Uwe Post

Eine Spiele-App von A bis Z

# Android-Apps entwickeln

## Ideal für Programmiereinsteiger geeignet

- Schritt für Schritt zu eigenen Apps und Spielen
- Inkl. Sprachgrundlagen von Java
- Animationen, Sounds, Zeichnen, Kamera, Bewegungssensoren, Highscores u. v. m.

DVD ROM Komplettes Startpaket mit der benötigten Software und allen Beispielen

Galileo Computing

Uwe Post

# Android-Apps entwickeln

Galileo Press

# Liebe Leserin, lieber Leser,

Sie steigen in die App-Entwicklung ein, und ich kann Ihnen versprechen: Sie werden nicht geschont! Es gilt, die erste Spiele-App zu programmieren und das mit allem Drum und Dran: Sound und Animation, Beschleunigungs- und Lagesensoren, Kamera, Magnetfeldsensoren… ein echtes Spiel eben. Das Ganze programmieren Sie in Java. Sie werden nach der Lektüre kein Java-Entwickler sein, aber Sie werden alle Anforderungen, die das Spiel an Sie stellt, programmierend in Java bewältigen können.

Ich kann Ihnen außerdem versprechen: Sie haben zwar ein Fachbuch erworben, aber Sie werden blendend unterhalten bei Ihrer Lektüre. Das Spiel ist anspruchsvoll, die Lernkurve steil und der Spaß bleibt bis zur letzten Seite!

Dieses Buch wurde mit großer Sorgfalt geschrieben, geprüft und produziert. Sollte dennoch einmal etwas nicht so funktionieren, wie Sie es erwarten, freue ich mich, wenn Sie sich mit mir in Verbindung setzen. Ihre Kritik und Ihre konstruktiven Anregungen sind uns jederzeit herzlich willkommen!

Doch jetzt will ich Sie nicht weiter aufhalten. Auf in den Kampf »Tomaten gegen Mücken«!

Viel Freude beim Lesen und Programmieren wünscht

**Ihre Judith Stevens-Lemoine**
Lektorat Galileo Computing

judith.stevens@galileo-press.de
www.galileocomputing.de
Galileo Press · Rheinwerkallee 4 · 53227 Bonn

# Auf einen Blick


| | | |
|---|---|---|
| 1 | **Einleitung** | 13 |
| 2 | **Ist Java nicht auch eine Insel?** | 37 |
| 3 | **Vorbereitungen** | 65 |
| 4 | **Die erste App** | 85 |
| 5 | **Ein Spiel entwickeln** | 127 |
| 6 | **Sound und Animation** | 173 |
| 7 | **Internet-Zugriff** | 205 |
| 8 | **Kamera und Augmented Reality** | 253 |
| 9 | **Sensoren und der Rest der Welt** | 271 |
| 10 | **Tipps und Tricks** | 317 |
| 11 | **Apps veröffentlichen** | 347 |

Der Name Galileo Press geht auf den italienischen Mathematiker und Philosophen Galileo Galilei (1564–1642) zurück. Er gilt als Gründungsfigur der neuzeitlichen Wissenschaft und wurde berühmt als Verfechter des modernen, heliozentrischen Weltbilds. Legendär ist sein Ausspruch *Eppur si muove* (Und sie bewegt sich doch). Das Emblem von Galileo Press ist der Jupiter, umkreist von den vier Galileischen Monden. Galilei entdeckte die nach ihm benannten Monde 1610.

**Lektorat** Judith Stevens-Lemoine
**Korrektorat** Alexandra Müller, Olfen
**Einbandgestaltung** Barbara Thoben, Köln
**Titelbild** Johannes Kretzschmar, Jena
**Typografie und Layout** Vera Brauner
**Herstellung** Norbert Englert
**Satz** Typographie & Computer, Krefeld
**Druck und Bindung** Bercker Graphischer Betrieb, Kevelaer

Dieses Buch wurde gesetzt aus der Linotype Syntax Serif (9,25/13,25 pt) in FrameMaker.

**Gerne stehen wir Ihnen mit Rat und Tat zur Seite:**
judith.stevens@galileo-press.de bei Fragen und Anmerkungen zum Inhalt des Buches
service@galileo-press.de für versandkostenfreie Bestellungen und Reklamationen
britta.behrens@galileo-press.de für Rezensions- und Schulungsexemplare

**Bibliografische Information der Deutschen Nationalbibliothek**
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.d-nb.de* abrufbar.

**ISBN 978-3-8362-1813-9**


1. Auflage 2012

# Inhalt


Vorwort ..... 11

**1 Einleitung** ..... 13

1.1 Für wen ist dieses Buch? ..... 13
Magie? ..... 14
Große Zahlen ..... 14
Technologie für alle ..... 15
Die Grenzen der Physik ..... 16
1.2 Unendliche Möglichkeiten ..... 17
Baukasten ..... 18
Spiel ohne Grenzen ..... 19
Alles geht ..... 22
1.3 Was ist so toll an Android? ..... 22
MapDroyd ..... 23
Google Sky Map ..... 24
Bump ..... 25
c:geo ..... 27
barcoo ..... 28
Öffi ..... 29
Wikitude ..... 30
Sprachsuche ..... 32
Cut the Rope ..... 33
Shaky Tower ..... 35

**2 Ist Java nicht auch eine Insel?** ..... 37

2.1 Warum Java? ..... 37
2.2 Grundlagen ..... 39
Objektorientierung: Klassen und Objekte ..... 40
Konstruktoren ..... 42
2.3 Pakete ..... 43
Packages deklarieren ..... 43
Klassen importieren ..... 44
2.4 Klassen implementieren ..... 45
Attribute ..... 45
Methoden ..... 48
Zugriffsbeschränkungen ..... 50

Eigene Konstruktoren ..... 53
Lokale Variablen ..... 54
2.5 Daten verwalten ..... 56
Listen ..... 56
Schleifen ..... 58
2.6 Vererbung ..... 59
Basisklassen ..... 59
Polymorphie ..... 62

**3 Vorbereitungen** ..... 65

3.1 Was brauche ich, um zu beginnen? ..... 65
3.2 JDK installieren ..... 67
3.3 Eclipse installieren ..... 69
3.4 Tour durch Eclipse ..... 71
3.5 Android Development Tool installieren ..... 74
3.6 Android SDK installieren ..... 76
3.7 SDK Tools installieren ..... 77
3.8 Ein virtuelles Gerät erzeugen ..... 78
3.9 Eclipse mit dem Handy verbinden ..... 81
3.10 Was tun, wenn mein Eclipse verrücktspielt? ..... 82
Unerklärliche Unterstreichungen ..... 82
Ein Handy namens Fragezeichen ..... 83
Eclipse hängt sich auf ..... 84
Eclipse findet Resource-Dateien nicht ..... 84

**4 Die erste App** ..... 85

4.1 Sag »Hallo«, Android! ..... 85
Ein neues Android-Projekt erstellen ..... 85
Die StartActivity ..... 87
Der erste Start ..... 92
4.2 Bestandteile einer Android-App ..... 94
Versionsnummern ..... 95
Activities anmelden ..... 96
Permissions ..... 97
Ressourcen ..... 99
Generierte Dateien ..... 101
4.3 Benutzeroberflächen bauen ..... 105
Layout bearbeiten ..... 105
String-Ressourcen ..... 109

Layout-Komponenten ..... 113
Weitere visuelle Komponenten ..... 116
4.4 Buttons mit Funktion ..... 117
Der OnClickListener ..... 117
4.5 Eine App installieren ..... 121
Start mit ADT ..... 121
Installieren per USB ..... 121
Installieren mit ADB ..... 122
Drahtlos installieren ..... 123

**5 Ein Spiel entwickeln** ..... 127

5.1 Wie viele Stechmücken kann man in einer Minute fangen? ..... 127
Der Plan ..... 127
Das Projekt erzeugen ..... 128
Layouts vorbereiten ..... 129
Die GameActivity ..... 130
5.2 Grafiken einbinden ..... 133
Die Mücke und der Rest der Welt ..... 134
Grafiken einbinden ..... 135
5.3 Die Game Engine ..... 137
Aufbau einer Game Engine ..... 137
Ein neues Spiel starten ..... 138
Eine Runde starten ..... 139
Den Bildschirm aktualisieren ..... 140
Die verbleibende Zeit herunterzählen ..... 146
Prüfen, ob das Spiel vorbei ist ..... 150
Prüfen, ob eine Runde vorbei ist ..... 152
Eine Mücke anzeigen ..... 152
Eine Mücke verschwinden lassen ..... 157
Das Treffen einer Mücke mit dem Finger verarbeiten ..... 160
»Game Over« ..... 161
Der Handler ..... 163
5.4 Der erste Mückenfang ..... 167
Retrospektive ..... 168

**6 Sound und Animation** ..... 173

6.1 Sounds hinzufügen ..... 174
Sounds erzeugen ..... 174
Sounds als Ressource ..... 176
6.2 Sounds abspielen ..... 177
Der MediaPlayer ..... 178

MediaPlayer initialisieren ..... 179
Zurückspulen und Abspielen ..... 179
6.3 Einfache Animationen ..... 181
Views einblenden ..... 182
Wackelnde Buttons ..... 184
Interpolation ..... 186
6.4 Fliegende Mücken ..... 191
Grundgedanken zur Animation von Views ..... 191
Geschwindigkeit festlegen ..... 191
Mücken bewegen ..... 193
Bilder programmatisch laden ..... 195
If-else-Abfragen ..... 197
Zweidimensionale Arrays ..... 198
Resource-IDs ermitteln ..... 200
Retrospektive ..... 201

**7 Internet-Zugriff** ..... 205

7.1 Highscores speichern ..... 205
Highscore anzeigen ..... 205
Activities mit Rückgabewert ..... 207
Werte permanent speichern ..... 207
Rekordhalter verewigen ..... 209
7.2 Bestenliste im Internet ..... 214
Ein App Engine-Projekt ..... 215
URL-Parameter entgegennehmen ..... 217
Daten im High Replication Datastore speichern ..... 218
Highscores aus dem Datastore auslesen ..... 220
Die Internet-Erlaubnis ..... 222
Der Android-HTTP-Client ..... 223
Background-Threads ..... 228
Die Oberfläche aktualisieren ..... 230
Highscores zum Server schicken ..... 232
HTML darstellen ..... 234
HTML mit Bildern ..... 237
7.3 Listen mit Adaptern ..... 240
ListViews ..... 240
ArrayAdapter ..... 244
Eigene Adapter ..... 247
Recyceln von Views ..... 251

**8 Kamera und Augmented Reality** ..... 253

8.1 Die Kamera verwenden ..... 253
Der CameraView ..... 254
CameraView ins Layout integrieren ..... 258
Die Camera-Permission ..... 260
8.2 Bilddaten verwenden ..... 261
Bilddaten anfordern ..... 261
Bilddaten auswerten ..... 263
Tomaten gegen Mücken ..... 265

**9 Sensoren und der Rest der Welt** ..... 271

9.1 Himmels- und sonstige Richtungen ..... 271
Der SensorManager ..... 272
Rufen Sie nicht an, wir rufen Sie an ..... 272
Die Kompassnadel und das Canvas-Element ..... 274
View und Activity verbinden ..... 278
9.2 Wo fliegen sie denn? ..... 278
Sphärische Koordinaten ..... 279
Die virtuelle Kamera ..... 281
Mücken vor der virtuellen Kamera ..... 282
Der Radarschirm ..... 286
9.3 Beschleunigung und Erschütterungen ..... 292
Ein Schrittzähler ..... 293
Mit dem SensorEventListener kommunizieren ..... 295
Schritt für Schritt ..... 297
9.4 Hintergrund-Services ..... 300
Eine Service-Klasse ..... 300
Service steuern ..... 303
Einfache Service-Kommunikation ..... 304
9.5 Arbeiten mit Geokoordinaten ..... 307
Der Weg ins Büro ..... 307
Koordinaten ermitteln ..... 309
Karten und Overlay ..... 311

**10 Tipps und Tricks** ..... 317

10.1 Fehlersuche ..... 317
Einen Stacktrace lesen ..... 318
Logging einbauen ..... 321
Schritt für Schritt debuggen ..... 323

10.2 Views mit Stil ... 325
Hintergrundgrafiken ... 325
Styles ... 326
Themes ... 327
Button-Zustände ... 329
9-Patches ... 330
10.3 Dialoge ... 332
Standard-Dialoge ... 332
Eigene Dialoge ... 337
Toasts ... 340
10.4 Layout-Gefummel ... 341
RelativeLayouts ... 341
Layout-Gewichte ... 343
10.5 Troubleshooting ... 344
Eclipse installiert die App nicht auf dem Handy ... 344
App vermisst existierende Ressourcen ... 345
LogCat bleibt stehen ... 345

**11 Apps veröffentlichen** ... 347

11.1 Vorarbeiten ... 347
Zertifikat erstellen ... 347
Das Entwickler-Konto ... 349
Die Entwicklerkonsole ... 350
11.2 Hausaufgaben ... 353
Updates ... 353
Statistiken ... 355
Fehlerberichte ... 357
11.3 In-App-Payment ... 359
In-App-Produkte ... 361
Der BillingService-Apparat ... 363
BillingReceiver und BillingResponseHandler ... 365
11.4 Alternative Markets ... 367
Amazon AppStore ... 368
AppsLib ... 368
AndroidPIT App Center ... 370
SlideME.org ... 371

Die Buch-DVD ... 373

Index ... 375

# Vorwort

Android ist in! Das Smartphone-Betriebssystem erobert immer mehr Jacken- und Hosentaschen – und das nicht nur wegen des sympathischen grünen Maskottchens.

Smartphones haben erstaunliche Fähigkeiten und beflügeln die Fantasie. Es ist nicht besonders schwer, eigene Apps zu basteln – es geht sogar ganz ohne Programmiervorkenntnisse. Okay, Sie sollten mit Ihrem PC umgehen können, aber alles andere erkläre ich Ihnen in diesem Buch.

Wenn Sie dieses Buch durchgearbeitet haben, können Sie eigene Apps schreiben, die mindestens so erstaunlich sind wie jene, die Sie bereits von Ihrem Smartphone kennen.

Dass der Spaß dabei nicht zu kurz kommt, garantiere ich Ihnen. Also, worauf warten Sie?

*»Jede hinreichend fortgeschrittene Technologie ist von Magie nicht mehr zu unterscheiden.«*
*(Arthur C. Clarke)*

# 1 Einleitung

Seit Ende 2008 existiert mit Android ein Betriebssystem für Smartphones und andere handliche Geräte, das sich schnell Millionen Freunde gemacht hat. Hauptgrund dafür sind die unzähligen Apps, die sich in Sekundenschnelle installieren lassen. Obwohl diese kleinen Programme auf den ersten Blick unscheinbar wirken, haben sie die Leben vieler Menschen verändert – und bei vielen den Wunsch geweckt: Das will ich auch!

Gehören Sie zu diesen Menschen, können aber überhaupt noch nicht programmieren?

Gratulation! Dann haben Sie sich für das richtige Buch entschieden.

## 1.1 Für wen ist dieses Buch?

Heerscharen von Programmierern, die ich kenne, haben sich auf Android-Entwicklung gestürzt. Im Handumdrehen bastelten sie erste Apps zusammen, denn die Einstiegshürde ist niedrig. Folglich kann es auch für Noch-Nicht-Programmierer nicht allzu schwer sein, mit der App-Entwicklung zu beginnen. Es ist wirklich nicht besonders kompliziert, und die nötige Programmiersprache, Java, lernen Sie praktisch nebenbei.

Selbst wenn Sie noch nie programmiert haben, können Sie sich am Ende dieses Buches zur wachsenden Gemeinde der Android-Entwickler zählen!

Wer schon programmieren kann, wird in diesem Buch ein paar Kapitel überschlagen können, dann aber auch anspruchsvolle Android-Techniken kennenlernen – begleitet von einigen heißen Tipps.

Experten wundern sich vermutlich zwischendrin über einige Vereinfachungen, die ich im Interesse der Einsteiger vor allem beim Erklären von Java vornehme – drücken Sie einfach ein Auge zu, und konzentrieren Sie sich auf die fortgeschrittenen Technologien und Tipps zur Android-Entwicklung, und auch Sie werden dieses Buch nicht als nutzlos empfinden.

## Magie?

»Moin auch, die Temperatur beträgt heute 22 Grad und die Regenwahrscheinlichkeit fünf Prozent.« Eine Frauenstimme neben dem Kopfkissen weckt Sie mit freundlichen Worten, die Sie selbst ausgesucht haben. Falls Sie wieder einschlafen, versucht die Stimme es kurz darauf etwas lauter und weniger diskret: »Raus aus den Federn, aber zackig!«

Die neue Müslipackung erwähnt Zucker schon an zweiter Stelle – Sekunden später finden Sie heraus, dass der Hersteller die Rezeptur geändert hat. Ihr Missfallen, per Touchscreen im sozialen Netzwerk veröffentlicht, hat schon das halbe Land durchquert, bevor Sie das Fahrrad aus der Garage geholt haben. Eile ist nicht geboten, denn Sie wissen bereits, dass die S-Bahn spät dran ist. Im Zug nutzen Sie die Zeit, um Ihren breitschultrigen Fantasy-Helden im Umgang mit seinem neuen Kriegshammer zu trainieren.

Beim Einkaufen stellt sich heraus, dass es Ihre Lieblingsschokolade im neuen Supermarkt ein paar Straßen weiter zum Einführungspreis gibt. Glücklicherweise ist der Laden auf Ihrem elektronischen Stadtplan eingezeichnet – nicht aber der Hundehaufen, in den Sie treten, weil Sie nur auf das Handy-Display starren.

Jetzt hätten Sie gerne Ihren Kriegshammer – und an diesem Punkt stoßen Sie dann doch an die Grenze zwischen Technologie und Magie.

Überflüssig zu erwähnen, dass die Generationen Ihrer Eltern und Großeltern den größten Teil dieser Geschichte als Ausgeburt Ihrer überbordenden Fantasie bezeichnen würden. Sie wissen es besser, denn Sie halten ein Stück Technologie in den Händen, dessen Fähigkeiten mehr als nur erstaunlich sind. Besser noch: Sie beherrschen sie. Sie erweitern sie. Sie fügen weitere Magie hinzu.

Solange der Akku reicht.

## Große Zahlen

Zum Zeitpunkt der Drucklegung dieses Buches (Herbst 2011) wurden weltweit über 550.000 neue Android-Geräte aktiviert. *Jeden Tag*. Über 130 Millionen sind insgesamt bereits auf der ganzen Welt in Betrieb. Das entspricht in etwa der verkauften Gesamtauflage von Tolkiens Roman »Der kleine Hobbit«.

500.000 Apps sind im Android Market verfügbar, den größten Teil davon können Sie kostenlos auf Ihr Smartphone runterladen und benutzen. Das funktioniert einfach und schnell. Deshalb wurden seit dem Erscheinen von Android bereits an die sechs Milliarden Apps installiert (Quelle: *androidlib.com*).

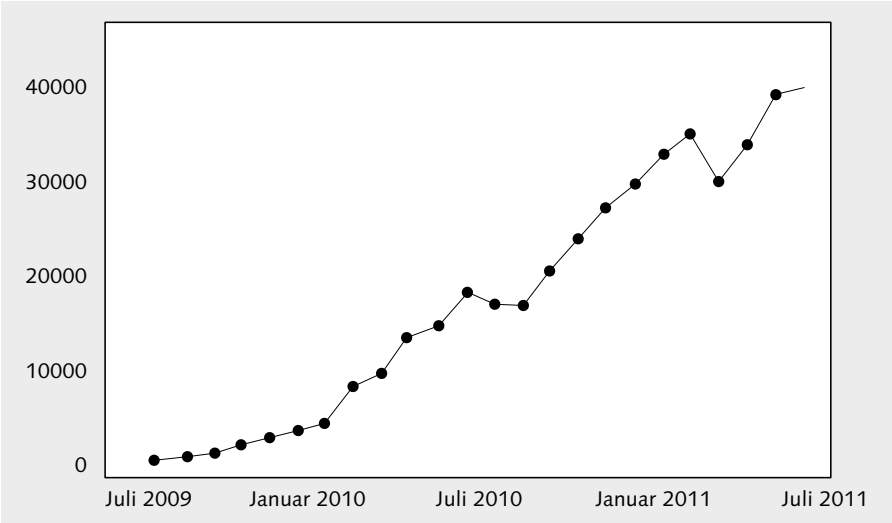



**Abbildung 1.1** Jeden Monat erscheint eine fünfstellige Anzahl neuer Apps im Android Market (Quelle: androidlib.com).

Sehr hoch ist aber noch eine andere Zahl: die Anzahl der wieder deinstallierten Apps. Anschauen kostet nichts, außer ein paar Minuten Zeit (Flatrate vorausgesetzt), Löschen noch weniger. Allein schon aufgrund des begrenzten Speichers üblicher Android-Geräte müssen Sie davon ausgehen, dass das Entsorgen einer App häufig zu ihrem Lebenszyklus gehört.

