCursor.getString(2);
      // TODO: Update der Eingabefelder der View...
    }
  }
}
```

Daten in der richtigen Methode speichern und wiederherstellen

Den Quellcode des Content Providers sparen wir uns an dieser Stelle. Die Activity zeigt immer den letzten Entwurf an, egal ob sie durch den Anwender oder durch Android beendet wurde. Wir nutzen die Methoden `onPause` und `onResume`, um den Entwurf zu speichern bzw. um die Eingabefelder »Empfängeradresse« und »E-Mail-Text« zu füllen. Grundsätzlich wäre dies der richtige Weg, sämtlichen Problemen aus dem Weg zu gehen. Die Daten würden immer gespeichert, sobald eine Activity inaktiv wird. Der Mehraufwand bei der Implementierung ist natürlich nicht gerade gering. Egal ob Content Provider, Datenbank oder Dateisystem, man hat einigen Programmieraufwand, um die Daten zu speichern und später wieder zu lesen. Es gibt einen weniger sauberen, aber schnelleren Weg…

13.3.3 Beispiel: Quick-and-dirty-Alternative

Wem das Speichern im Dateisystem oder in einer Datenbank zu mühselig ist, begnügt sich mit einer deutlich weniger aufwendigen Alternative. Wir haben den Quellcode auf die beiden wesentlichen Methoden reduziert.

Listing 13.3 *SharedPreferences-Objekt als Zustandsspeicher verwenden*

```
@Override
protected void onPause() {
  super.onPause();
  SharedPreferences.Editor editor =
      getPreferences(MODE_PRIVATE).edit();
  // TODO: Eingabefelder auslesen und setzen:
  // editor.putString("recipient", Empfaengeradresse...);
  // editor.putString("body", E-Mail-Text...);

  if (isFinishing()) {
    editor.putString("isFinishing", "true");
  }
  editor.commit();
}

@Override
protected void onResume() {
  super.onResume();
  SharedPreferences prefs =
      getPreferences(MODE_PRIVATE);
  String recipient = prefs.getString("recipient", null);
  String body = prefs.getString("body", null);

  String isFinishing =
      prefs.getString("isFinishing", null);
```

```
  if (isFinishing != null) {
    // TODO: Aktualisierung der Eingabefelder der View...
  }
}
```

Ein Android-Objekt missbrauchen

Die Klasse `SharedPreferences` haben wir schon in Abschnitt 6.5 kennengelernt. Sie ist eigentlich dafür gedacht, Einstellungsparameter einer Anwendung zu speichern. Also im E-Mail-Programm z.B. Schriftart und Schriftgröße.

Natürlich kann man alles mögliche über die zur Verfügung gestellten Methoden des Editors speichern, also auch längere Texte. Das Speichern von Objekten, Byte-Arrays, Bilder, Videos etc. ist glücklicherweise nicht möglich.

In der `onResume`-Methode holen wir uns schließlich die Werte zurück und könnten sie in die Textfelder der Activity setzen. Die Methode `getPreferences` gehört zur Activity und liefert uns das `SharedPreferences`-Objekt. Der Parameter `MODE_PRIVATE` zeigt an, dass wir das Objekt nur in unserer Anwendung verwenden und es für andere Anwendungen bzw. Prozesse nicht sichtbar sein soll. Dies funktioniert im vorliegenden Android-Release Version 1.0 auch noch nicht, wodurch die Paramter `MODE_WORLD_READABLE` und `MODE_WORLD_WRITEABLE` keine Wirkung haben.

Ein kleiner Trick...

Wir haben in dem Listing noch einen kleinen Trick mit Hilfe der Methode `isFinishing` eingebaut. Die Methode gehört zur Activity und wird am besten in der `onPause`-Methode verwendet. Wir können dadurch unterscheiden, ob die Activity mittels der `finish`-Methode beendet wurde, also ganz regulär verlassen wurde, oder ob sie inaktiv gemacht wurde, indem sie durch eine andere Activity überlagert oder sogar durch Android ganz zerstört wurde. Folglich nutzen wir die Methode, um die alten Werte der Activity nur dann wiederherzustellen, wenn sie nicht mit `finish` beendet wurde.

14 Iteration 6 – Netzwerk und Datenübertragung

Hinweis: Die 2. Auflage dieses Buchs erscheint unter dem Titel
»Android 2. Grundlagen und Programmierung«
Ende Mai 2010 im dpunkt.verlag, ISBN 978-3-89864-677-2

Ein mobiles Gerät ist kein PC, der zu Hause oder im Büro seinen festen Standort mit einer hohen Netzwerkverfügbarkeit hat. Wir sind mit einem Android-Gerät oft in Bewegung, wechseln zwischen WLAN-Netzen (zu Hause, bei Freunden, im Café oder im Büro) und der Internetverbindung über unseren Netzanbieter hin und her. Dazwischen reißt die Netzwerkverbindung ganz ab, weil wir mit dem Aufzug in die Tiefgarage fahren oder durch einen Tunnel.

Eine Anwendung, die Netzwerkkommunikation nutzt, muss auch berücksichtigen, dass Datenübertagung eine langwierige Sache sein kann.

Android berücksichtigt diese Schwierigkeiten und stellt uns Hilfsmittel zur Verfügung, die wir bei der Implementierung nutzen können. Auch kommen in dieser Iteration unsere Kenntnisse aus Kapitel 8 zur Anwendung. Wir werden nun ein Implementierungsmuster anhand von Beispielcode liefern, das die genannten Probleme berücksichtigt.

14.1 Iterationsziel

Unser Hauptaugenmerk wird nicht auf dem Aufbau einer Internetverbindung liegen. Dies ist Bestandteil des Java-SDK, und darüber findet man ausreichend Literatur und jede Menge Beispiele. Wir wollen uns vielmehr hier mit zwei Lösungen für den Datenaustausch über das Internet beschäftigen, die aus unserem Staumelder-Beispiel ohne Schwierigkeiten in andere Anwendungen übertragen werden können. Daher werden wir im Theorieteil ganz allgemein beschreiben, was bei Netzwerkprogrammierung auf mobilen Endgeräten zu beachten ist und wo die Probleme liegen. Im Praxisteil stellen wir dann zwei Implementierungsmuster anhand von Codebeispielen vor.

Zwei Beispiele

Wir werden uns zunächst mit dem Emulator beschäftigen und sehen, welche Möglichkeiten er bietet, Netzwerkverbindungen auf dem Entwicklungsrechner zu simulieren.

Ziele

Unsere Ziele für diese Iteration sind:

- mit dem Emulator eine Netzwerkverbindung aufzubauen und zu beenden.
- zu wissen, welche Netzwerkklassen im Android-SDK enthalten sind.
- die Probleme bei mobiler Netzwerkkommunikation zu verstehen.
- Anwendungen mit Datenübertragung programmieren zu können.

14.2 Theoretische Grundlagen

14.2.1 Das Emulator-Netzwerk

Der Android-Emulator simuliert nicht nur die Laufzeitumgebung (die DVM) und das Dateisystem, sondern ein komplettes Android-Gerät, inklusive der Netzwerkkomponenten. Um das Netzwerk des Rechners, auf dem die Entwicklungsumgebung läuft, vom Netzwerk des Android-Emulators zu trennen, besitzt der Emulator einen virtuellen Router. Dieser besitzt die IP-Adresse (Gateway-Adresse) *10.0.2.1*.

Getrennte Netzwerke

Damit haben wir auf einem Rechner zwei Netzwerke, die über den Router des Emulators in Verbindung stehen. Aus Sicht des Emulators (des Clients) adressiert die IP-Adresse *127.0.0.1* bzw. *localhost* folglich den Emulator selbst.

Auf unserem Entwicklungsrechner hingegen, auf dem der Emulator läuft, adressiert die IP-Adresse *127.0.0.1* den Rechner (den Host) selbst.

Router

Der Router schottet das Netzwerk des Emulators gegen das Netzwerk des Rechners ab, ist aber selbst Teil des Netzwerks. Um eine Verbindung zwischen beiden Netzwerken herstellen zu können, wurde dem Router eine feste IP-Adresse zugewiesen, die auf das Loopback-Interface des Rechners zeigt. Mit Hilfe dieser IP-Adresse kann sich der Emulator zum Beispiel mit einem Server verbinden, der auf dem Entwicklungsrechner läuft.

> Wenn sich der Emulator mit einem Server im Netzwerk des Entwicklungsrechners verbinden will, muss er die vorgegebene IP-Adresse *10.0.2.2* verwenden.

Den Emulator erreichen

Um den Emulator von unserem Rechner aus erreichen zu können, wurde ihm ebenfalls eine feste IP-Adresse vergeben. Über diese können wir eine direkte Verbindung zum Emulator und somit zu einem Programm,

welches im Emulator läuft, aufnehmen. Es handelt sich um die Adresse *10.0.2.15*. Tabelle 14-1 zeigt die IP-Adressen in einer Übersicht.

IP-Adresse	Beschreibung
127.0.0.1	Abhängig davon, in welchem Netzwerk mein Programm läuft (Emulator oder Rechner)
10.0.2.1	IP-Adresse des Routers (Gateway-Adresse)
10.0.2.2	IP-Adresse des Rechners, auf dem der Emulator läuft (Entwicklungsrechner)
10.0.2.15	IP-Adresse des Emulators
10.0.2.3	Erster DNS-Server des Routers
10.0.2.4-6	Weitere DNS-Server des Routers

Tab. 14-1
IP-Tabelle des Emulators

Da der Android-Emulator die Netzwerkverbindung des Rechners nutzt, auf dem er läuft, wird es keine Probleme geben, wenn man sich vom Emulator aus über die IP-Adresse *10.0.2.2* mit einem Server auf dem Entwicklungsrechner verbindet oder eine Verbindung ins Internet herstellt.

Probleme mit der Firewall?

Es kann Probleme mit der Firewall geben, wenn ein Programm (z.B. der Testserver auf dem Rechner) mit dem Android-Emulator über das Netzwerk kommunizieren will. Dann muss man Folgendes berücksichtigen:

- Ein Verbindungsaufbau mit dem Emulator (IP: *10.0.2.15*) kann durch die Firewall des Rechners, auf dem der Emulator läuft, abgeblockt werden.
- Ein Verbindungsaufbau kann durch eine Firewall im Netzwerk des Rechners abgeblockt werden.

Nur TCP und UDP

Derzeit werden nur die Netzwerkprotokolle *TCP* und *UDP* unterstützt. Die Verwendung eines »Ping«, um zu testen, ob ein Rechner noch ansprechbar ist, ist nicht möglich, da dies über *ICMP* (*Internet Control Message Protocol*) läuft.

Android-Geräte sind keine Server.

Die oben genannten IP-Adressen gelten nur für den Emulator. Auf einem echten Gerät gelten sie nicht. Wir kommen damit auf ein Problem bei mobilen Endgeräten: Sie sind reine Clients und keine Server. Aber mehr dazu im nächsten Abschnitt.

14.2.2 Die Internet-Einbahnstraße

Kein Server

Mobiltelefone haben keine feste IP-Adresse, über die sie von außen wie ein Server erreichbar sind. Vermutlich wird es noch Jahre dauern, bis jedes netzwerkfähige Gerät mit einer festen IP-Adresse ausgeliefert wird und jederzeit von außen ansprechbar ist. Bis es so weit ist und auch jeder Kühlschrank, Fernseher und Toaster fester Bestandteil des Internets ist, verhält sich das Mobiltelefon wie die meisten internetfähigen PCs zu Hause auf dem Schreibtisch: wie eine Internet-Einbahnstraße.

Immer erreichbar

Diesen Umstand nehmen wir bei unserem heimischen PC noch billigend in Kauf, beim Mobiltelefon sind wir anderes gewöhnt. Wir haben eine Telefonnummer und werden darüber angerufen. Eine E-Mail dagegen, verschickt über das Internet, erreicht uns jedoch nicht automatisch. Zwar gibt es inzwischen Push-Mail-Systeme, aber die sind für E-Mail und kosten meistens Geld. Wir haben noch eine große Lücke zwischen Sprachübertragung (Prozess wird von außen gestartet) und der Datenübertragung (man muss den Prozess selbst starten).

Gleichberechtigung

Was wir wollen, ist von einem Android-Gerät eine Verbindung zu einem anderen Android-Gerät aufzubauen.

Derzeit wird so etwas von den Netzbetreibern noch nicht unterstützt. Zwar stehen wir mit dem Mobiltelefon in (fast) ständiger Verbindung mit dem Netzbetreiber, da wir meist in mehreren Zellen (Antennenmasten) gleichzeitig eingebucht sind, aber eine Datenleitung zum Zielgerät kann nicht automatisch aufgebaut werden. Als Folge ist es nicht möglich, Daten von einem Mobiltelefon direkt an ein anderes Mobiltelefon zu senden. Dabei könnte man den Notification Manager gut nutzen, um über das Internet Notifications an jemanden zu schicken. Am Gerät leuchtet dann die Diode, und man weiß, dass neue Daten vorliegen (ein Freund ist in der Nähe, der Urlaubsantrag wurde genehmigt etc.). Mit SMS funktioniert es schließlich auch…

Lösung?

Eine Lösung für das Problem lässt sich tatsächlich mit SMS realisieren. Wir können einen Broadcast Receiver schreiben, der auf eingehende SMS reagiert. Die SMS werden von einem Server über ein SMS-Gateway verschickt. Der Server dient als Vermittler für den Datenaustausch zwischen den mobilen Endgeräten. Bei bestimmten SMS (z.B. mit einem vereinbarten Schlüssel im SMS-Text) startet eine bestimmte Anwendung und ruft die Daten aktiv vom Server ab. Leider kosten SMS in aller Regel Geld.

SMS sind bisher die einzige Möglichkeit, um Anwendungen auf anderen Mobiltelefonen zu starten.

Eine zweite Lösung könnten Peer-to-Peer-Netzwerke sein. Das Android-Gerät baut eine dauerhafte Verbindung zu einem Vermittlerservice auf, der die Daten und Nachrichten an den Empfänger weiterleitet. *XMPP* und *JINGLE* sind bekannte Protokolle für Peer-to-Peer-Kommunikation. Google stellt mit *GTalk* ein eigenes Peer-to-Peer-Netzwerk zur Verfügung. Das Thema würde den Rahmen des Buchs sprengen. Wir beschränken uns daher hier auf das aktive Aufbauen von Internetverbindungen, um Daten an einen Server zu senden oder Daten von einem Server abzuholen.

14.2.3 Androids Netzwerkunterstützung

Die Android-Plattform wird mit der *HttpComponents*-Bibliothek von *Jakarta Commons* ausgeliefert. Jakarta Commons ist ein Projekt der Apache Software Foundation. Die Klassen dazu finden sich im Paket `org.apache.http` bzw. in den Paketen darunter. Uns interessiert besonders ein Teilprojekt von Jakarta Commons, der *HttpClient*.

Eine ausführliche Dokumentation zum Jakarta-Commons-Projekt Http-Client findet man auf den Seiten von Apache [9]. Die Klassen des Http-Client unterstützen die Netzwerkprogrammierung.

Netzwerkverbindung überwachen

Android selbst steuert auch einige Klassen bei, die dabei helfen, den Status der Netzwerkverbindung zu überwachen. Wir müssen bedenken, dass es relativ normal ist, dass eine Netzwerkverbindung zusammenbricht. Betreten wir einen Aufzug, gehen in die Tiefgarage oder fahren mit dem Zug über Land, werden wir wahrscheinlich das ein oder andere Mal die Netzwerkverbindung verlieren.

Wenn eine Anwendung, wie z.B. ein Chat-Client, darauf angewiesen ist, eine dauerhafte Internetverbindung zu haben, so müssen wir auf so ein Unterbrechungsereignis reagieren können. Wir stellen hier zwei Android-Hilfsklassen vor, die für Anwendungen mit dauerhafter Netzwerkverbindung hilfreich sind.

android.net.ConnectivityManager

Kostenlose Netzwerküberwachung

Beim *Connectivity Manager* handelt es sich um eine der Manager-Klassen aus der Schicht des Anwendungsrahmens (vgl. Abb. 2-1) der Android-Plattform.

Der Connectivity Manager läuft im Hintergrund und überwacht den Zustand des Netzwerks. Ist keine Netzwerkverbindung mehr möglich, sendet er einen Broadcast Intent, den wir mit unserer Anwendung abfangen können, indem wir einen Broadcast Receiver (`android.content.BroadcastReceiver`) einsetzen. Wir können auf den Wegfall der Netzwerkverbindung reagieren, indem wir gar nicht erst

versuchen, eine Netzwerkverbindung aufzubauen. Oder wir informieren den Anwender darüber. Dies ist leicht möglich, da wir einen Broadcast Intent empfangen, wenn die Netzwerkverbindung wieder verfügbar ist.

Automatisches Failover

Der Connectivity Manager läuft im Hintergrund und sorgt für eine Ausfallsicherung der Netzwerkverbindung. Bricht eine Netzwerkverbindung weg, versucht er automatisch zur nächsten Netzwerkverbindung zu wechseln. Beim Betreten eines Gebäudes kann es sein, dass die Netzwerkverbindung nicht mehr über das GSM-Modul via GPRS oder UMTS läuft, sondern über ein WLAN.

Den Connectivity Manager erhalten wir mittels

```
ConnectivityManager cm =
    Context.getSystemService(
        Context.CONNECTIVITY_SERVICE)
```

Über ihn bekommen wir in der zweiten wichtigen Klasse Informationen über das Netzwerk.

android.net.NetworkInfo

Mittels der Methoden

```
NetworkInfo   getActiveNetworkInfo()
NetworkInfo[] getAllNetworkInfo()
NetworkInfo   getNetworkInfo(int networkType)
```

Nützliche Netzwerkinformationen

des Content Managers erhalten wir Objekte vom Typ `NetworkInfo`. Diese geben uns wichtige Hinweise, die wir nutzen können, wenn wir eine Verbindung aufbauen oder aufgebaut haben. Die wichtigsten sind:

- `getState`: liefert den Zustand des Netzwerks als `NetworkInfo.State`-Objekt, durch das wir etwas über die Phase des Aufbaus einer Verbindung erfahren
- `isConnected`: liefert true, wenn `getState NetworkInfo.State.CONNECTED` liefert. Nur jetzt kann man eine Verbindung aufbauen.
- `getType`: liefert den Netzwerktyp, über den die Internetverbindung erfolgt (WLAN oder GSM) als `int`
- `getTypeName`: liefert den Netzwerktyp als String

14.2.4 Arten der Netzwerkübertragung

Zwei Methoden

Wenn wir von Netzwerkkommunikation sprechen, können wir zwei grundsätzliche Verfahren unterscheiden:

- *Verfahren 1:* Man möchte einmalig Daten an einen Server schicken oder von einem Server abrufen.
- *Verfahren 2:* Man möchte eine dauerhafte Verbindung zu einem Server aufrechterhalten, um Daten auszutauschen.

Staumeldung im Hintergrund verschicken

Kommen wir zu unserem Staumelder-Beispiel. Wird vom Anwender eine Staumeldung verschickt, dann wird eine Verbindung zum Server aufgebaut, die Staudaten gesendet und abschließend die Verbindung wieder beendet. Dies sollte ein Service im Hintergrund erledigen.

Bei einem Chat-Client würde man anders vorgehen. Eine dauerhafte Verbindung wäre nötig, und sowohl der Server, der als Vermittler zwischen den Chat-Teilnehmern dient, als auch der Client warten, bis wieder Daten eintreffen.

Stauinfos periodisch abrufen

Übertragen auf unseren Staumelder heißt dies: Wir geben eine Route ein und fahren los. Aktuelle Staumeldungen werden angezeigt und periodisch beim Server nachgefragt, ob neue Meldungen vorliegen. Wir brauchen also eine dauerhafte Verbindung. Da wir über Land und durch Tunnel fahren, wird die Verbindung bisweilen abreißen und muss sich ohne unser Zutun neu aufbauen.

Vorsicht: Kostenfalle!

Wir möchten diesen zweiten Fall besonders hervorheben. Noch ist es so, dass Datenübertragung relativ teuer ist. Die meisten Mobilfunkanbieter rechnen in 10-Kilobyte-Datenblöcken ab. Das heißt, jede Datenübertragung nach Methode 1 kostet Geld. Wenn ein Megabyte Daten 40 Cent kostet, dann kostet jeder Verbindungsaufbau 0,4 Cent. Würden wir nun eine Anwendung nach Methode 1 schreiben, die jedesmal eine Verbindung zum Server aufbaut, um nach neuen Daten zu schauen, dann kann das teuer werden. Ein E-Mail-Programm, welches alle drei Minuten nach neuen Mails schaut, käme so in 30 Tagen auf 57,60 Euro!

Andererseits kann Methode 2 auch ungünstig sein. Wenn wir einmal am Tag unsere Arbeitszeit mit dem Android-Gerät erfassen bzw. an den Arbeitgeber senden, aber ständig beim Kunden abwechselnd im Keller im Serverraum und im ersten Stock arbeiten müssen, haben wir evtl. so viele Funklöcher, dass auf Dauer auch recht hohe Kosten entstehen. Und Funklöcher sind wirklich keine Seltenheit.

Wir geben für beide Fälle ein Praxisbeispiel, welches man auf eigene Anwendungsentwicklungen übertragen kann.

14.3 Netzwerken in der Praxis

Es ist möglich, zwei Android-Emulatoren zu starten und eine direkte Verbindung zwischen ihnen aufzubauen. Wir verweisen hier auf die

Android-Dokumentation, da dazu ein Port-Redirect definiert werden muss. Beide Emulatoren stellen dieselbe IP-Adresse mit demselben Port nach außen zur Verfügung. Sie sind folglich nicht unterscheidbar. Werden sie jedoch von außen über unterschiedliche Ports angesprochen, sind sie unterscheidbar. Intern muss einer der Emulatoren den geänderten Port wieder zurück auf den Port 80 lenken (redirect).

Tipp!

Sie können ein Funkloch (Abreißen der Netzwerkverbindung) simulieren, indem Sie im Emulator die Taste »F8« drücken. Damit können Sie wechselweise das Netzwerk ein- und ausschalten.

14.3.1 Verfahren 1: Stau melden

Vorbereitung

Erst die Berechtigung setzen

Als vorbereitende Maßnahme fügen wir zunächst dem Android-Manifest die notwendige Berechtigung zum Aufbau von Internetverbindungen hinzu:

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

Als Nächstes überlegen wir uns, wie die Netzwerkkomponente des Staumelders aufgebaut sein muss. Dazu können wir uns auf die Erkenntnisse der früheren Kapitel stützen. Da Netzwerkoperationen nichts mit der Anzeige der Daten in Activities zu tun haben, sondern diese Daten nur liefern, sollten sie von den Activities entkoppeln. Netzwerkoperationen sind langlaufende Programmteile, müssen also auch aus diesem Grund entkoppelt werden. Wir entschließen uns, einen Service als zentrale Stelle für die Netzwerkoperationen zu implementieren und die Netzwerkkommunikation selbst in Threads zu realisieren. Zusammen mit einer Fortschrittsanzeige (Progress-Dialog) während der Wartezeit, die bei der Netzwerkkommunikation entsteht, und einem Handler für den Callback aus dem Thread haben wir eine saubere Implementierung:

- Wir nutzen einen Service für eine bestimmte Aufgabe.
- Ein Thread im Service erledigt den langlaufenden Programmteil.
- Die Activity übernimmt die Anzeige des Progress-Dialogs.
- Über einen Callback-Handler informieren wir die Activity, wenn die Datenübertragung fertig ist.
- Wir riskieren keinen ANR.

Der Netzwerkservice

Schauen wir uns zunächst den Service an:

Listing 14.1 *Beispielimplementierung für einen Netzwerkservice*

```
public class NetzwerkService extends Service {

  private static final String url =
     "http://10.0.2.2:8081/staumelderserver/" +
     "StaumelderService";

  private final IBinder stauServiceBinder = // (1)
      new StauLocalBinder();

  public class StauLocalBinder extends Binder {
    public NetzwerkService getService() {
        return NetzwerkService.this;
    }

    public void staumeldungAbschicken(final Handler // (2)
        threadCallbackHandler, final GpsData gpsPosition,
        final StauPosition stauPosition, final String
        stauUrsache) {
      Thread thread = new Thread() { // (3)
        @Override
        public void run() {
          long result =
             _staumeldungAbschicken(gpsPosition,
                 stauPosition, stauUrsache);
          Message msg = new Message();
          Bundle bundle = new Bundle();
          bundle.putLong("stauId", result);
          msg.setData(bundle);
          threadCallbackHandler.sendMessage(msg); // (4)
        }
      };

      // starte den gerade definierten Thread:
      thread.start();
    }
  }

  @Override
  public IBinder onBind(Intent intent) {
    return stauServiceBinder;
  }
```

```
  // (5)
  private long _staumeldungAbschicken(final GpsData
      gpsPosition, final StauPosition stauPosition,
      final String stauUrsache) {