Wenn Sie jenen unzähligen Apps eigene hinzufügen möchten, die nicht so schnell wieder deinstalliert werden, dann ist dieses Buch für Sie genau richtig.

## Technologie für alle

Auch Handys des letzten Jahrhunderts waren prinzipiell programmierbar. Sie verfügten nicht über Bewegungssensoren oder Satellitenempfänger – grundsätzlich aber enthielten sie ähnliche Digitalelektronik wie Android-Handys: Mikroprozessoren, Speicher und Schnittstellen. Hardware, auf der Software lief und erstaunliche Dinge bewirkte. Es gibt jedoch einen entscheidenden Unterschied: Nur ausgewiesene Experten, die über teure Spezialgeräte verfügten, konnten solche Handys programmieren.

Bei Android-Smartphones sieht die Sache anders aus. Hersteller Google hat sich eine einfache Strategie überlegt: Je mehr Apps es gibt, desto interessanter sind Smartphones, und desto mehr Menschen kaufen welche. Also brauchen wir möglichst viele Leute, die Apps entwickeln. Daher gestalten wir das Entwickeln von Apps so einfach, dass es jeder kann!

**Abbildung 1.2** Nicht jedes Handy war vor dem Smartphone-Zeitalter nur zum Telefonieren geeignet – bloß könnten weder Sie noch ich Apps dafür entwickeln.

Ihr PC oder Laptop sind völlig ausreichende Arbeitsgeräte, um mit der App-Entwicklung zu beginnen. Die nötige Software ist kostenlos. Als Programmiersprache kommt Java zum Einsatz – eine Sprache, die leicht zu lernen ist und nicht jeden kleinen Fehler mit rätselhaften Abstürzen oder Datenverlust bestraft. Eine Sprache, die so verbreitet ist, dass es jede Menge Bücher dazu gibt – und Bekannte, die man fragen kann, wenn man nicht weiterkommt.

Wenn Sie ein solcher Bekannter werden wollen, dann ist dieses Buch für Sie ebenfalls genau richtig.

## Die Grenzen der Physik

Einstein hat herausgefunden, dass Masse den Raum krümmt, dass Raum und Zeit zusammenhängen und dass Genies auch ohne Friseure auskommen. Mithilfe eingebauter Sensoren kann ein Smartphone zwar das Schwerefeld der Erde messen, um oben und unten auseinanderzuhalten. Merklich beeinflussen kann aber selbst das massivste Gerät weder Raum noch Zeit, sodass Zeitmaschinen weiterhin nur in der Fantasie einiger Science-Fiction-Autoren existieren.

Daher kommen Sie auch mit dem modernsten Smartphone nicht daran vorbei, eine Menge Zeit zu investieren, bevor Sie Ihre ersten wirklich magischen Apps fertigstellen werden. Zu ersten vorzeigbaren Apps werden Sie schnell kommen, jedoch: Die Android-Technologie ist sehr mächtig, sonst könnte sie kaum ihre Wunder wirken. Manchmal werden Sie vor einem Rätsel stehen, aber ich werde Ihnen die nötigen Schlüssel in die Hand drücken, um all jene verborgenen Schatztruhen zu öffnen, die am Wegesrand stehen, und genug Plastiktüten, um all die Tretminen zu entfernen, die rücksichtslos in der Gegend deponiert wurden.


**Abbildung 1.3** Selbst besonders schwere Smartphones wie dieses Motorola Milestone krümmen nicht messbar den Raum (beachten Sie die untere Skala der Waage).

Auch wenn es eine lange, streckenweise beschwerliche Reise wird: Die gute Nachricht ist, dass die Steigung des Weges nicht mit jener der Zugspitzbahn (bis zu 250 Promille) zu vergleichen ist. Android-Entwicklung hat eine sehr flache Lernkurve, denn die wirklich kniffligen Dinge nimmt uns das System mit dem freundlichen, etwas pummeligen Roboter ab. Sie werden nur wenige Minuten benötigen, um Ihre erste App zu schreiben.

Und wenn Sie darüber staunen möchten, wie einfach Android-Entwicklung ist – auch dann ist dieses Buch für Sie genau richtig.

## 1.2 Unendliche Möglichkeiten

Smartphones bieten Unmengen Funktionen – genau das hat ihnen schließlich ihren Namen eingebracht. Sie sind *smart.* Apps können die bereitgestellten Funktionen ausnutzen, ganz nach Bedarf. Natürlich bringt Ihnen dieses Buch nicht jede dieser Funktionen im Detail bei. Aber Sie erhalten die nötige Grundausbildung.

## Baukasten

Vielleicht haben Sie als Kind mit einem Metallbaukasten gespielt. Oder mit Lego. Entwickler gehen nicht anders vor, um einfache Apps zu erstellen: Sie nehmen vorhandene Teile und stecken sie zusammen. Wenn etwas klemmt, liegt ein Hammer bereit.

Wenn Sie sich die Apps auf Ihrem Smartphone in Ruhe anschauen, werden Sie sehen, dass sie eine ganze Reihe Komponenten gemeinsam haben. Buttons in Apps mögen unterschiedlich groß sein, unterschiedliche Beschriftungen tragen und natürlich verschiedene Wirkung entfalten, wenn sie jemand drückt. Aber alle sind **Buttons**. Sie müssen sich nicht darum kümmern, wie der Button auf den Bildschirm kommt, wie der Touchscreen funktioniert oder dass er unterschiedlich aussehen kann, abhängig davon, ob er deaktiviert oder gerade gedrückt wurde.

Dasselbe gilt für Textbausteine, Eingabefelder, Bilder, Fortschrittsbalken etc. Mit wenigen Mausklicks lassen sich einfache Formulare, Listen und Bildschirmdarstellungen zusammenbauen und justieren. Um solche Benutzeroberflächen mit Leben zu füllen, kommt Java-Programmcode zum Einsatz. Da sich das Android-System um den größten Teil des Ablaufs ganz allein kümmert, sind oft nur wenige Zeilen einfacher Programmanweisungen notwendig, um eine sinnvolle App zu erhalten. So entstehen elektronische Einkaufszettel, Vokabeltrainer oder Zähler für erschlagene Mücken in einem feuchten Sommerurlaub in Skandinavien (wenn Sie glauben, so etwas sei überflüssig, waren Sie noch nie in Schweden).

Viele auf den ersten Blick einfache Apps nutzen Dienste im Internet. Prinzipiell kann fast jede Funktion, die auf einer Webseite zur Verfügung steht, in eine Android-App verpackt werden. Zwar bietet Android auch einen Browser, aber oft ist es im Mobilfunknetz viel effizienter, keine ganzen Webseiten abzurufen, sondern nur die gewünschte Funktion. Auf diese Weise lässt sich beispielsweise sehr einfach ein Synonymwörterbuch (Thesaurus) bauen, das genauso funktioniert wie die entsprechende Funktion auf der Webseite *wortschatz.uni-leipzig.de*.

Solche öffentlich und kostenlos zugänglichen Internet-Komponenten können Sie ähnlich den oben aufgeführten visuellen Komponenten als »Blackbox« in Ihre App integrieren, ohne über die internen Vorgänge Bescheid zu wissen oder sie gar selbst zu bauen.

Softwareentwicklung im 21. Jahrhundert hantiert nicht mehr mit einzelnen Bits und Bytes. Das wäre viel zu aufwendig, Sie würden niemals auch nur mit der einfachsten App fertig werden. Vielmehr besteht die Herausforderung darin, die richtigen Komponenten zu kennen, die optimal zueinander passen, und sie dann so zusammenzusetzen, dass sie fehlerfrei funktionieren.

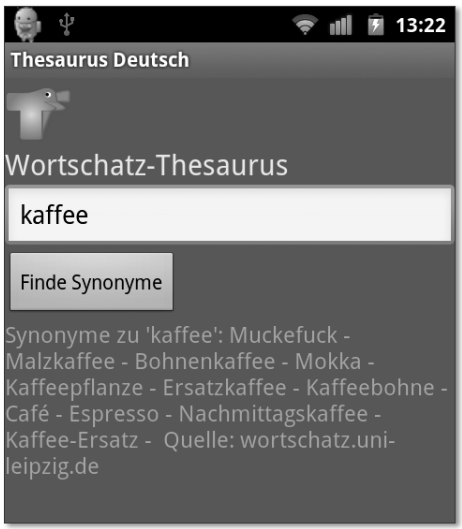



**Abbildung 1.4** Frei zugängliche Internet-Dienste lassen sich leicht in Android-Apps verpacken und ersparen dem Benutzer den Umweg über Browser und Webseite.

## Spiel ohne Grenzen

Im Laufe dieses Buches werden Sie die Android-Entwicklung zunächst anhand eines Spielprojekts kennenlernen. Das macht mehr Spaß als eine vergleichsweise trockene Synonyme-Anwendung, die meist weder von Lagesensoren noch vom Kompass besonders profitiert. Der Wow-Effekt ist größer, und die Themen sind herausfordernder.

Zu den herausragenden Eigenschaften von Smartphones gehören die eingebauten Sensoren. Vielleicht haben Sie gelegentlich in der TV-Serie **Star Trek: Das nächste Jahrhundert** das höchst erstaunliche Gerät namens Tricorder bewundert. Eine technische Meisterleistung im handlichen Hosentaschenformat und gleichzeitig ein **Deus ex Machina** für die Drehbuchautoren, wenn ihnen für eine herbeigeführte Situation keine andere Lösung einfiel.

Vielleicht kennen Sie bereits die Android-App Tricorder – zugegebenermaßen kann man damit weder den Gesundheitszustand von Personen scannen noch das Smartphone zur Explosion bringen (zum Glück), aber die App hat doch mehr als Aussehen und Geräuschkulisse mit dem Gerät gemein, mit dem Picard, Riker und Data so gerne spielen.

Sie finden die kostenlose App leicht anhand ihres Namens im Android Market, sodass einer Installation nichts im Weg steht.

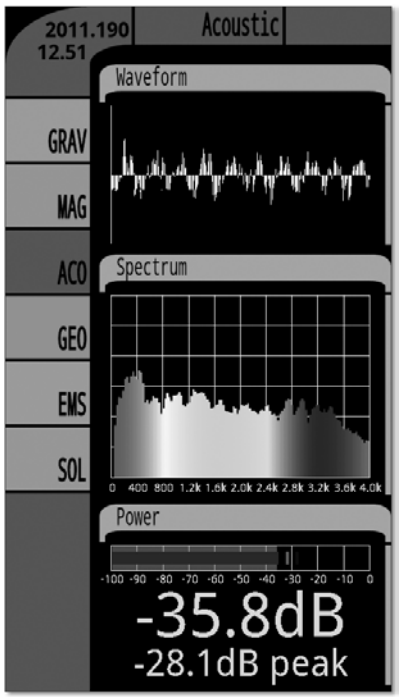



**Abbildung 1.5** Der Tricorder visualisiert meine Schreibtisch-Beschallung, in diesem Fall »Sweet Dreams« von Eurythmics, was Sie sicher sofort am Spektrum erkannt haben.

Der Tricorder ist die einfachste Möglichkeit, die eingebauten Sensoren zu besichtigen. Wählen Sie links einen Modus aus, zum Beispiel GRAV für den Beschleunigungssensor. Tippen Sie auf SCAN, um die Messung zu starten. Neigen Sie Ihr Gerät, und Sie sehen, wie sich die Anzeigen verändern. Wenn Sie das Handy flach auf den Tisch legen, ist die Beschleunigung in Richtung x und y natürlich 0, aber in Richtung z beträgt sie etwa 9,9 – das ist die Erdbeschleunigung. Die Werte sind freilich nicht genau genug für ernsthafte physikalische Messungen, aber sie genügen, um beispielsweise ein Spiel zu simulieren, in dem Sie eine Kugel durch Neigen des Geräts durch ein Labyrinth manövrieren. Auch Spiele, die das Handy als Steuerrad eines Autos oder anderen Gefährts verwenden, nutzen diese Sensoren.

Außerdem können Sie die Beschleunigungssensoren nutzen, um Erdbeben zu erkennen: Legen Sie das Handy flach hin, dann schlagen Sie mit der Faust auf den Tisch (bitte nicht zu fest, meine Versicherung zahlt ungern für dabei auftretende Schäden). Sie sehen einen deutlichen Ausschlag in der Fieberkurve.

Probieren Sie als Nächstes den Magnetfeldsensor aus: Starten Sie den Scan, halten Sie das Handy dann über ein Objekt aus Metall, beispielsweise eine Schachtel Schrauben oder Besteck. Besonders spannend sind 1- und 2-Euro-Münzen. Sie werden sehen, dass Münzen sehr unterschiedlich magnetisiert sind, je nach ihrem bisherigen Schicksal (Abbildung 1.6). Mit etwas Glück können Sie auf diese Weise einen Schatz finden, allerdings nur, wenn der höchstens einen Zentimeter tief vergraben ist – viel empfindlicher ist der Sensor nicht.

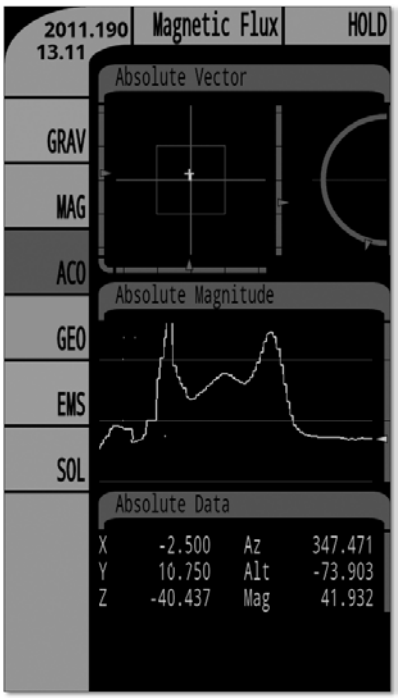



**Abbildung 1.6** Mit dem Scannen von unterschiedlichen Münzen kann man sich stundenlang beschäftigen. Ich habe hier vier verschiedene 2-Euro-Stücke der Reihe nach gescannt. Das magnetisch schwache ganz links war ein spanisches, aber ich bezweifle, dass man vom Magnetfeld auf die Herkunft schließen kann ...

Ohne Metall in der Nähe funktioniert der Magnetfeldsensor wie ein Kompass, was sich zahllose Apps im Android Market zunutze machen. Darauf kommen wir in einem späteren Kapitel zurück.

Der akustische Sensor (auch bekannt als Mikrofon) ist eine hübsche Spielerei, aber vom eingebauten Mini-Mikro kann man natürlich keine hohe Auflösung erwarten. Falls Sie auf die Idee kommen, mit dem Tricorder die Lautstärke eines startenden Flugzeugs in Dezibel zu messen, nehmen Sie bitte Gehörschutz mit zum Flughafen.

Für viele Anwendungen sehr spannend ist die Positionsmessung. Schalten Sie GPS (**Global Positioning System**) im Handy an, um auf wenige Meter genaue Geokoordinaten zu erhalten. Leichten Verfolgungswahn kann man empfinden, wenn man sogar ohne eingeschaltete Satellitenmessung eine relativ genaue Angabe erhält. Geben Sie ruhig mal die angezeigten Koordinaten bei Google Maps (*http://maps.google.de*) ein (genau wie angezeigt, also mit ° und Dezimalpunkt). Sie werden sehen, dass das Handy anhand der Mobilfunkzelle und in der Nähe befindlicher WLAN-Router ziemlich genau weiß, wo es sich befindet. Einerseits ist das unheimlich praktisch, weil es im Gegensatz zu GPS praktisch keine Energie verbraucht, andererseits ist es nur eine Frage der Zeit, bis jemand eine brillante Idee hat, wie er das missbrauchen kann.

Derzeit ist die beliebteste Anwendung der Positionsbestimmung mit Smartphones vermutlich **Geocaching** (*http://geocaching.com*), und das ist eine ziemlich harmlose Schnitzeljagd, die neuerdings sogar Kinder zur freiwilligen Teilnahme an einem Spaziergang animieren kann.

### Alles geht

Wenn Sie sich die Funktionen von Smartphones vor Augen halten, werden Sie schnell einsehen, dass es eigentlich nur eine Grenze für mögliche Anwendungsfälle gibt: Ihre Fantasie.

Smartphones sind eine sehr neue Technologie, die nur noch wenige Jahre davon entfernt ist, bisher übliche Mobiltelefone in der Hosentasche der meisten Menschen abzulösen.

Selbst Science-Fiction-Autoren oder Zukunftsforscher grübeln manchmal so lange über die neuen Möglichkeiten nach, dass ein inspirierter Entwickler an seinem Schreibtisch viel schneller damit fertig ist, eine revolutionäre App zu bauen.

Indem Sie dieses Buch lesen, sind Sie mittendrin in dieser Entwicklung – und können sie künftig mitgestalten.

Was für ein Spaß!

## 1.3 Was ist so toll an Android?

Was erklärt den Erfolg von Android? Rein numerisch gibt es mehr Android-Handys als beispielsweise iPhones – allein schon durch die große Anzahl verschiedener Hersteller. Die bieten unterschiedliche Designs und Ausstattungen, sodass jeder sein Lieblingsgerät findet.

Der Android Market ist vorinstalliert, und dort bekommen Sie Hunderttausende Apps aus einer Hand, die meisten sogar kostenlos. Sie müssen keine Apps aus obskuren Quellen installieren, wie das noch in der Zeit vor den Smartphones der Fall war – ohne es zu merken, hatten Sie da schnell mal ein Abo für 2,99 die Woche am Hals, obwohl Sie doch nur einen einzigen Klingelton haben wollten.

Die Apps im Android Market werden andererseits nicht von einer Prüfinstanz abgenickt, wie es beim AppStore von Apple der Fall ist. Was gleichzeitig ein Vorteil ist, bringt leider die Gefahr mit sich, dass Ihnen eine böswillige App unterkommt. Da hilft es nur, sich genau zu überlegen, welche Befugnisse Sie einer App erteilen – bei der Installation werden diese angezeigt, und wenn sie Ihnen zu freizügig erscheinen, verzichten Sie lieber auf die App. Ein simples Memory-Spiel dürfte Schwierigkeiten haben zu erklären, warum es auf die SMS-Funktion Ihres Handys zugreifen muss, und ein einfacher elektronischer Einkaufszettel benötigt wohl kaum Internet-Zugriff und Ihre genaue Position.

Wenn Sie schon länger ein Android-Gerät besitzen, dann kennen Sie sicher schon eine Menge sinnvoller Apps – lassen Sie mich Ihnen trotzdem eine kleine Auswahl vorstellen, die aus Sicht eines Entwicklers spannende Aspekte aufweist. Alle hier vorgestellten Apps sind kostenlos. Sie finden Sie im Android Market, indem Sie ihren Namen ins Suchfeld eingeben.

### MapDroyd

Sicher kennen Sie Google Maps – es ist schließlich auf fast jedem Android-Smartphone vorinstalliert. Diese App hat allerdings einen entscheidenden Nachteil: Sie lädt die jeweils benötigten Kartendaten aus dem Internet herunter.

Wenn Sie sich in einem Gebiet ohne Mobilfunkabdeckung befinden oder im Ausland, wo die Roaming-Kosten für Datenverkehr gerne mal die Kosten für Flug und Hotel übersteigen können, sollten Sie Google Maps tunlichst nicht aktivieren.

Die Alternative – MapDroyd – erlaubt es, Karten vorab herunterzuladen und auf der SD-Karte zu speichern. So können Sie bequem die Daten für alle Gegenden, die Sie zu besuchen gedenken, zu Hause im WLAN runterladen und verbrauchen später vor Ort keinen Cent Mobilfunkkosten.

MapDroyd (Abbildung 1.7) verwendet das Kartenmaterial von *openstreetmap.org* (OSM), das von einer offenen Community gepflegt wird. Sie werden staunen, was Sie darauf alles finden: Bushaltestellen mit Linieninformation, Restaurants, Sehenswürdigkeiten. Manchmal finden Sie dort sogar Fuß- oder Radwege, die in Google Maps fehlen.