    // hier erfolgt die HTTP-Verbindung zum Server.
  }
}
```

Einen IBinder verwenden

Kern der Klasse ist der IBinder `stauServiceBinder`, der uns später die Verbindung zum Service ermöglicht (1). Er implementiert unsere Methode `staumeldungAbschicken` (2). In der Methode starten wir einen Thread (3). In ihm wird die Datenübertragung zum Server mit Hilfe der Methode `_staumeldungAbschicken` erledigt (5). Dadurch wird die Methode nicht blockiert und erledigt die Arbeit im Hintergrund. Nachdem die Daten übertragen wurden, stellen wir mit Hilfe des Handlers `threadCallbackHandler` eine Message in die Message Queue des Threads, zu dem der Handler gehört (siehe Abschnitt 8.3.2) (4). Der Handler gehört zum UI-Thread, der die Activity beherbergt, die den Service verwendet. Wir bekommen also in der Activity Rückmeldung, wenn die Datenübertragung erfolgt ist.

Die Activity »StaumeldungErfassen«

Schauen wir uns die relevanten Teile der Activity an.

Listing 14.2 *Activity `StaumeldungErfassen`*

```
public class StaumeldungErfassen extends Activity {

  private ProgressDialog fortschrittDlg;
  private Stauverwaltung stauVerwaltung; // (1)

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.staumeldung_erfassen);
         stauVerwaltung = new StauverwaltungImpl(
        this.getApplicationContext());
    ...
  }

  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
      case ABSCHICKEN_ID:
             stauMelden();
             return true;
```

```
      case ...
    }
    return super.onOptionsItemSelected(item);
  }

  private void stauMelden() {
    final Spinner stauUrsache =
        (Spinner)findViewById(R.id.stauUrsache);
    final RadioGroup aktuellePosition =
       (RadioGroup)findViewById(R.id.position);
    final StauPosition stauPosition =
        StauPosition.STAUENDE;
    if (R.id.stauAnfang ==
        aktuellePosition.getCheckedRadioButtonId()) {
      stauPosition = StauPosition.STAUANFANG;
    }
    fortschrittDlg = ProgressDialog.show(this, // (2)
        "Datenübertragung...", "Melde Stau an Server", true, false);
    stauVerwaltung.staumeldungAbschicken(getHandler(),
        getCurrentGps(), stauPosition,
        stauUrsache.getSelectedItem().toString());
  }

  private Handler getHandler() {
    final Handler threadCallbackHandler = new Handler() {
      @Override
      public void handleMessage(Message msg) { // (3)
        Bundle bundle = msg.getData();
        stauId = bundle.getLong("stauId");
        fortschrittDlg.dismiss(); // (4)
        final Intent intent =
            new Intent(getApplicationContext(),
            StauinfoAnzeigen.class);
        intent.putExtra(STAU_ID, stauId);
        startActivity(intent); // (5)
        super.handleMessage(msg);
      }
    };
    return threadCallbackHandler;
  }

  private String getCurrentGps() {
    // TODO: ermittle GPS-Position
    return "Koordinaten/Teststau";
  }
}
```

Zugriff über Interface

Gleich am Anfang der Klasse erfolgt die Deklaration des Attributs `stauVerwaltung` (1). Zu Grunde liegt das Interface `Stauverwaltung`. Es stellt uns alle Methoden zur Verfügung, die mit Staus zu tun haben. Da wir in unserem Beispiel nur die Methode `staumeldungAbschicken` betrachten, haben wir das Interface und seine Implementierung (`de.androidbuch.staumelder.stau.impl.StauverwaltungImpl`) zunächst auf diese eine Methode reduziert.

Die Implementierung des Interface erfüllt zwei Zwecke. Zum einen werden die Methodenaufrufe an den `NetzwerkService` durchgereicht. Zum anderen wird dieser Service automatisch gestartet, sobald in der Staumelder-Anwendung mittels

```
stauVerwaltung = new StauverwaltungImpl(
    this.getApplicationContext());
```

eine Instanz der Klasse `StauverwaltungImpl` ins Leben gerufen wird (siehe Listing 14.3).

Fortschrittsanzeige mit dem Progress-Dialog

In der Methode `stauMelden` lesen wir die Anwendereingaben aus und melden mittels der Stauverwaltung den Stau. Für die Zeit der Datenübertragung zeigen wir eine Fortschrittsanzeige (`ProgressDialog`) an (2). Da die Staumeldung (Datenübertragung an den Server) im `NetzwerkService` stattfindet, müssen wir diesem einen Callback-Handler mitgeben. Er definiert in der Methode `handleMessage(Message msg)` (3), was nach der Datenübertragung passieren soll. Zum einen wollen wir die Fortschrittsanzeige wieder entfernen (4) und zum anderen eine Folge-Activity anzeigen, auf der wir unsere Staumeldung erneut anzeigen (5). Dazu haben wir im `NetworkConnectorService` dafür gesorgt, dass die auf Serverseite vergebene Stau-Id zurück an unsere Anwendung übertragen wird. Abschließend beenden wir die Methode `handleMessage`, indem wir die Ausführung an die Oberklasse weiterreichen.

Zu einer Staumeldung gehört auch die aktuelle GPS-Position. Wir kommen darauf später in Kapitel 15 zurück. Hier holen wir uns nur über die Methode `getCurrentGps` einen Platzhalter.

Die Stauverwaltung

Schauen wir uns nun abschließend die Implementierung des Interface `Stauverwaltung` an.

Listing 14.3 *Implementierung der Stauverwaltung*

```
public class StauverwaltungImpl
    implements Stauverwaltung {
  private NetzwerkService.StauLocalBinder
      mNetzwerkServiceBinder;
```

```
  protected StauverwaltungImpl() { }

  public StauverwaltungImpl(Context context) {
    Intent intent = new Intent(context,
        NetzwerkService.class);
    context.bindService(intent, localServiceConnection,
        Context.BIND_AUTO_CREATE); // (1)
  }

  @Override
  public void staumeldungAbschicken(Handler
      threadCallbackHandler, String gpsPosition,
      StauPosition stauPosition, String stauUrsache) {
    if (mNetzwerkServiceBinder != null) {
      mNetzwerkServiceBinder.staumeldungAbschicken( // (2)
          threadCallbackHandler, gpsPosition,
          stauPosition, stauUrsache);
    }
  }

  private ServiceConnection localServiceVerbindung =
      new ServiceConnection() {
    @Override
    public void onServiceConnected(ComponentName // (3)
        className, IBinder binder) {
      mNetzwerkServiceBinder =
        (NetzwerkService.StauLocalBinder)binder;
    }
  }
}
```

Den Service automatisch starten

In dem Moment, in dem wir zum ersten Mal eine Instanz der Klasse `StauverwaltungImpl` erzeugen, wird der `NetzwerkService` per Intent im Konstruktor der Klasse gestartet. Dadurch, dass wir die `bindService`-Methode mit `Context.BIND_AUTO_CREATE` aufrufen, wird der Service gestartet und gleichzeitig die Service-Connection `localServiceVerbindung` aufgebaut (1).

Wird eine Verbindung zum `NetzwerkService` hergestellt, wird die Methode `onServiceConnected` aufgerufen (3). Wir erhalten über die Variable `binder` ein Objekt vom Typ `NetzwerkService.StauLocalBinder`. Dieses implementiert das Interface `Stauverwaltung` und ermöglicht uns den Aufruf von Methoden im Service. Daher speichern wir uns in dem Attribut `mNetzwerkServiceBinder` diesen Binder und können ihn dann in der Methode `staumeldungAbschicken` verwenden, um einfach an den Service durchzureichen (2).

14.3.2 Daten zum Stauserver übertragen

Nun fehlt uns noch die eigentliche Methode, die die Daten zum Server überträgt. Im `NetworkConnectorService` (siehe Listing 14.1 (5)) haben wir die Methode `_staumeldungAbschicken` nicht implementiert, sondern offen gelassen. Wir erinnern uns: Sie läuft innerhalb eines eigenen Threads im Service, damit der Service nicht blockiert ist. Er soll für die gesamte Anwendung die Netzwerkprozesse übernehmen und ständig verfügbar sein.

Listing 14.4 *Daten zum Server übertragen...*

```
private long _staumeldungAbschicken(final GpsData
      gpsPosition, final StauPosition stauPosition,
      final String stauUrsache) {
    HttpURLConnection httpVerbindung = null;
    OutputStream out = null;
    try {
      StringBuffer contentBuffer = new StringBuffer();
      contentBuffer.append(gpsPosition);
      contentBuffer.append("#");
      contentBuffer.append(stauPosition.toString());
      contentBuffer.append("#");
      contentBuffer.append(stauUrsache);

      URL url = new URL(urlString);
      httpVerbindung =
          (HttpURLConnection) url.openConnection();
      byte[] buff;
      httpVerbindung.setRequestMethod("POST");
      httpVerbindung.setDoOutput(true);
      httpVerbindung.setDoInput(true);
      httpVerbindung.connect();
      out = httpVerbindung.getOutputStream();
      buff = contentBuffer.toString().getBytes("UTF8");
      out.write(buff);
      out.flush();
    } catch (MalformedURLException e) {
      // TODO...
    } catch (UnsupportedEncodingException e) {
      // TODO...
    } catch (IOException e) {
      // TODO...
    } finally {
      if (out != null) {
        try {
          out.close();
        } catch (IOException e) {
          // TODO...
```

```
    }
  }
}

// lese die Antwort vom Server:
String response = null;
if (httpVerbindung != null) {
  InputStream in = null;
  try {
    byte[] respData = null;
    if (httpVerbindung.getResponseCode() ==
        HttpURLConnection.HTTP_OK) {
      in = httpVerbindung.getInputStream();
      int length = httpVerbindung.getContentLength();
      if (length > 0) {
        respData = new byte[length];
        int total = 0;
        while( total < length ){
          total += in.read(respData, total,
              length - total);
        }
      }
      else {
        ByteArrayOutputStream bytestream =
            new ByteArrayOutputStream();
        int ch;
        while ((ch = in.read()) != -1) {
          bytestream.write(ch);
        }
        respData = bytestream.toByteArray();
        bytestream.close();
      }
      in.close();
      response = new String(respData);
    }
  } catch (IOException e) {
    // TODO...
  }
  finally {
    if (in != null) {
      try {
        in.close();
      } catch (IOException e) {
        // TODO...
      }
    }
```

```
        if (httpVerbindung != null) {
          httpVerbindung.disconnect();
        }
      }
    } // end if

    long result = 0;
    if (response != null) {
      try {
        result = Long.parseLong(response);
      } catch (NumberFormatException e) {
        // TODO...
      }
    }
    return result;
  }
}
```

Datenformat

Am Anfang der Methode erzeugen wir einen String, der unsere Staumeldung enthält. Diesen schicken wir an den Server. Auf Serverseite wird die Staumeldung gespeichert, und es wird eine Id generiert und an den Staumelder als Antwort zurückgegeben.

14.3.3 Verfahren 2: dauerhafte Verbindungen

Regelmäßiges Nachfragen beim Server

Setzt sich der Anwender hinter das Steuer des Autos und wählt im Staumelder eine Route, die er gleich fahren möchte, dann erwartet er immer die aktuellsten Staumeldungen zu sehen. Hierfür ist es sinnvoll, eine dauerhafte Verbindung zum Server aufzubauen und in regelmäßigen Abständen nach den aktuellen Staumeldungen für diese Route zu fragen.

Wir lassen den Staumelder periodisch beim Server nach Staus fragen, obwohl dies nicht der cleverste Weg ist. Besser wäre, wenn sich der Server beim Staumelder meldet, sobald sich die Staus für die gewählte Route geändert haben. Schließlich weiß der Server, wann sich Staus auf einer Route geändert haben. Wir drucken hier aber nicht den Code des Servers ab, dieser liegt auf der Webseite zum Buch zum Download bereit (`www.androidbuch.de`). Daher verlagern wir den aktiven Teil in unsere Anwendung. Außerdem wollen wir zeigen, wie man eine Activity periodisch aktualisiert.

Funklöcher bemerken

Weiterhin brauchen wir einen Detektor, der registriert, wenn die Netzwerkverbindung zusammenbricht und wenn sie wieder zur Verfügung steht.

Staudaten vom Server abrufen

Wir erweitern den aus Abschnitt 14.1 bekannten `NetzwerkService`. Wir haben ihn um den Code für die Staumeldung bereinigt, damit das folgende Listing nicht zu lang wird. Da dieser neue `NetzwerkService` recht viele Programmzeilen enthält, haben wir ihn in einzelne Module aufgeteilt.

Wir starten mit der Implementierung, indem wir die Klasse `NetzwerkService` anlegen.

Listing 14.5 *Netzwerkservice für dauerhafte Verbindungen*

```
public class NetzwerkService extends Service {

  private static final String SERVER_IP = "10.0.2.2";
  private static final int PORTNUM = 9379;

  private static StauSpeicherSqliteImpl stauSpeicher;
  private long routenId;
  private static Socket sock;
  private static BufferedReader br;
  private static PrintWriter pw;

  @Override
  public void onCreate() {
    stauSpeicher = new StauSpeicherSqliteImpl(this);

    netzwerkVerbindungHerstellen();
  }

  public void setRoutenId(long routenId) {
    this.routenId = routenId;
  }

  private static void netzwerkVerbindungHerstellen() {
    try {
      sock = new Socket(SERVER_IP, PORTNUM);
      br = new BufferedReader(new
          InputStreamReader(sock.getInputStream()));
      pw = new PrintWriter(sock.getOutputStream(),
          true);
    }
    catch (UnknownHostException e) { }
    catch (IOException e) { }
  }
}
```

Modul 1: Verbindung aufbauen

Der Service baut in der `onCreate`-Methode eine Netzwerkverbindung zum Server auf. Es handelt sich um eine Socket-Verbindung, da wir kei-

nen Request-Response-Zyklus mit dem hier unnötigen HTTP-Header brauchen. Wir setzen allgemeine Kenntnisse der Klasse `java.net.Socket` voraus, da sie zum Java-SDK gehört.

Modul 2: Periodische Abfrage

Wir schicken in periodischen Abständen die Routen-Id zum Stauserver. Der Stauserver sucht alle Staus zur Routen-Id und schickt die Staumeldungen zurück an den Staumelder auf dem Android-Gerät. Erweitern wir nun den `NetzwerkService` um einen Handler (siehe Abschnitt 8.3.2), der alle 10 Sekunden ein `Runnable`-Objekt in die Message Queue des Threads stellt.

Listing 14.6 *Periodischer Abruf der Stauinformation*

```
private static Handler mHandler = new Handler();

private Runnable timerTask = new Runnable() {
  public void run() {
    staudatenFuerRouteAbrufen();
    mHandler.postDelayed(this, 10000); // (1)
  }
};

private void staudatenFuerRouteAbrufen() {
  if (sock != null && !sock.isClosed()) {
    pw.println(String.valueOf(routenId));
    pw.flush();
  }
}
```

Regelmäßiges Pollings dank eines Handlers

Wie in Abschnitt 8.3.2 schon gesagt, verwaltet ein Handler die Message Queue des Threads, zu dem er gehört, also in diesem Fall die Message Queue des `NetzwerkService`. Wir definieren ein Runnable-Objekt mit Namen `timerTask`, in dessen `run`-Methode die Anfrage an den Server mittels der Methode `staudatenFuerRouteAbrufen` erfolgt. Diese nutzt den `PrintWriter pw` des Sockets, um die Routen-Id an den Server zu schicken.

Anschließend übergeben wir dem Handler `mHandler` das Runnable-Objekt über die `postDelayed`-Methode (1). Durch diese Methode wird das Runnable-Objekt ausgeführt, und anschließend wartet der Handler 10.000 Millisekunden, bis er wieder in die Message Queue schaut und alle darin befindlichen Runnable-Objekte ausführt. Da wir das Runnable `timerTask` nicht aus der Message Queue entfernt haben, wird es alle 10 Sekunden erneut ausgeführt.

Wir müssen nun noch die `onCreate`-Methode erweitern, damit sie erst malig eine Routen-Id zum Server schickt.

```
@Override
public void onCreate() {
```

```
  ...
  netzwerkVerbindungHerstellen();
  mHandler.post(timerTask);
}
```

Der Server empfängt die Routen-Id, ermittelt alle aktuellen Staumeldungen auf dieser Route und ruft auf seinem Ende der Socket-Verbindung ebenfalls einen `PrintWriter` auf.

Beachtenswert ist, dass die Hin- und Rückrichtung der Kommunikation vollkommen entkoppelt ist. Es würde auch reichen, wenn wir die Routen-Id nur zum Server schicken, wenn sie sich ändert, wir also eine andere Route wählen.

Eine Socket-Verbindung funktioniert in beiden Richtungen. Jede Seite, Client und Server, verwendet ein Objekt vom Typ `InputStream`, um Daten vom Socket zu lesen. In Listing 14.5 sehen wir, wie er erzeugt wird. Um später die Möglichkeit zu haben, so lange zu warten, bis neue Daten über den Socket ankommen, wird der `InputStream` in einen `BufferedReader` gekapselt. Dies gibt uns die Möglichkeit, einen *Listener* zu implementieren, der eingehende Daten registriert:

Modul 3: Listener für eingehende Staudaten

Listing 14.7
Listener für eingehende Stauinformationen

```
private static void starteStaudatenListener() {
  new Thread() {
    public void run() {
      try {
        String line;
        // blockiert, bis eine neue Zeile ankommt:
        while (sock != null && !sock.isClosed() &&
            (line = br.readLine()) != null) {
          stauSpeicher.getWritableDatabase().
              beginTransaction();
          if (line.startsWith("DELETE#")) {
            String[] infos = line.split("#");
            stauSpeicher.getWritableDatabase().
              delete(StauSpeicherSqliteImpl.TABLE_NAME,
              StauSpeicherSqliteImpl.COL_ROUTE_ID +
              "=" + Long.valueOf(infos[1]), null);
          }
          else {
            String[] stauDaten = line.split("#");
            Stau staumeldung = new Stau(null,
                Long.valueOf(stauDaten[0]),
                Long.valueOf(stauDaten[1]),
                Integer.valueOf(stauDaten[2]),
                stauDaten[3],
                Integer.valueOf(stauDaten[4]),
                Integer.valueOf(stauDaten[5]),
```

```
                    new Date(Long.valueOf(stauDaten[6])),
                    stauDaten[7],
                    Long.valueOf(stauDaten[8]));
                    stauSpeicher.
                        schreibeStaumeldung(staumeldung);
                }
                stauSpeicher.getWritableDatabase().
                    setTransactionSuccessful();
                stauSpeicher.getWritableDatabase().
                    endTransaction();
              }
            }
            catch (IOException ex) { }
            finally {
              if (stauSpeicher.getWritableDatabase().
                  inTransaction()) {
                stauSpeicher.getWritableDatabase().
                    endTransaction();
              }
            }
          }
        }.start();
    }
```

Daten vom Server empfangen

In der Methode `starteStaudatenListener` starten wir einen Thread, der im Hintergrund darauf wartet, dass zeilenweise Daten vom Server kommen. Wir übertragen in unserem Staumelder-Beispiel alle Daten in Strings verpackt, was die Programmierung hier etwas einfacher macht. Die `while`-Schleife läuft, solange unsere Socket-Verbindung mit dem Server steht. Immer wenn eine Textzeile vom Server an unsere Android-Anwendung geschickt wird, durchlaufen wir einmal den Programmcode der Schleife.

Empfangene Staudaten speichern

In der Schleife starten wir zunächst eine Transaktion auf der SQLite-Datenbank. Der Server schickt immer mehrere Zeilen an den Client. Die erste davon besteht aus dem String »`DELETE`« und der Routen-Id, getrennt durch ein Rautensymbol. Daraufhin löschen wir alle Stauinformationen zu der Routen-Id aus der Datenbank. Ohne großen Mehraufwand könnte man hier einen Update-Mechanismus einbauen. Alle anderen Zeilen enthalten Stauinformationen, mit denen wir ein Stau-Objekt initialisieren. Der letzte Parameter jeder Zeile ist dabei die Routen-Id. Wir speichern das Stau-Objekt in der Datenbank und können uns der nächsten Zeile zuwenden. Nach Verlassen der Schleife schließen wir die Transaktion, notfalls auch im Fehlerfall.

Modul 4: Umgang mit Funklöchern

Als letztes Modul erweitern wir den `NetzwerkService` um einen Broadcast Receiver, der Systemnachrichten vom Connectivity Manager

empfängt, die über verlorene und wiederhergestellte Netzwerkverbindungen informieren.

Listing 14.8
Auf Funklöcher reagieren

```
private ConnectionBroadcastReceiver mBroadcastReceiver;

private final IntentFilter intentFilter =
    new IntentFilter("android.net.conn.
        CONNECTIVITY_CHANGE");

private static class ConnectionBroadcastReceiver extends
    BroadcastReceiver {
  @Override
  public void onReceive(Context ctxt, Intent intent) {
    try {
      boolean isOffline =
          intent.getBooleanExtra(
          ConnectivityManager.EXTRA_NO_CONNECTIVITY,
          false);
      if (isOffline) {
        if (sock != null && sock.isConnected()) {
          sock.close();
        }
      }
      else {
        netzwerkVerbindungHerstellen();
        starteStaudatenListener();
      }
    }
    catch (Exception e) { }
  }
};
```

`ConnectionBroadcastReceiver` ist eine statische innere Klasse, damit wir die Intents direkt im `NetworkConnectorService` fangen können. Da das Android-SDK verlangt, dass der Broadcast Receiver statisch ist, müssen wir daher auch alles statisch deklarieren, was wir dort verwenden. Um die Intents fangen zu können, müssen wir die nötige Berechtigung im Android-Manifest mittels

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

setzen. Der `NetzwerkService` enthält einen Intent-Filter, mit dessen Hilfe nur Broadcast Intents des Typs `android.net.conn.CONNECTIVITY_CHANGE` durchgelassen werden.

Dadurch wird die `onReceive`-Methode des `ConnectionBroadcastReceiver` immer ausgeführt, wenn eine Netzwerkverbindung wegbricht oder wiederhergestellt wird. Innerhalb der `onReceive`-Methode müssen wir feststellen können, ob die Verbindung zusammengebrochen ist oder wiederhergestellt wurde. Dazu werten wir den Intent aus:

```
intent.getBooleanExtra(ConnectivityManager.
    EXTRA_NO_CONNECTIVITY, false)
```

Entweder schließen wir dann den Socket falls nötig oder stellen die Verbindung wieder her.

Staudaten werden in der Datenbank gespeichert.

Die `onCreate`-Methode nutzen wir, um den Service zu initialisieren. Die Staudaten sollen in einer Datenbank gespeichert werden. Wie das geht, hatten wir in Kapitel 11 gezeigt. Hier erzeugen wir nun ein Objekt vom Typ `ConnectionBroadcastReceiver` und registrieren es unter Verwendung des definierten Intent-Filters beim Service.