**Abbildung 1.7** MapDroyd zeigt Ihnen eine Menge Details in der Umgebung – ganz ohne Internet-Kosten.

Die heruntergeladenen Karten sind dabei keineswegs große Grafikdateien, sondern Geodaten in einem speziellen Format. Sie werden dynamisch auf den Bildschirm gezeichnet, was Sie beim Zoomen sehr gut erkennen können. Dahinter steckt eine ziemlich umfangreiche Programmlogik, zumal die Karten auch noch abhängig von der Himmelsrichtung frei gedreht werden können. Über den eingebauten Kompass wird in einem späteren Kapitel noch zu reden sein – und auf die Kartendaten von OSM komme ich noch mal ausführlich zurück. Sie werden diese Karten sogar in einer eigenen App verwenden.

## Google Sky Map

Während MapDroyd Ihnen die Navigation zu Lande erleichtert, benötigen Sie ein virtuelles Planetarium wie Google Sky Map, um sich am Himmel zurechtzufinden. Planeten, Sterne, Sternbilder – Sie werden den Himmel mit anderen Augen betrachten, wenn Sie abends bewaffnet mit Ihrem Handy nach oben schauen.

Sie richten das Handy einfach in die richtige Richtung, und dank der Lagesensoren erscheint der passende Himmelsausschnitt, bloß mit Beschriftung. So können

Sie sehr schnell herausfinden, ob das helle Licht im Osten ein leuchtkräftiger Stern ist, die Venus (auch bekannt als Abend- bzw. Morgenstern) oder ein UFO (wenn das Objekt in Sky Map fehlt).

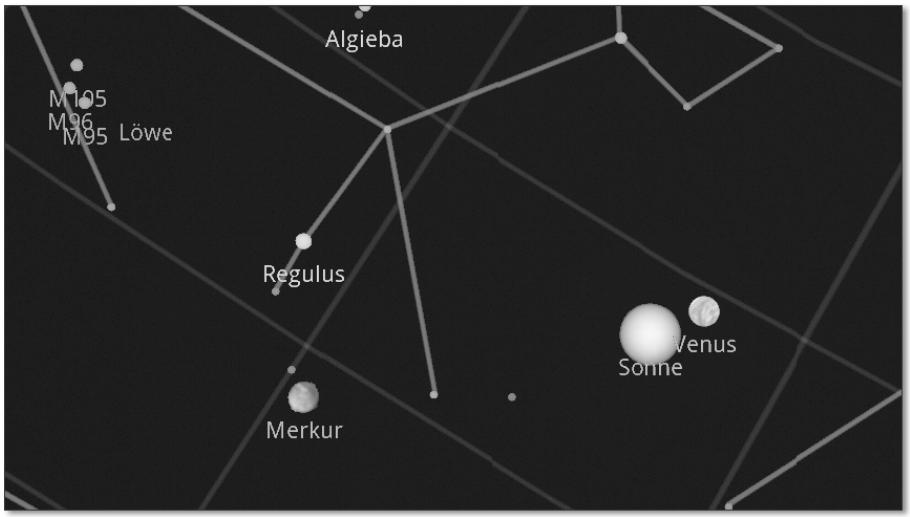



**Abbildung 1.8** Google Sky Map verrät Ihnen nicht nur die Positionen der Planeten, sondern auch die der Sonne – für den Fall, dass Sie die mal vor lauter Wolken nicht finden.

Sky Map bietet einen Nachtmodus für das an die Dunkelheit gewöhnte Auge (Rot auf Schwarz) und eine Galerie mit Fotos vom Hubble-Teleskop. Leider fehlt die Möglichkeit, ein Objekt anzutippen, um Informationen wie Entfernung oder Leuchtkraft zu erfahren – dazu müssen Sie dann doch Wikipedia (oder eine andere Informationsquelle) bemühen.

Ähnlich einer Karten-App verwendet Sky Map die Lagesensoren Ihres Geräts und die Geokoordinaten, um den richtigen Himmelsausschnitt anzuzeigen. Dahinter steckt eine Menge Mathematik, die selbst die meisten ausgebildeten Astronomen nicht auswendig programmieren könnten. Deshalb werde ich Ihnen in diesem Buch nur eine einfache Variante des Himmelskugel-Effekts erklären.

### Bump

Es gibt eine ganze Menge Möglichkeiten, Daten zwischen zwei Handys auszutauschen. Möchten Sie einem Freund Ihre Telefonnummer oder E-Mail-Adresse geben? Oder ein Foto, das Sie gerade geschossen haben? Bevor Sie mit Kabeln hantieren oder Ziffernfolgen diktieren, starten Sie doch einfach beide eine App, und stoßen Sie dann die Handys gegeneinander.


**Abbildung 1.9** Bump schickt Daten von einem Handy zum anderen.

Wie funktioniert das? Woher weiß Bump, an welches Handy es Daten übertragen muss? Was ist, wenn zur gleichen Zeit zwei Leute neben mir auch »bumpen«? Ist das nicht furchtbar unsicher?

Diese Fragen sind aus Sicht eines Entwicklers sehr spannend. Offensichtlich verwendet die App den Beschleunigungssensor Ihres Handys, um das Anstoßen zu erkennen. Aber das genügt nicht, denn sobald eine große Anzahl Anwender die App innerhalb eines kurzen Zeitraums verwendet, wäre es unmöglich, echte Bumps von zufälligen zu unterscheiden. Deshalb muss die App auf Ihre Geokoordinaten zugreifen. Sobald der Beschleunigungssensor Ihres Handys einen Stoß meldet, schickt Bump die Uhrzeit (auf einige Nanosekunden genau) und Ihren Standort an eine mächtige Internet-Applikation. Diese **Serveranwendung** wertet Geokoordinaten und Zeitpunkte aus und führt passende Paare zusammen. Falls nicht genau zwei Ereignisse zueinander passen, sondern beispielsweise nur eines, drei oder mehr, wird keine Verbindung aufgebaut.

Der Anwender wird dann noch einmal gefragt, ob er wirklich mit der erkannten Gegenstelle Kontakt aufnehmen möchte. Falls ja, kann die eigentliche Datenübertragung beginnen, die dann über eine verschlüsselte Verbindung stattfindet.

Sie sehen, dass Bump nicht ohne eine mächtige Serveranwendung auskommt. Aber gerade die schnelle, problemlose Verbindung von Smartphone-Sensoren und Rechnern im Internet ist es, die diese (und andere) neuartige Anwendungen ermöglicht.

Und weil die Synergie zwischen Smartphone und Internetanwendungen so wichtig ist, zeige ich Ihnen noch weitere Apps, die sie ausnutzen, und erkläre Ihnen in einem späteren Kapitel genau, wie Sie so was selbst bauen können.

### c:geo

Es gibt mehrere neuartige Spiele, die das Zusammenspiel von Smartphone-Features und einer Serveranwendung nutzen, und eines der bekanntesten dürfte **Geocaching** sein.

Geocaching ist ein Spiel, das vor Beginn des Smartphone-Zeitalters so gut wie unbekannt war. Es ist nichts anderes als eine Schatzsuche, wobei Ihr Smartphone die Schatzkarte darstellt und Sie dank Satellitennavigation zum Ziel führt. Dort ist dann unter einem Baumstamm oder in einer Mauerritze ein kleines Kästchen versteckt – der **Cache**. Darin befinden sich je nach Größe kleine Geschenke, von denen Sie eins gegen ein mitgebrachtes austauschen können. Außerdem gibt es eine Finder-Liste, in die Sie sich eintragen können. Gleichzeitig können Sie per Smartphone Ihren Fund auf der Webseite von *geocaching.com* bekanntgeben (wenn Sie sich dort angemeldet haben). Manchmal finden Sie in der Beschreibung eines Caches ein kleines Rätsel, das Sie lösen müssen, um den genauen Fundort zu identifizieren.

Man mag es kaum glauben, aber die GPS-Schatzsuche lockt sogar notorische Stubenhocker ins Grüne: Achten Sie mal beim nächsten Spaziergang darauf, wie viele Familien mit Smartphones durch den Wald laufen, auf der Suche nach dem nächsten Cache.

Es gibt eine ganze Reihe Apps, die die Schatzsuche mit dem Handy ermöglichen, und die beliebteste dürfte **c:geo** sein.

Für Entwickler gibt es eine ganze Reihe interessanter Fragen, die die Programmierung einer solchen App betreffen. Dass sie den GPS-Sensor des Smartphones verwenden muss, um die aktuellen Geokoordinaten zu erhalten, ist offensichtlich. Anschließend muss die App bei einer Serveranwendung fragen, welche Caches sich in der Umgebung befinden. Spannend ist die Bildschirmanzeige: Dort dient die von Google Maps zur Verfügung gestellte Karte als Grundlage, und für die Caches werden an der richtigen Stelle kleine Grafiken eingeblendet.


**Abbildung 1.10** c:geo blendet die nahen Geocaches in eine Karte ein. Wie Sie sehen, befinden sich im Neandertal gleich mehrere Fundorte.

Manchmal sind die GPS-Koordinaten nicht genau genug – weniger als zehn Meter Genauigkeit liefern sie nicht. Wenn Sie einen Cache von fünf Zentimetern Größe finden wollen, brauchen Sie Hilfestellung. Dazu blendet c:geo auf Wunsch einen Kompass ein, der Sie in die richtige Richtung leitet.

Auf den Kompass werden wir im Laufe dieses Buches noch zurückkommen, und Geokoordinaten dürfen natürlich auch nicht fehlen.

## barcoo

Jedes Smartphone besitzt eine eingebaute Kamera, und jedes Produkt im Regal besitzt einen Barcode. Diese Codes identifizieren Produkte eindeutig, und dank einer umfangreichen Datenbank im Internet können Sie beliebige Informationen nachschlagen.

Eine der Apps, die Ihnen dabei hilft, ist barcoo. Scannen Sie ein Produkt, und sofort schickt barcoo den erkannten Code zu seiner Serveranwendung, die postwendend alles dazu ausspuckt, was sie weiß. Je nach Art des Produkts erhalten

Sie aktuelle Preise, bei Lebensmitteln eine Ampel (größer und farbiger als auf den meisten Produkten) und Informationen zur ökologischen Verträglichkeit. Bei Büchern oder DVDs finden Sie manchmal Rezensionen und Bewertungen.

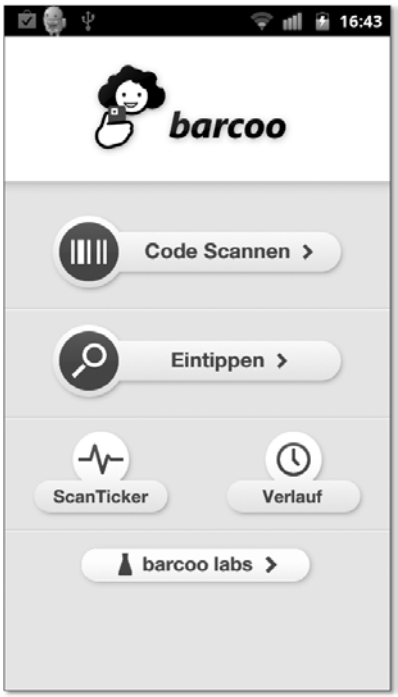



**Abbildung 1.11** barcoo scannt Barcodes und verrät Ihnen manchmal mehr, als Sie wissen wollten.

Wenn Sie barcoo ausprobieren, wird Ihnen sicher auffallen, dass Sie die Kamera nicht auslösen müssen, um einen Barcode zu scannen. Die App arbeitet nämlich mit dem Vorschaubild der eingebauten Digicam! Natürlich haben Vorschaubilder bei Weitem nicht die Auflösung von endgültigen Schnappschüssen, aber sie genügen, um die verhältnismäßig einfachen Muster eines Barcodes zu erkennen. In der App steckt einiges an Programmierarbeit, denn es ist keineswegs trivial, die Balken in einem verrauschten, grobpixeligen Bild richtig zu erkennen. In diesem Buch werden Sie in einem späteren Kapitel erfahren, wie das funktioniert.

### Öffi

Eine weitere App, die das Zusammenspiel von Ortsdaten, Google Maps und einer Serveranwendung nutzt, ist Öffi. Wenn Sie viel mit öffentlichen Verkehrsmitteln unterwegs sind, kann Ihnen diese App unschätzbare Dienste erweisen.

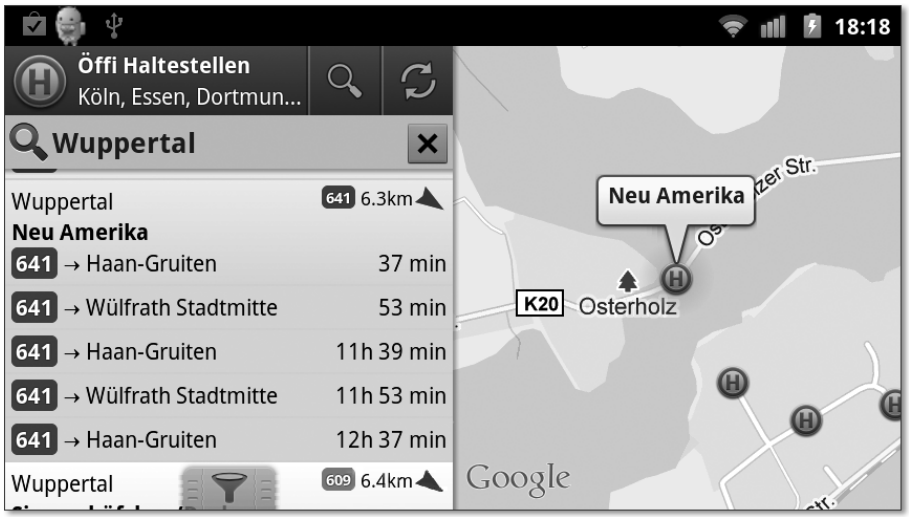



**Abbildung 1.12** Öffi weiß immer, wann der letzte Bus kommt – und wann der erste.

Öffi genügt in den meisten Fällen die netzwerkbasierte Ortsbestimmung, d. h., es kommt ohne GPS aus, das Ihren Akku ziemlich schnell leeren kann. Eine Serveranwendung verrät Öffi die Koordinaten der nahe gelegenen Haltestellen mitsamt den dort verkehrenden Linien und Fahrplänen. Die Haltestellen blendet Öffi in eine Karte von Google Maps ein, ähnlich wie c:geo dies mit Geocaches tut.

Hinzu kommt eine Auskunft über die schnellste Verbindung zu einer beliebigen Zielhaltestelle. Wenn Sie im Zug oder Bus sitzen, können Sie sich den Verlauf der Fahrt und gegebenenfalls Informationen zum Umsteigen anschauen. Noch dazu werden aktuelle Verspätungen berücksichtigt.

Es ist offensichtlich, dass der größte Batzen Arbeit hier in der Implementierung der Kommunikation mit den Servern der Verkehrsunternehmen steckt.

### Wikitude

Eine weitere App, die sich die Synergie zwischen Ortsbestimmung und einer Internet-Datenbank zunutze macht, ist Wikitude**.** Genau genommen, greift Wikitude sogar auf eine ganze Reihe Datenquellen zu – so ungefähr auf alles, was mit Geokoordinaten versehen ist. Alles, was Wikitude in der Nähe findet, wird dann in Form von kleinen Fähnchen ins Kamerabild eingeblendet.

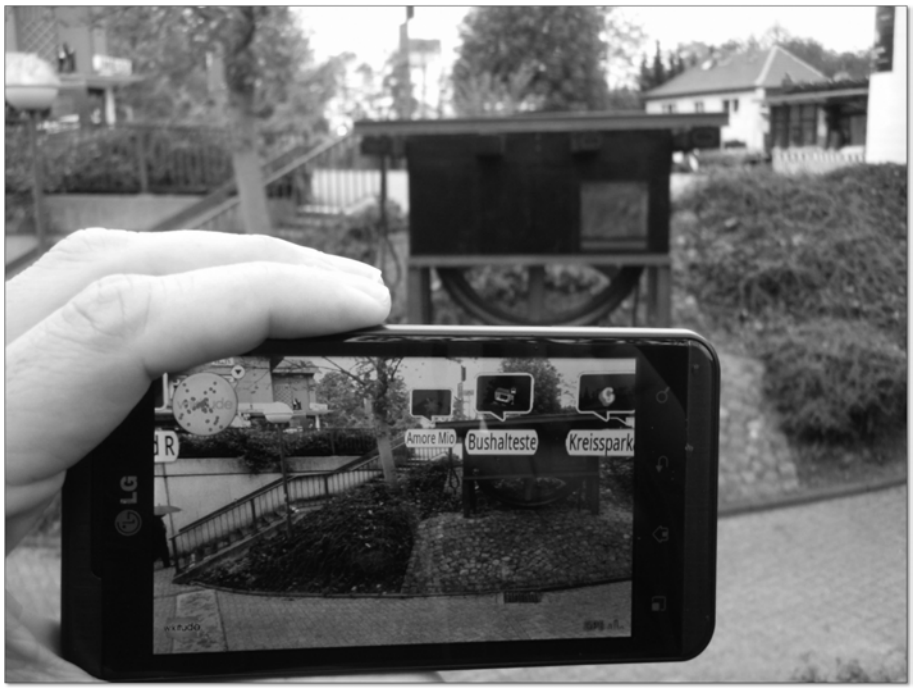



**Abbildung 1.13** Wikitude zeigt Ihnen, wo der Bus hält, selbst wenn ein Gegenstand den Blick versperrt.

Freilich ist nicht alles besonders sinnvoll: Beispielsweise ist es meist von geringem Interesse, sich Fotos von dem Ort anzusehen, an dem Sie ohnehin gerade stehen. Doch wer weiß, vielleicht ergibt sich eine überraschende Perspektive, wenn Sie sich anschauen, wie die Benutzer von Panoramico oder flickr.com Ihren Standort in Szene gesetzt haben.

Tendenziell lustig ist die Anzeige nahe gelegener Internet-Webcams: Theoretisch könnten Sie sich in deren Erfassungsbereich stellen und auf diese Weise sich selbst zuwinken.

Deutlich nützlicher sind Fähnchen, die zur nächsten Pizzeria, Apotheke oder Übernachtungsmöglichkeit führen.

Wenn Sie in der Fremde unterwegs sind, können Sie an Sehenswürdigkeiten sofort den zugehörigen Wikipedia-Artikel aufrufen.

Ähnlich wie Öffi verwendet Wikitude Ihre Geokoordinaten, um auf dem Server eine Umgebungssuche durchzuführen. Die Spezialität ist das Einblenden von Hinweisfähnchen ins Vorschaubild der Kamera. Dabei ist es allerdings keineswegs so, dass Wikitude Objekte im Kamerabild erkennt (dessen können Sie sich

vergewissern, indem Sie einfach das Objektiv mit dem Finger verdecken). Vielmehr ermittelt die App den Winkel, in dem Sie Ihr Handy halten, und berechnet dann, welche Fähnchen an welcher Stelle auf dem Bildschirm einzublenden sind.

Diese Vermischung von Kameravorschau und hineinplatzierten zusätzlichen Inhalten, **Augmented Reality**, ist die Königsdisziplin der Smartphone-Entwicklung. In einem späteren Kapitel werden wir darauf zurückkommen: Dieses spannende Konzept möchte ich Ihnen keinesfalls vorenthalten.

### Sprachsuche

Haben Sie sich auch schon einmal darüber geärgert, dass Computer heutzutage dermaßen neunmalklug sind, dass sie Ihnen alles über das Wetter am anderen Ende der Welt verraten können, aber erst, nachdem Sie ihnen mit der Tastatur oder Maus die entsprechende Frage gestellt haben? Wo sind die sprechenden Computer, die schon in der ersten Serie **Raumschiff Enterprise** vorkamen, die jedes Wort verstanden, Kommandos interpretierten und sogar wussten, wann sie *nicht* gemeint waren? Gerade ein Smartphone, das meist auf eine Tastatur verzichten muss, ist unheimlich unpraktisch, wenn Sie einen Text eingeben möchten – umso schlimmer, je länger der Text ist, und je mehr das Transportmittel, in dem Sie sich gerade befinden, wackelt.

Wir brauchen Spracheingabe!

Nun beherrschen selbst manche Telefone der Vor-Smartphone-Ära schon Sprachwahl, d. h., Sie sprechen dem Telefon einen Namen vor, und wenn Sie den später wiederholen, erkennt das Gerät den Klang Ihrer Worte wieder und wählt die zugehörige Nummer.

Die Google-Sprachsuche geht ein Stück weiter: Hier müssen Sie dem Gerät nicht erst jedes Wort beibringen, das wäre auch etwas viel verlangt. Stattdessen ermittelt die Sprachsuche auf dem Smartphone gewisse Charakteristika Ihrer gesprochenen Worte und versucht, sie mithilfe einer serverseitigen Datenbank zu identifizieren. Die ist verdammt mächtig, filtert sogar Nebengeräusche aus und kann aus ihren Fehlern lernen. Wenn Sie nicht gerade einen starken Dialekt sprechen und Ihren Suchbegriff direkt ins Mikrofon sprechen, funktioniert die Suche erstaunlich gut.

Seit Android 2.1 gibt es sogar einen recht unscheinbaren Mikrofon-Button auf der virtuellen Tastatur. Egal, in welcher App Sie sich befinden – E-Mail, SMS, Notizblock –, tippen Sie das Mikrofon an, und sprechen Sie. Wenn Sie deutlich artikuliert Hochdeutsch sprechen und keine außergewöhnlichen Worte verwenden, wird Sie die Qualität der Spracherkennung überraschen.


**Abbildung 1.14** Jede Notizblock-App wird ab Android 2.1 zum Einkaufszettel mit Spracheingabe.

Die Datenmenge, mit der die Google-Server bei der Sprachsuche umgehen müssen, übersteigt die Vorstellungskraft der meisten App-Entwickler. Das macht aber nichts, denn entscheidend ist: Sie können die Spracherkennung leicht in Ihre App integrieren! Das gilt nicht nur für alle Texteingabefelder, die die Bildschirmtastatur verwenden (dort ist die Spracheingabe automatisch verfügbar), sondern auch für jegliche Anwendung, die Sie sich vorstellen können. Alles ist vorstellbar: Auch ein Abenteuer-Rollenspiel, in dem Sie zur Wirtin marschieren und lautstark »Bier!« verlangen – wenn Sie Pech haben, antwortet die: »So was wie dich bedienen wir hier nicht« – in gesprochenen Worten, denn Sprach*ausgabe* beherrscht Android natürlich auch. Darauf werden wir bereits bei der ersten Beispiel-App zurückkommen, die Sie mit diesem Buch programmieren werden.

### Cut the Rope

Stellvertretend für das wachsende Genre der sogenannten Physik-Spiele sei hier Cut the Rope genannt. Sie müssen in diesem Spiel eine Süßigkeit in das hungrige Maul eines knuffigen, grünen Haustiers befördern. Knifflig daran ist, dass die

Süßigkeit allen möglichen Kräften ausgesetzt ist: der Schwerkraft, dem Auftrieb einer Seifenblase oder gummiartigen Schnüren, die Sie mit dem Finger durchtrennen können.

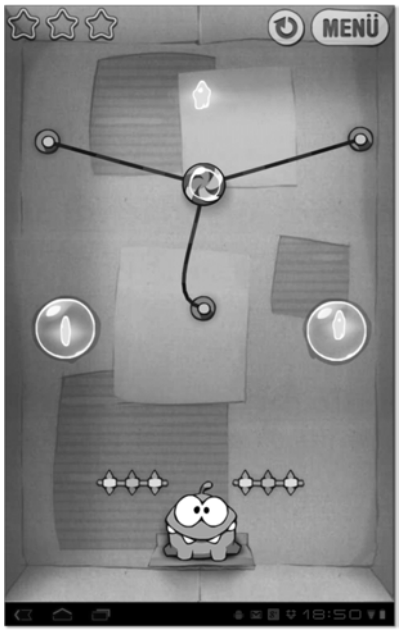



**Abbildung 1.15** Om Nom steht auf Süßigkeiten. Der Fütterung des knuffigen Wesens stehen allerdings knifflige Physik-Rätsel im Wege.

Der Clou an Physik-Spielen wie diesem ist, dass sie praktisch intuitiv spielbar sind, weil jeder aus seiner täglichen Lebenserfahrung die Spielregeln kennt. Das A und O ist natürlich eine akkurate Simulation der physikalischen Kräfte. Den Job erledigt eine Physik-Engine, die alle Kräfte, die auf Spielobjekte wirken, kennt und daraus die natürliche Bewegung errechnet.

Diese Regeln und Formeln sind überraschend einfach, allerdings hilft es, ein bis zwei Semester Physik studiert zu haben oder zumindest in der Schule im Physik-Unterricht gut aufgepasst zu haben. Für Mathe-Muffel gibt es fertige Physik-Engines, die wie die frei verfügbare andengine (*andengine.org*) bereits mit einer Grafik-Engine gebündelt sind. Im einfachsten Fall müssen Sie nur ein hübsches Bildchen malen, es der Engine unterschieben und ihr mitteilen, welche Kräfte wirken sollen.

Prinzipiell arbeiten die meisten Engines derzeit zweidimensional, aber der dahintersteckenden Physik ist das egal. Es ist allerdings nur eine Frage der Zeit, bis die 3-D-Grafikfähigkeiten von Smartphones so mächtig sein werden, dass auch 3-D-Physik-Spiele alltäglich werden.

### Shaky Tower

Ein weiteres Physik-Spiel, das sogar die Sensoren des Smartphones nutzt, ist Shaky Tower. Sie müssen gewisse Aufgaben lösen, die meistens damit zu tun haben, dass Sie grinsende Quadrate aufeinandersetzen. Dabei spielt die Schwerkraft eine entscheidende Rolle: Ein schiefer Turm kippt unweigerlich um. Wie schon bei Cut the Rope werkelt hier im Hintergrund eine Physik-Engine, die sich um die Berechnung der Kräfte und der Bewegungen kümmert.

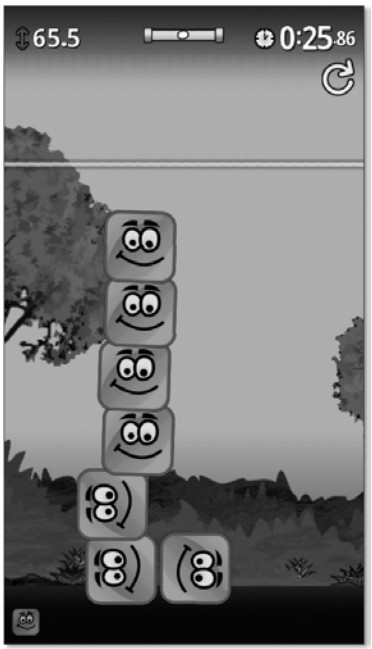



**Abbildung 1.16** Shaky Tower lässt Sie Türmchen bauen, die so lange stehenbleiben, bis Sie das Handy schräg halten.

Der Unterschied: Nicht nur die Physik des virtuellen Raums wird berücksichtigt, sondern auch die reale. Die Lagesensoren des Handys werden ausgelesen und fügen dem Geschehen auf dem Bildschirm weitere Gravitationskräfte hinzu.

Es gibt eine ganze Reihe von Spielen, die auf dieses Konzept setzen, allerdings verbraucht sich der Effekt recht schnell. Technisch ist interessant, dass Sie der verwendeten Physik-Engine äußere Kräfte hinzufügen müssen, die sich aus den Werten der Lagesensoren berechnen. Die größte Herausforderung besteht wie bei den meisten Spielen im Balancing: Wenn die Bewegung des Handys zu starken Einfluss auf das Spielgeschehen hat, wird es viele Spieler überfordern. Ist der Einfluss zu gering, wird der Effekt als zu gering empfunden. Hier ist das Fingerspitzengefühl des Programmierers gefragt – und jede Menge Testen.

*»Vorhin hat's noch funktioniert …!«*
*(Inschrift auf dem Grabstein des Unbekannten Programmierers)*

# 2 Ist Java nicht auch eine Insel?

Dieses Kapitel wird Ihnen einen ersten Überblick über die Programmiersprache Java geben. Selbst wenn Sie vorher noch nie programmiert haben, wissen Sie am Ende alles Nötige, um mit der App-Entwicklung zu starten.

## 2.1 Warum Java?

Die Erfinder von Java waren vermutlich passionierte Trinker asiatischen Kaffees, denn entsprechende Begriffe finden sich überall in ihrem Umfeld: Da ist mal von Beans (Bohnen) die Rede, eine Entwicklergemeinde nennt sich nach Jakarta, der Hauptstadt von Indonesien, die wiederum auf der Insel namens Java liegt. Ich möchte auf die Legenden über den Kaffeekonsum der Java-Erfinder an dieser Stelle nicht weiter eingehen. Tatsache ist, dass Java mehr ist als eine Programmiersprache.

Sie haben vielleicht die Namen diverser Programmiersprachen gehört: C, C++, Pascal, Fortran. Sie alle haben eine Gemeinsamkeit: Der Programmierer schreibt Anweisungscode in Textform, und ein spezielles Programm namens Compiler übersetzt das in Maschinensprache. Denn nur Maschinensprache kann die CPU, die Denkmaschine jedes Computers, verstehen und ausführen. Nun gibt es allerdings eine ganze Menge unterschiedlicher CPUs, die verschiedene Maschinensprache-Dialekte sprechen. Damit ein Programm auf jedem Computer läuft, müsste der Code also für jede existierende CPU einmal vom passenden Compiler übersetzt werden. Das ist eine sehr aufwendige Sache.

Computer verfügen von Haus aus über eine ganze Menge Funktionen, die Programme benutzen können: Bildschirmausgabe, Drucken, Internet-Zugriff etc. Allerdings sind all diese Funktionen bei jedem Betriebssystem anders. Deshalb läuft ein Programm, das auf einem Windows-PC von einem Compiler übersetzt wurde, nicht ohne Weiteres auf einem Mac, selbst wenn die gleiche CPU drinsteckt.

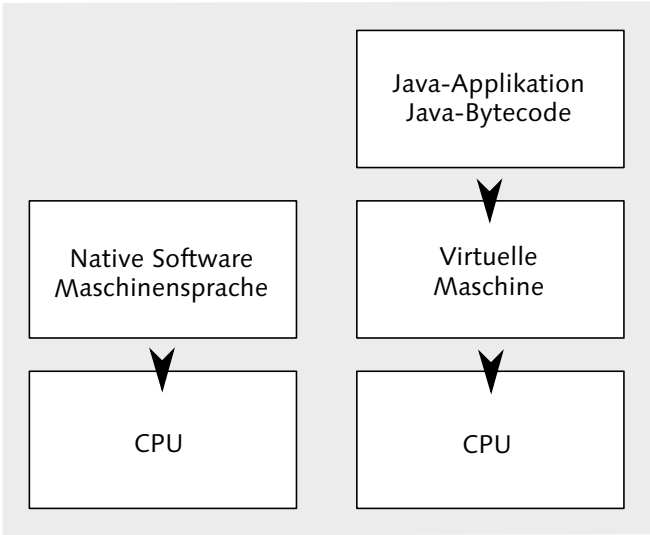



**Abbildung 2.1** Native Software ist in Maschinensprache geschrieben und wird direkt von der CPU ausgeführt. Java-Bytecode dagegen läuft in einer virtuellen Maschine.

Java löst beide Probleme, indem es eine einheitliche Plattform schafft. Jedes Java-Programm läuft auf jedem Rechner, wenn die **Java Runtime Environment** (JRE) vorhanden ist. Die JRE ist freilich für jedes Betriebssystem unterschiedlich, muss aber nur einmal installiert werden. Java-Programme bestehen aus einem speziellen Bytecode, der von der **virtuellen Maschine**, die in der JRE steckt, ausgeführt wird – egal, auf welchem Rechner (Abbildung 2.1).

Für die Erfinder von Android war es logisch, sich an dieser Idee zu orientieren. Deshalb kommt jedes Android-Gerät mit einem von uns weitgehend unbemerkten Linux-Betriebssystem daher, verfügt aber auch über eine JRE namens **Dalvik VM**. Allerdings unterscheidet es sich in gewisser Hinsicht von JREs auf PCs, sodass nicht jedes Java-Programm auf Anhieb läuft. Vor allem dann nicht, wenn es mit dem Benutzer interagieren möchte. Das liegt aber in der Natur der Sache, denn sowohl die Bedienung als auch die Bildschirmdarstellung auf einem Handy einerseits und einem PC andererseits unterscheiden sich fundamental. Über die Details der Unterschiede zwischen Dalvik und dem Original-Java streiten sich derzeit eine Menge Firmen vor hohen Gerichten um noch höhere Geldbeträge.

Glücklicherweise haben die Erfinder von Java noch mehr schlaue Ideen gehabt. Vor allem haben sie die vernünftigsten Eigenschaften existierender Programmiersprachen übernommen und die kniffligsten weggelassen. In C oder C++ findet man beispielsweise oft Programmcode, den auch ein Experte nur mit Zeit und Mühe verstehen kann. In Java passiert das selten. Java ist leicht lesbar und

deshalb leicht zu erlernen. Sicher hatten die Android-Macher auch diese Tatsache im Hinterkopf, als sie ihr System konzipiert haben. Denn: Je mehr App-Entwickler es gibt, desto erfolgreicher wird Android.

Mindestens eine Million Entwickler weltweit verdienen ihren Lebensunterhalt mit Java. Seit der ersten Version 1996 haben immer mehr Programmierer die Vorzüge von Java erkannt. Version 7 erschien Mitte 2011 und enthält seit der Vorversion eine ganze Reihe interessanter Neuerungen, die Technologie ist also alles andere als eingeschlafen.

Schließlich lassen sich in Java nicht nur Android-Apps schreiben, sondern auch Desktop-Anwendungen wie Eclipse (das werden Sie in Kürze näher kennenlernen), hochkomplexe Serveranwendungen (z. B. eBay) oder auch Browser-Spiele (wie Runescape).

**Abbildung 2.2** Viel Betrieb herrscht im in Java geschriebenen 3-D-Browser-Spiel »Runescape«.

Sie sehen: Java kann eine ganze Menge. Bloß die Steuerung von Atomreaktoren schließen die Java-Lizenzbedingungen explizit aus. Man weiß ja nie ...

## 2.2 Grundlagen

Viele Java-Kurse bringen Ihnen die Programmiersprache anhand von Mini-Anwendungen und Code-Schnipseln bei, die Lottozahlen generieren (leider meist die falschen), Stundenpläne ausgeben oder Kunden namens Max Muster-

mann Rechnungen für vermutlich irrtümlich gelieferte Schiffscontainer ausstellen. Wir drehen den Spieß um: Sie lernen Java von Anfang an anhand »androidische« Beispiele. Nur die allernötigsten Grundbegriffe erkläre ich mit einem Rundumschlag vorab. Wenn Sie es eilig haben, können Sie diese Seiten überblättern und später zurückkehren, wenn Sie nur noch Bahnhof verstehen.

## Objektorientierung: Klassen und Objekte

Java ist eine **objektorientierte** Sprache. Diese Art zu programmieren hat sich seit Jahren bewährt, weil sie sich stark an der Realität orientiert, die aus miteinander in Beziehungen stehenden **Objekten** besteht. Denn Ihr Auto ist ein **Objekt**, Ihre Schreibtischlampe ist eins, die Tomaten in Ihrem Kühlschrank sind Objekte – selbst Sie sind ein Objekt. Sie können intuitiv mit Objekten hantieren, ihnen Attribute zuweisen oder sie manipulieren. Deshalb erleichtert Objektorientierung das Verständnis zwischen Mensch und Maschine.

Entscheidend ist, dass Objekte einer Sorte eine ganze Menge gemeinsam haben. Jedes Auto hat beispielsweise ein Kennzeichen, jede Lampe hat einen Einschalter, jede Tomate einen Reifegrad etc. Diese Verallgemeinerungen oder Vorlagen nennt man **Klassen**. Klassen sind allgemeine Beschreibungen, und Objekte sind konkrete Instanzen von Klassen. Eine Klasse deklariert man in Java mit dem Schüsselwort `class`:

```
class Auto { };
```

Eine Klasse ist aber nicht mehr als eine Blaupause, eine Vorlage (in diesem Fall eine ziemlich leere). Stellen Sie sich die Klasse wie einen Bestellschein vor. Sie füllen beispielsweise einen Auto-Bestellschein aus, reichen ihn beim zuständigen Schalterbeamten (der Java Runtime) ein und erhalten das bestellte Fahrzeug. In Java sieht das dann so aus:

```
Auto meinCabrio = new Auto();
```

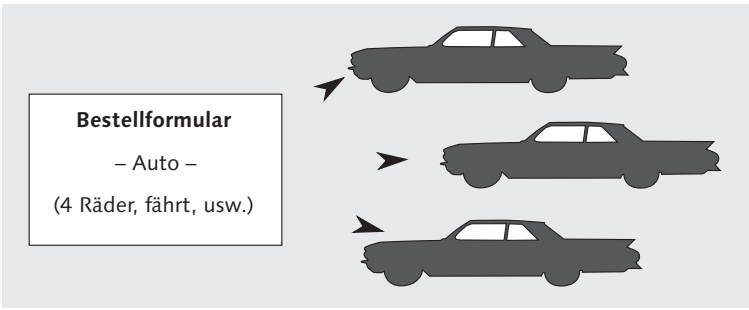



**Abbildung 2.3** Eine Klasse (das Bestellformular) dient zum Erzeugen konkreter Objekte (der Autos) mit »new«-Anweisungen.

Gewöhnen Sie sich daran, Sätze nicht mit einem Punkt zu beenden, sondern mit einem Semikolon, wenn Sie mit Ihrem Java-Compiler sprechen. Das Semikolon markiert das Ende einer Anweisung.

**Denglisch oder was?**

Ein professioneller Java-Entwickler würde an dieser Stelle den Zeigefinger heben. Denglischer Programmcode? So nicht, meine Damen und Herren!

In der Tat ist Englisch die Weltsprache der Programmierer. Alle Namen von Klassen, Objekten etc. und sogar erklärende Kommentare haben gefälligst in Englisch formuliert zu sein. Das klingt auf den ersten Blick übertrieben, aber glauben Sie mir: Wenn Sie zum ersten Mal in fremdem Programmcode französische, italienische oder sonst wie unverständliche Buchstabenkombinationen angetroffen haben, unterschreiben Sie diese Forderung, ohne mit der Wimper zu zucken.

Allerdings richtet sich dieses Buch an Einsteiger. Java lernen ist nicht schwer, aber immerhin anspruchsvoll genug, um sich nicht mit unnötigen Verständnisproblemen herumschlagen zu wollen.

Darüber hinaus können Sie auf den ersten Blick unterscheiden, was Java-Schlüsselwörter sind und wobei es sich um »unseren« Programmcode handelt, denn Letzterer ist in Deutsch geschrieben.

Denglischer Programmcode klingt schräg und tut ein bisschen in den Augen weh – aber für den Anfang können wir prima damit leben.

Wenn dieser Code später ausgeführt wird, erledigt die Java Runtime mehrere Dinge:

1. Ein neues Objekt von der Klasse `Auto` wird erzeugt und bis auf Weiteres im Speicher abgelegt.
2. Dieses Objekt erhält die Bezeichnung `meinCabrio`. Unter diesem Variablennamen können Sie Ihren fahrbaren Untersatz später verwenden, denn Java merkt sich, an welcher Stelle im Speicher der zugehörige Wert abgelegt ist.

Sie können ohne Weiteres mehrere Objekte einer Klasse erzeugen (instanziieren):

```
Auto meinCabrio = new Auto();
Auto alteKarreDesNachbarn = new Auto();
```

Sowohl Ihr Cabrio als auch die Karre des Nachbarn sind nun unabhängige Objekte, die aber beide eine gemeinsame Vorlage haben. Sie können mit `meinCabrio` alles tun, was Sie mit einem Auto tun können, und mit `alteKarreDesNachbarn` ebenfalls. Beide können beispielsweise fahren (auch wenn Sie das bislang noch nicht programmiert haben), aber wenn das eine fährt, kann das andere stehen und umgekehrt. Beide stammen von ein und derselben Vorlage, können sich aber unabhängig voneinander verhalten.

Variablennamen dürfen übrigens keine Leerzeichen enthalten. Deshalb schreiben Sie `meinCabrio` in einem Wort – wie die Deutsche Bahn es mit dem Regional-Express tut, bloß hat das vermutlich andere Gründe. Umlaute und Eszett sind zwar erlaubt, aber tun Sie sich einen Gefallen, und verzichten Sie darauf. Ohne auf die komplizierten Hintergründe einzugeben, kann ich Ihnen garantieren, dass Sie keinen Spaß an Objekten namens `Spaß` haben werden.

Die im Speicher angelegten Objekte müssen irgendwann, spätestens aber bei Programmende, wieder entfernt werden, um Platz für andere Daten zu schaffen. Und damit sind wir bei einem der großen Vorteile von Java gegenüber anderen Programmiersprachen: Das Reinemachen geschieht von alleine!

Bitte wundern Sie sich also nicht, wenn Sie einen Java-Programmierer zu Hause besuchen und der Mülleimer überläuft: Er ist es von Java gewohnt, dass er sich nicht um das Aufräumen kümmern muss. In der Java Runtime läuft die ganze Zeit im Hintergrund der Garbage Collector. Wenn der etwas findet, das nicht mehr benötigt wird, gibt er den belegten Speicher frei.