```
@Override
public void onCreate() {
  ...
  mBroadcastReceiver =
      new ConnectionBroadcastReceiver();
  this.registerReceiver(mBroadcastReceiver,
      intentFilter);
  ...
}
```

Staudaten anzeigen

Entkopplung dank Datenbank

Was wir bisher implementiert haben, ist ein Service, der in regelmäßigen Abständen die aktuellen Staudaten vom Server empfängt und in die Datenbank unserer Anwendung schreibt. Damit ist der Service weitgehend entkoppelt von sonstigen Aufgaben. Wenn wir nun die aktuellen Staudaten anzeigen möchten, können wir einfach in regelmäßigen Abständen in die Datenbank schauen und die Staudaten von dort laden und in einer Activity darstellen. Dazu können wir das gleiche Verfahren wie im `NetzwerkService` nutzen und ein Runnable in die Message Queue unseres UI-Threads stellen. Das Runnable sorgt dann für das Laden der Staudaten aus der Datenbank und die Darstellung. Somit aktualisiert sich die Activity in gleichen Intervallen mit den neuesten Meldungen.

Listing 14.9
Selbstaktualisierende Activity zum Anzeigen des Stauberichts

```
public class StauberichtAnzeigen extends ListActivity {
  private static Context context;

  private StauSpeicher stauSpeicher;

  private
      ConnectionBroadcastReceiver mBroadcastReceiver;
  private final IntentFilter intentFilter =
      new IntentFilter(
          "android.net.conn.CONNECTIVITY_CHANGE");

  private long routenId;

  private static Handler mHandler = new Handler();
  private Runnable timerTask = new Runnable() {
      public void run() {
        zeigeStaubericht();
        mHandler.postDelayed(this, 5000); }
  };

  private static class ConnectionBroadcastReceiver
      extends BroadcastReceiver {
    private Toast mToast = null;
    @Override
    public void onReceive(Context ctxt, Intent intent) {
      try {
        boolean isOffline = intent.getBooleanExtra(
            ConnectivityManager.EXTRA_NO_CONNECTIVITY,
            false);
        if (isOffline) {
          if (mToast == null) {
            mToast = Toast.makeText(context,
                "Verbindung verloren!", Toast.LENGTH_LONG);
          }
          mToast.show();
          ((ListActivity)context).setTitle(
              context.getString(
              R.string.stauberichtanzeigen_titel) +
              " - Offline!");
        }
```

```
      else {
        if (mToast != null) {
          mToast.cancel();
          ((ListActivity)context).setTitle(
          context.getString(
          R.string.stauberichtanzeigen_titel) +
          " - Online...");
        }
      }
    }
    catch (Exception e) { }
  }
};

@Override
public void onCreate(Bundle icicle) {
  super.onCreate(icicle);

  setContentView(R.layout.staubericht_anzeigen);
  setTitle(R.string.stauberichtanzeigen_titel);

  context = this;
  stauSpeicher = new StauSpeicherSqliteImpl(this);

  mBroadcastReceiver =
      new ConnectionBroadcastReceiver();
  context.registerReceiver(mBroadcastReceiver,
      intentFilter);

  mHandler.post(timerTask);

  final Bundle extras = getIntent().getExtras();
  if (extras != null) {
    routenId = extras.getLong("ROUTEN_ID");
    zeigeStaubericht();
  }
}

private void zeigeStaubericht() {
  Cursor stauberichtCursor =
      stauSpeicher.ladeStauberichtFuerRoute(routenId);
  startManagingCursor(stauberichtCursor);

  String[] from = new String[] {
      StauSpeicherSqliteImpl.COL_STAUURSACHE };
  int[] to = new int[] { android.R.id.text1 };
```

```
    SimpleCursorAdapter dspStaubericht =
        new SimpleCursorAdapter(this,
            android.R.layout.simple_list_item_1,
            stauberichtCursor, from, to);
    setListAdapter(dspStaubericht);
  }
}
```

In der `StauberichtAnzeigen`-Activity bedienen wir uns der gleichen Hilfsmittel wie im `NetzwerkService`. Wir verwenden einen Broadcast Receiver, der uns informiert, wenn eine Netzwerkverbindung zusammenbricht bzw. wiederhergestellt wird, und einen Handler, der in regelmäßigen Abständen die Activity aktualisiert. Da die Activity von `ListActivity` abgeleitet ist, können wir die Staudaten direkt über einen Adapter in die Liste der Staudaten schreiben. Wir führen dies hier nicht weiter aus. Die Implementierung von `ListActivity` war Thema des Abschnitts 6.4.

Ein »Toast« als Meldung

Den Broadcast Receiver verwenden wir nur, um den Anwender auf der Oberfläche über den Verbindungsstatus in Kenntnis zu setzen. Wir haben dazu ein Toast (siehe Abschnitt 9.3) verwendet. Zusätzlich ändern wir die Titelzeile der Activity, damit man auch nach dem Verschwinden des Toasts über den Status der Netzwerkverbindung informiert ist.

Auch hier fügen wir der Message Queue des UI-Threads mit Hilfe eines Handlers ein Runnable-Objekt hinzu. Dadurch, dass wir auf dem Handler die `postDelay`-Methode verwenden, wird das Runnable alle 5 Sekunden ausgeführt und aktualisiert mittels Aufruf der Methode `zeigeStaubericht` die Oberfläche der Activity.

14.4 Fazit

Wir haben in diesem Kapitel gezeigt, was beim Emulator zu beachten ist, wenn man Netzwerkprogrammierung mit ihm testen möchte. Wir haben etwas über die Probleme bei der Datenübertragung in Mobilfunknetzen gelernt und wie man mit ihnen umgehen kann. Zahlreiche Codebeispiele decken die wesentlichen Kernpunkte ab und bilden einen Einstieg für die Entwicklung stabiler, professioneller Anwendungen mit Datenübertragung.

Wir haben uns weniger auf das Programmieren von HTTP-Verbindungen konzentriert, sondern mehr auf eine stabile, serviceorientierte Architektur, die Netzwerkprozesse im Hintergrund ablaufen lässt. Zum einen haben wir gesehen, wie man einen Callback-Handler

einsetzt, um nach erfolgter Datenübertragung informiert zu werden. Zum anderen haben wir eine dauerhafte Internetverbindung mit einer Socket-Connection aufgebaut, die sich automatisch wiederaufbaut, sobald wir nach einem Funkloch wieder Daten senden können.

Beide Methoden der Datenübertragung haben ihr jeweiliges Anwendungsfeld, bei dem wir auch die Kosten für die Datenübertragung in unsere Überlegungen einfließen lassen müssen. Wir haben für beide Methoden der Datenübertragung, Request-Response-Zyklus und dauerhafte Netzwerkverbindung, jeweils ein Beispiel gegeben, welches als Implementierungsmuster für eigene Entwicklungen dienen kann.

15 Iteration 7 – Location Based Services

Hinweis: Die 2. Auflage dieses Buchs erscheint unter dem Titel
»Android 2. Grundlagen und Programmierung«
Ende Mai 2010 im dpunkt.verlag, ISBN 978-3-89864-677-2

Location Based Services gewinnen zunehmend an Bedeutung. Immer mehr Mobiltelefone sind mit einem GPS-Empfänger ausgestattet, und es ist reizvoll, über Anwendungen nachzudenken, die standortspezifische Informationen verwenden.

Wir wollen uns in diesem Kapitel auf zwei Kernbereiche beschränken und diese in unseren Staumelder integrieren. Zum einem ist dies die Ermittlung von Positionsdaten und zum anderen die Anzeige des aktuellen Standorts in einer Straßenkarte (*Google Maps*). Die verwendeten Beispiele lassen sich leicht auf eigene Anwendungen übertragen und entsprechend anpassen.

15.1 Iterationsziel

Für unseren Staumelder heißt das, dass wir bei einer Staumeldung automatisch die aktuelle Position ermitteln und mit der Staumeldung an den Server senden. Wir brauchen also eine Komponente, die uns die Positionsdaten ermittelt. Außerdem wollen wir unsere aktuelle Position in einer Landkarte anzeigen. Das könnte uns helfen, eine Alternativroute zu finden, falls wir kein Navigationsgerät dabei haben oder gar keins besitzen.

Bevor wir jedoch mit der Programmierung starten, sind einige Vorbereitungen nötig, die wir unserer Struktur entsprechend in den theoretischen Teil gepackt haben. Wir wollen die Möglichkeiten des Emulators ausschöpfen, um effizient entwickeln zu können. Außerdem sollten wir ein wenig über GPS und Google Maps wissen.

Unsere Ziele für dieses Kapitel lauten wie folgt:

- verstehen, wie man mit Positionsdaten umgeht
- die Möglichkeiten des Emulators ausschöpfen können
- lernen, wie man die aktuelle Position ermittelt
- lernen, wie man Google Maps verwendet

15.2 Theoretische Grundlagen

15.2.1 GPS, KML und GPX

Längen- und Breitengrade

Da Android *GPS* (Global Positioning System) unterstützt, werden viele Hersteller ein GPS-Modul in ihre mobilen Computer einbauen. Ist dies der Fall, haben wir einen Lieferanten für Positionsdaten. Die wichtigsten Parameter sind dabei der Längen- und der Breitengrad. Längengrade werden mit »*longitude*« bezeichnet, Breitengrade mit »*latitude*« und können in östlichen Längengraden bzw. in nördlichen Breitengraden (jeweils mit Bogenminuten und -sekunden) oder im Dezimalsystem angegeben werden. Beispielsweise stellt [7°06'54.09" Nord, 50°42'23.57" Ost] denselben Geopunkt dar wie [7.1152637, 50.7066272] im Dezimalsystem.

GPS-Datenformate

Viele GPS-Geräte bieten zudem die Möglichkeit, die Positionsdaten über einen Zeitraum aufzuzeichnen, zu »tracken«. Meist werden diese dann als *GPX*-Datei (GPS Exchange Format) von *Google Earth* exportiert. Eine solche Datei kann viele Wegpunkte einer zurückgelegten Strecke enthalten.

Mit Google Earth KML-Dateien erzeugen

Ein weiteres Format ist *KML* (Keyhole Markup Language) von *Google Earth*. Das Programm Google Earth bietet die Möglichkeit, KML-Dateien zu erzeugen. Man fügt in dem Programm einfach über »*Ortsmarke hinzufügen*« eine neue Ortsmarke hinzu. Diese erscheint im linken Panel unter dem Reiter »*Orte*«. Klickt man auf diesen Ortspunkt mit der rechten Maustaste und wählt »*Speichern unter*«, kann man als Zieldatei den Typ »*KML-Datei*« auswählen.

Höheninformation

Der *Location Manager* von Android, den wir gleich vorstellen, verwendet nur die Dezimalschreibweise. Neben den Parametern »*longitude*« und »*latitude*« ist »*altitude*« noch ein wichtiger Geopunkt-Parameter. Er gibt die Höhe in Metern über Normal-Null an.

15.2.2 Entwickeln im Emulator

Wenn man Anwendungen erstellt, die Positionsdaten aus dem Location Manager verwenden, dann wird man diese meist erst im Emulator testen wollen. Im Emulator müssen Positionsdaten simuliert werden. Den komfortablen Weg beschreiben wir in Abschnitt 16.2. Allerdings funktioniert dieser Weg mit den Releases bis einschließlich *1.1rc1* aufgrund eines Fehlers mit deutschen Spracheinstellungen im Betriebssystem noch nicht. Daher beschreiben wir hier einen alternativen Weg über die Telnet-Konsole, da ein Testen der Anwendung sonst nicht möglich ist.

GPS-Modul per Telnet simulieren

Geodaten mittels Telnet an den Emulator senden

Öffnen Sie eine Betriebssystem-Konsole und schauen Sie in der Titelzeile Ihres Emulators, auf welchem Port er läuft (normalerweise 5554). Geben Sie

```
> telnet localhost 5554
```

ein. Dadurch wird die Android-Konsole gestartet. In dieser geben Sie dann beispielsweise

```
> geo fix <longitude> <latitude>\index{geo fix}
```

Dezimalzahlen als Eingabe

an. Denken Sie daran, dass Android nur Dezimalzahlen versteht. Eine Eingabe der Form

```
> geo fix 7.1152637 50.7066272
```

wäre also zulässig.

Wir brauchen dies erst später zum Testen, wenn wir unsere Anwendung erstellt haben. Einfacher geht es natürlich mit einem echten Gerät, aber das Laden der Karten von Google Maps erfolgt über das Internet und kann auf Dauer nicht unbeträchtliche Kosten verursachen.

15.2.3 Debug Maps API-Key erstellen

Mitgeliefertes Entwicklerzertifikat verwenden

Wir werden später Google Maps nutzen und eine Straßenkarte mit unserer aktuellen Position anzeigen. Dazu bedarf es einiger Vorbereitungen. Um von Google Maps Kartenmaterial abrufen zu dürfen, braucht man einen Maps API-Key. Wir zeigen nun, wie man diesen möglichst einfach erstellt. Grundlage für die Erstellung des Maps API-Key ist ein Zertifikat, mit dem man seine fertige Anwendung signiert. Das Signieren von Anwendungen ist Thema des Kapitels 17. Hier müssen wir jedoch schon wissen, dass das Eclipse-Plug-in zum Android-SDK ein Zertifikat für Entwicklungszwecke mitbringt, hier Debug-Zertifikat genannt. Anwendungen, die man mit dem Eclipse-Plug-in erstellt, werden automatisch mit dem Debug-Zertifikat signiert und laufen im Emulator oder in einem per USB-Kabel angeschlossenen Android-Gerät.

Zum Erstellen des Maps API-Keys brauchen wir ein Zertifikat. Da wir zum Entwickeln und Testen ein Debug-Zertifikat zur Verfügung haben, verwenden wir dieses, um den Debug Maps API-Key zu erzeugen.

Debug-Zertifikat finden

Das Debug-Zertifikat befindet sich in einem sogenannten *Keystore*. Wir müssen zunächst den Speicherort für den Keystore ermitteln. Sein Name ist »`debug.keystore`«. In Eclipse findet man die Angabe des Verzeichnises unter »*Window->Preferences->Android->Build*« (siehe Abb. 15-1).

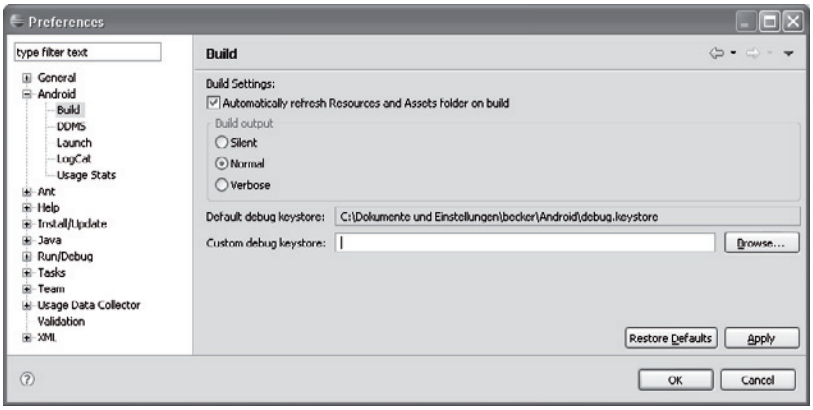


Abb. 15-1
Speicherort vom Debug-Keystore ermitteln

Um den Maps API-Key zu erzeugen, verwenden wir ein Tool, welches bei der Java-Runtime bzw. bei jedem JDK dabei ist und im `bin`-Verzeichnis von Java liegt. Das Tool heißt »`keytool.exe`«. Man kann im System die `PATH`-Variable erweitern und auf das `bin`-Verzeichnis zeigen lassen, damit man `keytool.exe` überall aufrufen kann. Am besten, man kopiert sich die Datei `debug.keystore` in ein Verzeichnis, welches keine Leerzeichen enthält, da sonst `keystore.exe` die Parameterliste nicht auflösen kann. Wir haben die Datei nach `D:\\Projekte\\androidbuch` kopiert und generieren nun mit `keystore.exe` einen MD5-Fingerabdruck, den wir für die Erzeugung des Maps API-Key brauchen.

MD5-Fingerabdruck

Ein MD5-Fingerabdruck ist eine Zahl (meist in hexadezimaler Schreibweise angegeben), die eine Checksumme von Daten oder Dateien darstellt. Verändern sich die Daten oder der Inhalt einer Datei, ändert sich auch der Fingerabdruck. Besitzt man den Fingerabdruck einer Datei aus sicherer Quelle und die Datei selbst aus unsicherer Quelle, kann man den Fingerabdruck der Datei erzeugen. Stimmen nun beide Fingerabdrücke überein, besitzt man mit größter Wahrscheinlichkeit die Originaldatei.

```
> keytool -list -alias androiddebugkey -keystore
    D:\Projekte\androidbuch\debug.keystore -storepass
    android -keypass android
```

Ein MD5-Fingerabdruck

Alle anderen Parameter außer dem Pfad zur `debug.keystore`-Datei bleiben unverändert. Das Ergebnis ist eine Ausgabe in der Konsole der Art:

```
Zertifikatsfingerabdruck (MD5):
    D1:E1:A3:84:B2:70:59:25:66:F0:E5:49:7D:B2:F1:36
```

Um nun an den Maps API-Key zu kommen, geht man auf die Google-Internetseite `http://code.google.com/android/maps-api-signup.html` und gibt den MD5-Fingerabdruck in das dafür vorgesehene Eingabefeld ein. Nach einem Klick auf die Schaltfläche »*Generate API Key*« erhält man auf der Folgeseite den gewünschten Key. Diesen brauchen wir später, wenn wir ein Layout für eine Activity entwerfen, welche die Straßenkarte anzeigen soll. Daher merken wir uns den Maps API-Key.

15.3 Praxisteil

Im folgenden Praxisteil erweitern wir den Staumelder um einen Service, der uns auf Anfrage die aktuelle Ortsposition zur Verfügung stellt. In einem zweiten Schritt implementieren wir die Funktion hinter dem Hauptmenüpunkt »*Straßenkarte anzeigen*« des Staumelders.

15.3.1 Vorbereitung

Um GPS in einer Android-Anwendung nutzen zu können, muss man im Android-Manifest ein paar kleine Erweiterungen vornehmen. Zum einen müssen wir folgende Berechtigung setzen:

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

Berechtigungen und Bibliotheken

Dies ermöglicht den Zugriff auf das GPS-Modul des Android-Geräts. Zusätzlich muss auch die Android-Berechtigung `android.permission.INTERNET` gesetzt sein, wenn wir Google-Maps verwenden, da das Kartenmaterial über das Internet geladen wird. Google-Maps erfordert auch das Einbinden einer zusätzlichen Bibliothek, die Android zwar mitbringt, aber nicht automatisch in eine Anwendung einbindet. Die Bibliothek binden wir innerhalb des `<application>`-Tags ein:

```
<application android:icon="@drawable/icon"
             android:label="@string/app_name"
             android:debuggable="true">

  <uses-library
    android:name="com.google.android.maps" />
  ...
```

15.3.2 Der Location Manager

Provider liefern Positionsdaten.

Der Location Manager (`android.location.LocationManager`) ist eine Schnittstelle zu zum GPS-Modul des mobilen Computers. Es kann durchaus mehrere GPS-Module in einem Gerät geben. In den meisten Android-Geräten wird in Zukunft sicherlich ein GPS-Empfänger eingebaut sein. Zusätzlich könnte man über Bluetooth einen zweiten Empfänger anbinden. Darüber hinaus gibt es auch die Möglichkeit, Informationen über die aktuelle Position über das Netzwerk zu beziehen. Der Location Manager verwaltet die Lieferanten von Positionsdaten unter der Bezeichnung »*Provider*«. Ein Provider liefert uns also die Ortsposition.

Im Grunde hat der Location Manager drei Funktionen:

- Er liefert die letzte bekannte Ortsposition (Geopunkt).
- Auf Wunsch wirft er einen selbstdefinierten `PendingIntent`, wenn wir den Radius um einen bestimmten Ortspunkt unterschreiten, uns ihm also nähern.
- Man kann bei ihm einen Listener registrieren, der mehr oder weniger periodisch die Ortsposition liefert.

Listener registriert Ortsveränderungen.

Wir schauen uns den letzten Punkt etwas genauer an, da wir diese Funktion für die Anzeige unserer Straßenkarte gut brauchen können. Der Location Manager besitzt unter anderem folgende Methode:

```
public void requestLocationUpdates(String provider,
    long minTime, float minDistance,
    LocationListener listener)
```

Diese Methode veranlasst den Location Manager, in periodischen Abständen die Ortsposition zu ermitteln. Das Ermitteln der Ortsposition verbraucht sehr viel Systemressourcen, sowohl was die Prozessorlast als auch was den Stromverbrauch angeht, denn das GPS-Modul verbraucht sehr viel Energie.

Die Methode hilft, Strom zu sparen, und man sollte die Parameter dem Einsatzzweck entsprechend setzen, wie wir gleich sehen werden.

Parameter	Beschreibung
`provider`	Name des GPS-Providers
`minTime`	0: Der Location Manager liefert so oft wie möglich die aktuelle Position. Stromverbrauch ist maximal. >0: Wert in Millisekunden, der festlegt, wie lange der Location Manager mindestens warten soll, bevor er wieder nach der aktuellen Position fragt. Je größer der Wert, desto geringer der Stromverbrauch.
`minDistance`	0: Der Location Manager liefert so oft wie möglich die aktuelle Position. Stromverbrauch ist maximal. >0: Distanz in Metern, um die sich die Position mindestens verändert haben muss, damit der Location Manager eine neue Position liefert. Je größer der Wert, desto geringer der Stromverbrauch.
`listener`	Ein selbst programmierter Location Listener überschreibt die Methode `onLocationChanged(Location location)`. Das Objekt `location` enthält die aktuelle Position.

Tab. 15-1
Parameter der Methode requestLocationUpdates

Die Tabelle 15-1 erklärt die Parameter der Methode. Will man die aktuelle Position so oft wie möglich erhalten, so setzt man die Parameter `minTime` und `minDistance` beide auf Null.

Tipp

Schalten Sie das GPS-Modul des mobilen Computers nur ein, wenn Sie es wirklich brauchen. Der Akku hält dann wesentlich länger! Beenden Sie Ihre Activity mit der Methode `finish` in der `onPause`-Methode. Dadurch greift Ihre Anwendung nicht auf das GPS-Modul zu und Sie sparen Strom.

Eigener GPS-Service

Wir werden nun den `GpsPositionsServiceLocal` aus Abschnitt 8.3.1 auf Seite 110 erweitern. Ziel ist es, innerhalb einer Activity über einen Callback-Handler (siehe Abschnitt 8.3.2) immer mit aktuellen GPS-Daten aus einem eigenen GPS-Service versorgt zu werden. Wir können dann diesen Service später für andere Anwendungen weiterverwenden und eine Activity schreiben, die den Callback-Handler definiert und die GPS-Daten automatisch vom `GpsPositionsServiceLocal` erhält. Die Activity kann dann die Daten beispielsweise verwenden, um die aktuelle Position in einer Straßenkarte darzustellen.

Wir erweitern zunächst den schon bekannten `GpsPositionsServiceLocal`. Dazu fügen wir folgende Zeile ein:

```
Handler uiServiceCallbackHandler;
```

Listener registriert Ortsveränderung.

Nun fügen wir dem Service eine innere Klasse hinzu. Die Klasse implementiert einen `LocationListener` und muss vier Methoden überschreiben. Davon interessiert uns eigentlich nur eine Methode, die Methode `onLocationChanged(Location location)`. Wie der Name vermuten lässt, wird die Methode aufgerufen, wenn sich die GPS-Position geändert hat. Dem Location Manager übergeben wir gleich den hier implementierten Listener mittels der `requestLocationUpdates`-Methode, die weiter oben beschrieben wurde.

Listing 15.1
Eigenen Listener für Ortsveränderungen implementieren

```
private class MyLocationListener
    implements LocationListener {

  @Override
  public void onLocationChanged(Location location) {
    if (uiServiceCallbackHandler != null) {
      Message message = new Message();
      message.obj = location;
      Bundle bundle = new Bundle();
      bundle.putParcelable("location", location);
      message.setData(bundle);
      uiServiceCallbackHandler.sendMessage(message);
    }
  }

  @Override
  public void onProviderDisabled(String provider) { }

  @Override
  public void onProviderEnabled(String provider) { }

  @Override
  public void onStatusChanged(String provider,
      int status, Bundle extras) { }
};
```

Nun implementieren wir noch die `onCreate`-Methode des `GpsPositionsServiceLocal` neu:

Listing 15.2
Den Location Manager initialisieren

```
@Override
public void onCreate() {
  locationManager = (LocationManager)getSystemService(
      Context.LOCATION_SERVICE);
```

```
  locationListener = new MyLocationListener();
  locationManager.requestLocationUpdates(LocationManager.
      GPS_PROVIDER, 5000, 10, locationListener);
}
```

Wir holen uns darin vom System einen Verweis auf den Location Manager. Dies erfolgt wie üblich über die Methode `getSystemService(String name)`. Nun erzeugen wir ein Exemplar unseres `MyLocationListeners` und rufen auf dem Location Manager die `requestLocationUpdates`-Methode auf. Mit den verwendeten Parametern wird der `locationListener` minimal alle fünf Sekunden (meist seltener) über die Ortsposition informiert. Er wird ebenfalls aufgerufen, wenn sich die Position um mehr als 10 Meter seit dem letzten Aufruf der Methode `onLocationChanged` verändert hat.

Location-Objekt für Ortspunkte

Um nun die GPS-Daten an die Activity zu übertragen, die sich um die Darstellung unserer aktuellen Position kümmert, implementieren wir die `onLocationChanged`-Methode. Wenn unser Callback-Handler nicht null ist, erzeugen wir eine Message und ein Bundle. Wir verzichten hier auf unsere eigene Klasse `GpsData` (siehe Listing 8.8 in Abschnitt 8.3.1), sondern können den Übergabeparameter `location` direkt verwenden, da er das Interface `Parcelable` schon implementiert. Folglich fügen wir dem Bundle die `location` hinzu, vervollständigen die Message und stellen die Message in die Message Queue des aufrufenden Threads, also unserer Activity, die wir gleich implementieren.

Als Nächstes erweitern wir den `IBinder`. Er ist unsere Schnittstelle nach außen und soll auch die Möglichkeit bieten, aktuelle GPS-Daten per Methodenaufruf und nicht über den Callback-Handler zu liefern.

Listing 15.3
Den Location Manager initialisieren

```
public class GpsLocalBinder extends Binder {

  public GpsPositionsServiceLocal getService() {
      return GpsPositionsServiceLocal.this;
  }

  public void setCallbackHandler(Handler
      callbackHandler) {
    uiServiceCallbackHandler = callbackHandler;
  }

  public GpsData getGpsData() {
    if (locationManager != null) {
      Location location = locationManager.
          getLastKnownLocation(LocationManager.
          GPS_PROVIDER);
      GpsData gpsData = new GpsData(location.getTime(),
```

```
            (float)location.getLongitude(), (float)location.
            getLatitude(), (float)location.getAltitude());
        return gpsData;
      }
      return null;
    }
}
```

Letzte bekannte Position ermitteln

Zum einen können wir hier unserem Service den Callback-Handler übergeben, den wir oben schon gebraucht haben, zum anderen können wir hier jederzeit aktuelle GPS-Daten abrufen. Die Methode `getLastKnownLocation` liefert uns ebenfalls ein Objekt vom Typ `Location`. Aus Kompabilitätsgründen und um die wichtigsten Methoden des `Location`-Objekts vorzustellen, erzeugen wir das schon bekannte `GpsData`-Objekt und geben es an den Aufrufer zurück.

Abschließend sei der Vollständigkeit halber noch erwähnt, dass wir die `GpsPositionsServiceLocal`-Klasse noch um folgende Deklaration erweitern müssen:

```
private LocationManager locationManager;
private LocationListener locationListener;
```

15.3.3 Google Maps

In einigen Anwendungen wird man seine aktuelle Position auf einer Landkarte darstellen wollen. Dies können wir mit Hilfe von Google Maps machen. Nachdem wir uns am Anfang des Kapitels den Maps API-Key zum Testen unserer Anwendung besorgt haben, können wir die Anzeige einer Straßenkarte mit unserer aktuellen Position in Angriff nehmen.

`MapView` und `MapActivity`

Auf einfache Weise ist dies mit einer `MapView` möglich. Dies ist eine spezielle View, die wir in ein Layout integrieren können. Zur Darstellung eines Layouts mit `MapView` sollte man unbedingt eine `MapActivity` verwenden. Die `MapActivity` hat einen speziellen Lebenszyklus, da im Hintergrund das Kartenmaterial über das Internet geladen und geladenes Kartenmaterial im Dateisystem abgelegt wird. Diese langwierigen Aufgaben erledigen Threads, die im Lebenszyklus der `MapActivity` verwaltet werden.

Schauen wir uns zunächst die Definition eines einfachen Layouts zur Anzeige einer Straßenkarte mit Google Maps an.

Listing 15.4 *Layout zur Anzeige einer MapView*

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

```
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">

  <com.google.android.maps.MapView
     android:id="@+id/mapview_strassenkarte"
     android:layout_width="fill_parent"
     android:layout_height="fill_parent"
     android:clickable="true"
     android:apiKey="
         0xYgdiZYM8ZDqhlJWsm7qdgRzBrMEkTJCdqpD6w"/>
</LinearLayout>
```

Zwei Attribute sind dabei neu: `android:clickable` und `android:apiKey`. Das erste dient unter anderem dazu, dass wir später die Straßenkarte verschieben und ein Oberflächenelement anzeigen können, welches den Zweck hat, die Karte zu vergrößern oder zu verkleinern. Das zweite Attribut ist der Maps API-Key, den wir am Anfang des Kapitels auf den Google-Maps-Internetseiten haben erzeugen lassen. Fügen Sie hier Ihren eigene Maps API-Key ein, den Sie wie in Abschnitt 15.2.3 beschrieben selbst erzeugt haben.

Eigenen Maps API-Key einfügen

Was wir damit haben, ist ein View-Element, welches wir in unserer Activity ansprechen können.

15.3.4 MapActivity

Wie wir weiter oben gesehen haben, sollten wir für die Darstellung der Straßenkarte eine `MapActivity` einsetzen. Um die `MapActivity` zu implementieren, müssen wir folgende Schritte ausführen:

- Wir müssen die `MapView` in der `MapActivity` anzeigen.
- Wir müssen den Callback-Handler implementieren, so dass er die Straßenkarte mit den aktuellen GPS-Daten versorgt und die Darstellung aktualisiert.
- Wir müssen der Klasse `GpsPositionsServiceLocal` den Callback-Handler übergeben, um in der Activity automatisch aktuelle GPS-Positionsdaten zu erhalten.
- Wir müssen unseren aktuellen Standort in die Karte malen.

Starten wir mit der `MapActivity` und dem Zugriff auf die `MapView`.

Listing 15.5 *`MapActivity` zur Anzeige einer Straßenkarte*

```
public class StraßenkarteAnzeigen extends MapActivity {

  private MapView mapView;
  private MapController mapController;
  private MapViewOverlay mapViewOverlay;
  private Paint paint = new Paint();
```

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.strassenkarte_anzeigen);
    setTitle(R.string.text_strassenkarte_titel);

    Intent intent = new Intent(this,
        GpsPositionsServiceLocal.class);
    bindService(intent, localServiceVerbindung,
        Context.BIND_AUTO_CREATE);

    mapView = (MapView) findViewById(
        R.id.mapview_strassenkarte); // (1)
    mapController = mapView.getController();

    int zoomlevel = mapView.getMaxZoomLevel();
    mapController.setZoom(zoomlevel - 2);

    mapViewOverlay = new MapViewOverlay(); // (2)
    mapView.getOverlays().add(mapViewOverlay);
    mapView.postInvalidate();

    LinearLayout zoomView = (LinearLayout) mapView.
        getZoomControls();
    zoomView.setLayoutParams(new ViewGroup.LayoutParams(
        ViewGroup.LayoutParams.WRAP_CONTENT,
        ViewGroup.LayoutParams.WRAP_CONTENT));
    zoomView.setGravity(Gravity.BOTTOM |
        Gravity.CENTER_HORIZONTAL);
    mapView.addView(zoomView);

    mapView.setStreetView(true);
  }
}
```

Wir erweitern nun die Activity um die fehlenden Methoden. Um eine Verbindung zu unserem Service aufzubauen, müssen wir noch das Attribut `localServiceVerbindung` angeben. Sobald die Verbindung zum Service steht, setzen wir unseren Callback-Handler.

Listing 15.6 *Verbindung zum lokalen GPS-Service aufbauen*

```
private ServiceConnection localServiceVerbindung =
    new ServiceConnection() {
  @Override
  public void onServiceConnected(ComponentName className,
      IBinder binder) {
    ((GpsPositionsServiceLocal.GpsLocalBinder)binder).
        setCallbackHandler(uiThreadCallbackHandler);
  }
```

```
  @Override
  public void onServiceDisconnected(
      ComponentName className) { }
};
```

Wenn wir zur `onCreate`-Methode zurückkehren, sehen wir als Nächstes, wie wir uns die `MapView`, die wir im Layout deklariert haben, holen (1). Eigenschaften der `MapView` können wir über einen Controller steuern. Nachdem wir die `MapView` und den `MapController` haben, können wir die Straßenkarte für die Ansicht präparieren. Wir holen uns mittels der Methode `getMaxZoomLevel` die höchste Auflösung, die uns für so eine View zur Verfügung steht. Es handelt sich hier um die maximal verfügbare Auflösung in der Mitte der Karte. Da Google Maps nicht für alle Orte auf der Welt die gleiche Auflösung zur Verfügung stellt, bietet es sich an, hier evtl. einen festen Wert zu nehmen. Wenn dieser höher als der höchste verfügbare Zoomlevel liegt, fällt die Anwendung auf diesen Zoomlevel zurück.

Zoomlevel

Als Nächstes instanziieren wir ein Objekt vom Typ `MapViewOverlay` (2). Die Deklaration des Objekts fügen wir als innere Klasse unserer Activity hinzu.

Listing 15.7
Ein Overlay zur Anzeige des eigenen Standorts in der Karte

```
public class MapViewOverlay extends Overlay {
  @Override
  public void draw(Canvas canvas, MapView mapView,
      boolean shadow) {
    super.draw(canvas, mapView, shadow);

    // Wandel Geopunkt in Pixel um:
    GeoPoint gp = mapView.getMapCenter();
    Point point = new Point();
    // in Point werden die relativen Pixelkoordinaten
    // gesetzt:
    mapView.getProjection().toPixels(gp, point);

    // Zeichenbereich definieren:
    RectF rect = new RectF(point.x - 5, point.y + 5,
        point.x + 5, point.y - 5);

    // roter Punkt fuer eigenen Standort
    paint.setARGB(255, 200, 0, 30);
    paint.setStyle(Style.FILL);
    canvas.drawOval(rect, paint);
```

```
    // schwarzen Kreis um den Punkt:
    paint.setARGB(255,0,0,0);
    paint.setStyle(Style.STROKE);
    canvas.drawCircle(point.x, point.y, 5, paint);
  }
}
```

Was ist ein Overlay?

Ein *Overlay* können wir uns als durchsichtige Folie vorstellen, die wir über unsere `MapView` legen. Mittels der `draw`-Methode können wir wie auf einem Overhead-Projektor zeichnen und malen. Wir können einer `MapView` potenziell beliebig viele Overlays hinzufügen und wieder entfernen, ohne dass sich die `MapView` verändert. Daher holen wir uns in der `onCreate`-Methode der `MapActivity` erst alle Overlays und fügen dann unseres hinzu.

Das Overlay selbst gehört zur `MapView` und bekommt diese als Parameter in der `draw`-Methode übergeben. Wir holen uns mittels `mapView.getMapCenter` das Zentrum als Geo-Punkt (Objekt `GeoPoint`) und nutzen die `MapView` für die Umrechnung in Pixel (Methode `getProjection(GeoPoint gp, Point point)`). Der Rest der Methode dient dem Zeichnen eines roten Punkts mit schwarzer Umrandung in der Mitte der Karte, also dort, wo sich laut GPS-Modul unser aktueller Standort befindet.

Die eigene Position anzeigen

Was uns nun noch fehlt, ist der Callback-Handler. Er liefert uns in seiner `handleMessage`-Methode den Ortspunkt. Da diese Methode automatisch durch den Service aufgerufen wird, entweder wenn wir uns bewegen oder wenn eine bestimmte Zeitspanne vergangen ist, brauchen wir nur dafür zu sorgen, dass die Straßenkarte in der `MapView` auf unsere aktuellen GPS-Koordinaten zentriert wird. Den Rest übernimmt dann das Overlay, indem es unseren Standort als roten Punkt einzeichnet.

Listing 15.8
Der Callback-Handler liefert uns die GPS-Daten aus unserem GPS-Service.

```
private final Handler uiThreadCallbackHandler =
    new Handler() {
  @Override
  public void handleMessage(Message msg) {
    super.handleMessage(msg);
    Bundle bundle = msg.getData();
    if (bundle != null) {
      Location location =
          (Location)bundle.get("location");
      GeoPoint geoPoint = new GeoPoint(
          (int) (location.getLatitude() * 1E6),
          (int) (location.getLongitude() * 1E6));
```

```
      mapController.animateTo(geoPoint);
      mapView.invalidate();
    }
  }
};
```

Um die `MapView` zu zentrieren, holen wir uns das Location-Objekt aus dem Bundle, welches uns der Service liefert. Das Zentrieren erledigt der Controller der `MapView`, und dieser erwartet ein `GeoPoint`-Objekt. Die Umrechnung ist einfach. Das `location`-Objekt liefert die Werte `longitude` und `latitude` in Grad. `GeoPoint` erwartet aber Mikrograd. Durch die Multiplikation mit dem Wert `1E6` (1 Million) erhalten wir aus Grad Mikrograd. Abschließend rufen wir noch auf der `MapView` `invalidate` auf und erzwingen dadurch ein Neuzeichnen der Straßenkarte. Die Overlays werden nach wie vor angezeigt, so dass uns auch der Standortpunkt nicht verloren geht.

Abb. 15-2
Anzeige der aktuellen Position mit Google Maps im Emulator

Abbildung 15-2 zeigt das fertige Ergebnis unserer Bemühungen im Emulator, nachdem wir mit `telnet` eine Ortskoordinate eingegeben haben.

15.4 Fazit

Wir haben das Kapitel mit ein wenig Theorie über GPS begonnen, damit wir später beim Entwickeln und Testen mit Ortskoordinaten umgehen können. Denn wir können auch im Emulator Location Based Services testen, da wir den Location Manager über die Android-Konsole mit Geodaten füttern können.

Wir haben dann einige Vorbereitungen getroffen, damit wir später Google Maps in unserer Anwendung zur grafischen Anzeige von Karten nutzen können. Wir haben uns einen Maps API-Key aus unserem Debug-Zertifikat des Android-SDK erstellt.

Nach diesen Vorbereitungen haben wir uns dem Location Manager gewidmet. Er liefert uns die Ortskoordinaten unseres aktuellen Standorts. Normalerweise greift er dazu auf das GPS-Modul des Android-Geräts zurück, allerdings sind auch andere Verfahren der Positionsermittlung möglich, z.B. die Berechnung der Position über die Funkmasten, in die das Android-Gerät gerade eingebucht ist.

Wir haben dann mit der `MapActivity` experimentiert. Durch sie war es uns möglich, die von Google Maps geladene Karte darzustellen. Dank Overlays konnten wir dann der Karte noch Informationen in grafischer Form hinzufügen, was wir beispielhaft durch einen roten Punkt in der Kartenmitte, der unsere aktuelle Position anzeigt, getan haben.

Teil III

Android für Fortgeschrittene

16 Debugging und das DDMS-Tool

Hinweis: Die 2. Auflage dieses Buchs erscheint unter dem Titel
»Android 2. Grundlagen und Programmierung«
Ende Mai 2010 im dpunkt.verlag, ISBN 978-3-89864-677-2

Wir werden uns in diesem Kapitel damit beschäftigen, wie man sich das Leben bei der Entwicklung von Android-Anwendungen etwas erleichtern kann. Es wird erklärt, wie man ein Android-Gerät per USB an den Entwicklungsrechner anschließt. Das Debugging kann dann auch direkt auf dem Endgerät oder weiterhin wie gewohnt im Emulator erfolgen.

Wir werden eine weitere Perspektive namens *DDMS* vorstellen, die das Eclipse-Plug-in mitbringt. DDMS steht für *Dalvik Debug Monitor Service*. In dieser Perspektive können wir mehr über die Anwendung und ihr Laufzeitverhalten erfahren.

16.1 Anschluss eines Android-Geräts per USB

Das Android-SDK bringt den notwendigen USB-Treiber bereits mit. Dieser liegt im Ordner `usb_driver` auf der gleichen Ebene wie der Ordner `tools`. Nachdem man ihn installiert hat, kann man das Android-Gerät erstmals anschließen.

USB mounten

Da es sich bei Android um ein Linux-Betriebssystem handelt, muss die USB-Verbindung gemountet werden. Das Android-Gerät erkennt automatisch, dass ein USB-Kabel angeschlossen wurde, und sendet sich selbst eine Notification. Das Mounten des USB-Anschlusses erfolgt, indem man den Statusbalken in der Anzeige des Android-Geräts nach unten zieht und die Notification öffnet (siehe Abb. 16-1).

Wenn das Gerät korrekt verbunden wurde, kann man die Anwendung ganz normal starten. Läuft gleichzeitig der Emulator, wird man gefragt, ob man das Programm im Emulator oder auf dem angeschlossenen Gerät laufen lassen möchte.

On-Device-Debugging aktivieren

Um direkt auf dem Gerät debuggen zu können, muss man noch eine Systemeinstellung im Android-Gerät kontrollieren. Wir geben dies für das G1 von HTC an, da wir auf diesem Gerät unsere Beispiele entwickelt haben: *Hauptmenü -> Settings -> Applications -> Development*. Dort den Menüpunkt »*USB Debugging*« aktivieren.

Abb. 16-1
Notification nach Anschluss des Geräts über USB und das Mounten des Anschlusses

16.2 DDMS: Dalvik Debug Monitor Service

Es handelt sich bei dem DDMS um ein eigenständiges Tool, welches von der Kommandozeile aus gestartet oder in Eclipse verwendet werden kann. Es liegt im `tools`-Verzeichnis des SDK.

Verwendet man Eclipse mit dem Android-Plug-in, kann man den DDMS als eigene Perspektive über *Windows -> Open Perspective -> Other... -> DDMS* starten (siehe Abb. 16-2 auf Seite 281).

Emulator beenden kann helfen.

Der DDMS verbindet sich automatisch sowohl mit dem Emulator als auch mit einem angeschlossenen Android-Gerät. Allerdings stehen bei gleichzeitiger Nutzung vom Emulator und angeschlossenem Android-Gerät für das Android-Gerät nur die beiden Views *Threads* und *Heap* zur Verfügung. Will man aber auch die Threads und den Heap des angeschlossenen Endgeräts anzeigen, muss man den Emulator schließen, sonst bleiben die beiden Masken leer.

Tipp!

Wollen Sie Informationen über Threads und Heap eines angeschlossenen Endgeräts anzeigen, müssen Sie den Emulator schließen. Bisweilen reicht auch das nicht, und Sie müssen die USB-Verbindung neu mounten. Ziehen Sie dann das USB-Kabel und verbinden Sie das Gerät neu. Nach dem Mounten sollten nun auch die Threads und der Heap angezeigt werden. Klicken Sie dazu im Panel *Devices* auf das Paket Ihrer Anwendung.

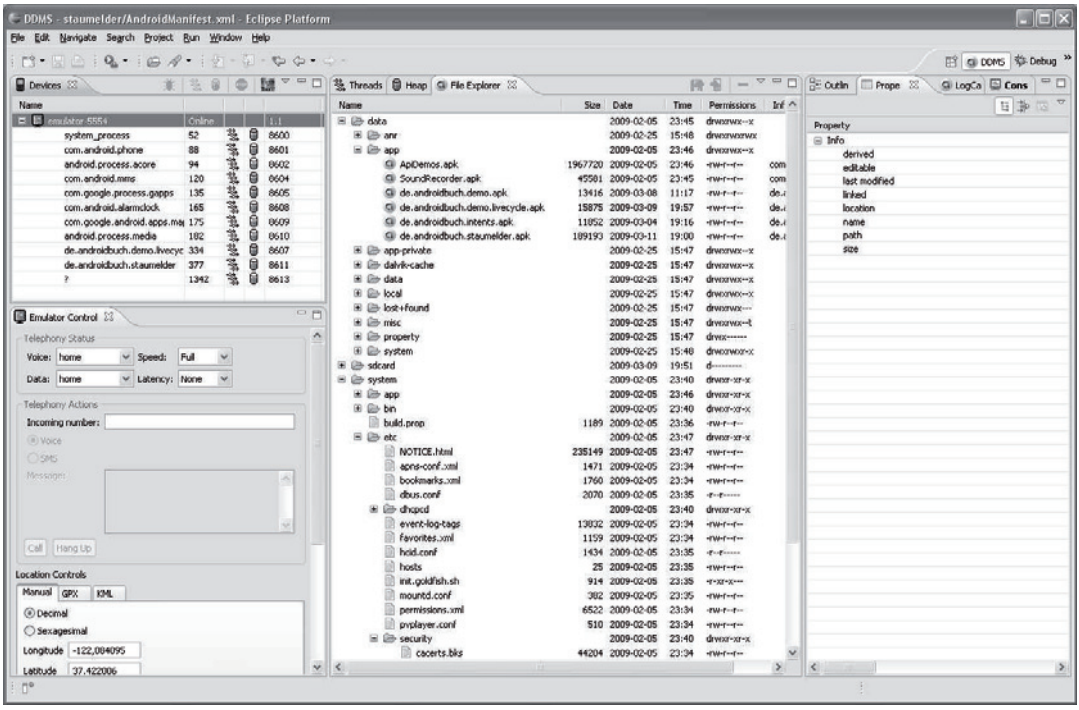


Abb. 16-2
Ansicht des DDMS in Eclipse

Die wichtigsten Teilansichten des DDMS sind:

- Anzeige der *LogCat* (siehe Abschnitt 16.2.1)
- Informationen über Threads und Heap
- Anzeige der laufenden Prozesse
- Simulation eingehender Telefonanrufe oder SMS
- Simulation eines GPS-Moduls mit eigenen Daten
- File-Explorer

16.2.1 Emulator Control

Die View `Emulator Control` verdient eine etwas genauere Betrachtung. Wir können sie nutzen, um unsere Anwendung zu testen.

Telephony Actions

Telefonanrufe und SMS simulieren

Im Panel *Telephony Actions* hat man die Möglichkeit, Telefonanrufe zu simulieren oder SMS an die Anwendung zu schicken. Verständlicherweise funktioniert dies nur für den Emulator und nicht für ein angeschlossenes Gerät. Gerade das Simulieren von Telefonanrufen eignet sich in der Praxis gut, den Lebenszyklus der implementierten Activities

zu testen, da man mit zwei Klicks (»*Call*« und »*Hang Up*«) die Anwendung unterbrechen und wieder fortsetzen kann.

Location Controls

Zu Testzwecken kann man dem Emulator mitteilen, welchen Geo-Punkt der Location Manager liefern soll. Es stehen drei Eingabemöglichkeiten zur Verfügung:

- Geodaten von Hand eintragen
- eine GPX-Datei laden
- eine KML-Datei laden

GPX- und KML-Dateien

Mittels des Programms *Google Earth* kann man sich KML-Dateien erzeugen lassen. Wie dies geht, wurde in Kapitel 15 auf Seite 261 beschrieben. Viele GPS-Geräte lassen einen Export der aufgezeichneten Daten oder Ortspunkte per GPX-Datei zu. Im Internet gibt es zahlreiche Möglichkeiten, KML nach GPX und umgekehrt zu konvertieren.

Tipp!

Will man Geodaten mit Hilfe des DDMS an den Emulator senden, gibt es zumindest bis zur Version *1.1* des Android-SDK einen Fehler. Die Zahlenformate werden intern in das Zahlenformat konvertiert, welches dem Betriebssystem zugrunde liegt. Das hat in Deutschland z.B. zur Folge, dass der Emulator alle Zahlen im deutschen Format (mit Komma statt Dezimalpunkt) erhält und der Location Manager des Emulators sie nicht als Zahl erkennt und ignoriert. Folglich lassen sich keinerlei Geodaten aus der DDMS an den Emulator senden, und man ist auf `telnet` angewiesen (siehe Kap. 15.2.2). Oder man verwendet folgenden Workaround: Man kann die Länder und Spracheinstellung im Betriebssystem ändern. Wählen Sie als Land z.B. England, so können Sie die Location Controls nutzen.

LogCat

Debug-Ausgabe in einer Konsole

Die *LogCat* gehört zum Eclipse-Plug-in und kann über `Window -> Show View -> Other... -> Android -> LogCat` aufgerufen und so in eine Perspektive eingebunden werden. In der *LogCat* werden Ausgaben angezeigt, die im Programmcode generiert werden. Dazu deklariert man am besten innerhalb jeder Klasse eine statische Variable namens `TAG`, die den Namen der Klasse ohne Paketnamen erhält.

```
private static final String TAG = "StauinfoAnzeigen";
```

Eine Ausgabe innerhalb des Programmcodes erfolgt über die statischen Methoden der Klasse `android.util.Log`. Wir verweisen hier auf die API-Dokumentation, wollen aber trotzdem den Hinweis geben, dass man sich über den Einsatz der Methoden der Klasse `Log` Gedanken machen sollte. Ein Beispiel:

```
double hoehe = getAltitude();
Log.i(TAG, "Höhe: " + hoehe + " Meter.");
```

Die String-Konkatenation wird immer ausgeführt, auch wenn die Anwendung später auf dem Endgerät ausgeführt wird und keine Ausgabe sichtbar wird. Folglich handelt man sich durch den Einsatz der Klasse `Log` einen Performanzverlust und einen höheren Speicherverbrauch ein. Zwar verwendet Android intern die Klasse `StringBuffer`, um die Ausgabe zu erzeugen, jedoch ist dies kein Allheilmittel. Wir empfehlen, wie bei gewohnter Java-Programmierung mit Logging-Tools, eine statische Variable zu deklarieren, die es dem Compiler erlaubt, die gesamte Logging-Zeile zu entfernen. Bevor man die Anwendung als fertiges Release in den produktiven Einsatz gibt, kann man den Wert der Variablen umsetzen, und alle Zeilen à la `Log.d(TAG, »irgendwas...«)` werden beim Kompilieren nicht berücksichtigt.

Ressourcen schonen

Man könnte dazu in einer Klasse `Einstellungen` die statische Variable unterbringen und von überallher auf diesen Wert zugreifen.

```
public class Einstellungen {
  public static final boolean DEBUG = true;
  ...
```

Logging erfolgt dann innerhalb einer Klasse nach folgendem Pattern:

```
public class StauinfoAnzeigen {
  public static final boolean DEBUG =
      Einstellungen.DEBUG;
  public static final String TAG =
      "StauinfoAnzeigen";

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    if (DEBUG) Log.d(TAG,
        "StauinfoAnzeigen->onCreate(): irgendwas...");
    ...
  }
  ...
}
```

Ein zentrales Umschalten der Variable `DEBUG` in der Klasse `Einstellungen` bereinigt den kompletten Code beim Kompilieren um die Ausgabe durch die Klasse `Log`. Das Ganze lässt sich natürlich auch selektiv einsetzen, um Meldungen vom Level »*Error*« oder »*Warning*« zu erhalten.

16.2.2 Debugging

Debuggen auf dem Endgerät

Debugging ist sowohl im Emulator als auch auf einem echten Android-Gerät möglich. Die Anwendung wird einfach im Debug-Modus gestartet und das Zielgerät (Emulator oder Endgerät) ausgewählt. Hier ist es egal, ob der Emulator läuft, wenn man die Anwendung auf dem Endgerät debuggen möchte. Sowohl die Breakpoints als auch die *LogCat* funktionieren.

Spezielle Einstellungen für den Debugger muss man nicht vornehmen. Sobald man in Eclipse für seine Anwendung eine *Run*-Konfiguration angelegt hat, kann man die Anwendung auch im Debugger starten.

17 Anwendungen signieren

Hinweis: Die 2. Auflage dieses Buchs erscheint unter dem Titel
»Android 2. Grundlagen und Programmierung«
Ende Mai 2010 im dpunkt.verlag, ISBN 978-3-89864-677-2

Während der Entwicklungsphase werden Anwendungen automatisch signiert. Das Eclipse-Plug-in verwendet dazu das Debug-Zertifikat. Dies reicht zunächst aus, um Anwendungen im Emulator oder auf dem via USB angeschlossenen Android-Gerät zu testen.

Für »marktreife« Anwendungen, die über das Internet installiert werden, reicht dies aber nicht. Insbesondere, wenn Anwendungen über den Android-Market vertrieben werden, müssen sie mit einem eigenen Zertifikat signiert werden. Dabei spielt es keine Rolle, ob es sich um ein durch eine offizielle Zertifizierungsstelle (engl. *Certification Authority*, *CA*) beglaubigtes Zertifikat handelt oder ob man die Anwendung mit einem selbst erstellten Zertifikat signiert.

Wir werden in diesem Kapitel zeigen, wie man eine Anwendung signiert. Als vorbereitenden Schritt zeigen wir, wie man ein eigenes Zertifikat zum Signieren von Anwendungen erstellt. Wer sich noch ausführlicher mit dem Thema beschäftigen möchte, kann sich die Google-Dokumentation durchlesen ([17]).

17.1 Vorbereitung

Wenn eine Anwendung fertig entwickelt und getestet wurde, kann man sie einer größeren Gruppe zugänglich machen. Android-Anwendungen haben die Endung `.apk` und werden während der Entwicklungsphase automatisch erzeugt, wenn wir sie im Emulator laufen lassen. Da allerdings während der Entwicklung das Debug-Zertifikat verwendet wurde, um die Anwendung zu signieren, kann man sie nicht auf Geräten installieren, die nicht an den Entwicklungsrechner direkt angeschlossen sind. Will man die Anwendung z.B. im Android-Market[1] oder auf der eigenen Internetseite zum Download anbieten, muss man ein eigenes Zertifikat zum Signieren der Anwendung erstellen und die Anwendung

[1] Der Android-Market (`www.android.com/market/`) ist Googles zentrale Internetseite, um Anwendungen zu vertreiben. Entwickler können sich dort anmelden und ihre Anwendungen zum Download anbieten.

damit signieren, sonst ist eine Installation über das Internet nicht möglich.

Eigene Zertifikate sind erlaubt.

Glücklicherweise kann man das Zertifikat selbst erstellen und ist nicht auf eine kostenpflichtige Zertifizierungsstelle (CA) angewiesen. Dies ist z.B. ein großer Unterschied zum *Java Verified Program*, welches Midlets auf Basis von J2ME durchlaufen müssen, um maximale Berechtigungen für den Zugriff auf Funktionen des Mobiltelefons zu bekommen.

Haben wir nun also eine marktreife Anwendung fertiggestellt, müssen wir die `.apk`-Datei erneut erstellen, diesmal aber ohne Zuhilfenahme des Standardzertifikats aus dem Debug-Keystore.

Erklärung:

Ein Keystore (Zertifikatspeicher) speichert unter einem Aliasnamen Zertifikate zum Signieren bzw. Verschlüsseln. Ein Keystore kann viele Zertifikate verschiedener Hersteller beherbergen. Der von Android mitgelieferte `debug.keystore` enthält das Debug-Zertifikat unter dem Aliasnamen `androiddebugkey`.

Ein Release erzeugen

Wir erzeugen nun zunächst eine unsignierte Version unserer Android-Anwendung, die wir dann später mit dem selbst erstellten Zertifikat signieren können. Dazu markieren wir in Eclipse im »*Package-Explorer*« unser Projekt mit der rechten Maustaste und wählen im Kontextmenü unter »*Android Tools*« den Punkt »*Export Unsigned Application Package...*«.

Release-Verzeichnis anlegen

Wir empfehlen, nicht das `bin`-Verzeichnis als Speicherort zu wählen, sondern auf der gleichen Verzeichnisebene oder im Ordner `bin` ein Verzeichnis `release` anzulegen. Auf diese Weise verhindert man, dass beim nächsten Start des Emulators die Release-Version überschrieben werden kann. Als Name wählt man am besten einen temporären Namen, da wir die endgültige `.apk`-Datei noch signieren müssen. Für unser Staumelder-Beispiel würden wir z.B. `staumelder_unsigned.apk` wählen.

Was nun folgt, ist etwas Handarbeit. Glücklicherweise reicht es völlig, ein Zertifikat für alle Anwendungen zu erzeugen, die wir auf den Markt bringen wollen. Denn das Zertifikat liefert Informationen über den Hersteller der Anwendung, nicht über die Anwendung selbst.

17.2 Ein eigenes Zertifikat erzeugen

Im Abschnitt 15.2.3 haben wir schon gelernt, wie wir das Hilfsprogramm `keytool.exe` aus dem JDK bzw. der Java-Runtime-Umgebung

finden bzw. aufrufen können. Wir verwenden es nun mit anderen Parametern, um einen RSA-Schlüssel zum Signieren der Anwendung zu erstellen. Denn für eine digitale Signatur in der Anwendung ist ein privater Schlüssel notwendig, der Teil des zu erstellenden Zertifikats ist.

keytool.exe zur Schlüsselgenerierung nutzen

Wir öffnen eine Konsole und geben Folgendes ein:

```
> keytool -genkey -v -keystore visionera.keystore
      -alias visionerakey -keyalg RSA -validity 18250
```

Parameter	Beschreibung
`-genkey`	Veranlasst `keytool.exe`, ein Schlüsselpaar, bestehend aus öffentlichem und privatem Schlüssel, zu erzeugen.
`-v`	Steht für »verbose«. Umfangreiche Ausgabe während der Laufzeit von `keytool.exe` einschalten.
`-keystore`	Speicherort und Name des Keystores. Existiert die Datei nicht, wird sie angelegt.
`-alias`	Ein Aliasname, anhand dessen das Zertifikat im Keystore wiedergefunden werden kann.
`-keyalg`	Verschlüsselungsalgorithmus für den Schlüssel. Gültige Werte sind *RSA* und *DSA*.
`-validity`	Gültigkeitsdauer des Zertifikats in Tagen (hier: 18250 Tage = 50 Jahre). Dieser Wert sollte ausreichend groß gewählt werden, da Anwendungen nur installiert werden können, wenn das Zertifikat noch gültig ist. Will man seine Android-Anwendung im Android-Market veröffentlichen, so muss das Gültigkeitsdatum nach dem 22. Oktober 2033 liegen, da sonst die Anwendung dort nicht hochgeladen werden kann.

Tab. 17-1
keytool-Parameter zum Erzeugen eines eigenen Zertifikats

Das Programm `keytool` verlangt von uns einige Eingaben (siehe Tabelle 17-1). Hier geht es um eine eindeutige Identifikation der Firma bzw. der Person, die später das Zertifikat verwendet. Wenn wir alles richtig gemacht haben, gibt uns `keytool` am Ende eine Statuszeile folgender Art aus:

```
Erstellen von Schlüsselpaar (Typ RSA, 1.024 Bit) und
selbstunterzeichnetem Zertifikat (SHA1withRSA) mit
einer Gültigkeit von 18.250 Tagen für: CN=Arno Becker,
OU=Mobile Business, O=visionera gmbh, L=Bonn, ST=NRW,
C=DE
Geben Sie das Passwort für <visionerakey> ein.
```

```
(EINGABETASTE, wenn Passwort dasselbe wie für den
Keystore):
```

Aus Bequemlichkeit kann man die letzte Frage mit der Eingabetaste quittieren, wenn alle Zertifikate im Keystore über dasselbe Passwort erreichbar sein sollen. Wählen Sie auf jeden Fall ein ausreichend kompliziertes Passwort und verwenden Sie `keytool` nur an einem sicheren Rechner, an dem das Passwort für Ihren Keystore nicht ausgespäht werden kann.

17.3 Eine Android-Anwendung signieren

jarsigner zum Signieren verwenden

Zum Signieren einer Android-Anwendung brauchen wir das Werkzeug `jarsigner.exe`. Während `keytool` sowohl in der JRE als auch im JDK vorhanden ist, befindet sich `jarsigner` nur im `bin`-Verzeichnis des JDK.

Wir gehen davon aus, dass die unsignierte Android-Anwendung unter dem Namen `staumelder_unsigned.apk` im `release`-Ordner liegt und wir uns im Projektverzeichnis des Staumelders befinden.

```
> jarsigner -verbose -keystore androidbuch.keystore
      -signedjar .\release\staumelder.apk
      .\release\staumelder_unsigned.apk visionerakey
```

Mittels des Parameters `-signedjar` geben wir den Ziel-Dateinamen an. Wir können allerdings auch die unsignierte Datei einfach überschreiben, indem wir diesen Parameter weglassen. Mit den letzten beiden Parametern geben wir den Namen der zu signierenden Datei und den Aliasnamen (`visionerakey`) an, um aus dem Keystore (`androidbuch.keystore`) den richtigen Schlüssel zu erhalten.

Ergebnis überprüfen

Wenn wir prüfen wollen, ob die Signierung geklappt hat oder ob die Anwendung schon signiert ist, dann können wir in der Konsole Folgendes eingeben:

```
> jarsigner -verify staumelder.apk
```

Wenn daraufhin in der Konsole `jar verified` ausgegeben wird, ist unsere Anwendung signiert, und wir können sie in Zukunft über das Internet zum Download anbieten oder im Android-Market veröffentlichen.

18 Sicherheit und Verschlüsselung

Hinweis: Die 2. Auflage dieses Buchs erscheint unter dem Titel
»Android 2. Grundlagen und Programmierung«
Ende Mai 2010 im dpunkt.verlag, ISBN 978-3-89864-677-2

In diesem Kapitel geht es um den Schutz unserer Daten. Es stellt im Grunde die Fortsetzung des Kapitels 3 in Teil 1 des Buchs dar, in dem wir das Prinzip der Sandbox für Android-Anwendungen kennengelernt haben. Durch die Sandbox ist ein laufendes Programm samt seiner Daten erst mal gut vor unerlaubten Zugriffen geschützt. Auch Daten, die wir selbst erzeugen und nur im Speicher halten, sind relativ gut geschützt, solange nicht eine Sicherheitslücke im System entdeckt wird. Sobald wir aber Daten persistent machen oder über das Internet verschicken, wird unsere Anwendung unsicher.

Daher betrachten wir erst mal die potenziellen Risiken im Allgemeinen und überlegen dann, wie wir selbst für die Sicherheit unserer Anwendung bzw. unserer Daten sorgen.

18.1 Sicherheit

Mobiltelefone übertragen Sprache und Daten mit Hilfe der *GSM*-Mobilfunktechnik (*GSM*: *Global System for Mobile Communication*) zu einer Basisstation. Eine Basisstation ist ein Sende- und Empfangsmast, zu dem das Mobiltelefon eine Funkverbindung aufbaut. Die meisten Basisstationen bestehen aus drei Zellen (auch »*Sektoren*« genannt). Sie haben Antennen in drei Richtungen, die jeweils etwa 120 Grad abdecken. Eine Zelle ist somit ein Sende- und Empfangskorridor in Form eines großen Kuchenstücks mit einer Reichweite von maximal 35 km. Diese Zellen haben eine eindeutige Id, die auch Cell-Id genannt wird und zur Positionsbestimmung mittels Dreieckspeilung (als Alternative zu GPS, aber sehr viel ungenauer) eingesetzt werden kann.

Mobilfunktechnik

Eine Basisstation ist an einen Vermittlungsknoten angebunden. Vermittlungsknoten sind Teil des Telefon-Festnetzes und kümmern sich zum Beispiel um die Wegsuche, Teilnehmerermittlung und die Verbindung ins Internet.

Der Weg ins Internet

Um mit einem Mobiltelefon zu telefonieren oder Daten zu übertragen, muss man eine *SIM*-Karte (*SIM*: *Subscriber Identity Module*) einlegen. Eine SIM-Karte erfüllt folgende Zwecke:

- Speichern der eigenen Kundennummer
- Speichern der eigenen Telefonnummer
- Authentifizierung
- Nutzdatenverschlüsselung
- Speichern der kryptographischen Algorithmen zur Verschlüsselung

Die SIM-Karte ist somit unabhängig vom Mobiltelefon und enthält alle Daten, um technisch gesehen in jedes Mobiltelefon eingebaut werden zu können. Das Mobiltelefon selbst hat eine international eindeutige Identifikationsnummer, die hier jedoch keine Rolle spielt.

Standardmäßig verschlüsselt, aber...

Normalerweise baut das Mobiltelefon eine verschlüsselte Verbindung für Sprach- und Datenübertragung zur Basisstation auf. Dazu findet zunächst eine Authentifizierung statt und anschließend wird der Verschlüsselungsalgorithmus ausgehandelt. Verwendet wird dabei ein GSM-A5/1-Verschlüsselungsalgorithmus, der als hinreichend sicher angesehen werden kann. Jedoch ist im GSM-Standard festgelegt, dass z.B. bei hoher Netzlast auf die Verschlüsselung verzichtet werden kann. Dann werden die Daten *unverschlüsselt* per Funk zur Basisstation übertragen.

Theoretisch ist es möglich, den Funkverkehr zwischen Basisstation und allen beteiligten Mobiltelefonen abzuhören und ein bestimmtes Mobiltelefon herauszufiltern. Im Falle unverschlüsselter Datenübertragung könnte man so den Funkverkehr abhören.

Keine 100-prozentige Garantie

Strafverfolgungsbehörden und Nachrichtendienste setzen sogenannte IMSI-Catcher ein [3]. Dies sind technische Geräte, die sich gegenüber dem Mobiltelefon wie eine Basisstation verhalten und die Verschlüsselung ausschalten. Telefongespräche und übertragene Daten lassen sich mit IMSI-Catchern abhören. Auch eine ungefähre Positionsbestimmung des Mobiltelefons ist möglich.

Leichter lassen sich übertragene Daten abfangen, wenn man Zugang zur Technik des Netzbetreibers hat. Dann stehen einem alle Wege offen. Von Polizei und Behörden wird dies genutzt, so dass auf diesem Weg ein ungefähres Bewegungsprofil erstellt und Anrufe und übertragene Daten mitgeschnitten werden können.

Haben unsere Daten schließlich den Weg über Basisstation und Vermittlungsknoten ins Internet gefunden, sind sie dort den üblichen Gefahren ausgesetzt. Auch wenn sie beim Empfänger ankommen, sind sie dort im internen Netz möglicherweise nicht sicher vor unerlaubtem Zugriff.

Als Fazit können wir festhalten, dass eine unverschlüsselte Datenübertragung ein Risiko birgt. Das Risiko liegt weniger im Bereich der GSM-Technik, sondern eher im Internet bzw. beim Empfänger.

SSL-Verbindungen nutzen

Abhilfe können wir schaffen, indem wir die Daten verschlüsselt übertragen. Dies können wir in Android-Anwendungen genau wie in Java-SE entweder über eine HTTPS-Verbindung machen oder über eine SSL-Socket-Verbindung (`javax.net.ssl.SSLSocket`).

Jedoch haben wir an beiden Enden der Verbindung die Daten wieder unverschlüsselt vorliegen. Sobald wir sie persistent machen, sind sie unzureichend gesichert. Genauso sieht es aus, wenn wir Daten an einen Server übertragen. Auch dort können sie in falsche Hände geraten.

Um das Problem auf der Android-Seite noch etwas zu konkretisieren, sei Folgendes gesagt: Wenn wir aus der Anwendung heraus Daten speichern, dann wird der Speicherort eine Datenbank oder eine normale Datei im Dateisystem, meist auf einer Speicherkarte, sein. Wer an die Speicherkarte kommt, wird es nicht schwer haben, auch an die Daten zu kommen.

Sicherheitsrisiko Speicherkarte

Wichtig!

SQLite bietet keine Möglichkeit, Daten automatisch beim Speichern zu verschlüsseln. Da die Datenbank eine Datei ist, auf die man relativ leicht zugreifen kann, wenn man im Besitz des Android-Geräts ist, liegt hier ein Sicherheitsrisiko vor, und sensible Daten sollten nur verschlüsselt gespeichert werden.

18.2 Verschlüsselung

Glaubt man den Studien und Umfragen in Unternehmen, dann ist Sicherheit von mobilen Anwendungen eines der zentralen Themen, wenn es um die Erstellung professioneller mobiler Unternehmenssoftware geht. Mobile Geräte gehen leicht verloren, werden oft gestohlen, und eine Übertragung von Daten per Funk und über das Internet ist nicht vollkommen sicher. Daher widmen wir einen etwas längeren Abschnitt der Verschlüsselung und wollen den Einstieg durch ein Codebeispiel erleichtern.

Das JDK beinhaltet seit der Version 1.4 die *Java Cryptography Extension (JCE)*. JCE steht auch im Android-SDK zur Verfügung. Aufbauend auf der *Java Cryptography Architecture (JCA)* liefert die JCE Erweiterungen, um Texte oder Objekte zu verschlüsseln. Dabei bringt die JCE schon Provider mit. Provider stellen die Implementierungen von Verschlüsselungsalgorithmen zur Verfügung. Die JCE lässt aber das Einbinden weiterer Provider und damit weiterer Verschlüsselungsalgorithmen zu.

Android nutzt Bouncy Castle.

Wer sich schon mal ein wenig mit Verschlüsselung beschäftigt hat, wird schon auf *Bouncy Castle* gestoßen sein. Die »*Bouncy Castle Crypto APIs*« sind eine Sammlung von APIs für Verschlüsselung. Es gibt eine *C#*- und eine Java-Variante. Auch Bouncy Castle stellt einen Provider zur Verfügung, den man mit der JCE nutzen kann. Da Android zwar die Java-API in seiner DVM versteht und umsetzt, aber aus rechtlichen Gründen nicht die Implementierungen von Sun nutzen kann, hat man sich dazu entschlossen, Teile der Verschlüsselungsimplementierung auf Bouncy Castle aufzubauen. Das hat zur Folge, dass die API, die wir implementieren, zwar JCE-konform ist, die darunterliegende Implementierung, also die Verschlüsselungsalgorithmen und alles, was sonst noch nötig ist, zu beträchtlichen Teilen von Bouncy Castle stammt.

Konflikte möglich!

Folglich wird es zu Problemen kommen, wenn man versucht, Bouncy Castle als Bibliotheken seiner eigenen Anwendung einzubinden und die Anwendung laufen zu lassen. Es kommt zu Konflikten, da Klassen der eigenen Anwendung mit Klassen der darunterliegenden Implementierung kollidieren. Wir vertiefen dieses Thema hier nicht weiter und möchten den Leser nur auf mögliche Probleme vorbereiten. Denn generell gilt, dass die Implementierungen von Bouncy Castle sehr vertrauenswürdig sind und der Quellcode offenliegt. Es ist also durchaus sehr zu begrüßen, dass Android Bouncy Castle als Provider für die JCE verwendet.

Fassen wir aber zunächst zusammen, was eine sichere Android-Anwendung können sollte:

- verschlüsselte Datenübertragung per HTTPS
- verschlüsselte Datenübertragung per SSL via Sockets
- Daten oder Objekte verschlüsseln

Bevor wir uns aber per SSL mit einem Server verbinden, verlieren wir noch ein paar Worte über Zertifikate.

18.2.1 Verschlüsselte Datenübertragung

Android und SSL-Zertifikate

Für eine verschlüsselte Datenübertragung brauchen wir ein SSL-Zertifikat. Ein Zertifikat besteht aus strukturierten Daten, die den Eigentümer eines öffentlichen Schlüssels bestätigen. Der öffentliche Schlüssel dient der *asymmetrischen* Verschlüsselung von Daten.

Asymmetrische Verschlüsselung (Public-Key-Verfahren) Bei der asymmetrischen Verschlüsselung benötigt man zwei Schlüssel, einen privaten, geheimen Schlüssel und einen öffentlichen Schlüssel. Das

bekannteste asymmetrische Verfahren ist *RSA* (benannt nach seinen Erfindern Rivest, Shamir, Adleman). Der Sender verschlüsselt die Nachricht mit dem öffentlichen Schlüssel des Empfängers. Nur der Empfänger, der im Besitz des dazu passenden privaten Schlüssels ist, kann die Nachricht dekodieren.
Vorteil: Der Schlüsselaustausch ist einfach und sicher. Man holt sich den öffentlichen Schlüssel vom Empfänger, z.B. über das Internet.
Nachteil: Unter anderem ist das Verfahren sehr rechenaufwendig.

Symmetrische Verschlüsselung Sender und Empfänger müssen im Besitz des gleichen geheimen Schlüssels sein. Der Sender verschlüsselt die Nachricht mit dem geheimen Schlüssel und der Empfänger entschlüsselt sie mit dem gleichen Schlüssel.
Vorteil: Das Verfahren ist relativ schnell.
Nachteil: Der sichere Schlüsselaustausch stellt ein Problem dar.

SSL-Verschlüsselung beginnt mit dem sogenannten *SSL-Handshake*. Dieser sorgt auf Basis eines asymmetrischen Verschlüsselungsverfahrens für die Identifikation und Authentifizierung des Kommunikationspartners. Beim Handshake wird der öffentliche Schlüssel durch den Sender über das Internet vom Empfänger geladen.

X.509 ist der derzeit bedeutendste Standard für digitale Zertifikate. X.509-Zertifikate sind SSL-Zertifikate, die nach diesem Standard aufgebaut sind. Ein X.509-Zertifikat ist immer an eine E-Mail-Adresse oder einen DNS-Eintrag gebunden. Außerdem enthält der X.509-Standard ein Verfahren, welches es erlaubt, mit Hilfe von Sperrlisten der Zertifizierungsstellen einmal ausgelieferte Zertifikate wieder ungültig zu machen.

Mit Hilfe des Zertifikats, welches beim Sender liegt, wird der Eigentümer des öffentlichen Schlüssels verifiziert. Auf diese Weise wird eine sogenannte *Man-in-the-middle-Attacke* verhindert, bei der jemand versucht, sich als der Empfänger auszugeben. Nach dem Handshake werden die Daten für die Übertragung zum Empfänger symmetrisch verschlüsselt.

Alles hängt also davon ab, dass wir das Zertifikat auch wirklich vom Empfänger unserer Daten haben. Zertifikate werden in einem *Zertifikatspeicher* (engl. *keystore*) abgelegt. Wir werden im Folgenden die Bezeichnung »Keystore« verwenden, da sie allgemein gebräuchlich ist, auch wenn sie nicht ganz passend ist.

Zertifikatspeicher = Keystore

Ein Keystore ist zum Beispiel Teil jedes Browsers, jedes JDK, und damit besitzt auch die Android-Plattform einen Keystore.

Darin befinden sich, egal ob Emulator oder gekauftes Android-Gerät, vorinstallierte Zertifikate von vertrauenswürdigen Zertifizie-

rungsstellen (*CA: Certificate Authority*). Bei diesen Zertifizierungsstellen kann man, in aller Regel gegen Geld, SSL-Zertifikate erhalten. Die eigene Identität wird von der Zertifizierungsstelle sichergestellt und ein Zertifikat generiert, welches man auf seinem Server installieren kann. Baut man nun mit dem Android-Gerät oder dem Emulator eine SSL-Verbindung zum Server auf, wird die Gültigkeit des Serverzertifikats anhand der vorinstallierten Zertifikate im Keystore des Android-Geräts geprüft.

Vorinstallierte Zertifikate beim Client

Der Server liefert beim Verbindungsaufbau das eigene Zertifikat aus und der Client (z.B. der Webbrowser) schaut in seinem Keystore nach, ob er ein Zertifikat der gleichen Zertifizierungsstelle gespeichert hat, welche das Serverzertifikat erstellt hat. Falls ja, weiß der Client, dass das Ziel seiner Verbindung das Zertifikat von der CA bekommen hat. Damit ist die Identität des Betreibers des Servers sichergestellt. Zuvor muss man natürlich das Zertifikat in seine Serversoftware (Webserver, Mailserver, Anwendungsserver etc.) einbauen.

Was aber ist, wenn wir kein Geld ausgeben wollen oder den Zertifizierungsstellen nicht vertrauen? Dann können wir ein eigenes SSL-Zertifikat erstellen und auf unserem Server installieren. Wenn unser Client dann eine verschlüsselte Verbindung dorthin aufbaut, müssen wir das Serverzertifikat vorher in den Keystore des Clients importieren, indem wir es explizit akzeptieren. Im Falle eines Webbrowsers bekommen wir eine Meldung, dass die Identität des Servers nicht sichergestellt ist (siehe Abb. 18-1). Das ist richtig, wir haben das Zertifikat ja selbst erstellt und müssen uns nun auch selbst vertrauen.

Selbst erstellte Zertifikate importieren

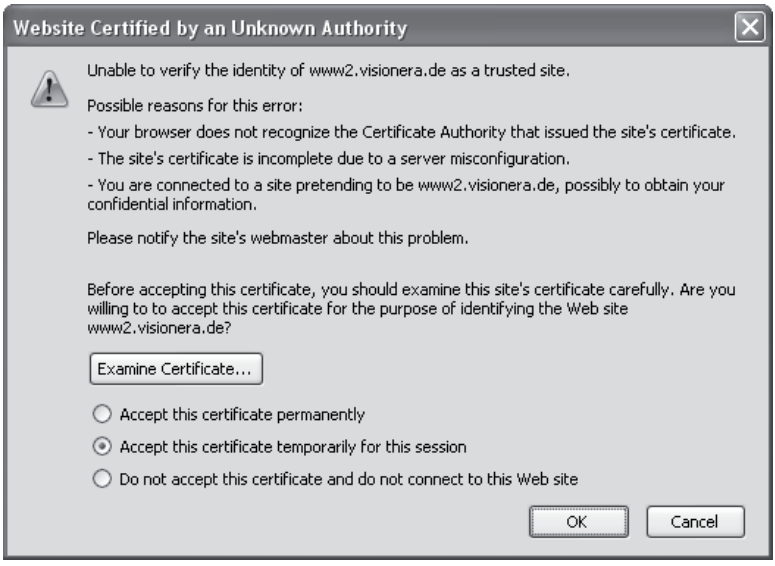


Abb. 18-1
Browser-Zertifikat akzeptieren

Ein Browser bietet automatisch den Import des SSL-Zertifikats an und überspringt damit die gesicherte Authentifizierung. Jeder Browser? Nein, leider nicht! *Android gestattet es nicht, eigene Zertifikate zu importieren.* Wenn wir mit den vorinstallierten Programmen (Browser, E-Mail-Client) eine SSL-Verbindung zu einem Server mit einem selbst erstellten SSL-Zertifikat aufbauen, werden wir scheitern. Ein automatisches Akzeptieren des Zertifikats würde dieses dem Keystore des Android-Geräts hinzufügen, und dies ist untersagt.