### Konstruktoren

Wenn Sie ein neues Objekt mit `new` erzeugen, ruft Java ohne Ihr Zutun einen Programmcode auf, der **Konstruktor** heißt (Constructor). Dieser Code kann notwendige Initialisierungen vornehmen, hauptsächlich aber belegt er den nötigen Hauptspeicher, um das Objekt zu verwalten.

Klassen können mehrere Konstruktoren besitzen, die sich durch zusätzliche Parameter unterscheiden:

```
Auto karre = new Auto();
Auto rotesAuto = new Auto( "rot" );
Auto wunschauto = new Auto( luxusmarke, "schwarz" );
```

Was die Konstruktoren mit den ihnen übergebenen Parametern anstellen, ist deren Sache. Meist setzen sie aber grundlegende Attribute des Objekts, die oft später nicht mehr verändert werden können.

Manche Klassen stellen überhaupt keinen Konstruktor ohne Parameter zur Verfügung, weil sie sonst keinen Sinn ergäben:

```
Drache drache = new Drache( schatz );
```

## 2.3 Pakete

Java-Programme bestehen aus einer, meist aber mehreren Klassen. Üblicherweise schreibt man den Programmcode jeder Klasse in eine zugehörige Textdatei, die genauso heißt wie die Klasse. Diese Textdatei erhält die Dateiendung *.java*. Bei großen Programmen kann sich eine ganze Menge Dateien ansammeln, sodass der Überblick schnell verloren geht. Glücklicherweise lassen sich Klassen in einer Ordnerhierarchie speichern.

### Packages deklarieren

Ordner mit Klassen darin heißen in Java **Pakete** (Packages). Allerdings stehen zwischen den einzelnen Ordnern im Gegensatz zum Dateisystem keine Schrägstriche (unter Windows \ und unter Linux /), sondern Punkte. Außerdem erwartet der Java-Compiler, dass am Anfang jeder Quellcodedatei der Paketname explizit deklariert wird. Es hat sich in der Entwicklergemeinde eingebürgert, Pakete wie umgekehrte Domainnamen zu erfinden, zum Beispiel `de.androidnewcomer` oder `de.namemeinerfirma.demoapp`. Mit Internet-Domains, die Sie in einem Browser öffnen können, hat das übrigens nichts zu tun. Deshalb sieht die Quellcodedatei für unsere Auto-Klasse wie folgt aus:

```
package de.androidnewcomer;
public class Auto { };
```

Diese Datei muss *Auto.java* heißen und gehört in ein Verzeichnis namens *androidnewcomer*, das wiederum in einem Verzeichnis namens *de* liegt. Letzteres liegt im Hauptverzeichnis unseres Projekts.

**Groß oder klein?**

Vielleicht haben Sie schon einmal schmerzhaft bemerkt, dass verschiedene Betriebssysteme Groß- und Kleinschreibung sehr unterschiedlich handhaben. Unter Windows ist es egal, ob Sie große oder kleine Buchstaben verwenden, unter Linux nicht. Deshalb halten Sie sich unbedingt an die folgenden Regeln:

- Ordnernamen: Kleinbuchstaben
- Klassennamen: erster Buchstabe jedes Wortes groß, Rest klein
- Variablennamen: wie Klassennamen, aber erster Buchstabe klein
- Generell: keine Leerzeichen
- keine Sonderzeichen, ausgenommen der Unterstrich: _ (auf den verzichten Java-Programmierer aber meistens)
- keine Umlaute und Eszett
- Ziffern sind erlaubt, aber nicht am Anfang.

In obigem Listing habe ich Ihnen noch einen sogenannten **Modifizierer** (englisch Modifier) untergejubelt, nämlich das Wörtchen `public`: Es sorgt dafür, dass die Klasse `Auto` auch von allen anderen Klassen unseres Projekts verwendet werden kann. Da das in den meisten Fällen gewünscht ist, schreiben Sie immer `public class`.

Übrigens dürfen prinzipiell zwei Klassen auch denselben Namen haben, solange sie sich in zwei unterschiedlichen Paketen befinden. Wenn Sie beide Klassen in einer dritten verwenden möchten, müssen Sie sie irgendwie auseinanderhalten. Dazu können Sie den **qualifizierten Namen** (Qualified Name) verwenden:

```
de.verkehr.Auto cabrio = new de.verkehr.Auto();
de.modelle.Auto modellauto = new de.modelle.Auto();
```

In diesem Beispiel erzeugen Sie zwei Objekte der Klasse `Auto`, allerdings handelt es sich im ersten Fall um ein »richtiges« Auto und im zweiten um ein Modell. Um Verwirrung zu vermeiden, sollten Sie solche Namen aber besser nicht verwenden.

### Klassen importieren

Möchte eine Klasse in einem Paket eine Klasse in einem anderen Paket verwenden, muss sie sie explizit importieren. Nehmen wir für den Augenblick an, dass Sie eine App für Gemüsetransporter schreiben möchten. Dann könnte es sein, dass Sie zusätzlich die folgende Klasse benötigen:

```
package de.androidnewcomer.transportgut;
public class Tomate { };
```

Die Klasse `Tomate` befindet sich in einem anderen Paket als `Auto`, nämlich in `de.androidnewcomer.transportgut`. Wenn Sie später der Auto-Klasse beibringen möchten, wie man Tomaten transportiert, werden Sie Java erklären müssen, wie Autos Näheres über dieses rote Gemüse herausfinden können. Daher importieren Sie die Tomaten-Klasse:

```
package de.androidnewcomer;
import de.androidnewcomer.transportgut.Tomate;
public class Auto { };
```

Falls Sie nicht nur Tomaten, sondern auch anderes Transportgut verwenden möchten, legen Sie die zugehörigen Klassen der Ordnung halber ebenfalls in das Paket `de.androidnewcomer.transportgut`. Dann können Sie alles auf einmal importieren:

```
package de.androidnewcomer;
import de.androidnewcomer.transportgut.*;
public class Auto { };
```

Der Stern fungiert als Platzhalter für beliebige Klassennamen, wie Sie es vielleicht von Dateinamen her kennen: **.** meint alle Dateien, **.txt* nur Texte und **.jpg* nur (JPEG-)Bilder.

**Weniger ist mehr**

Keine Sorge: Das sieht nach viel Tipparbeit aus, aber wie ich bereits sagte, sind Programmierer ziemlich faul. Moderne Entwicklungsumgebungen nehmen uns auf Knopfdruck die Import-Arbeit ab. In Eclipse drücken Sie später einfach Strg + O (oder im Menü SOURCE • ORGANIZE IMPORTS), und die `import`-Anweisungen werden automatisch erzeugt. Immer wenn Eclipse eine Klasse des gewünschten Namens in mehreren Paketen findet, bittet es Sie, eine Wahl zu treffen.

## 2.4 Klassen implementieren

Bisher herrscht in der Beispielklasse `Auto` gähnende Leere zwischen den geschweiften Klammern. Das ist genau die Stelle, die eine Klasse mit Leben füllt. An die Arbeit!

### Attribute

Wenn eine Klasse Daten speichern soll, tut sie das gewöhnlich in **Attributen.** Wie funktioniert das?

Dazu verlassen wir das Beispiel mit dem Autotransporter und stellen uns ein waschechtes Abenteuerspiel vor, bei dem der Spieler auf einem Schreibtisch eine Geheimmitteilung finden muss – allerdings im Dunkeln. Deshalb stellen wir eine Lampe auf den Schreibtisch, die der Spieler anschalten muss. Es handelt sich bei der Lampe um ein Objekt, das einen bestimmten Zustand hat – »an« oder »aus«.

Zwei Zustände – da denken Programmierer sofort an Boolean (ausgesprochen »Buhliän«, Betonung auf der ersten Silbe). Ein **Boolean** ist der einfachste Datentyp, den ein Computer kennt. Er wird durch ein einzelnes Bit dargestellt, das entweder gesetzt ist oder auch nicht. Je nach Anwendungsfall spricht der Programmierer von den logischen Werten 1 oder 0, **wahr** oder **falsch**, **ja** oder **nein**, **an** oder **aus**. Gemeint ist immer dasselbe: ein mikroskopisch kleiner Schalter, dessen Zustand sich der Computer dauerhaft merken kann. Übrigens besitzt ein sehr einfach ausgestatteter PC heutzutage über acht Milliarden solcher Schalter (wenn er ein Gigabyte RAM eingebaut hat), von denen auf der Festplatte (acht Billionen = ca. ein Terabyte) will ich gar nicht erst anfangen. Sie brauchen nur einen einzigen Schalter, und den bauen Sie wie folgt in Ihre neue Klasse `Lampe` ein:

```
package de.androidnewcomer.abenteuer;
public class Lampe {
   Boolean angeschaltet;
};
```

Die Klasse namens `Lampe` besitzt jetzt ein **Attribut** namens `angeschaltet`. Wie schon bei der Deklaration unseres Cabrios steht auch hier der Name der Klasse `Boolean` vor dem Variablennamen, gefolgt von einem Semikolon. `Boolean` ist eine Klasse, die in Java fest eingebaut ist, daher müssen Sie sie nicht extra importieren.

Wenn Sie eine Schreibtischlampe erzeugen mit

```
Lampe schreibtischlampe = new Lampe();
```

wird diese zunächst nicht angeschaltet sein, weil Java jedes Boolean-Objekt automatisch beim Programmstart auf den Wert 0 (oder »falsch«) setzt. Allerdings schadet es nicht, zur Sicherheit und der besseren Lesbarkeit wegen das Objekt explizit zu initialisieren, ihm also einen Startwert zuzuweisen. Das sieht dann so aus:

```
package de.androidnewcomer.abenteuer;
public class Lampe {
  Boolean angeschaltet = false;
};
```

An irgendeiner Stelle in Ihrer App wird der Spieler nun den richtigen Knopf drücken, um nicht mehr im Dunkeln zu stehen. Sie benötigen also einen Befehl, um das Boolean-Attribut im Objekt `schreibtischlampe` auf »wahr«, »an« oder »logisch 1« zu setzen. Und der lautet wie folgt:

```
schreibtischlampe.angeschaltet = true;
```

Der Punkt steht zwischen dem Namen des Objekts und dem Namen des Attributs in diesem Objekt. Der neue Wert steht auf der rechten Seite des **Zuweisungsoperators** =.

Als Attribute innerhalb von Klassen können Sie beliebige andere Klassen oder primitive Datentypen verwenden. Die neben `Boolean` am häufigsten eingesetzten Klassen, die Java von Haus aus mitbringt, sind `String` und `Integer`.

`Integer` ist eine Klasse, die einen ganzzahligen Wert speichern kann, der zwischen –2.147.483.648 und +2.147.483.647 liegt. Unsere Lampe verfügt über eine Glühbirne, dessen Leistung mit einem solchen `Integer`-Objekt bequem gespeichert werden kann:

```
package de.androidnewcomer.abenteuer;
public class Lampe {
   Boolean angeschaltet = false;
   Integer leistung = 40;
};
```

**Klassen und Datentypen**

In Java ist nicht alles ein Objekt oder eine Klasse. Es gibt außerdem primitive Datentypen, die nicht die Eigenschaften von Klassen haben. Neue Instanzen müssen nicht mit `new` erzeugt werden, und man kann nicht mit dem Punkt ».« auf Methoden oder Attribute zugreifen. Teilweise ist der Unterschied leicht zu übersehen, etwa bei der Klasse `Boolean` und dem primitiven Datentyp `boolean`. Wann Sie am besten eine Klasse bzw. ein Objekt einsetzen und wann einen primitiven Datentypen, hängt von der jeweiligen Aufgabe ab. In den folgenden Kapiteln werde ich jeweils auf dieses Thema eingehen, wenn es sich anbietet.

Hier zunächst die obligatorische Liste der Datentypen in Java:

- `boolean`: `true` oder `false`
- `byte`: eine Zahl zwischen –128 und +127
- `short`: eine Zahl zwischen –32.768 und +32.767
- `int`: eine Zahl zwischen –2.147.483.648 und +2.147.483.647
- `long`: eine Zahl zwischen –9.223.372.036.854.775.808 und +9.223.372.036.854.775.807
- `float`: eine Kommazahl mit normaler Präzision
- `double`: eine Kommazahl mit doppelter Präzision
- `char`: ein Unicode-Zeichen oder -Buchstabe

In diesem Beispiel schrauben Sie standardmäßig 40-Watt-Birnen in Ihre Lampen. Jedes mit `new Lampe()` erzeugte Lampen-Objekt wird zunächst im Attribut für die Leistung den Wert 40 aufweisen. Falls unser wackerer Abenteuerspieler auf die Idee käme, eine Energiesparlampe mit einer Nennleistung von acht Watt einzusetzen, würden Sie schreiben:

```
schreibtischlampe.leistung = 8;
```

Eine solche Zuweisung darf beliebig oft erfolgen, wobei jegliche Erinnerung an den vorangegangenen Wert ausgelöscht wird. Kommazahlen – also etwa Preise wie € 1,50 – erfordern allerdings eine andere Klasse als `Integer`, nämlich `Double`. Da Java von amerikanischen Entwicklern erfunden wurde, müssen Sie jedoch das englische Dezimaltrennzeichen verwenden: den Punkt anstelle des hierzulande üblichen Kommas. Pedanten würden also die Leistung der Lampe beispielsweise wie folgt festlegen:

```
Double leistung = 39.249;
```

Äußerst flexibel ist die Klasse `String`. In ein Objekt dieser Klasse passen beliebige Zeichenfolgen, also Buchstaben, Ziffern, Wörter oder ganze Sätze. Lassen Sie uns als einfaches Beispiel eine Beschriftung hinzufügen:

```
package de.androidnewcomer.abenteuer;
public class Lampe {
   Boolean angeschaltet = false;
   Integer leistung = 40;
   String beschriftung = "";
};
```

Im Gegensatz zu konstanten `Boolean`-, `Integer`- oder `Double`-Werten müssen konstante `String`-Werte in doppelte Anführungszeichen gesetzt werden. Die Beschreibung ist standardmäßig leer (`""`) und lässt sich mit einer einfachen Zuweisung füllen:

```
schreibtischlampe.beschriftung = "Maximal 40 Watt";
```

Sie haben jetzt gesehen, wie Klassen Vorlagen für Objekte bieten können, die bestimmte Eigenschaften haben, und wie diese manipuliert werden. Ein paar Seiten liegen noch zwischen Ihnen und der ersten lauffähigen App, aber Sie wissen immerhin schon einmal, wie Sie das Licht anschalten können.

**Der kleine Unterschied**

Strings können alles Mögliche enthalten, auch Ziffern. Deren Wert ist dem String allerdings fürchterlich egal. Deshalb können Sie mit Zahlen, die in Anführungszeichen stehen, nicht rechnen: Java betrachtet sie nicht als Zahlen. Schauen Sie sich den kleinen, aber gerne übersehenen Unterschied an:

```
Integer Summe = 40 + 40;
```

Während diese Addition erwartungsgemäß den Wert 80 in das Objekt namens `summe` schreibt, passiert mit Strings etwas anderes:

```
String Summe = "40" + "40";
```

Der Operator `+` ist bei Strings anders definiert als bei Zahlen: Er fügt Strings aneinander. Das Resultat der String-Addition lautet also:

```
"4040"
```

## Methoden

Papier ist geduldig. Was hindert einen arglistigen Spieler daran, eine falsche Glühbirne in die Schreibtischlampe zu schrauben? Im Moment lediglich die Beschriftung. Also könnte dies passieren:

```
schreibtischlampe.leistung = 100;
```

Das zu verhindern, ist die Aufgabe dieses Abschnitts. Dazu werden Sie der Klasse `Lampe` etwas Neues hinzufügen: eine Methode. Im Gegensatz zu Attributen, die lediglich Daten speichern, sind Methoden *aktiv*. Sie führen Aktionen aus, manipulieren Daten, liefern Ergebnisse zurück oder interagieren mit dem Benutzer. Methoden werden in Klassen definiert und fassen ein paar Zeilen Programmcode unter einem eindeutigen Namen zusammen. Sie funktionieren dann in allen Objekten, die aus der Klasse erzeugt werden:

```
Package de.androidnewcomer.abenteuer;
Public class Lampe {
   Boolean angeschaltet = false;
   ovoid schalten() {
      angeschaltet = true;
   }
};
```

Vor dem Namen der eingefügten Methode `schalten()` steht das Schlüsselwort `ovoid`. Es legt fest, dass diese Methode keinen Wert zurückgibt. Stattdessen könnte hier ein Klassenname stehen, aber die Methode `schalten()` hat kein konkretes Ergebnis, das sie zurückliefern müsste. Ebenso wenig erwartet sie Parameter, deren Platz zwischen den runden Klammern wäre. Zwischen den geschweiften Klammern steht der Programmcode, den die Java Runtime immer dann ausführt, wenn die Methode aufgerufen wird. Und das geht so:

```
schreibtischlampe.schalteAn();
```

Sie vermuten richtig, dass diese Zeile dieselbe Wirkung hat wie die im vorangegangenen Abschnitt verwendete einfache Zuweisung des Wertes `true` an das Attribut `schreibtischlampe.angeschaltet`. Allerdings benötigen Sie eine weitere Methode zum Ausschalten der Lampe, oder Sie müssten das weiterhin mit der Zuweisung `schreibtischlampe.angeschaltet = false` erledigen. Warum also so viel Schreibarbeit?

Weil Methoden deutlich mehr können als einfache Zuweisungen. Um das zu demonstrieren, kommen wir zurück zum Einschrauben der falschen Birne. Zunächst schreiben Sie eine Methode dafür:

```
package de.androidnewcomer.abenteuer;
public class Lampe {
   Integer leistung = 40;
   void neueBirne(Integer neueLeistung) {
      leistung = neueLeistung;
   }
};
```

Der Übersicht halber habe ich jetzt den Anschalter weggelassen. Mit der folgenden Zeile könnte unser Spieler jetzt eine andere Glühbirne einschrauben:

```
schreibtischlampe.neueBirne(8);
```

Die Methode `neueBirne()` erwartet einen `Integer`-Parameter. Als Name für diesen Parameter dient das Objekt `neueLeistung`, das beim Aufruf hier den Wert 8 erhält. Dieser Wert von `neueLeistung` wird dann dem Attribut `leistung` zugewiesen. Bei der Deklaration des Parameters steht wie immer die Klasse vor dem Variablennamen.

Der Parameter ist ein ähnliches Objekt wie ein Attribut, allerdings existiert er nur, solange die Methode abgearbeitet wird, und er kann bei jedem Aufruf einen anderen Wert aufweisen.

### Zugriffsbeschränkungen

Im Moment hindert nichts unseren Spieler daran, zum Beispiel eine 900-Watt-Birne einzuschrauben:

```
schreibtischlampe.neueBirne(900);
```

Gegen derartige Stromverschwendung sollten Sie unbedingt vorgehen. Geben Sie also der Methode `neueBirne()` die Macht, sich über den Willen des Spielers hinwegzusetzen und höchstens 40 Watt zu erlauben:

```
package de.androidnewcomer.abenteuer;
public class Lampe {
   private Integer leistung = 40;
   void neueBirne(Integer neueLeistung) {
      if(neueLeistung <= 40) {
         leistung = neueLeistung;
      }
   }
};
```

Das Schöne an Java-Code ist, dass er oft auf Anhieb verständlich ist. Übersetzen Sie einfach im Kopf, was dort in der Methode `neueBirne()` steht: Falls die gewünschte Leistung kleiner oder gleich 40 ist, weise dem Attribut Leistung die neue Leistung zu (sonst nicht).

Die Hauptrolle spielt hier das Schlüsselwort `if`. Hinter `if` steht in runden Klammern eine Bedingung, deren Wahrheitswert (vom primitiven Typ `boolean`) der Computer beim Ablauf des Programms auswertet. Trifft die Bedingung zu, führt die Java Runtime den anschließenden Codeblock aus, sonst nicht. Wie jeder Codeblock ist auch dieser in geschweifte Klammern gerahmt und der Übersicht halber eingerückt. Die Methode tut also bei dem folgenden Aufruf rein gar nichts:

```
schreibtischlampe.neueBirne(100);
```

Hingegen führt sie die Energiesparversion ohne Murren aus:

```
schreibtischlampe.neueBirne(8);
```

Es gibt noch ein Loch zu stopfen. Was hindert den Spieler daran, wie zuvor direkt die Leistung zu ändern und unsere neue Methode zu umgehen, indem er schreibt:

```
schreibtischlampe.leistung = 100;
```

Die Lösung habe ich schon oben in den Programmcode gemogelt: Das Schlüsselwort `private` vor der Deklaration des Attributs für die Leistung verhindert die direkte Zuweisung, ausgenommen durch Methoden innerhalb unserer eigenen Klasse.