Importieren nicht möglich!

Der Grund ist recht einfach. Ein Browser auf einem Nicht-Android-Gerät bringt seinen eigenen Keystore gleich mit. Diesem kann man dann eigene Zertifikate hinzufügen, indem man diese explizit akzeptiert. Jeder andere Browser und jede andere Anwendung bekommt davon nichts mit. Bei Android ist der Keystore aber Teil des Betriebssystems, und das Hinzufügen eines weiteren Zertifikats hätte zur Folge, dass jede andere Android-Anwendung ebenfalls diesem Zertifikat vertrauen würde, da sie ja auf eben diesen zentralen Keystore zugreift. Damit hätte man sofort ein großes Sicherheitsleck, und daher hat sich Google entschlossen, dies in Android nicht zuzulassen.

Da aber gerade kleinere Unternehmen oft mit selbst erstellten Zertifikaten arbeiten, um damit z.B. den E-Mail-Server zu betreiben (*IMAP über SSL*), hat dies zur Folge, dass einige Android-Standardprogramme nicht verwendet werden können. Ein Verbindungsaufbau über SSL ist nicht möglich.

Achtung!

Android gestattet es nicht, selbst erstellte SSL-Zertifikate zu verwenden. Nur Zertifikate einer offiziellen Zertifizierungsstelle können zum Aufbau einer SSL-Verbindung genutzt werden. Dies ist in aller Regel mit Kosten und zusätzlichem Aufwand für den Betreiber des Servers verbunden.

Folglich wäre die einzige Möglichkeit, ein Zertifikat zu kaufen, wenn wir SSL-verschlüsselte Verbindungen aufbauen wollen. Dieses muss natürlich passend zu einem der im Android-Keystore vorinstallierten Zertifikate gewählt werden.

Bevor wir zeigen, wie man doch selbst erstellte Zertifikate nutzen kann, möchten wir eine »Lösung« vorstellen, die im Internet kursierte. Sie basierte auf einer Sicherheitslücke, die inzwischen geschlossen wurde. Es handelt sich um den sogenannten »*Android Jailbreak*«. Damit ist es möglich, Root-Rechte für das *G1*-Mobiltelefon von *HTC* zu erhalten. Auf diese Weise wäre es zum Beispiel möglich, den Keystore des Android-Geräts um eigene Zertifikate zu erweitern. Dazu schaltet

Der Android-Jailbreak

man am *G1* das WLAN an und ermittelt die IP-Adresse des Geräts. Mittels eines frei verfügbaren Programms namens `PTerminal`, welches man vom Android-Market herunterladen kann, kann man den Telnet-Dämon `telnetd` auf dem *G1* starten. Nun kann man sich von einem beliebigen Rechner im WLAN per Telnet über das WLAN mit dem Android-Gerät verbinden und hat Root-Rechte.

Mittels eines Remount der Systemverzeichnisse mit Lese- und Schreibrechten kann man einfach den zentralen Android-Keystore austauschen bzw. erweitern, so dass er anschließend das eigene Zertifikat enthält. Man riskiert bei einem solchen Eingriff ins System allerdings, das Android-Gerät zu zerstören, da man Systemdateien ändert. Ein Fehler bei der Durchführung könnte fatale Folgen haben.

Wir erwähnen den Android-Jailbreak hier, da es sich um eine Sicherheitslücke handelte. Auch wenn sie inzwischen geschlossen wurde, könnte Android noch andere Sicherheitslücken enthalten, die bisher noch nicht gefunden wurden. Diese potenziellen Sicherheitslücken sind ein wichtiges Argument für Abschnitt 18.2.2, in dem es um Verschlüsselung der Daten im Speicher des Android-Geräts geht.

Und es geht doch!

Um doch selbst erstellte Zertifikate verwenden zu können, geben wir der Anwendung einfach einen *eigenen* Keystore mit. Dieser enthält das selbst erstellte Zertifikat. Damit ist der Keystore nur für die Zertifikate dieser einen Anwendung zuständig, und wir haben kein Sicherheitsleck.

Wir brauchen dazu als Erstes einen Keystore, der das Zertifikat der gewünschten Webseite aufnimmt. Wir werden im Beispiel den Keystore mit einem selbst erstellten Zertifikat bestücken, welches wir uns vorher von der Webseite, zu der wir die SSL-Verbindung aufbauen wollen, geholt haben. Der Keystore wird dann der Android-Anwendung als Ressource mitgegeben.

Mit selbst erzeugten Zertifikaten arbeiten

Ziel

Eine Anwendung, die eine SSL-Verbindung mit einem Server aufbauen kann, der ein selbst erstelltes Zertifikat verwendet.

Was ist zu tun?

- Schritt 1: Zertifikat von der Webseite abrufen
- Schritt 2: Zertifikat konvertieren
- Schritt 3: Den Bouncy-Castle-Provider im JDK installieren
- Schritt 4: Der Anwendung den Keystore als Ressource hinzufügen

Lösung

Schritt 1: Zertifikat von der Webseite abholen

Ein SSL-Zertifikat abholen

Zunächst brauchen wir das passende Zertifikat. Wenn wir es nicht schon selbst haben, können wir den Internet Explorer verwenden, um ein SSL-Zertifikat von der Webseite abzuholen, zu der wir später die SSL-Verbindung aufbauen möchten. Wir sprechen hier nun von X.509-Zertifikaten, um deutlich zu machen, dass nahezu alle Webbrowser mit SSL-Zertifikaten arbeiten, die nach dem X.509-Standard erstellt wurden.

Wir rufen im Browser die URL der Webseite auf. Wenn der Webserver ein selbst erstelltes X.509-Zertifikat verwendet, wird in der Browserleiste ein Zertifikatsfehler angezeigt. Wenn wir daraufklicken, können wir das Zertifikat im Keystore des Browsers installieren (siehe Abb. 18-2).

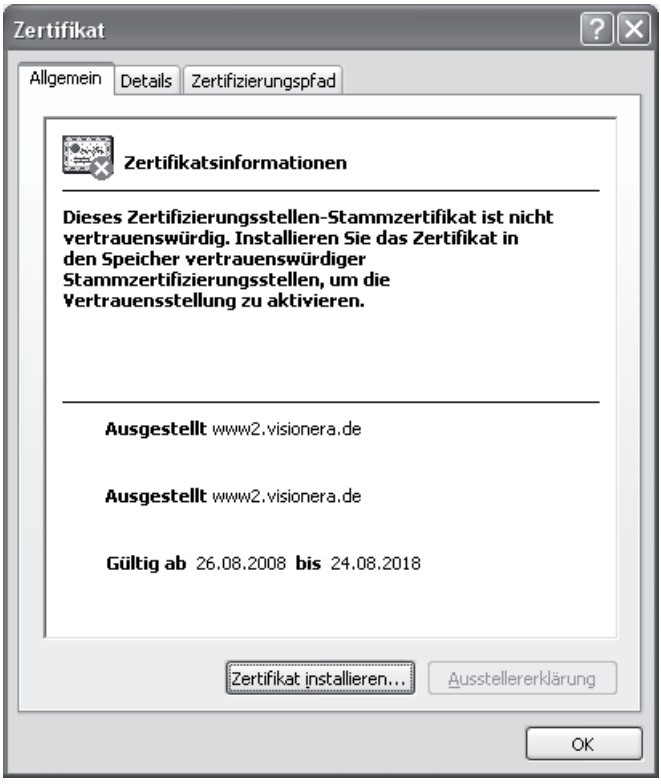


Abb. 18-2
Browser-Zertifikat installieren (Internet Explorer)

Schritt 2: Zertifikat konvertieren Um das Zertifikat im Emulator verwenden zu können, müssen wir es als X.509-Zertifikat aus dem Keystore des Internet Explorer exportieren und dabei gleichzeitig konvertieren. Dazu geht man auf `Extras -> Internetoptionen -> Inhalte -> Zertifikate`. Unter dem Reiter *Vertrauenswürdige Stammzertifikate* finden wir dann unser Zertifikat. Wir klicken auf die Schaltfläche *Exportieren...* (siehe Abb. 18-3).

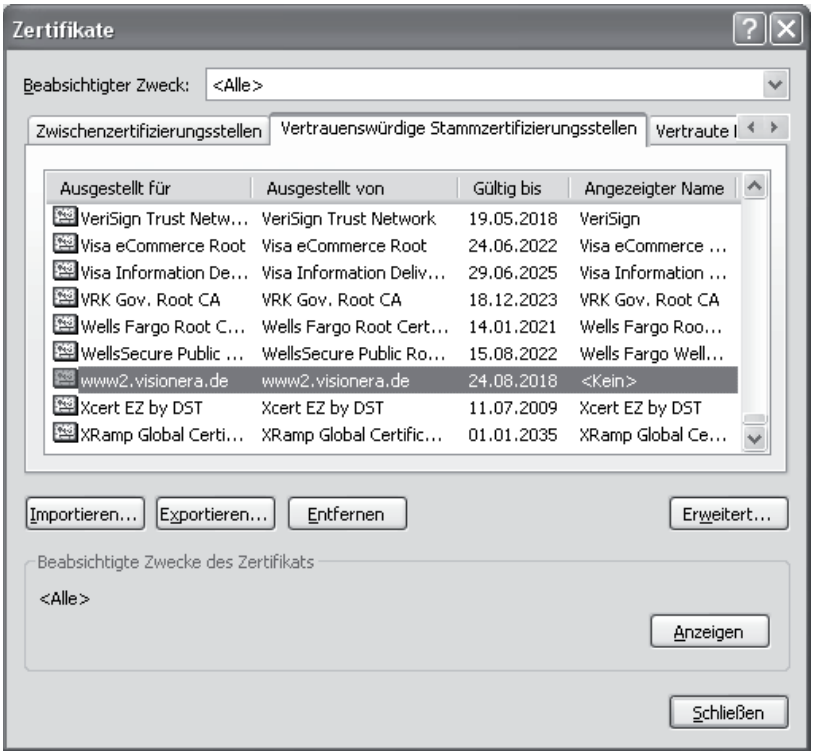


Abb. 18-3 *Browser-Zertifikat exportieren (Internet Explorer)*

Zertifikatsexport mit dem Assistenten

Es wird der Zertifikatsexport-Assistent gestartet. Als *gewünschtes Format* sollte man *»DER-codiert-binär X.509 (.CER)«* wählen. Anschließend wählt man einen Dateinamen (ohne Endung) und den Speicherort aus.

Schritt 3: Bouncy-Castle-Provider im JDK installieren
Laden Sie den BKS-Provider von *Bouncy Castle* herunter (`http://bouncycastle.org`). Diesen finden Sie unter *»latest releases«* in der rechten Textspalte und dann in der Rubrik *»provider«*. Achten Sie darauf, dass Sie den Provider passend zu Ihrer Version des JDK auswählen. Es handelt sich um eine Jar-Datei (`bcprov-jdk[version].jar`).

Kopieren Sie nun den BKS-Provider in das `\jre\lib\ext`-Verzeichnis Ihres JDK.

Das JDK erweitern

Passen Sie im Verzeichnis `\jre\lib\security` die Datei `java.security` an, indem Sie in der Liste der Provider folgende Zeile hinzufügen:

```
security.provider.7=
    org.bouncycastle.jce.provider.BouncyCastleProvider
```

Passen Sie ggf. die fortlaufende Nummer (»7«) an.

Schritt 4: Einen Keystore selbst erstellen

Für unsere Zwecke reicht es, einen neuen Keystore anzulegen. Dieser Keystore enthält dann nur das eine X.509-Zertifikat als öffentlichen Schlüssel für unsere Anwendung, um später eine SSL-Verbindung aufbauen zu können. Das Zertifikat haben wir uns gerade von der Webseite geholt, zu der wir eine Verbindung aufbauen wollen, und fügen es dem neuen Keystore hinzu:

```
keytool -v -import -storetype BKS
    -storepass meinpasswort
    -trustcacerts -alias meinalias
    -file D:\temp\meinzertifikat.cer
    -keystore D:\workspace\meinProjekt\res\raw\
        owncerts.bks
```

Als `storetype` können wir nun BKS angeben, da unser JDK den BouncyCastle-Provider kennt. Mittels `storepass` setzen wir ein eigenes Passwort für den Keystore. Auch den Aliasnamen (`alias`) können wir frei vergeben. Über den Parameter `file` importieren wir unser in Schritt 2 abgespeichertes Zertifikat. Mit `keystore` geben wir dem Keystore schließlich seinen Namen. Wir speichern die Datei direkt in unserem Android-Projekt, und zwar im Ressourcen-Ordner. Dort müssen wir, falls noch nicht vorhanden, einen weiteren Ordner namens `raw` anlegen.

Keystore als Ressource hinzufügen

Nun können wir noch kontrollieren, ob das Zertifikat auch wirklich im Keystore gelandet ist:

```
keytool -v -list -storetype BKS
    -storepass meinpasswort
    -alias meinalias -keystore
    D:\workspace\meinProjekt\res\raw\owncerts.bks
```

Einen eigenen Keystore pro Anwendung verwalten

Kommen wir nun zum Programmcode. Wir werden insgesamt drei verschiedene Arten des SSL-Verbindungsaufbaus vorstellen, zwei über HTTPS und einen über eine Socket-Verbindung.

Keystore laden

Wir wollen, dass die Anwendung auf unseren eigenen Keystore zugreift, wenn eine SSL-Verbindung aufgebaut wird. Dazu müssen wir den selbst erstellten Keystore aus dem Ressourcenordner laden und verwenden.

Um eine SSL-Verbindung aufzubauen, brauchen wir einen `X509TrustManager` (`javax.net.ssl.X509TrustManager`). Trust Manager kennen die Zertifikate aus dem Keystore und kümmern sich bei einem SSL-Handshake mit dem Server um dessen Authentifizierung. Zusätzlich können sie eine Gültigkeitsprüfung des Serverzertifikats vornehmen. Ist dessen Gültigkeit abgelaufen, kann ein Verbindungsaufbau abgelehnt werden.

Da wir den Keystore mit unserem X.509-Zertifikat aus dem Dateisystem laden, können wir einen solchen `X509TrustManager` selbst schreiben und dafür sorgen, dass er beim Verbindungsaufbau verwendet wird. Darin liegt der ganze Trick.

Wenn wir einen `X509TrustManager` selbst implementieren, müssen wir dabei seine Methode `checkServerTrusted` überschreiben. Denn hier findet die Prüfung des Servers beim SSL-Handshake statt. Solange wir dafür sorgen, dass der selbst implementierte `X509TrustManager` unser eigenes Zertifikat aus dem Keystore verwendet, wird die Prüfung klappen. Da wir das Zertifikat vom Server abgerufen haben, muss es zum Serverzertifikat passen.

Trust Manager ausgetrickst

Der Server wird damit als vertrauenswürdig eingestuft, und einem Verbindungsaufbau steht nichts im Wege. Damit haben wir den Standardweg ausgehebelt, bei dem der `X509TrustManager` der Laufzeitumgebung verwendet wird. Dieser würde auf den Android-eigenen Keystore `cacerts.bks` zugreifen, der zentral für alle Anwendungen im Systemordner `\system\etc\security` liegt.

Trust Manager werden durch eine Trust-Manager-Factory verwaltet (`javax.net.ssl.TrustManagerFactory`). Diese wird beim SSL-Verbindungsaufbau benötigt. Wir werden nun eine eigene Trust-Manager-Factory implementieren, die unseren eigenen Trust Manager verwaltet.

Listing 18.1 *Eine eigene Trust-Manager-Factory implementieren*

```
package de.androidbuch.ssl;
...
public class TrustManagerFactory {

  private static X509TrustManager serverTrustManager;

  private static KeyStore keystore;
  private static final String keystorePasswort =
      "meinpasswort";
```

```
private static class MeinX509TrustManager // (6)
    implements X509TrustManager {

  private MeinX509TrustManager() {
  }

  public static X509TrustManager getInstance() {
    return new MeinX509TrustManager();
  }

  public void checkClientTrusted(X509Certificate[]
      chain, String authType) throws
      CertificateException {
  }

  public void checkServerTrusted( // (7)
      X509Certificate[] chain, String authType)
      throws CertificateException {
    try {
      serverTrustManager.checkServerTrusted(chain, // (8)
          authType);
      chain[0].checkValidity(); // (9)
    }
    catch (CertificateException e) {
      throw new CertificateException(e.toString());
    }
  }

  public X509Certificate[] getAcceptedIssuers() {
    return new X509Certificate[] {};
  }
} // Ende der inneren Klasse

public static void ladeKeyStore(InputStream is) // (1)
    throws CertificateException {
  try {
    keystore = KeyStore.getInstance(
        KeyStore.getDefaultType());
    keystore.load(is, // (2)
        keystorePasswort.toCharArray());

    javax.net.ssl.TrustManagerFactory tmf =
       javax.net.ssl.TrustManagerFactory.getInstance(
       "X509"); // (3)

    tmf.init(keyStore); // (4)
```

```
      TrustManager[] tms = tmf.getTrustManagers();
      if (tms != null) {
        for (TrustManager tm : tms) {
          if (tm instanceof X509TrustManager) {
            serverTrustManager =
                (X509TrustManager)tm; // (5)
            break;
          }
        }
      }
    }
    catch (Exception e) {
      keystore = null;
      throw new CertificateException(e);
    }
  }

  public static X509TrustManager get() {
    return MeinX509TrustManager.getInstance();
  }

  public static KeyStore getKeyStore() {
    return keystore;
  }
}
```

Betrachten wir zunächst die Methode `ladeKeyStore(InputStream is)` (1). Wir initialisieren das Attribut `keystore` und laden mit Hilfe des Passworts und eines `InputStream` (wird in Listing 18.2 erzeugt) den Keystore aus dem Dateisystem (2).

TrustManagerFactory initialisieren

Als Nächstes verwenden wir eine Instanz der Klasse `javax.net.ssl.TrustManagerFactory` (3). Diese initialisieren wir mit dem gerade geladenen Keystore (4). Über die Factory können wir uns ein Array vom Typ `TrustManager` geben lassen. Da wir in unserem Keystore mindestens ein X.509-Zertifikat haben, werden wir auch einen `X509TrustManager` erhalten. Daher durchlaufen wir in der `for`-Schleife das Array und speichern den `X509TrustManager` in dem Attribut `serverTrustManager` (5).

Kommen wir nun zur inneren Klasse `MeinX509TrustManager` (6). Sie implementiert das Interface `javax.net.ssl.X509TrustManager`, d.h., wir müssen einige Methoden überschreiben. Wichtig ist hierbei vor allem die Methode `checkServerTrusted` (7). Sie wird aufgerufen, wenn wir eine SSL-Verbindung zum Server aufbauen. Wir reichen den Methodenaufruf an unseren eigenen `X509TrustManager`, gespeichert in dem Attribut namens `serverTrustManager`, weiter. Der Übergabeparameter `chain`

enthält die Zertifikatskette des Servers, `authType` den Namen des Verschlüsselungsalgorithmus (z.B. »*RSA*«). Unser Zertifikat wird so gegen die Zertifikatskette des Servers geprüft (8).

Selbst verantwortlich für die Sicherheit

An dieser Stelle im Code haben wir die Sicherheit selbst in der Hand. Die Zeile (`chain[0].checkValidity()`) prüft, ob das Zertifikat gültig oder abgelaufen ist (9).

Nun haben wir alle Bausteine zusammen, die wir brauchen. Wenn wir jetzt von außen die `get`-Methode auf unserer `de.androidbuch.ssl.TrustManagerFactory` aufrufen, erhalten wir eine Instanz von `MeinX509TrustManager`, die die Zertifikatsprüfung mittels der Methode `checkServerTrusted` gegen einen `X509TrustManager` (gespeichert in `serverTrustManager`) vornimmt, den wir selbst aus dem Keystore erzeugt haben.

Als nächsten Schritt nehmen wir an, dass wir die SSL-Verbindung aus einer Activity heraus aufbauen wollen. Für die folgenden Beispiele setzen wir voraus, dass das Laden des Keystores in der `onCreate`-Methode der Activity erfolgt:

Listing 18.2
Initialisierung der eigenen TrustManagerFactory

```
public void onCreate(Bundle icicle) {
  super.onCreate(icicle);
  ...
  is = this.getResources().
        openRawResource(R.raw.owncerts);
  try {
    de.androidbuch.ssl.TrustManagerFactory.
        ladeKeyStore(is);
  }
  catch (CertificateException e) { }
  ...
}
```

Keystore als Ressource laden

Da wir unseren Keystore namens `owncerts.bks` als Ressource im Projektordner `\res\raw` abgelegt haben, erhalten wir über die Methode `getResources().openRawResource(r.raw.owncerts)` einen `InputStream` zum Laden des Keystores. Diesen übergeben wir unserer `TrustManagerFactory`.

HTTPS-Client mit javax.net.ssl

Kommen wir nun zu unserem ersten SSL-Client, der eine HTTPS-Verbindung zum Server aufbauen soll.

Listing 18.3
HTTPS-Client mit javax.net.ssl

```
try {
  javax.net.ssl.SSLContext sslContext =
      SSLContext.getInstance("TLS"); // (1)
  sslContext.init(null, new TrustManager[] {
```

```
        de.androidbuch.ssl.TrustManagerFactory.get()},
        new SecureRandom()); // (2)
    javax.net.ssl.SSLSocketFactory socketFactory =
        sslContext.getSocketFactory();

    HttpsURLConnection.setDefaultSSLSocketFactory(
        socketFactory); // (3)
    HttpsURLConnection.setDefaultHostnameVerifier(
        new TestX509HostnameVerifier()); // (4)

    HttpsURLConnection httpsUrlConnection =
        (HttpsURLConnection) new URL(
        https://www2.visionera.de/android/").
        openConnection();
    httpsUrlConnection.connect();

    // HTML der Webseite auslesen:
    InputStream in = httpsUrlConnection.getInputStream();
    BufferedReader br =
        new BufferedReader(new InputStreamReader(in));
    String line;
    while((line = br.readLine()) != null) {
      Log.d(TAG, line);
    }
    in.close();
  }
  catch (Exception e) { }
```

Unser Client holt sich zunächst einen `SSLContext` (1). Als Protokoll geben wir »*TLS*« an, was für *Transport Layer Security* steht und von Android unterstützt wird. TLS ist eine Weiterentwicklung von SSL 3.0.

Prüfung gegen das eigene Zertifikat

Nun initialisieren wir den `SSLContext` mit unserem eigenen `X509TrustManager` aus unserer `TrustManagerFactory` (2). Die Prüfung, ob der Server vertrauenswürdig ist, erfolgt also letztendlich gegen unser eigenes Zertifikat. Die `init`-Methode verlangt entweder ein Array von Key Managern oder ein Array von Trust Managern. Daher setzen wir den ersten Parameter auf `null`. Als zweiten Parameter übergeben wir das Trust Manager-Array. Wir initialisieren ein solches Array, indem wir uns über unsere eigene `TrustManagerFactory` den als innere Klasse implementierten `MeinX509TrustManager` holen (siehe Listing 18.1).

Die Variable `sslContext` liefert uns nun eine `SSLSocketFactory`, die wir einer `HttpsURLConnection` mittels einer statischen Methode übergeben (3).

Da wir unserem Server vertrauen, schreiben wir uns unseren eigenen `X509HostnameVerifier` (4). Er muss nicht viel können, da wir ihn

eigentlich nicht brauchen. Es ist ja schließlich unsere eigene Webseite und unser eigenes Zertifikat.

Listing 18.4 Dummy-X509HostnameVerifier

```
class TestX509HostnameVerifier implements
    X509HostnameVerifier {
  public boolean verify(String host,
      SSLSession session) {
    return true;
  }

  public void verify(String host, SSLSocket ssl) throws
      IOException { }

  public void verify(String host, String[] cns,
      String[] subjectAlts) throws SSLException { }

  public void verify(String host, X509Certificate cert)
      throws SSLException { }

  public void verify(String arg0, java.security.cert.
      X509Certificate arg1) throws SSLException { }
}
```

Auch den `TestX509HostnameVerifier` übergeben wir mittels einer statischen Methode an `HttpsURLConnection`. Dies ist notwendig, da wir sonst eine `SSLException` erhalten, die uns mitteilt, dass dem Serverzertifikat nicht vertraut wurde.

Verbindung kommt zustande.

Nun öffnen wir eine Verbindung zu unserem Server und starten sie mittels der Methode `connect`. Stimmen nun Serverzertifikat und das Zertifikat in unserem Keystore überein, so kommt eine Verbindung zustande, und wir können uns aus dem Verbindungsobjekt `httpsUrlConnection` einen `InputStream` holen und den Inhalt der Webseite auslesen. Das reicht uns als Test, um zu prüfen, ob eine SSL-Verbindung mit der Webseite hergestellt werden konnte. Im Erfolgsfall erhalten wir von der URL `www2.visionera.de/android/` eine einfache HTML-Seite zurück, die den Text »*Hallo Android*« enthält.

HTTPS-Client mit org.apache.http.conn.ssl

Als Alternative wollen wir noch zeigen, wie man mit den Implementierungen von Apache, die in Android enthalten sind, eine HTTPS-Verbindung zum Server aufbauen kann. Wir lassen die `onCreate`-Methode, wie sie ist, können aber auf den `TextX509HostnameVerifier` verzichten.

Listing 18.5 *HTTPS-Client mit den Apache-SSL-Klassen*

```
try {
  KeyStore keyStore =
      TrustManagerFactory.getKeyStore(); // (1)
  org.apache.http.conn.ssl.SSLSocketFactory
      socketFactory;

  socketFactory = new
      org.apache.http.conn.ssl.SSLSocketFactory(
      keyStore); // (2)
  socketFactory.setHostnameVerifier(org.apache.http.
      conn.ssl.SSLSocketFactory.
      ALLOW_ALL_HOSTNAME_VERIFIER); // (3)

  Scheme https =
      new Scheme("https", socketFactory, 443); // (4)
  DefaultHttpClient httpclient =
      new DefaultHttpClient();
  httpclient.getConnectionManager().
      getSchemeRegistry().register(https);

  HttpGet httpget = new HttpGet(
      "https://www2.visionera.de/android/");

  HttpResponse response = httpclient.execute(httpget);
  HttpEntity entity = response.getEntity();

  // HTML der Webseite auslesen:
  InputStream in = entity.getContent();

  // weiter wie oben...

}
catch (Exception e) { }
```

Bei der Apache-Implementierung gehen wir ein wenig anders vor. Wir holen uns aus der in Listing 18.1 implementierten `TrustManagerFactory` nur den Keystore (1). Wir können also auf einen Teil dieser Klasse verzichten.

Keine Hostnamen-Verifizierung

Anschließend wird die `SSLSocketFactory` mit dem Keystore initialisiert (2). Um auf die Verifizierung des Hostnamens zu verzichten, können wir in der `SSLSocketFactory` `ALLOW_ALL_HOSTNAME_VERIFIER` setzen (3). Als Verbindungsschema wählen wir HTTPS und übergeben dem Schema im Konstruktor die `socketFactory`, die den Keystore enthält (4).

Mit dem `DefaultHttpClient` haben wir ein mächtiges Werkzeug, um HTTP(S)-Verbindungen herzustellen und zu verwenden. Dadurch, dass wir das HTTPS-Schema im Connection Manager des `DefaultHttpClient`

registrieren, kann der Client auf den Keystore zurückgreifen, um das Serverzertifikat zu verifizieren. Nachdem der Client initialisiert ist, können wir ganz normal einen `GET`-Befehl definieren und ausführen. Wir lesen die Response des Servers über das `HttpEntity`-Objekt aus und holen uns wieder einen `InputStream`. Der Rest der Methode zum Ausgeben des Inhalts der Webseite ist analog zum vorherigen Beispiel.

SSL-Socket-Verbindung mit javax.net.ssl

Nicht immer will man über HTTPS eine Verbindung aufbauen. Für viele Anwendungsfälle benötigt man Socket-Verbindungen. Wir zeigen nun noch, wie man eine SSL-Socket-Verbindung aufbaut. Das Beispiel nutzt den gleichen Code wie im ersten Beispiel, um eine `SSLSocketFactory` zu erhalten.

Listing 18.6
Ein SSL-Socket-Client

```
try {
  javax.net.ssl.SSLContext sslContext =
    SSLContext.getInstance("TLS");
  sslContext.init(null, new TrustManager[] {
      TrustManagerFactory.get() }, new SecureRandom());
  javax.net.ssl.SSLSocketFactory socketFactory =
      sslContext.getSocketFactory();

  Socket mSocket =
      socketFactory.createSocket("www.verisign.com", 443);

  BufferedWriter out = new BufferedWriter(new
  OutputStreamWriter(mSocket.getOutputStream()));
  BufferedReader in = new BufferedReader(
      new InputStreamReader(mSocket.getInputStream()));
  out.write("GET / HTTP/1.0\n\n");
  out.flush();

  String line;
  while((line = in.readLine()) != null) {
      Log.d(TAG, line);
  }
  out.close();
  in.close();
}
catch (Exception e) {
  e.printStackTrace();
}
```

Diesmal mit beglaubigtem Zertifikat

Wir bauen diesmal eine Verbindung zu `www.verisign.com` auf. *Verisign* ist eine bekannte Zertifizierungsstelle (CA). Android hat in seinem eingebauten Keystore (`cacerts.bks`) natürlich auch ein Zertifikat von Verisign. Somit klappt der Handshake zwischen Verisign und unserer Anwendung, ohne dass wir unsere selbst implementierte `TrustManagerFactory` einsetzen müssen. Allerdings müssen wir das Import-Statement

```
import javax.net.ssl.TrustManagerFactory
```

anpassen und die `onCreate`-Methode bereinigen, damit wieder die `TrustManagerFactory` des Systems verwendet wird.

18.2.2 Daten oder Objekte verschlüsseln

Nun wissen wir zwar, wie wir Daten sicher mit einem Server austauschen, aber wenn unser Android-Gerät in falsche Hände fällt, könnten auch die Daten ausgespäht werden. Speichern wir die Daten unverschlüsselt im Dateisystem oder in einer Datenbank auf dem Gerät, so gehen wir ein weiteres Sicherheitsrisiko ein.

Einfache Sache...

Zum Glück ist das Verschlüsseln keine schwere Sache. Wir wollen hier nur ein einfaches Beispiel geben, was aber für den Einstieg durchaus ausreichend ist. Es gibt sehr viel Literatur, die das Thema vertieft, falls ein besonders hoher Sicherheitsstandard erreicht werden soll.

Um eine Nachricht zu verschlüsseln, brauchen wir einen Schlüssel. Diesen kann man sich automatisch mit Hilfe der Klasse `KeyGenerator` generieren lassen. Wenn wir den Schlüssel haben, ist das Ver- und Entschlüsseln eine Kleinigkeit. Aber schauen wir uns zunächst den Quellcode an.

Listing 18.7 *Eine Methode zum Verschlüsseln von Texten*

```
private byte[] verschluesseln(String text) {
  try {
    KeyGenerator keyGen =
        KeyGenerator.getInstance("AES");
    keyGen.init(128);
    secretKey = keyGen.generateKey();

    Cipher cipher = Cipher.getInstance("AES");
    cipher.init(Cipher.ENCRYPT_MODE, secretKey);

    byte[] verschluesselt =
        cipher.doFinal(text.getBytes());
    Log.d(TAG, "verschluesselter Text: " +
        new String(verschluesselt));
```

```
    FileOutputStream fos;
    fos = openFileOutput("geheimnis.enc", MODE_PRIVATE);
    BufferedOutputStream bos =
        new BufferedOutputStream(fos);
    CipherOutputStream cos =
        new CipherOutputStream(bos, cipher);

    cos.write(text.getBytes());
    cos.close();

    return verschluesselt;
  }
  catch (Exception e) { }

  return null;
}
```

Einen geheimen Schlüssel automatisch generieren

Wir übergeben der Methode den Text, den wir verschlüsseln wollen. Mit Hilfe der Klasse `KeyGenerator` erzeugen wir automatisch unseren geheimen Schlüssel und speichern ihn in dem Attribut `secretKey`, damit wir später beim Entschlüsseln darauf zurückgreifen können. Statt des Verschlüsselungsalgorithmus »*AES*« können wir den Key-Generator mit »*DES*« initialisieren, aber dann muss der erzeugte Schlüssel eine bestimmte Länge haben. »*RSA*« käme auch in Frage, eignet sich aber nur für sehr kleine Datenpakete. RSA wird meist zur Generierung von Schlüsseln für die symmetrische Verschlüsselung verwendet und eignet sich nicht zum Verschlüsseln großer Datenmengen.

Für unser Beispiel nehmen wir den AES-Algorithmus. Mit diesem Algorithmus erzeugen wir ein Objekt vom Typ `javax.crypto.Cipher`. Mit Hilfe dieser Klasse haben wir ein mächtiges Werkzeug in der Hand. Wir initialisieren es im Verschlüsselungsmodus (`Cipher.ENCRYPT_MODE`) zusammen mit dem geheimen Schlüssel.

doFinal-Methode

Um nun den Text zu verschlüsseln, rufen wir die Methode `doFinal` auf. Sie bekommt den zu verschlüsselnden Text als Byte-Array übergeben. Das Ergebnis ist wiederum ein Byte-Array. Um zu prüfen, ob eine Verschlüsselung stattgefunden hat, geben wir es in der *LogCat* als Text aus. Dieses Byte-Array könnten wir jetzt in einer Datenbank speichern und hätten unsere Daten sicher persistiert.

Eine Datei als Ziel

Wir wollen aber noch mehr. Wenn wir keine Datenbank brauchen, reicht es vielleicht, den verschlüsselten Text im Dateisystem zu speichern. Hier hilft uns die Klasse `javax.crypto.CipherOutputStream`. Sie ist von `OutputStream` abgeleitet. Beim Anlegen nimmt sie einen ebenfalls von `OutputStream` abgeleiteten Stream entgegen. Zusätzlich wird sie mit unserem `Cipher`-Objekt erzeugt. Dadurch kann `CipherOutputStream` alles

verschlüsseln, was die Klasse durchläuft. Also kann unser Ziel auch ein `FileOutputStream` sein, durch den wir unsere verschlüsselten Daten in eine Datei schreiben.

Wie wir in Kapitel 10 gelernt haben, können wir innerhalb einer Activity die Methode `openFileOutput` nutzen und erhalten einen `FileOutputStream`. Wir setzen die Streamkette fort, wie im Beispielcode gezeigt. Um das Byte-Array zu speichern, können wir direkt die Klasse `CipherOutputStream` verwenden. Mit Hilfe ihrer `write`-Methoden kann man problemlos Byte-Arrays verarbeiten. Wir übergeben unseren Klartext als Byte-Array, und `CipherOutputStream` verschlüsselt dieses für uns. Die Streamkette wird durchlaufen, und schließlich schreibt `FileOutputStream` die verschlüsselte Zeichenkette in unsere Datei `geheimnis.enc`.

Das Entschlüsseln erfolgt analog:

Listing 18.8 *Eine Methode zum Entschlüsseln von Texten*

```
private String entschluesseln(byte[] verschluesselt) {
  try {
    Cipher cipher = Cipher.getInstance("AES");
    cipher.init(Cipher.DECRYPT_MODE, secretKey);

    byte[] entschluesselt =
        cipher.doFinal(verschluesselt);
    String original = new String(entschluesselt);
    Log.d(TAG, "Originaltext: " + original);

    FileInputStream fis;
    fis = openFileInput("geheimnis.enc");
    BufferedInputStream bis = new
        BufferedInputStream(fis);
    CipherInputStream cis =
        new CipherInputStream(bis, cipher);

    byte[] buffer = new byte[128];
    StringBuffer sb = new StringBuffer();
    while ((cis.read(buffer)) >= 0) {
      sb.append((new String(buffer)));
    }
    cis.close();

    original = sb.toString().trim();
    Log.d(TAG, "Originaltext: " + original);

    return original;
  }
  catch (Exception e) { }
```

```
  return null;
}
```

Für die Entschlüsselung gehen wir den umgekehrten Weg. Da wir nicht wissen, wie lang die Datei ist, die die verschlüsselten Daten enthält, erfolgt das Lesen des verschlüsselten Textes blockweise in der `while`-Schleife. Wir nutzen zur Pufferung einen `StringBuffer`, der uns abschließend den kompletten Originaltext liefert. Der Aufruf der Methode `trim` ist wichtig, da aufgrund der festen Blocklänge wahrscheinlich Leerzeichen am Ende des dekodierten Textes stehen.

18.2.3 Verschlüsselung anwenden

Wir wissen nun, wie wir mit einer Android-Anwendung sicher Daten per HTTPS oder SSL-Sockets an einen Server übertragen bzw. sicher von dort empfangen können. Weiterhin wissen wir, wie wir die Daten in unserer Anwendung ver- und entschlüsseln können, um sie zu speichern. Potenziell haben wir aber immer noch ein Sicherheitsleck. Kurzzeitig liegen die Daten in unserer Anwendung im Klartext vor, da an den Endpunkten der SSL-Verbindung eine automatische Entschlüsselung stattfindet. Besser wäre es doch, wenn wir die Daten verschlüsseln, dann ggf. per SSL übertragen (doppelte Sicherheit) und schließlich verschlüsselt in unserer Anwendung speichern. Erst wenn wir die Daten innerhalb der Anwendung anzeigen wollen, laden wir sie, immer noch verschlüsselt, aus der Datenbank, entschlüsseln sie und bringen sie im Klartext zur Anzeige.

Sicherheitsleck

Wir wollen hier nur kurz ein Verfahren vorstellen, wie wir unabhängig von HTTPS und SSL-Sockets Daten verschlüsselt austauschen können. Alles, was dafür notwendig ist, haben wir schon. Wenn wir den Server entsprechend programmieren, können wir mit ihm Daten sicher austauschen.

Ein gängiges Verfahren

In der Praxis verwendet man häufig eine Kombination aus asymmetrischer und symmetrischer Verschlüsselung. Man verwendet die asymmetrische Verschlüsselung für den Schlüsselaustausch des geheimen Schlüssels des symmetrischen Verfahrens. Ein geheimer Schlüssel ist nicht sehr lang, und daher stellt die asymmetrische Verschlüsselung einer so kleinen Datenmenge auch auf einem mobilen Computer kein großes Problem dar, was die Laufzeit angeht. Die einzelnen Schritte sind folgende:

Beide Verfahren kombinieren

Asymmetrischer Teil

- Der Empfänger erzeugt einen privaten und einen öffentlichen Schlüssel.
- Der Sender lädt den öffentlichen Schlüssel des Empfängers über das Internet.
- Der Sender erzeugt einen neuen, geheimen Schlüssel für das symmetrische Verschlüsselungsverfahren.
- Der Sender verschlüsselt seinen geheimen Schlüssel mit dem öffentlichen Schlüssel des Empfängers und schickt das Ergebnis an den Empfänger.
- Der Empfänger entschlüsselt mit seinem privaten Schlüssel den geheimen Schlüssel des Senders. Ein sicherer Schlüsselaustausch für eine symmetrische Verschlüsselung hat stattgefunden.

Symmetrischer Teil

- Der Sender verschlüsselt seine Nachricht mit seinem geheimen Schlüssel und schickt das Ergebnis an den Empfänger.
- Der Empfänger dekodiert die Nachricht mit dem eben vom Sender erhaltenen geheimen Schlüssel.

Mit den hier vorgestellten Beispielen lässt sich schon sehr viel erreichen. Wir können unsere Android-Anwendung sicher machen, sowohl bei der Datenübertragung als auch beim Verarbeiten und Speichern der Daten auf dem Gerät. Wir haben zwar nur an der Oberfläche des Themas Sicherheit und Verschlüsselung gekratzt, konnten aber feststellen, dass sich sichere Anwendungen schon mit relativ wenig Codezeilen erreichen lassen.

19 Unit- und Integrationstests

Hinweis: Die 2. Auflage dieses Buchs erscheint unter dem Titel
»Android 2. Grundlagen und Programmierung«
Ende Mai 2010 im dpunkt.verlag, ISBN 978-3-89864-677-2

Aus der professionellen Softwareentwicklung sind Entwicklertests nicht mehr wegzudenken. Wir wollen uns daher auch in diesem Einsteigerbuch kurz mit diesem Thema auseinandersetzen. Grundkenntnisse bei der Durchführung von JUnit-Tests setzen wir im Folgenden voraus ([1], [20]).

Nach der Lektüre dieses Kapitels werden Sie in der Lage sein, JUnit-Testfälle für alle Elemente einer Android-Anwendung zu erstellen und durchzuführen. Leider würde eine ausführliche Diskussion aller Testmöglichkeiten den Rahmen dieses Buches sprengen. Daher endet das Kapitel mit einer Auflistung möglicher Einstiegspunkte zum Selbststudium.

19.1 Allgemeines

Testen ja! Aber was?

Bevor wir uns den technischen Details der Testimplementierung zuwenden, sollten wir uns fragen, *was genau* wir testen wollen. Schließlich ist jeder Test mit einigem Programmier- und Pflegeaufwand verbunden.

Testwürdige Elemente einer Anwendung sind zunächst einmal alle Android-Komponenten, von der Activity bis zum Service. Darüber hinaus sollten auch alle Klassen verprobt werden können, die Hilfs- oder Querschnittsfunktionalität für die Android-Komponenten bereitstellen. Wir unterscheiden fortan drei Kategorien:

1. Klassen, die ohne Android-SDK lauffähig sind,
2. Klassen, die keine Android-Komponenten sind, aber den Anwendungskontext benötigen,
3. Android-Komponenten.

Die Tests werden mit Hilfe des JUnit-Frameworks (`www.junit.org`) erstellt und durchgeführt.

Eigenes Testprojekt

Wir empfehlen, alle Tests in ein separates Projekt auszulagern. Auf diese Weise wird die resultierende .apk-Datei der Hauptanwendung nicht unnötig durch Testcode vergrößert. Weitere Gründe dafür werden wir in den folgenden Abschnitten nennen. Le-

gen wir uns also, parallel zum Projekt `staumelder`, in Eclipse ein weiteres Android-Projekt `staumelder.tests` an. Als Package sollte `de.androidbuch.staumelder.tests` gewählt werden. Die Start-Activity ist nicht von Belang und kann beliebig benannt werden. Wir müssen nun in Eclipse noch die Projektabhängigkeit zum Hauptprojekt `staumelder` eintragen, damit unsere zu testenden Klassen im Classpath liegen.

19.2 Tests von Nicht-Android-Komponenten

Starten wir mit einem Test für eine Klasse, die komplett ohne Android-SDK funktionsfähig ist: der Klasse `RoutenDatei` aus Kapitel 12. Für diese Klasse wollen wir beispielhaft einen Test für die Methode `neuePositionHinzufuegen` schreiben.

Dazu erzeugen wir uns eine Unterklasse von `org.junit.TestCase` und implementieren den Testcode wie in Listing 19.1 beschrieben.

Listing 19.1 *Unit-Test für RoutenDatei*

```
package de.androidbuch.staumelder.routenspeicher;

import de.androidbuch.staumelder.commons.GpsData;
import junit.framework.TestCase;

public class RoutenDateiTest extends TestCase {
  private static final long routenId = 4711L;
  private RoutenDatei datei;

  @Override
  protected void setUp() throws Exception {
    super.setUp();

    datei = new RoutenDatei(routenId);
  }

  public void testPositionHinzufuegen_ok()
      throws Exception {
    float laenge = 435643.5f;
    float hoehe = 45345439.2f;
    float breite = 100000.89f;
    GpsData gpsPos =
      new GpsData(
          System.currentTimeMillis(),
        laenge, breite, hoehe);
    assertEquals(0, datei.getAnzahlPositionen());
    assertTrue(datei.neuePositionHinzufuegen(gpsPos));
    assertEquals(1, datei.getAnzahlPositionen());
  }
}
```

Der Test enthält keinerlei Android-Spezifika, wird also jetzt nicht weiter kommentiert. Nun wollen wir ihn mit Hilfe des JUnit-Plug-in von Eclipse ausführen (`Run as... -> Junit Test...`). Leider erhalten wir einen Fehler der Laufzeitumgebung. Was ist geschehen?

Falsche Bibliotheken

Das Problem ist, dass Eclipse für Android-Projekte automatisch die Android-Laufzeitumgebung in den Classpath legt. Die Bibliotheken der klassischen JVM sind nicht enthalten. Die Android-Umgebung sieht nur eine Rumpf-Implementierung für JUnit vor, die wir hier nicht verwenden können. Daher schlägt die Ausführung des Tests frühzeitig fehl.

Laufzeit-Konfiguration anpassen

Die Lösung ist relativ einfach. Wir müssen in der Laufzeit-Konfiguration des Tests die Java-Systembibliotheken und die vollständige JUnit-Bibliothek anbinden. Dazu öffnen wir die Konfiguration über `Run... -> Open Run Dialog...` durch Klick auf `RoutenDateiTest`. Dann wählen wir den Reiter »Classpath«, klicken auf »Bootstrap Entries« und fügen über die Schaltfläche `Advanced... -> Add Library...` die fehlenden Bibliotheken `JUnit` und `JRE System Library` hinzu (Abbildung 19-1).

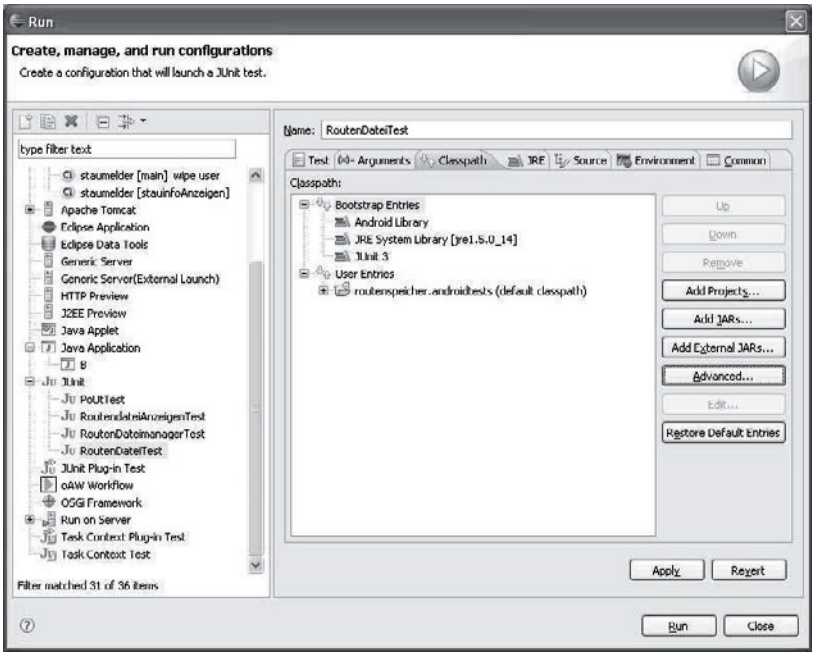


Abb. 19-1
Laufzeit-Classpath-Einstellung

Danach wiederholen wir den Testlauf und erhalten ein erfolgreiches Testergebnis. Diese Konfiguration muss für alle Testklassen wiederholt werden, die ohne das Android-SDK auskommen. Falls es sich dabei um viele Klassen handelt, kann man diese auch in ein eigenes Nicht-Android-Testprojekt auslagern.

19.3 Tests von Android-Komponenten

Wenden wir uns nun den Tests Android-spezifischer Klassen zu. Wir hatten eingangs gesagt, dass alle Android-Komponenten, von der Activity bis zum Service, potenzielle Testkandidaten seien.

Laufzeitumgebung gesucht

Android-Komponenten sind meist nur im Umfeld der gesamten Laufzeitumgebung sinnvoll nutzbar. Über den Anwendungskontext werden Infrastruktur-Ressourcen wie Datenbanken, Einstellungen und Dateien verwaltet. Android-Komponenten führen in den seltensten Fällen komplexe Geschäftstransaktionen aus, die isoliert getestet werden können.

Test auf dem Gerät

Darüber hinaus sollten die Testfälle nicht nur auf dem Emulator, sondern auch auf den Zielgeräten erfolgreich durchlaufen, bevor eine Anwendung ausgeliefert wird.

Integrationstests

Wir stehen also vor dem Problem, dass wir automatisierte, wiederholbare Tests so formulieren müssen, dass sie unter »Realbedingungen« der Android-Umgebung die Anforderungen der zu testenden Komponente isoliert verproben können. Es handelt sich also mehr um Integrations- und Funktionstests als um »reine« Unit-Tests.

19.3.1 Instrumentierung

Fernsteuerung

Das Android-SDK stellt im Package `android.test` eine Reihe von Klassen bereit, mit deren Hilfe Testfälle implementiert werden können. Diese laufen als *Instrumentierung* der Zielanwendung parallel zu dieser in einem eigenen Anwendungskontext ab. Die Instrumentierung einer Klasse kann mit einer »Fernsteuerung« für diese Referenzklasse betrachtet werden. Implementiert wird diese Funktionalität in der Klasse `android.app.Instrumentation`. Diese kann einerseits zur Beobachtung der Prozesse der Referenzklasse, aber andererseits auch zu deren Fernsteuerung (z.B. über Methoden wie `callActivityOnStart`, `callActivityOnResume` etc.) genutzt werden.

Zwei Anwendungen installieren

Da wir ja die Tests von der eigentlichen Anwendung trennen wollen, bedeutet dies, dass zur Testdurchführung immer zwei Anwendungen auf dem Zielgerät installiert sein müssen: die Testanwendung und die zu testende Anwendung. Schauen wir uns nun einmal die Instrumentierungs-Anwendung `staumelder.tests` aus der Vogelperspektive an.

InstrumentationTest und InstrumentationTestRunner

Prinzipiell sind alle auf Instrumentierung basierenden Testfälle von `android.test.InstrumentationTestCase` abgeleitet. Die Konventionen des JUnit-Frameworks gelten auch hier ohne Einschränkung. Daher existiert auch ein `InstrumentationTestRunner`, der alle korrekt definierten Testfälle unabhängig voneinander ausführt.