Eine solche **Zugriffsbeschränkung** mag schizophren erscheinen, denn der Programmierer, dem der Quellcode der gesamten Anwendung zur Verfügung steht, könnte einfach das Wörtchen `private` löschen und wieder tun, was ihm Spaß macht.

Das Stichwort hierzu lautet: Verantwortung. Unsere Klasse beansprucht die Verantwortung für das Attribut namens `leistung` für sich. Das bedeutet zwei Dinge: Erstens kann sich jede andere Klasse darauf verlassen, dass sich jede Lampe korrekt und unabhängig vom Rest der Welt um ihre eigene Leistung kümmert. Zweitens geht jeder Programmierer, der sich über die bewusst eingebaute `private`-Einschränkung hinwegsetzt, das Risiko ein, dass die Klasse nicht mehr wie vorgesehen arbeitet. Wer einer Klasse die Kontrolle über ihre privaten Attribute entzieht, bringt im schlimmsten Fall sein ganzes Projekt in Gefahr.

### Gleich und gleicher

Java enthält einen kleinen Seitenhieb auf die Mathematik. Mathematiker und Java-Programmierer meinen nämlich nicht dasselbe, wenn sie schreiben:

```
a = b
```

In Java ist das = ein Zuweisungsoperator. Der Operator wertet aus, was auf der rechten Seite steht, und weist das Ergebnis dem links stehenden Objekt zu. Mathematiker meinen, dass a und b denselben Wert haben – von Anfang an. Das = in der Mathematik ist ein Vergleichsoperator.[1]

Den Vergleichsoperator gibt es auch in Java, um Verwechslungen zu vermeiden, sieht er allerdings etwas anders aus:

```
a == b
```

1 Um der Wahrheit die Ehre zu geben, dürfen auch in der Mathematik links und rechts vom = unterschiedliche Werte stehen. 1 + 1 = 13 ist aus Sicht der Mathematik schlicht eine falsche Aussage, anders ausgedrückt: Der Wahrheitswert der Aussage ist falsch. Trotzdem darf man diese falsche Aussage aufschreiben – man sollte bloß nicht behaupten, dass sie wahr ist, denn das wäre, Sie ahnen es: falsch.

### Gleich und gleicher

Dieser Vergleichsoperator kommt meist in `if`-Bedingungen zum Einsatz, beispielsweise so:

```
if(a == b) {
}
```

Falls die Objekte `a` und `b` den gleichen Wert haben, wird der Code in den geschweiften Klammern ausgeführt, sonst nicht.

Die Leerzeichen vor und hinter dem `==` dienen der Übersicht. Sie sind nicht notwendig, aber sinnvoll.

Java kennt eine ganze Reihe weiterer Vergleichsoperatoren: `>=`, `<=`, `<` und `>`.

Da Vergleichsoperatoren ein `boolean`-Ergebnis haben, können Sie dies prinzipiell auch Attributen oder lokalen Variablen zuweisen:

```
boolean vergleichsresultat = (a == b);
if(vergleichsresultat) ...
```

Die Klammern um `a == b` sind nicht unbedingt erforderlich, aber sie helfen, den Überblick zu behalten.

### Statische Methoden

Bei der App-Entwicklung kommt es andauernd vor, dass Zahlen in Strings stehen (zum Beispiel in Texteingabefeldern). Um mit den Werten zu arbeiten, müssen sie in Integer konvertiert werden. Das geht so:

```
Integer wert = Integer.parseString("40");
```

Dies schreibt uns die Zahl 40 in das Objekt `wert`. () Auch für den umgekehrten Weg gibt es eine sehr einfache Methode:

```
String wert = Integer.toString(40);
```

Anschließend enthält das `String`-Objekt `wert` den `String`-Wert `"40"`.

`parseString()` und `toString()` sind *statische Methoden*. Während andere Methoden nur auf ihren Objekten arbeiten, können Sie statische Methoden verwenden, indem Sie den Klassennamen vor den Punkt setzen. Daher steht hier `Integer`, ein Klassenname.

Statische Methoden funktionieren im Kontext einer Klasse, nicht eines Objekts. Folglich dürfen Sie in statischen Methoden keine Eigenschaften des Objekts verwenden. Üblicherweise setzen Sie eine statische Methode ein, wenn Sie eine allgemeine Funktionalität bereitstellen möchten. Verwenden Sie das Schlüsselwort `static`:

```
public class Lampe {
   static Lampe erzeugeLampeMit80erBirne() {
      Lampe lampe = new Lampe();
      lampe.leistung = 80;
      return lampe;
   }
}
```

Dieses Beispiel einer statischen Methode ist eine mögliche Abkürzung, um später im Code mit einer einzigen Zeile eine Lampe mit einer 80-Watt-Birne zu erzeugen:

```
Lampe lampe = Lampe.erzeugeLampeMit80erBirne();
```

### Eigene Konstruktoren

Bisher besitzt Ihre Klasse `Lampe` lediglich einen Standard-Konstruktor, denn Sie haben keinen eigenen geschrieben. Deshalb ist derzeit nur ein `new`-Aufruf möglich, der eine generische Lampe erzeugt:

```
Lampe lampe = new Lampe();
```

Sie können dem Anwender der Klasse `Lampe` aber die Möglichkeit geben, beispielsweise die Leistung der Birne schon im Konstruktor festzulegen:

```
Lampe lampe = new Lampe(60);
```

Diesen speziellen Konstruktor, der einen `int`-Parameter entgegennimmt, müssen Sie explizit implementieren:

```
public class Lampe {
   private int leistung = 80;
   public Lampe(int wert) {
      leistung = wert;
   }
}
```

Der Konstruktor ist nichts anderes als eine spezielle Methode, die beim Erzeugen des Objekts mit `new` aufgerufen wird. Der Name des Konstruktors ist dabei immer der Name der Klasse.

Sie können auch mehrere Konstruktoren anbieten, die sich durch ihre Parameterliste unterscheiden. Java erkennt dann beim Erzeugen von Objekten anhand der verwendeten Parameter, welchen Konstruktor es benutzen muss:

```
public class Lampe {
   private int leistung = 80;
   private boolean eingeschaltet = false;
   public Lampe(int wert) {
      leistung = wert;
   }

  public Lampe(int wert, boolean an) {

      leistung = wert;
      eingeschaltet = an;
   }
```

```
}
...
Lampe lampe = new Lampe(100);
Lampe angeschalteteLampe = new Lampe(40, true);
```

Den Standard-Konstruktor müssen Sie nicht explizit hinschreiben. Würden Sie es tun, sähe er auch ziemlich langweilig aus:

```
public class Lampe {
   ...
   public Lampe() {
   }
}
```

Manchmal ist es allerdings sinnvoll, im Standard-Konstruktor Programmcode auszuführen. Beispielsweise können Sie hier Vorgabewerte setzen, die Sie ansonsten Attributen bei der Deklaration zuweisen würden:

```
public class Lampe {
   private leistung;
   public Lampe() {
      leistung = 80;
   }
}
```

Der Unterschied ist meist nicht von Bedeutung, sodass Sie die kürzere Schreibweise ohne expliziten Konstruktor bevorzugen können.

### Lokale Variablen

Es gibt noch eine dritte Art von Objekten neben Attributen und Methodenparametern: lokale Objekte (meist Variablen genannt). Im Grunde haben Sie solche Objekte bereits gesehen, immer wenn Sie in irgendeiner Methode ein neues Objekt erzeugt haben:

```
public Schreibtisch erzeugeSchreibtisch() {
   Schreibtisch schreibtisch = new Schreibtisch();
   schreibtisch.setLampe( new Lampe() );
   return schreibtisch;
}
```

Hier ist `schreibtisch` ein lokales Objekt, das unter diesem Namen nur existiert, solange die Methode `erzeugeSchreibtischMitLampe()` ausgeführt wird. In den meisten Fällen dienen lokale Objekte dazu, Code zu vereinfachen oder übersichtlicher zu schreiben. Denn die obige Methode könnte man auch wie folgt schreiben:

```
public Schreibtisch erzeugeSchreibtischMitLampe() {
   return (new Schreibtisch()).setLampe(new Lampe());
}
```

Das sieht kompakter aus, so etwas geht aber spätestens dann schief, wenn Sie mehrere `set`-Methoden aufrufen wollen:

```
public Schreibtisch erzeugeSchreibtisch() {
   Schreibtisch schreibtisch = new Schreibtisch();
   schreibtisch.setLampe( new Lampe() );
   schreibtisch.setApfel( new Apfel() );
   return schreibtisch;
}
```

In diesem Beispiel ist es nicht ohne Weiteres möglich, alles in eine Zeile zu zwingen.

**Lebensdauer**

Der Schreibtisch aus dem fiktiven Abenteuerspiel ist zwar ein lokales Objekt, das unter dem Namen `schreibtisch` nur zur Verfügung steht, bis die Methode beendet ist. Allerdings wird die Java Runtime das zugehörige Objekt nicht aus dem Speicher entfernen. Denn die Methode `erzeugeSchreibtisch()` gibt das Objekt ja zurück an irgendeine andere Methode, die sicher irgendetwas mit dem Schreibtisch anzustellen gedenkt und ihn möglicherweise in einem Attribut speichert:

```
class Raum {
   private Schreibtisch schreibtisch = erzeugeSchreibtisch();
   public Schreibtisch erzeugeSchreibtisch() {
      ...
   }
}
```

Solange eine Referenz auf ein Objekt existiert, wird es vom Garbage Collector nicht weggeräumt. Erst wenn kein `Raum`-Objekt mehr vorhanden ist, ist auch der `Schreibtisch` entbehrlich.

Aber es gibt auch den anderen Fall:

```
public void pruefeLampe() {
   boolean istNacht = ermittleTageszeit();
   if(istNacht) {
      lampe.schalteAn();
   }
}
```

Dieses Beispiel verwendet die lokale Variable `istNacht` lediglich als Bedingung in der `if`-Verzweigung. Da es beim Beenden der Methode keine Referenz mehr auf `istNacht` gibt, wird der davon belegte Speicher bei nächster Gelegenheit freigegeben.

## 2.5 Daten verwalten

In jedem Abenteuerspiel gibt es so etwas wie ein Inventar, in dem der Spieler nützliche Dinge wie Tomaten und Energiesparbirnen transportieren kann. Wie lassen sich also solche Objekte einem anderen (dem Inventar) zuordnen?

### Listen

Dafür gibt es (unter anderem) die Java-Klasse `ArrayList`. `ArrayList`-Objekte verwalten eine beliebige Anzahl Objekte sowie deren Reihenfolge. Sie können Objekte hinzufügen, entfernen oder prüfen, ob sie in der Liste vorhanden sind. Bevor ich Ihnen den zugehörigen Java-Code zeige, muss ich zugeben, dass ich Ihnen eine veraltete, vereinfachte Variante unterschiebe, die Ihnen die Entwicklungsumgebung mit »so lieber nicht« quittiert. Aber es funktioniert, und die moderne, etwas kompliziertere Variante kommt noch früh genug.

```
ArrayList inventar = new ArrayList();
```

Diese Zeile wird Sie nicht überraschen, denn Sie erzeugen lediglich ein neues `ArrayList`-Objekt mit dem Namen `inventar`. Fügen Sie dem Inventar nun alles hinzu, was Ihnen einfällt:

```
Lampe schreibtischlampe = new Lampe();
inventar.add(schreibtischlampe);
Auto meinCabrio = new Auto();
inventar.add(meinCabrio);
Auto alteKarreVomNachbarn = new Auto();
inventar.add(alteKarreVomNachbarn);
```

Verflixt, jetzt haben Sie glatt Ihrem Nachbarn den Wagen gemopst. Das sollten Sie lieber wieder rückgängig machen, bevor er's merkt:

```
inventar.remove(alteKarreVomNachbarn);
```

Listen sind sehr wichtige Klassen. Sie benötigen sie in zahlreichen Fällen in Android-Apps, und es gibt eine ganze Menge Methoden, um Listen zu manipulieren. So können Sie Listen löschen, sortieren oder duplizieren. Ich werde Ihnen die betreffenden Methoden jeweils dort vorstellen, wo Sie sie erstmals brauchen. Zudem gibt es eine ganze Reihe enge Verwandte der Klasse `ArrayList`. Sie unterscheiden sich in wichtigen Details, allerdings sind die in den meisten Fällen für unsere Zwecke uninteressant. Deshalb werden Sie fast immer die gute alte `ArrayList` verwenden, es sei denn, sie bietet nicht die Funktionen, die Sie gerade benötigen.

Sehr oft möchten Sie mit jedem Objekt in einer Liste dasselbe anstellen. Beispielsweise möchten Sie berechnen, ob der Spieler den ganzen Kram, den er

durch die Gegend schleppt, überhaupt tragen kann oder ob er unter der Last zusammenbricht. Dazu muss natürlich jedes Objekt im Inventar ein Gewicht haben, das Sie mit einer Methode `getGewicht()` in Kilogramm ermitteln können:

```
class Item {
   private float gewicht;
   public float getGewicht() {
      return gewicht;
   }
}
```

Lassen Sie sich nicht durch den englischen Begriff **Item** irritieren – der hat sich international für all die Dinge eingebürgert, die Abenteurer mit sich herumschleppen. Die deutsche Alternative, »Ding«, klänge etwas profan. Sicher wird die Klasse `Item` noch weitere Methoden und Attribute besitzen, aber wir beschränken uns auf das, was wir unbedingt brauchen.

Es ist sicher eine gute Idee, Kommazahlen als Gewicht zu erlauben, daher verwende ich den primitiven Datentyp `float` als Rückgabewert der Methode `getGewicht()`.

**Arrays**

Listen wie `ArrayList` möchten mit Objekten gefüllt werden, primitive Datentypen lassen sich nicht hinzufügen. Der Unterschied mag auf den ersten Blick marginal wirken, aber Sie sollten das im Hinterkopf haben.

Oft sind Listen primitiver Typen die einfachste Lösung, und das zugehörige Sprachelement von Java heißt *Array*. Sehr oft werden Arrays verwendet, wenn sich die Länge einer Werteliste nicht ändert, denn ein einmal erzeugtes Array lässt sich nicht gern vergrößern.

Entscheidend an Arrays ist, dass Sie mit einem 0-basierten Index auf die einzelnen Elemente zugreifen können. Erzeugen Sie ein Array mit `new`:

```
String[] farben = new String[3];
farben[0] = "rot";
farben[1] = "gelb";
farben[2] = "grün";
```

Sie sehen, dass bei Arrays eine Menge eckiger Klammern im Spiel sind. Bei der Deklaration des Arrays dürfen Sie die Klammern nach Belieben hinter den gewünschten Typ (hier: `String`) schreiben oder hinter den Bezeichner:

```
String farben[] = new String[3];
```

Bei `new` geben Sie die Größe des gewünschten Arrays mit an, wobei eine Größe von 3 Elemente mit den Indizes 0, 1 und 2 erzeugt – das Element `farben[3]` existiert nicht.

Sie können einem Array feste Werte auch mit geschweiften Klammern in einer Zeile zuweisen:

```
int[] lottozahlenNaechstenSamstag = { 1, 8, 16, 34, 37, 44 };
```

Wenn Sie diese Zahlen tippen und zufälligerweise etwas gewinnen, müssen Sie mir nichts abgeben.

### Schleifen

Das Inventar eines Spielers wird eine Liste sein:

```
ArrayList inventar = new ArrayList();
```

Der Spieler hat nun irgendwelche Items eingesammelt, die mit `inventar.add(item)` in der Liste gelandet sind. Lassen Sie uns nun das Gesamtgewicht berechnen. Dazu gibt es eine einfache Kontrollstruktur, die alle Elemente in der Liste nacheinander durchgeht – die `for`-Schleife:

```
for( Item item : inventar ) {
}
```

In den Klammern hinter dem Schlüsselwort `for` stehen drei Dinge: Der Name der Klasse der Objekte im Inventar (`Item`), ein Bezeichner `item` und die Liste `inventar`.

Die `for`-Schleife können Sie etwa wie folgt übersetzen: »Für jedes Element in der Liste `inventar` durchlaufe den anschließenden Codeblock und stelle das betreffende `Item` im Objekt `item` zur Verfügung«.

Der Code in den geschweiften Klammern (den Sie noch schreiben müssen) wird also so oft ausgeführt, wie es Elemente im Inventar gibt, und darin steht ein lokales Objekt `item` zur Verfügung, das eine Referenz auf jeweils eines der Elemente ist. Wenn das Inventar leer ist, wird der Codeblock überhaupt nicht durchlaufen.

Nun können Sie die eigentliche Berechnung des Gewichts hinschreiben:

```
float gesamtgewicht = 0.0;
for( Item item : inventar ) {
   gesamtgewicht += item.getGewicht();
}
```

Für jedes der Items im Inventar wird damit die Methode `getGewicht()` aufgerufen. Das Ergebnis, also das Gewicht des jeweiligen Items, wird zur lokalen Variablen `gesamtgewicht` hinzuaddiert.

Wenn die Schleife beendet ist, enthält `gesamtgewicht` die gewünschte Summe. Sie könnten also als Nächstes schreiben:

```
if(gesamtgewicht > zulaessigesGewicht) {
   brichZusammen();
}
```

Vielleicht ist Ihnen aufgefallen, dass ich für dieses Beispiel weder `Lampe` noch `Auto` ins Inventar gepackt habe, sondern generische `Item`-Objekte. Der Grund ist nicht, dass sowieso niemand ein Auto tragen könnte, sondern dass Sie die Methode `getGewicht()` nicht in jede einzelne dieser Klassen einbauen möchten. Das wäre bei zahlreichen unterschiedlichen Inventarobjekten ein fürchterlicher Mehraufwand. Daher wäre es praktisch, wenn Sie Methoden wie `getGewicht()`, die eine Gemeinsamkeit verschiedener Klassen ist, nur einmal schreiben müssten.

Dabei hilft die wohl wichtigste Eigenschaft objektorientierter Sprachen wie Java: die **Vererbung**.

## 2.6 Vererbung

Menschen und Affen haben dieselben Vorfahren, haben daher ähnliche Gene geerbt und besitzen z. B. dieselbe Anzahl Gliedmaßen. Dieses Prinzip ist aus der objektorientierten Programmierung nicht wegzudenken. Dahinter steckt der Wunsch der Programmierer nach mehr Freizeit, denn das Prinzip der Vererbung spart jede Menge Entwicklungszeit.

### Basisklassen

Kehren wir zu unserem Gemüsetransporter zurück und stellen die alles entscheidende Frage: Ist es nicht ungesund, sich ausschließlich von Tomaten zu ernähren?

Fügen Sie dem Speiseplan also Blumenkohl und Spinat hinzu. Die für einen Gemüsehändler interessanteste Gemeinsamkeit dürfte der Kilopreis sein. Da ich Ihnen noch nichts über Vererbung in Java erzählt habe, könnten Sie auf die Idee kommen, folgende Klassen anzulegen:

```
public class Tomate {
  public float kilopreis;
}
public class Blumenkohl {
  public float kilopreis;
}
public class Spinat {
  public float kilopreis;
}
```

```
...
Spinat spinat = new Spinat();
spinat.kilopreis = 0,89;
Tomate tomate = new Tomate();
tomate.kilopreis = 0,99;
Blumenkohl blumenkohl = new Blumenkohl();
blumenkohl.kilopreis = 0,79;
```

Jede der drei Klassen hat genau ein Attribut, und zwar ein `float` namens `kilopreis`, das Euro- und Cent-Beträge mit Dezimalkomma aufnehmen kann.