Keine integrierte Oberfläche

Leider gibt es derzeit noch keine komfortable Oberfläche, wie wir sie vom klassischen JUnit kennen. Die Testergebnisse müssen von der Konsole abgelesen werden. Der `InstrumentationTestRunner` stellt die Startklasse der Instrumentierungs-Anwendung dar. Wenn Ziel- und Testanwendung auf dem Zielsystem installiert sind, lässt sich die Testsuite über die `adb`-Konsole wie folgt starten:

```
$ adb shell am instrument
    -w TestPackageName/android.test.InstrumentationTestRunner
```

Die vollständige Syntax dieses Aufrufs ist im JavaDoc von `InstrumentationTestRunner` beschrieben. Das Ergebnis des Testlaufs erhalten wir unmittelbar auf der Konsole angezeigt.

```
de....routenspeicher.RoutenDateiTest:.
de.....routenspeicher.RoutendateiAnzeigenTest:
Failure in testInitWithValidParam:
junit.framework.AssertionFailedError:
  expected:<Inhalt der Beispieldatei> but was:<>
  at ...RoutendateiAnzeigenTest.
        testInitWithValidParam(RoutendateiAnzeigenTest.java:36)
  at ...InstrumentationTestCase.runMethod()
  at ...InstrumentationTestCase.runTest()
  at ...AndroidTestRunner.runTest()
  at ...InstrumentationTestRunner.onStart()
  at ...Instrumentation$InstrumentationThread.run()
.
Test results for InstrumentationTestRunner=..F.
Time: 0.584

FAILURES!!!
Tests run: 3,  Failures: 1,  Errors: 0
```

Damit all dies funktioniert, muss im Manifest der Testanwendung beschrieben werden, auf welches Ziel-Package sich die Instrumentierung beziehen soll (Listing 19.2).

Listing 19.2 *Manifest der Testinstrumentierung*

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
  xmlns:android=
  "http://schemas.android.com/apk/res/android"
  android:versionCode="1"
  android:versionName="1.0.0"
  package="de.androidbuch.staumelder.tests"
>
```

```
  <application>
    <uses-library
      android:name="android.test.runner" />
  </application>

  <instrumentation
    android:name=
      "android.test.InstrumentationTestRunner"
    android:targetPackage=
      "de.androidbuch.staumelder"
    android:label=
      "Staumelder Oberflächentests"
  >
  </instrumentation>
</manifest>
```

19.3.2 Wahl der Testklasse

Je nach Komplexität des Testfalls haben wir die Wahl zwischen verschiedenen Oberklassen für die Testimplementierung. Die erste Option haben wir bereits im letzten Beispiel kennengelernt.

junit.framework.TestCase Ein von dieser Klasse abgeleiteter Testfall benötigt keinerlei Anwendungskontext von Android. Er ist, als einziger der hier vorgestellten Testfälle, mit Hilfe des normalen JUnit-`TestRunner` lauffähig. Dadurch lassen sich diese Tests bequem in der Eclipse-Umgebung ausführen.

android.test.AndroidTestCase Wird ein `android.content.Context` von der zu testenden Klasse benötigt, so sollten deren Testfälle von einer Unterklasse von `AndroidTestCase` realisiert werden. Diese Tests sind immer noch recht leichtgewichtig, aber schon mächtiger als die zuletzt erwähnten. Man kann hier bereits auf Ressourcen der Anwendung, wie z.B. Layouts, Datenbanken etc., zugreifen.

android.test.InstrumentationTestCase Hauptaufgabe dieser Testklasse ist es, ihren Unterklassen die Instrumentierung der Zielklasse bereitzustellen. Nach Aufruf von `getInstrumentation` erhält der Tester die Fernbedienung für die Zielklasse. Des Weiteren bietet diese Klasse über die Methoden `sendKeys(...,...)` die Möglichkeit, Tastatureingaben für Komponenten der gleichen Anwendung durchzuführen. Somit sind auch umfangreichere Oberflächentests realisierbar.

Keine Trennung mehr

Leider lässt sich dann aber unser Konzept der getrennten Test- und Zielanwendungen nicht mehr durchführen, da es Anwendungen aus Sicherheitsgründen nicht gestattet ist, Tastatureingaben für andere Anwendungen auszuführen. Vielfach lässt sich die Nutzung expliziter Tastatureingaben in Tests jedoch vermeiden. Wir werden später ein Beispiel für eine solche Alternative geben.

Basisklasse für Tests

Der `InstrumentationTestCase` dient als Oberklasse für viele Testklassen des Android-SDK, die bereits auf bestimmte Komponententypen spezialisiert sind. Wir nennen hier nur die Implementierungen für Activities und überlassen die Beschreibung der restlichen Testfälle der Online-Dokumentation.

android.test.ActivityUnitTestCase<Activity> Hierbei handelt es sich um einen `InstrumentationTestCase`, der seine zu testende Komponente vom Zielsystem isoliert startet. Auf diese Komponente kann innerhalb des Tests zugegriffen werden. Ihr Zustand kann verändert und mit den Soll-Vorgaben abgeglichen werden. Dabei bedient man sich der Zielklasse direkt oder deren Instrumentierung. Der `ActivityUnitTestCase` eignet sich also gut dazu, interne Prozesslogik (z.B. Berechnungen auf der Oberfläche, Reagieren auf leere/volle Ergebnislisten etc.) sowie die Außenschnittstelle einer Activity zu testen. Es ist aber hier nicht möglich, mit anderen Komponenten der Anwendung zu kommunizieren.

android.test.ActivityInstrumentationTestCase<Activity> Für diesen Instrumentierungstestfall sorgt die Testumgebung bereits dafür, dass die Activity für jeden Testdurchlauf gestartet und danach korrekt beendet wird.

19.3.3 Beispiel: Test einer Activity

Das Szenario

Im folgenden Abschnitt wollen wir am Beispiel eines Testfalls für die Activity `RoutenDateiAnzeigen` die Implementierungsdetails vertiefen. Erinnern wir uns: Diese Activity dient dazu, den Inhalt einer GPX-Datei auf der Oberfläche darzustellen. Der dazu passende Schlüsselwert für die Routenidentifikation wird über den aufrufenden Intent übergeben.

Die Testfälle

Testen wollen wir zunächst nur zwei Eigenschaften der Activity: den korrekten Aufruf anhand einer Routen-Id und die korrekte Reaktion auf die Auswahl der Menüoption »Einstellungen bearbeiten«.

Wir fügen also unserem im letzten Abschnitt erstellten Projekt `staumelder.tests` die Testklasse `RoutenDateiAnzeigenTests` hinzu. Listing 19.3 stellt den Code dar, den wir gleich genauer analysieren wollen.

Listing 19.3 *Test einer Activity*

```
package de.androidbuch.staumelder.routenspeicher;
...
import android.test.ActivityUnitTestCase;
...
public class RoutendateiAnzeigenTest
  extends // (1)
    ActivityUnitTestCase<RoutendateiAnzeigen> {

  private static Long testDateiId = new Long(1234L);

  public RoutendateiAnzeigenTest() {
    super(RoutendateiAnzeigen.class); // (2)
  }

  public void testInitWithoutParam() throws Exception {
    Intent emptyIntent = new Intent();
    Bundle emptyBundle = new Bundle();
    RoutendateiAnzeigen activity =
        startActivity( // (3)
                emptyIntent,
        null, null);
    assertNotNull(activity);
    TextView uiDateiInhalt =
      (TextView) activity // (4)
        .findViewById(R.id.dateiInhalt);
    assertNotNull(uiDateiInhalt);
    assertEquals("", uiDateiInhalt.getText());
  }
  ...
}
```

(1) Definition der Testklasse: Wir wählen hier einen `ActivityUnitTestCase`, da wir lediglich den Aufruf der Activity testen wollen.

(2) Übergabe der Zielklasse: Die zu instrumentierende Klasse `RoutendateiAnzeigen` muss im Konstruktor mitgegeben werden. Ab jetzt kann auf eine Instanz der Activity zugegriffen werden.

(3) Start der Activity: Da wir einen `ActivityUnitTestCase` nutzen, müssen wir uns um den Start der Ziel-Activity selbst kümmern. Das ist ja auch in unserem Interesse, da wir genau dort das korrekte Verhalten von `RoutendateiAnzeigen` verproben wollen. Wir geben daher beim ersten Aufruf der Activity keine Parameter mit und erwarten, dass die Routenanzeige leer bleibt.

(4) Zugriff auf Activity: Die im vorherigen Schritt gestartete Activity wird nun untersucht. Wir können gezielt den Inhalt der `TextView` untersuchen und erwarten, dass diese keinen Text enthält.

Testausführung

Starten wir nun den Test. Dazu benötigen wir die `adb`-Konsole. Zunächst werden Ziel- und Testanwendung installiert.

```
adb install staumelder.apk
adb uninstall staumelder.tests.apk
adb install staumelder.tests.apk
```

Nach erfolgreicher Installation können wir unsere Testfälle starten.

```
$ adb shell am instrument -w
  de.androidbuch.staumelder.tests/
  android.test.InstrumentationTestRunner
```

Wir wir sehen können, startet der `InstrumentationTestRunner` nicht nur `InstrumentationTestCases`, sondern alle Unit-Tests, die den Namenskonventionen von JUnit genügen.

> Wir dürfen nicht vergessen, nach Änderungen der Hauptanwendung diese auch wieder neu zu starten (oder zumindest zu deployen). Sonst operiert die Testanwendung auf einer veralteten Version.

Instrumentierung

Zum Schluss dieses Kapitels wollen wir noch einen Blick auf die »Fernsteuerung« durch Instrumentierung werfen. Wir erweitern dazu unseren Testfall um eine Methode, die absichern soll, dass nach Klick auf die Menüoption »Einstellungen bearbeiten« die dazu passende Activity aufgerufen würde. Den tatsächlichen Aufruf können wir ja nicht nachstellen, da es sich immer noch um einen isolierten Testfall handelt. Aber darauf kommt es uns ja auch nicht so sehr an. Wichtig ist, dass der korrekte Intent für den Aufruf generiert wird. Der Rest ist Infrastruktur und sollte ohnehin nicht Bestandteil eines Unit-Tests sein.

Werfen wir zunächst wieder einen Blick auf die Implementierung des beschriebenen Testfalls.

Listing 19.4
Test einer Menüoption

```
/**
 * prüfe, ob klick auf Menüoption "Einstellungen"
 * die korrekte Activity startet.
 */
public void testMenueEinstellungen() throws Exception {
  Intent emptyIntent = new Intent();
  Bundle emptyBundle = new Bundle();
  RoutendateiAnzeigen activity = startActivity(emptyIntent,null, null);
```

```
  assertTrue(
      "Optionsmenue nicht nutzbar",
      getInstrumentation().invokeMenuActionSync( // (1)
      activity,
      R.id.opt_einstellungenBearbeiten,
      0));

  Intent targetIntent =
    getStartedActivityIntent(); // (2)
  assertNotNull(
      "Es wurde kein Intent ausgeloest!",
      targetIntent);

  assertEquals(
      "Intent EinstellungenBearbeiten nicht gestartet.",
      EinstellungenBearbeiten.class.getName(),
      targetIntent.getComponent().getClassName()
      );
}
```

Ferngesteuerte Menüs

Der Voodoo findet in der Zeile (1) statt. Hier holen wir uns die Instrumentierung der Zielklasse und steuern den Menü-Aufruf fern. Das Ergebnis holen wir uns in (2) ab. `getStartedActivityIntent` liefert den zuletzt gestarteten Intent zurück. Das müsste, eine korrekte Implementierung von `RoutendateiAnzeigen` vorausgesetzt, der Intent zum Aufruf der Activity `EinstellungenBearbeiten` sein. Diese Prüfung schließt den Testfall ab.

19.4 Ausblick

Wie bereits zu Beginn des Kapitels erwähnt, wollen wir Ihnen hier nur das Basiskonzept von Tests für die Android-Plattform skizzieren. Für detailliertere Ausführungen fehlt uns leider der Platz. Die mit dem SDK ausgelieferten Beispielanwendungen unterhalb des `ApiDemos`-Verzeichnisses bieten einen guten Einstiegspunkt für weitere Studien. Bitte beachten Sie, dass die dort vorliegenden Testfälle erst in ein eigenes Projekt ausgelagert werden müssen, damit sie sauber durchlaufen.

Das Thema *Tests* steht eher noch am Anfang und wird auch in den Entwicklerforen und -blogs rege diskutiert. Wir sind sicher, dass sich hier in Zukunft noch einiges tun wird.

20 Optimierung und Performance

Hinweis: Die 2. Auflage dieses Buchs erscheint unter dem Titel
»Android 2. Grundlagen und Programmierung«
Ende Mai 2010 im dpunkt.verlag, ISBN 978-3-89864-677-2

In diesem Kapitel geben wir einige Hinweise zur Optimierung von Android-Anwendungen. In vielen Fällen müssen wir gewohnte Vorgehensweisen durch, manchmal recht »unschöne«, Alternativen ablösen, um die Performanz der Anwendung nicht in Mitleidenschaft zu ziehen.

20.1 Erste Optimierungsregeln

Ein Telefon ist kein Server!

Es sollte jedem Entwickler sofort klar sein, dass er auf einem mobilen Computer nicht über die gleichen Hardware-Ressourcen verfügt wie auf einem Desktop- oder Serversystem. Er muss also dafür Sorge tragen, dass er Speicherverbrauch und Prozessorlast so gering wie möglich hält. Daher definieren wir die »Oberste Regel der Android-Entwicklung« wie folgt:

> *Keep it simple!* Schreiben Sie nur Code, wenn Sie diesen unbedingt benötigen.

Wir vermeiden wenn möglich

- die Verwendung von Interfaces,
- elegante, aber nicht unbedingt erforderliche Methoden (z.B. Wrapper, Delegator etc.),
- den Aufruf von Getter/Setter-Methoden zum Zugriff auf Attribute der *eigenen* Klasse,

es sei denn, es gibt einen guten Grund dafür. Ein solcher könnte z.B. sein, wenn eine Programmierschnittstelle für andere Anwendungen bereitgestellt werden soll. In diesem Fall sollten z.B. Interfaces genutzt werden.

Unser Ziel muss es sein, eine kompakte und schlanke Anwendung zu erstellen.

20.2 Datenobjekte

Bei der Erstellung von Client-Server-Anwendungen wird beispielsweise die Datenbankzugriffsschicht häufig durch objektrelationale Mapping-Frameworks wie Hibernate, Toplink etc. gekapselt. Wir müssen uns dann nur mit Datenobjekten befassen, die auf magische Weise in Datenbanktabellen gespeichert werden.

Datenobjekt oder nicht?

Objekterzeugung kostet Zeit. Daher sollten wir versuchen, diese auf ein notwendiges Maß zu reduzieren. Wir müssen uns von Fall zu Fall fragen, ob *wirklich* überall und immer eine Klasse zur Datenrepräsentation benötigt wird. In vielen Fällen reicht die reine Datenübertragung z.B. in Form von Arrays vollkommen aus. Die Datenhaltung in Objekten ist nur dann wirklich sinnvoll, wenn

- konkretes Verhalten in Form von Methoden für die Datenklasse benötigt wird,
- fast alle Datenattribute angezeigt oder anderweitig verwendet werden sollen.

Eine zuweilen ausgesprochene Empfehlung ist, zugunsten der Performanz auf Getter/Setter-Methoden zu verzichten und stattdessen alle Attribute `public` zu definieren. Wir konnten diese Aussage in Lasttests jedoch nicht bestätigen und empfehlen weiterhin die Verwendung von Zugriffsmethoden auf private Attribute.

20.3 Cursor oder Liste?

Android bietet für den Zugriff auf Datenmengen das Konzept eines *Cursors* an, das wir in Abschnitt 11.5.3 ab Seite 173 vorgestellt haben. Mit Hilfe eines Cursors kann man auf einzelne Ergebnisdatensätze bzw. konkrete Spaltenwerte des Ergebnisses einer Datenbankanfrage zugreifen, ohne diese erst in Objekte umzuwandeln. Ein Cursor sollte immer dann verwendet werden, wenn es darum geht, mehrelementige Ergebnismengen einer Anfrage darzustellen oder auszuwerten.

Der in vielen Persistenzframeworks sinnvollerweise gewählte Weg, Listen für den Datentransfer einzusetzen, ist für Android-Anwendungen nicht empfehlenswert. Der durch Objekterzeugung höhere Speicheraufwand und die damit verbundenen Laufzeiteinbußen rechtfertigen den gewonnenen Komfort in den meisten Fällen nicht.

20.4 Time is Akku!

Wir bewegen uns auf Geräten mit eingeschränkten Hardware-Ressourcen. Dazu gehört auch die Abhängigkeit von einer sich schnell erschöpfenden Stromquelle.

Komplexe Operationen, z.B. Zugriffe auf Dateien oder Datenbanken, sind wahre Stromfresser. Eine SQL-Anfrage erfordert viele Speicherzugriffe und benötigt Platz im Hauptspeicher zur Ergebnisdarstellung. Das gilt auch für die Nutzung von Location-Based-Services oder dauerhaften Internetverbindungen.

Jede dieser Operationen muss also sorgfältig geplant werden.

21 Das Android-SDK

Hinweis: Die 2. Auflage dieses Buchs erscheint unter dem Titel
»Android 2. Grundlagen und Programmierung«
Ende Mai 2010 im dpunkt.verlag, ISBN 978-3-89864-677-2

Das Android-SDK deckt nicht den kompletten Sprachumfang des Java-SDKs ab. Einige Pakete fehlen. Dies kann Konsequenzen für die eigene Anwendungsentwicklung haben. Fügt man der eigenen Anwendung eine fremde Bibilothek hinzu, ist auf den ersten Blick nicht ersichtlich, welche Klassen diese referenziert. Wurde diese Bibliothek für das Java-SDK entwickelt, verwendet sie möglicherweise Klassen, die Bestandteil des Java-SDK sind, aber im Android-SDK nicht vorkommen. Dann ist eine Android-Anwendung, die eine solche Bibliothek verwendet, nicht lauffähig. Wie wir mit einem solchen Problem umgehen und wie eine Lösung aussehen kann, damit beschäftigen wir uns in diesem Kapitel.

21.1 Unterschiede zum Java-SDK

Die folgende Aufzählung listet die meisten Unterschiede zum Java-SDK der Version 1.5 auf. Nicht alle der aufgezählten Pakete fehlen vollständig.

- java.awt
- java.beans
- java.lang (teilweise)
- java.rmi
- javax.accessibility
- javax.activity
- javax.imageio
- javax.management
- javax.naming
- javax.print
- javax.rmi
- javax.sound
- javax.sql (teilweise)
- javax.swing
- javax.transaction
- javax.xml (teilweise)

- org.omg
- org.w3c (teilweise)

Vieles davon ist erklärbar. AWT und SWING wird nicht unterstützt. Daher fehlen im Android-SDK die Pakete `java.awt` und `javax.swing`. Android bringt eigene Oberflächenelemente mit, die mittels Layouts in Activities dargestellt werden. Die Oberflächengestaltung von Android ist eine Alternative zu AWT und Swing, und vermutlich braucht sie wesentlich weniger Ressourcen.

Das Fehlen des Pakets `javax.print` ist auch recht verständlich. Man wird wohl eher über das Netzwerk drucken und keinen Drucker direkt an ein Android-Gerät anschließen.

Dass das Paket `java.rmi` fehlt, ist auch verständlich, kann aber schon eher Probleme verursachen. Android arbeitet mit Intents. Direkte Aufrufe von Methoden erfolgen über Binder. Entfernte Aufrufe (Remote-Aufrufe) erfolgen über IPC. Android verzichtet somit auf RMI und nutzt alternative Möglichkeiten.

Problematisch wird es beim Paket `java.beans`. Seine Abwesenheit wird vermutlich zu den meisten Problemen führen. Beispielsweise nutzen die Bibliotheken des Apache-Commons-Projekts *BeanUtils* das Paket `java.beans` recht häufig. Ebenso gibt es zahlreiche weitere Bibliotheken, die wiederum die *BeanUtils* verwenden.

Wer könnte vor diesem Hintergrund zum Beispiel sagen, ob Axis, ein Webservice-Framework, auf Android läuft, wenn er sich damit nicht recht gut auskennt?

Wir sollten an dieser Stelle auch nicht vergessen, dass das Android-SDK nicht den Versionszyklen des Java-SDK unterliegt. Wir können derzeit von einem abgespeckten SDK der Version 1.5 ausgehen. Wir sollten also zunächst Java-Anwendungen, die für ein höheres JDK programmiert wurden, auf der Version 1.5 lauffähig machen.

Das Android-SDK ist speziell für die geringen Ressourcen in Bezug auf Speicher und Prozessorgeschwindigkeit optimiert. Der Einsatz von Java-Frameworks und zusätzlichen Bibliotheken auf einem Android-Gerät kann daher auch zu Problemen führen. XML-Parser verbrauchen bisweilen erstaunlich viel Speicher. Dies sollte man im Hinterkopf behalten, wenn man die Bibliotheken auf Android portiert. Lasttests und ein kritisches Auge auf den Speicherverbrauch sind unerlässlich.

21.2 Wege aus der Krise

Wir skizzieren nun eine Vorgehensweise, wie man Bibliotheken auf ein Android-Gerät portiert.

Grundsätzlich haben wir zwei Ausgangspunkte:

- Der fremde Code liegt als Quellcode vor.
- Es liegt nur ein Jar mit den kompilierten Klassen vor.

Im Falle von Open-Source-Projekten kann man sich den Quellcode von der Projektseite herunterladen und in das `src`-Verzeichnis des Android-Projekts kopieren. Der Compiler nimmt uns hier viel Arbeit ab, und beim Erstellen der Anwendung werden nur die benötigten Klassen in die .apk-Datei paketiert.

Ungünstiger ist es, wenn nur kompilierte Klassen vorliegen. Es empfiehlt sich in diesem Fall, das Jar zu entpacken und alle benötigten Klassen, den Abhängigkeiten folgend, dem Android-Projekt einzeln hinzuzufügen. Das kann sehr aufwendig sein, bietet aber den großen Vorteil, dass wirklich nur die benötigten Klassen im Android-Projekt sind. Beim Starten der Android-Anwendung lädt der Classloader nur die Klassen, die die Anwendung verwendet. Der Speicherverbrauch bleibt geringer, und die Gefahr, dass es zur Laufzeit zu Problemen kommt, ist geringer.

Frei von Problemen ist man dann natürlich oft nicht. Werden von den unbekannten Klassen per Reflection weitere Klassen erst zur Laufzeit geladen, fehlen diese, und die Anwendung reagiert mit einer `ClassNotFoundException`.

Literaturverzeichnis

Hinweis: Die 2. Auflage dieses Buchs erscheint unter dem Titel
»Android 2. Grundlagen und Programmierung«
Ende Mai 2010 im dpunkt.verlag, ISBN 978-3-89864-677-2

[1] Kent Beck. *Test Driven Development. By Example*. Addison-Wesley Longman, Amsterdam, 2002.

[2] DalvikVM.com. *Dalvik Virtual Machine*. Webseite, 2008. http://www.dalvikvm.com.

[3] diverse. *IMSI-Catcher*. Webseite, 16.2.2009. http://de.wikipedia.org/wiki/IMSI-Catcher.

[4] diverse. *sqlite3: A command-line access program for SQLite databases*. Webseite, 16.3.2009. http://www.sqlite.org/sqlite.html.

[5] diverse. *db4objects*. Webseite, 17.4.2009. http://www.db4o.com/.

[6] diverse. *Registry of intents protocols*. Webseite, 17.4.2009. http://www.openintents.org/en/intentstable.

[7] diverse. *SQLite: Available Documentation*. Webseite, 28.3.2009. http://www.sqlite.org/docs.html.

[8] diverse. *SQLite: Distinctive Features of SQLite*. Webseite, 3.3.2008. http://www.sqlite.org/different.html.

[9] Apache Software Foundation. *Jakarta Commons HttpClient*. Webseite, 8.2.2008. http://hc.apache.org/httpclient-3.x/.

[10] Google Inc. *The AndroidManifest.xml File*. Webseite, 13.4.2009. http://developer.android.com/guide/topics/manifest/manifest-intro.html.

[11] Google Inc. *Common Layout Objects*. Webseite, 13.4.2009. http://developer.android.com/guide/topics/ui/layout-objects.html.

[12] Google Inc. <*data*>. Webseite, 13.4.2009. http://developer.android.com/guide/topics/manifest/data-element.html.

[13] Google Inc. *Hello Views*. Webseite, 13.4.2009. http://developer.android.com/guide/tutorials/views/index.html.

[14] Google Inc. *Manifest.permission*. Webseite, 13.4.2009. http://developer.android.com/reference/android/Manifest.permission.html.

[15] Google Inc. *Reference of Available Intents*. Webseite, 13.4.2009. http://developer.android.com/reference/android/content/Intent.html.

[16] Google Inc. *Security and Permissions in Android*. Webseite, 13.4.2009. http://developer.android.com/guide/topics/security/security.html.

[17] Google Inc. *Signing and Publishing Your Applications*. Webseite, 13.4.2009. http://developer.android.com/guide/publishing/app-signing.html.

[18] Google Inc. *Installing the Android SDK*. Webseite, 17.4.2009. http://developer.android.com/sdk/1.1_r1/installing.html.

[19] Google Inc. *System styles and themes*. Webseite, 2009. http://developer.android.com/reference/android/R.style.html.

[20] Johannes Link. *Softwaretests mit JUnit*. 2. Auflage, dpunkt.verlag, 2005.

[21] Peter Rechenberg. *Technisches Schreiben*. 3. Auflage, Carl Hanser Verlag, München, 2006.

Index

Hinweis: Die 2. Auflage dieses Buchs erscheint unter dem Titel
»Android 2. Grundlagen und Programmierung«
Ende Mai 2010 im dpunkt.verlag, ISBN 978-3-89864-677-2