Offensichtlich hat das gesamte Gemüse eine Gemeinsamkeit. Nehmen wir das Konzept des Bestellformulars hinzu, das ich zu Beginn dieses Kapitels für ein Auto verwendet habe: Die Bestellformulare für Tomate, Blumenkohl und Spinat haben alle ein gemeinsames Attribut zum Eintragen des Kilopreises. Folglich ist es viel effizienter, zunächst ein gemeinsames Formular für Gemüse allgemein zu verwenden (dass das in der Realität nicht geschieht, sagt einiges über Bürokratie, finden Sie nicht?). Eine solche abstrakte Verallgemeinerung nennt man **Basisklasse**. Für unser Gemüse sieht sie wie folgt aus:

```
public abstract class Gemuese {
  public float kilopreis;
}
```

Das Schlüsselwort `abstract` sorgt dafür, dass niemand ein Gemüse-Objekt erstellen kann, denn was sollte das auch sein? Der folgende Befehl ist also unzulässig:

```
Gemuese abstraktesGemuese = new Gemuese();
```

Abstraktes Gemüse kann eben niemand transportieren oder gar essen. Das geht nur mit den **von der Basisklasse abgeleiteten Klassen**:

```
public class Tomate extends Gemuese {
}
public class Blumenkohl extends Gemuese{
}
public class Spinat extends Gemuese {
}
```

Das Schlüsselwort `extends` sorgt dafür, dass die jeweilige Klasse alle Methoden und Attribute der angegebenen Basisklasse **erbt**. Daher funktioniert jetzt der obige Code immer noch:

```
Spinat spinat = new Spinat();
spinat.kilopreis = 0,89;
Tomate tomate = new Tomate();
tomate.kilopreis = 0,99;
```

```
Blumenkohl blumenkohl = new Blumenkohl();
blumenkohl.kilopreis = 0,79;
```

**Refactoring**

Sie sehen am Beispiel des Gemüses, dass es möglich ist, die *Klassenhierarchie* nachträglich zu verändern. Vorher steckten die `kilopreis`-Attribute in jeder einzelnen Klasse, nun nur noch in der Basisklasse – aber der Code zum Erzeugen des konkreten Gemüses ist gleich geblieben, genau wie die Zuweisung der Preise.

Einen solchen nachträglichen Eingriff nennt man *Refactoring*, und dieser Prozess war früher von aufwendiger und fehleranfälliger Handarbeit an vielen Dateien geprägt – bis die integrierten Entwicklungsumgebungen wie Eclipse auf den Markt kamen, die dazu gemacht sind, dem Entwickler möglichst viel von dieser manuellen Arbeit zu ersparen.

Die Basisklasse ist für ihre Kinder wie ein offenes Buch: Jede abgeleitete Klasse, die von der Basisklasse erbt, kann auf deren Attribute und Methoden zugreifen. Wirklich auf *alle*? Kommt drauf an.

Möglicherweise hat die Basisklasse gewisse Intimitäten zu verbergen. Dann kann sie Methoden oder Attribute als `private` deklarieren:

```
public abstract class Gemuese {
  private float vitamingehalt = 0.0;
}
```

Mithilfe des Schlüsselworts `private` könnte die Gemüse-Klasse beispielsweise spannende Berechnungen über den Vitamingehalt pro Kilopreis anstellen, ohne dass die abgeleiteten Klassen oder gar fremde Klassen davon erfahren. Diese **Verkapselung** ist nicht zuletzt eine Sicherheitsvorkehrung. Eine Klasse kann nur dann einwandfreie Arbeit garantieren, wenn ihr niemand unvorhergesehen im Innenleben herumfummelt.

Es gibt auch den Mittelweg:

```
public abstract class Gemuese {
   protected String farbe;
   public String getFarbe() { return farbe; }
}
```

Eine Farbe zu haben ist unbestritten eine gemeinsame Eigenschaft allen Gemüses. Der `protected`-Modifier erlaubt es den abgeleiteten Klassen wie Tomate und Spinat, ihre Farbe zu setzen – aber fremde Klassen dürfen nur die Methode `getFarbe()` verwenden, um die Farbe auszulesen. Ein direkter Zugriff auf das Attribut `farbe` ist von außerhalb nicht erlaubt – egal, ob lesend oder schreibend:

```
tomate.farbe = "blau"; // Fehler
```

Wie Ihnen vielleicht aufgefallen ist, treibe ich hier das furchtbare Denglisch auf die Spitze, indem ich die Methode `getFarbe()` genannt habe. Der Grund: Solche Methoden, die nichts anderes tun, als den Wert eines nicht direkt zugänglichen Attributs auszulesen und fremden Klassen verfügbar zu machen, erhalten vereinbarungsgemäß immer den Namen `get` plus den Namen des Attributs. Die schreibende Variante heißt immer `set` plus Attributname, und die leicht zu merkenden Fachbegriffe für diese sehr häufig auftauchenden Methoden lauten **Getter** und **Setter**.

## Polymorphie

Was passiert, wenn sowohl Elternklasse als auch Kind eine Methode gleichen Namens implementieren? Sehen Sie sich das folgende Beispiel an:

```
public class Tomate extends Gemuese {
   private boolean istReif;
   @Override
   public String getFarbe() {
      if(istReif) {
      return "rot";
   } else {
      return "grün";
   }
}
```

Die Tomaten-Methode `getFarbe()` ignoriert völlig das `farbe`-Attribut der Elternklasse `Gemuese`! Stattdessen gibt sie eine Farbe zurück, die von einem privaten Attribut `istReif` abhängt. Darf die das?

Sie darf. Diese Art der Spezialisierung in abgeleiteten Klassen nennt man **Überschreiben** (Override). Die Methode `Tomate.getFarbe()` überschreibt die Methode `Gemuese.getFarbe()`. Als Hinweis für den Compiler dient die davor gesetzte Annotation `@Override`.

Und jetzt kommt der entscheidende Clou an der Sache:

```
Tomate tomate = new Tomate();
zeigeAn( tomate.getFarbe() );
```

Stellen Sie sich vor, dass die Methode `zeigeAn()` den Parameter irgendwo auf dem Bildschirm anzeigt. Natürlich wird je nach Reifegrad entweder `rot` oder `grün` angezeigt, denn offensichtlich wird die `getFarbe()`-Methode der Tomate verwendet. Aber jetzt sehen Sie sich das hier an:

```
Gemuese gemuese = new Tomate();
zeigeAn( gemuese.getFarbe() );
```

An diesen zwei Zeilen gibt es zwei wichtige Dinge zu lernen.

Erstens ist jedes Objekt der Klasse `Tomate` auch ein Objekt der Klasse `Gemuese`. Sie sind kompatibel, deshalb ist die Zuweisung erlaubt.

Und zweitens merkt das Objekt `gemuese`, aus welcher Klasse es erzeugt wurde: Es ist immer noch eine `Tomate`! Deshalb führt der Aufruf von `gemuese.getFarbe()` auch hier zur Tomaten-Methode, genau wie im Codebeispiel davor. Dieses wichtige Verhalten objektorientierter Programmiersprachen nennt man **Polymorphie**.

Kehren wir zum Schluss zum Beispiel des Inventars zurück und übertragen es auf Gemüse:

```
float gesamtgewicht = 0.0;
for( Gemuese gemuese : inventar ) {
   gesamtgewicht += gemuese.getGewicht();
}
```

Jetzt kann jede von `Gemuese` abgeleitete Klasse eine völlig unterschiedliche Methode `getGewicht()` besitzen. Tut sie es nicht, wird die einfache Version `getGewicht()` in der Basisklasse verwendet.

Sie sehen, dass Polymorphie sehr elegante und effiziente Programmierung ermöglicht. Dieses Konzept wird Ihnen im Laufe Ihrer gerade beginnenden Karriere als Android-Entwickler noch häufig über den Weg laufen.

Und jetzt wird es langsam Zeit für die erste App, finden Sie nicht auch?

*»Ja, ich habe die Lizenzbedingungen gelesen«*
*(Häufigste Lüge des 21. Jahrhunderts)*

# 3 Vorbereitungen

Alle Theorie ist grau, rote Tomaten hin oder her. Sobald Sie Ihren Rechner vorbereitet haben, können Sie mit der ersten App beginnen. Also ran an die Installation!

## 3.1 Was brauche ich, um zu beginnen?

Wenn Sie einen PC mit Internet-Anschluss haben, können Sie sofort loslegen. Allerdings sollte Ihr PC möglichst 2 Gigabyte RAM besitzen und eine mit etwa 2 Ghz getaktete CPU. Weniger geht auch, allerdings können dann hier und da Wartezeiten auftreten.

Sorgen Sie dafür, dass auf Ihrer Festplatte ein paar Gigabyte frei sind. Speicherplatz ist heutzutage so billig, dass ich keine Rücksicht darauf nehmen werde, wie viel davon Sie auf dem Entwicklungsrechner verbrauchen. Nach und nach werden sich eine Menge kleiner Dateien ansammeln, und niemand möchte Zeit damit verschwenden, sie einzeln aufzuräumen.

Um selbst entwickelte Apps auszuprobieren, benötigen Sie ein Android-Gerät. Prinzipiell kommt jedes Tablet oder Smartphone infrage, auf dem Android 1.6 oder neuer läuft (ältere Versionen berücksichtigen wir nicht, obwohl die meisten Beispiele damit auch funktionieren). Obligatorisch ist ein USB-Kabel, damit Ihr PC sich mit dem Androiden unterhalten kann. Glücklicherweise liefern die meisten Hersteller solche Kabel mit. Falls nicht, ist das nicht weiter tragisch: Mit wenigen Ausnahmen verfügen Android-Geräte über standardisierte Micro- oder Mini-USB-Stecker. Passende Kabel bekommen Sie preiswert in vielen Geschäften oder in Online-Shops.

**USB-Treiber**

Unter Windows müssen Sie möglicherweise einen USB-Treiber Ihres Handy-Herstellers installieren, damit die Kommunikation klappt.

Für Google-Telefone wie das Nexus One oder Nexus S finden Sie den Treiber hier:

*http://developer.android.com/sdk/win-usb.html*

Bei anderen Herstellern hilft ein Blick auf deren Homepage. Freundlicherweise hat Google eine Liste mit passenden Links zusammengestellt:

*http://developer.android.com/sdk/oem-usb.html*

**Abbildung 3.1** Suchen Sie sich aus, mit welchem Android-Gerät Sie Ihre ersten Apps basteln: Hier sehen Sie ein Tablet, ein altes und ein neueres Handy.

Sie benötigen nicht das neueste und schnellste Android-Gerät. Zur Entwicklung für mobile Endgeräte gehört nämlich eine wichtige Erkenntnis: Selbst moderne Smartphones sind nicht so gut motorisiert wie ein Desktop-PC oder Notebook. Während ein PC über praktisch unendlich viel Speicher und Plattenplatz verfügt, gilt das für ein Handy nicht. Dort sind fette und langsame Applikationen sogar ein echtes Ärgernis – und werden schnell wieder deinstalliert. Wenn Sie eine App entwickeln würden, die auf einem schnellen Gerät »einigermaßen« läuft, wird sie auf einem älteren nicht zu ertragen sein – und schneller gelöscht, als Sie »aber auf meinem ...« sagen können. Daher werden wir von Anfang an darauf achten, möglichst schlanke und schnelle Anwendungen zu schreiben.

Notfalls können Sie auch ohne Android-Gerät loslegen. Es existiert ein Emulator, den Sie auf dem PC oder Mac laufen lassen können. Er simuliert ein Android-System nach Wahl, läuft allerdings auf untermotorisierten PCs je nach Anwendung

quälend langsam. Da es für die Qualität einer App entscheidend ist, wie sie sich auf dem Handy bedienen lässt, ist der Emulator immer zweite Wahl bei der Entwicklung.

Für die entspannte Arbeit empfehle ich Ihnen darüber hinaus eine Kaffeemaschine oder einen Wasserkocher mitsamt opulentem Schwarzteevorrat. Übertriebene Taktfrequenzen sind dabei verzichtbar, aber Sie werden staunen, wie hilfreich diese Geräte sind.

Bei der Entwicklung wird uns ein Haufen Software unterstützen. Zum Glück wird hinsichtlich der Betriebssystemvoraussetzungen niemand ausgeschlossen: Alles läuft unter Windows, Linux und Mac OS und sieht auf allen Systemen weitgehend identisch aus.

Sie finden alle Pakete auf der Buch-DVD im Ordner *software*. Etwaige neuere Versionen können Sie aber auch kostenlos im Internet herunterladen. Da es sich zum Teil um erhebliche Datenmengen handelt, sollten Sie für das Herunterladen etwas Zeit einplanen. Alternativ finden Sie fast jederzeit in Zeitschriftenläden eine Ausgabe irgendeines Magazins, dem eine CD oder DVD mit den nötigen Programmen beigelegt ist.

Als da wären:

| Softwarepaket | Link |
|---|---|
| Java Development Kit (JDK 6u27) | *http://www.oracle.com/technetwork/java/javase/downloads/* |
| Eclipse IDE for Java Developers 3.7 (Indigo) | *http://www.eclipse.org/downloads/* |
| Android SDK R12 oder neuer | *http://developer.android.com/sdk* |

**Tabelle 3.1** Benötigte Softwarepakete

## 3.2 JDK installieren

Das Java Development Kit **(**JDK**)** ist das erste Paket, das Sie installieren müssen. Es umfasst nicht nur die Laufzeitumgebung von Java (**Java Runtime Environment**, JRE), sondern auch den Compiler, ohne den Java-Code nichts anderes ist als etwas sonderbare Textdateien. Erst der Compiler erzeugt daraus ausführbare Programme.

**Was ist noch im JDK?**

Das JDK enthält nicht nur den Compiler. Mit dabei sind auch eine ganze Reihe Programmbibliotheken, eine einfache Datenbank namens Derby, Demos und Beispielcode. Sogar ein sehr großer Teil des Programmcodes der Java-Standardbibliothek befindet sich mit im Paket. Um das meiste davon müssen Sie sich nicht kümmern, es wartet einfach geduldig auf der Festplatte, bis Sie es benötigen.

Und das geschieht manchmal schneller, als man denkt.

Für Apple-Rechner mit Mac OS werden Sie ein JDK bei Oracle vergeblich suchen – Sie erhalten es direkt bei Apple. Installieren Sie einfach die **Developer Tools** über die Apple-Website.

Unter Linux empfiehlt sich die Installation über die eingebaute Paketverwaltung (z. B. Ubuntu Software Center).

Wenn Sie das JDK für Ihr Betriebssystem von der DVD oder aus dem Internet installiert haben, werfen Sie einen Blick auf die mitgelieferten Demos. Wenn die laufen, wissen Sie, dass Sie alles richtig gemacht haben. Außerdem verschaffen Sie sich so leicht einen Überblick über die Fähigkeiten von Java.

Suchen Sie dazu im installierten JDK das Verzeichnis *demo/jfc/Java2D*. Darin befindet sich eine Datei *Java2Demo.html*, auf die Sie doppelklicken. Daraufhin demonstriert Ihr Browser Ihnen die Grafikfähigkeiten von Java. Wenn Sie sich für den Inhalt des JDK im Detail interessieren, öffnen Sie die Datei *README.html* im JDK-Verzeichnis im Browser.

Rein theoretisch können Sie Android-Apps auch ohne Entwicklungsumgebung erstellen. Es genügen ein Editor wie Notepad und ein Kommandozeilenfenster. Mir ist allerdings kein Entwickler bekannt, der auf diese Weise arbeitet, denn der Aufwand und die Fehleranfälligkeit sind viel zu groß.

Vielleicht erinnern Sie sich an die Achtziger, als Computerfreaks stunden- oder tagelang Zahlenkolonnen aus Zeitschriften abgetippt haben, weil es noch keine effizientere, erschwingliche Art der Datenübertragung gab. Seinerzeit enthielten abgedruckte Listings extra generierte Prüfsummen, die Tippfehler erkennen und sogar ungefähr lokalisieren konnten.

Android-Entwicklung ohne Entwicklungsumgebung ähnelt der Tipparbeit damaliger Zeiten, bloß ohne die Prüfsummen.

**App Inventor**

Eine etwas angenehmere Option ist der App Inventor (*http://appinventor.googlelabs.com/about/*, Abbildung 3.2). Das ist ein grafisches Entwicklungssystem, das einem Baukasten ähnelt, mit dem Sie ohne Programmcode Apps zusammenschrauben können. Zwar erscheint das verlockend, allerdings kommt man trotzdem nicht um ein tiefes Verständnis des Android-Systems herum. Schließlich stößt man früher oder später an die Grenzen des App Inventors – also warum nicht gleich den Weg der Profis wählen?

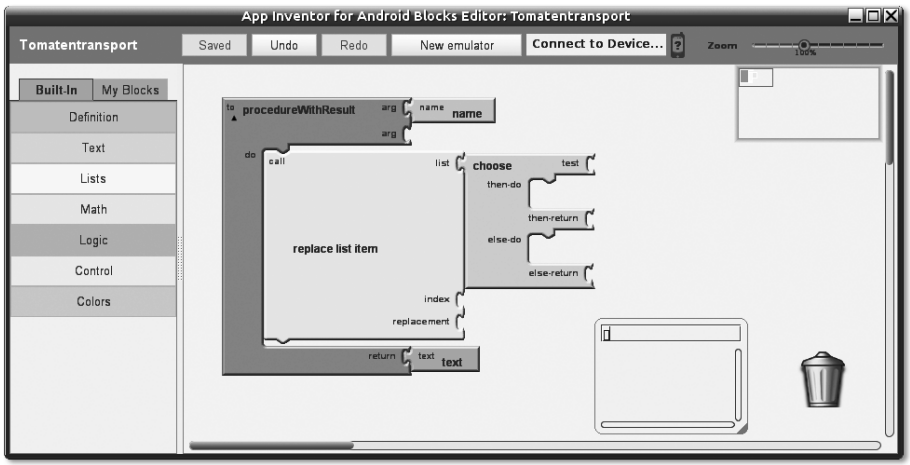



**Abbildung 3.2** Der App Inventor ist perfekt für Fans von Puzzlespielen, aber ernsthaft programmieren möchte man damit nicht.

Jetzt ist der Zeitpunkt gekommen, um die mitgelieferte DVD einzulegen und Eclipse zu installieren. Da Eclipse in Java geschrieben ist, funktioniert das erst, wenn Sie das JDK installiert haben – achten Sie also auf die richtige Reihenfolge.

## 3.3 Eclipse installieren

Eclipse ist die wohl verbreitetste Entwicklungsumgebung für Java-Zwecke. Gleichzeitig ist es kostenlos und Open Source. Während Ihr Computer sich mit dem Kopieren der umfangreichen Eclipse-Dateien von der Buch-DVD auf Ihre Festplatte abmüht (ein »richtiges« Setup gibt es nicht), erkläre ich Ihnen, was dieses Wunderwerk der Softwaretechnik alles leistet.

Eine Entwicklungsumgebung vereint alle Funktionen, die Sie benötigen, in einem Fenster und unterstützt Sie mit praktischen Hilfen beim Programmieren.

Eclipse ist modular angelegt, sodass Sie je nach Bedarf Plugins hinzufügen können. Es gibt sogar Plugins für andere Programmiersprachen, uns interessiert aber vor allem eines: das **Android Development Tool**, kurz ADT**.** Es wird von Google selbst angeboten und erweitert Eclipse um eine ganze Reihe wichtiger Funktionen, zum Beispiel:

- Basteln von App-Screens
- Unterstützung beim Editieren spezieller Android-Dateien
- Erstellen von App-Paketen (kurz **APKs**)
- Anlegen von Handy-Emulatoren
- Installieren und Starten von Apps auf Handy-Emulatoren und anderen Android-Geräten

Inzwischen dürfte Eclipse auf Ihrem Rechner angekommen sein. Damit Sie nicht immer erst in Ihr Eclipse-Verzeichnis wechseln müssen, legen Sie sich am besten eine Start-Verknüpfung auf den Desktop oder in die Schnellstartleiste. Unter Windows starten Sie Eclipse mit der `eclipse.bat`, unter Linux die ausführbare Datei `eclipse`. Auf dem Mac doppelklicken Sie auf das Eclipse-Icon, nachdem Sie das ausgepackte Archiv in den Programme-Ordner geschoben haben.

Beim ersten Start möchte Eclipse, dass Sie einen Workspace anlegen. Das ist ein Verzeichnis, in dem künftig alle Ihre Projekte gespeichert werden. Sie können den Vorschlag von Eclipse annehmen und das Häkchen setzen, damit Eclipse nicht bei jedem Start dieselbe Frage stellt, und weitermachen (Abbildung 3.3).

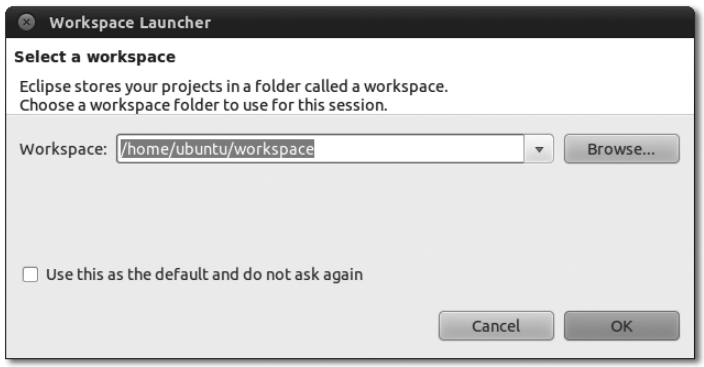



**Abbildung 3.3** Der Workspace Launcher

Eclipse begrüßt Sie mit einem Willkommensbildschirm (Abbildung 3.4), der vor allem eins tut: Er versteckt die komplexe Benutzeroberfläche. Schließen Sie den Willkommensschirm mit dem Kreuzchen oben im Anfasser, denn er führt uns im Moment nicht weiter. Anschließend erwartet Sie eine ziemlich leere Ansicht mit

einer großen Menge mysteriöser Buttons. Bitte drücken Sie jetzt keinen davon, sondern lassen Sie uns der Reihe nach vorgehen.

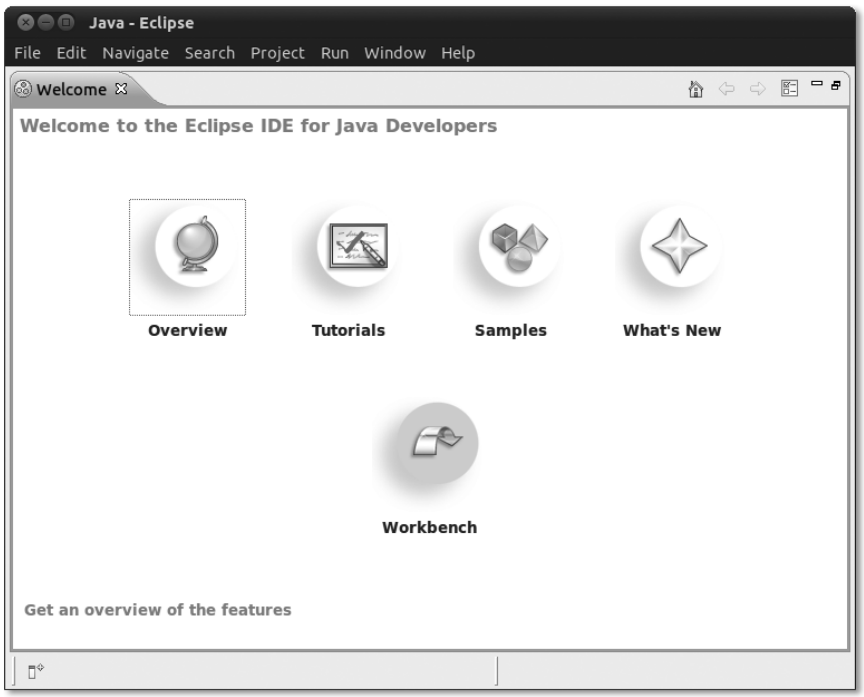



**Abbildung 3.4** Willkommen! Nie wieder wird Eclipse so übersichtlich sein wie jetzt. Auf in den Kampf ...

Die Screenshots in diesem Buch stammen übrigens zum großen Teil von einem Ubuntu **Linux 11.04.** Andere Betriebssysteme mögen die Bedienelemente optisch anders darstellen, im Großen und Ganzen sieht aber alles gleich aus und befindet sich am gleichen Ort.

## 3.4 Tour durch Eclipse

Maximieren Sie das Eclipse-Fenster, denn Platz können Sie gar nicht genug haben. Das Hauptfenster von Eclipse ist in unterschiedliche Bereiche aufgeteilt, die sich nach Wunsch verschieben, einklappen, verkleinern und vergrößern lassen. Unterschiedliche Anordnungen, sogenannte **Perspectives,** lassen sich speichern und wieder abrufen. Nach dem ersten Start befinden Sie sich in der Java-Perspective. Sie erkennen das an dem gedrückten Java-Button in der Leiste rechts oben (Abbildung 3.5).

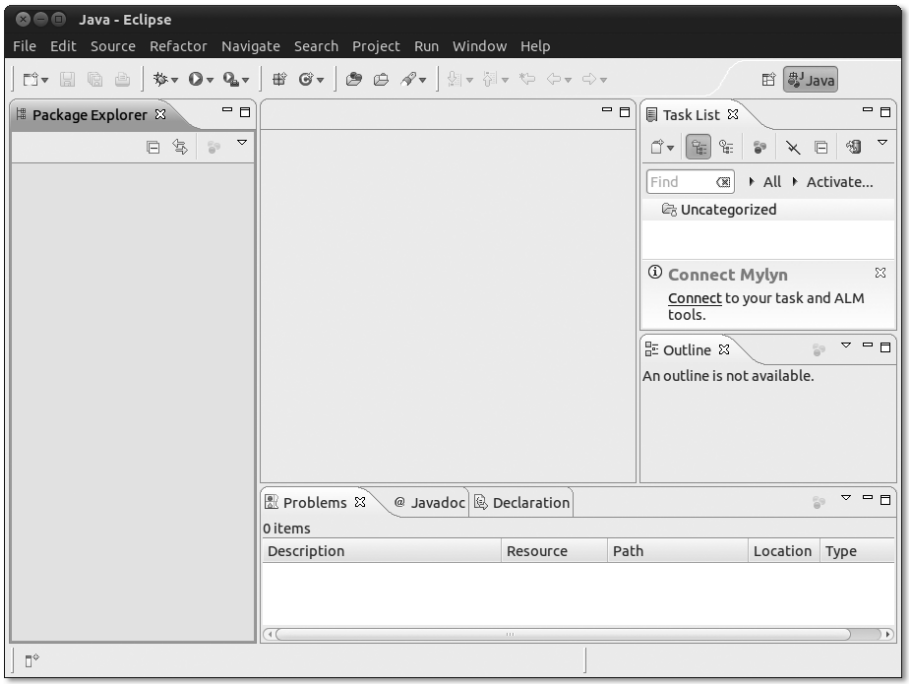



**Abbildung 3.5** Die unberührte Java-Perspective

Die verschiedenen Bereiche des Eclipse-Fensters können abhängig von der aktiven Perspektive völlig unterschiedliche Inhalte zeigen. Jeder Bereich kann noch dazu mehrere Registerkarten enthalten, von denen jeweils einer aktiv ist. Im Beispiel sehen Sie links den Package Explorer, in dem später in einer Baumstruktur unsere Projekte und Dateien zu sehen sein werden.

In der Mitte werden Sie Programmdateien bearbeiten, unten wartet mit der bedrohlichen Überschrift PROBLEMS eine Ansicht auf Sie, die derzeit glücklicherweise leer ist. Rechts sind gleich zwei weitere Bereiche eingeblendet: Einer zeigt eine Task List, die Sie vorläufig durch Klick auf das Kreuzchen schließen. Dann bleibt noch ein Bereich übrig, die OUTLINE, die Sie sich auf die gleiche Weise vom Hals schaffen.

Jeder Bereich verfügt über zwei kleine Icons oben rechts: einen schmalen, horizontalen Balken und ein weißes Rechteck rechts daneben. Klicken Sie auf den weißen Balken des Package Explorers, um ihn in eine schmale Leiste zu verwandeln. Dabei erscheint ein kleines Icon zum Wiederherstellen des vorherigen Zustands (RESTORE). Das Rechteck schließlich maximiert den Bereich, und auch das lässt sich leicht rückgängig machen – einfach noch mal klicken. In der Praxis werden Sie möglicherweise selten Gebrauch von diesen Funktionen machen,

aber wenn Sie damit vertraut sind, erschrecken Sie wenigstens nicht, wenn Sie versehentlich auf einen dieser kleinen Buttons drücken und Eclipse plötzlich anders aussieht als vorher.

Lassen Sie uns, bevor Sie diesen frischen, aufgeräumten Zustand von Eclipse ein für allemal beenden, kurz über die Menü- und Icon-Leiste sprechen, die Sie am oberen Rand des Fensters finden (beim Mac hängt das Menü wie üblich am oberen Bildschirmrand). Über das Menü sind eine Unmenge Funktionen erreichbar, die ich jetzt nicht im Einzelnen durchgehen werde. Ich werde Ihnen immer die jeweils nötigen Bedienschritte erklären, die für das Projekt von Bedeutung sind. Zunächst empfehle ich Ihnen, keine Menüpunkte unbedacht auszuprobieren, denn selbst wenn die meisten keine schlimmen Folgen haben, könnte es passieren, dass Eclipse Ihnen Fragen stellt, die Sie nicht beantworten können. Bei einer komplexen Software wie Eclipse ist die Gefahr, vom Wesentlichen abgelenkt zu werden, recht hoch. Sie könnten beispielsweise Stunden in den Tutorials verbringen. Wenn Sie die Zeit haben, werfen Sie ruhig einen Blick darauf (Menüpunkt HELP • WELCOME, Abbildung 3.6).

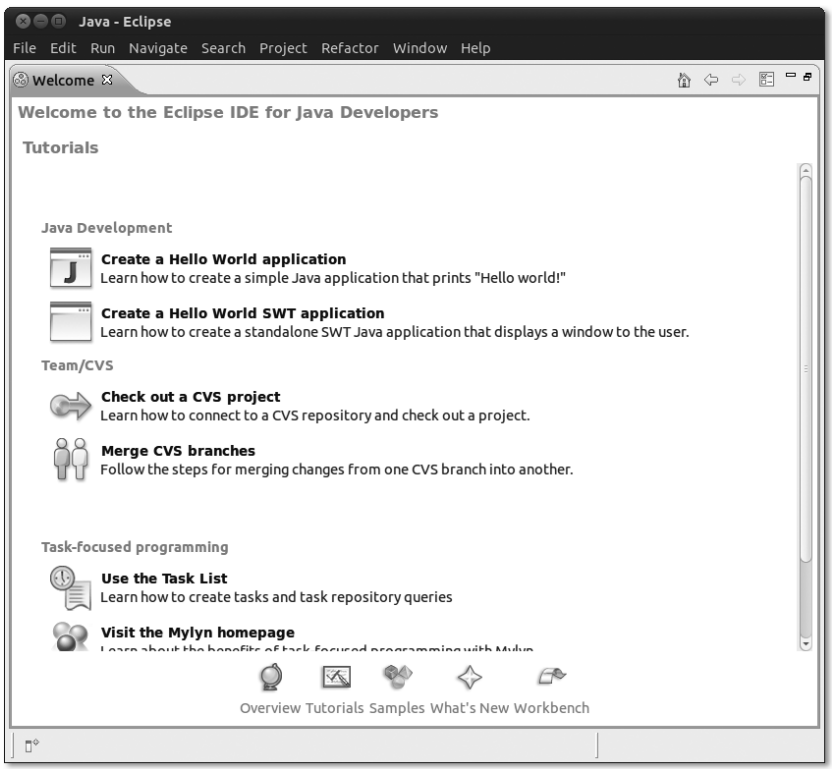



**Abbildung 3.6** Eclipse bietet eine Reihe von Tutorials für Leute, die es nicht so eilig haben wie wir.

Die Kurzfassung bekommen Sie jetzt von mir. Mich können Sie zwar nicht mit der Maus wegklicken, dafür spreche ich im Gegensatz zum Eclipse-eigenen Tutorial aber Deutsch.

Die Icon-Leiste enthält eine Untermenge der Menüpunkte für den schnelleren Zugriff. Natürlich können Sie die Icon-Leiste nach persönlichen Vorlieben anpassen, wie Sie es von Word oder anderen Programmen kennen. Ehrlich gesagt, werden Sie sich aber sehr schnell die Tastenkombinationen für die häufigsten Bedienschritte angewöhnen und die Icons gar nicht brauchen.

## 3.5 Android Development Tool installieren

In der Standard-Version kann Eclipse zwar Java-Programme erzeugen, aber hat nicht den geringsten Schimmer, was Android ist. Deshalb installieren Sie jetzt das **Android Development Tool**. Das ADT ist ein Eclipse-Plugin, das mit wenigen Mausklicks jene Funktionalität zu Eclipse hinzufügt, die Sie brauchen.

Leider steht das ADT nicht auf der Buch-DVD zur Verfügung, aber Sie können es im Handumdrehen aus dem Netz laden und ohne Umweg in Eclipse verewigen.

Wählen Sie im Menü HELP • INSTALL NEW SOFTWARE aus. Eclipse installiert neue Software nur von bekannten Software Sites. Die Android-Website gehört nicht dazu, sodass Sie sie hinzufügen müssen. Klicken Sie auf ADD..., und geben Sie die Adresse von Googles Download-Bereich an (Abbildung 3.7).

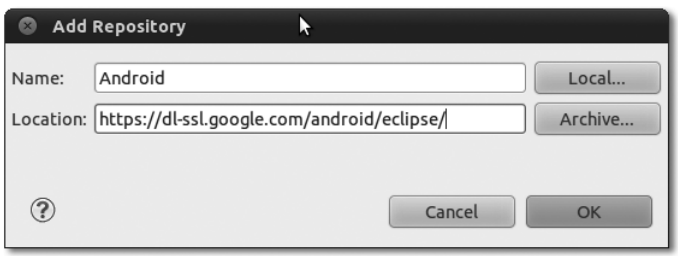



**Abbildung 3.7** Googles Repository ist die Quelle aller Android-Tools für Eclipse.

Bestätigen Sie den Dialog, und nach einer Weile präsentiert Eclipse Ihnen eine Liste der Developer Tools, die Google zur Verfügung stellt (Abbildung 3.8).

Setzen Sie die Häkchen wie in der Abbildung, und klicken Sie auf NEXT. Ihr Eclipse muss darüber eine ganze Weile nachdenken. Währenddessen lassen Sie mich darauf hinweisen, dass Sie neben den eigentlichen ADT gerade auch drei weitere Pakete installieren, nämlich das DDMS **(Dalvik Debug Monitor Server)**,

den **Hierarchy Viewer** und **Traceview**. Diese Zusätze sind nicht obligatorisch, und ich werde sie in diesem Buch nicht verwenden. Aber wenn Sie weiter mit Android arbeiten, können Sie die Pakete gut gebrauchen.

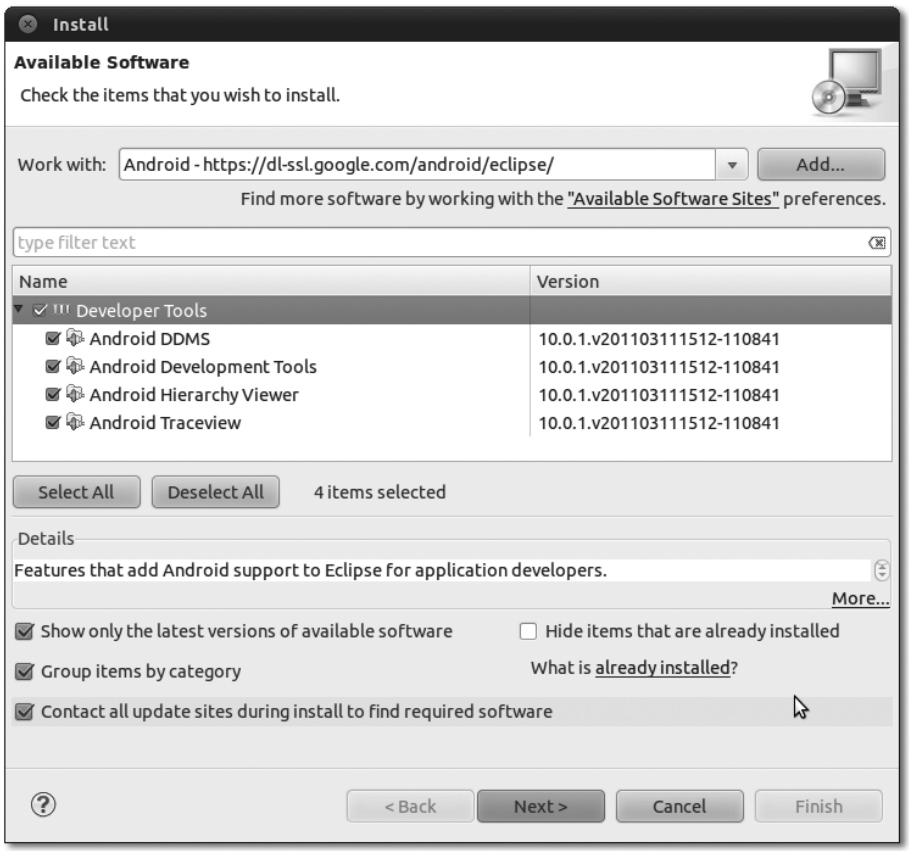



**Abbildung 3.8** Aus dem Android Repository installieren Sie alles, was Sie kriegen können.

Inzwischen hat Eclipse Sie gebeten, einige Open-Source-Lizenzen zu akzeptieren, und mit der eigentlichen Installation begonnen. Da eine ganze Menge Daten heruntergeladen wird, kann die Angelegenheit eine Weile dauern. Anschließend möchte Eclipse neu gestartet werden. Nachdem Sie das getan haben, müssen Sie ganz genau hinschauen – *noch* genauer –, und dann sehen Sie den kleinen hellgrünen Roboter, besser gesagt: seinen Kopf, der hinter einem grauen Rechteck hervorschaut.

Bevor Sie das Android-Icon anklicken, müssen Sie ihm den Installationsort des Android SDK verraten. Wählen Sie im Menü Window • Preferences, und klicken Sie links im Baum Android an. Rechts erscheint ein leeres Textfeld, in das

Sie den Pfad zu Ihrem installierten Android SDK eintragen müssen. Darum werden Sie sich als Nächstes kümmern.

## 3.6 Android SDK installieren

Ohne das Android SDK geht nichts. SDK steht für **Software Development Kit**. Dahinter verbirgt sich ein Bündel Programme und Dateien, die die Entwicklung von Android-Apps ermöglichen.

Kopieren Sie das Android SDK von der Buch-DVD, oder laden Sie das Android SDK für Ihr Betriebssystem herunter: *http://developer.android.com/sdk*.

Das SDK enthält unter anderem:

- den Emulator für Android-Geräte
- den apkbuilder, der installierbare App-Pakete bauen kann

Die Windows-Version installiert sich mit einem üblichen Setup, unter Mac OS X oder Linux müssen Sie das Archiv an einem Ort Ihrer Wahl entpacken. Tragen Sie diesen Pfad in den Android-Preferences von Eclipse ein (Abbildung 3.9).

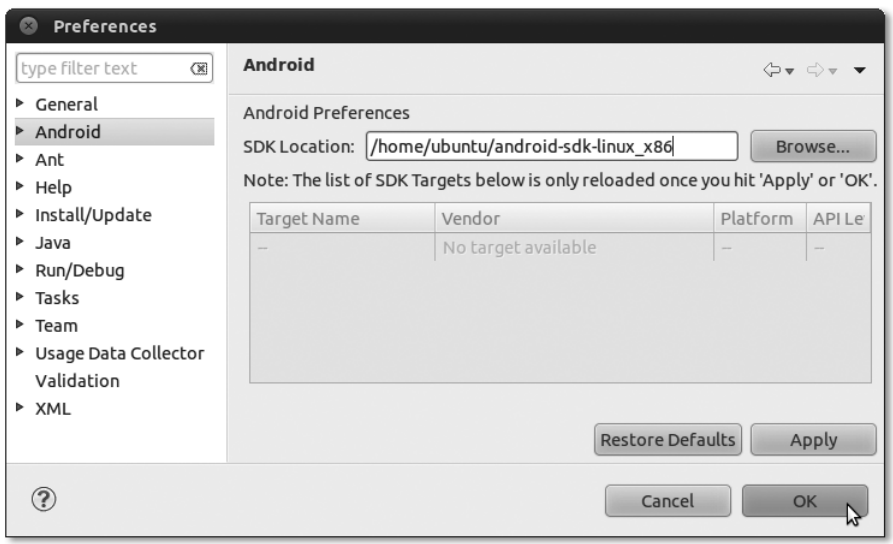



**Abbildung 3.9** Tragen Sie den Pfad zum SDK in den ADT-Einstellungen ein.

Sie erhalten anschließend den Hinweis, dass Sie noch keine SDK Platform Tools installiert haben. Dabei handelt es sich um ein Paket mit Hilfsprogrammen, ohne die Sie nicht weit kommen. Wir werden uns als Nächstes darum kümmern.

## 3.7 SDK Tools installieren

Das Android Developer Tools Plugin kennt jetzt den Aufenthaltsort Ihrer Kopie des Android SDK. Ihnen fehlen noch die SDK Tools, die Sie direkt aus dem Internet mithilfe des ADT installieren.

Endlich dürfen Sie das Android-Icon in der Symbolleiste von Eclipse anklicken. Es öffnet sich der **Android SDK and AVD Manager (**AVD **=** Android Virtual Device**)**. Wählen Sie links die Sektion Available Packages aus, klappen Sie das Android Repository auf, und wählen Sie die folgenden Komponenten zur Installation aus (Abbildung 3.10):

- Android SDK Platform-tools (allgemeine Hilfsprogramme)
- SDK Platform Android 2.2, API 8 (Android-Version 2.2)
- Samples for SDK API 8 (Beispielcode)

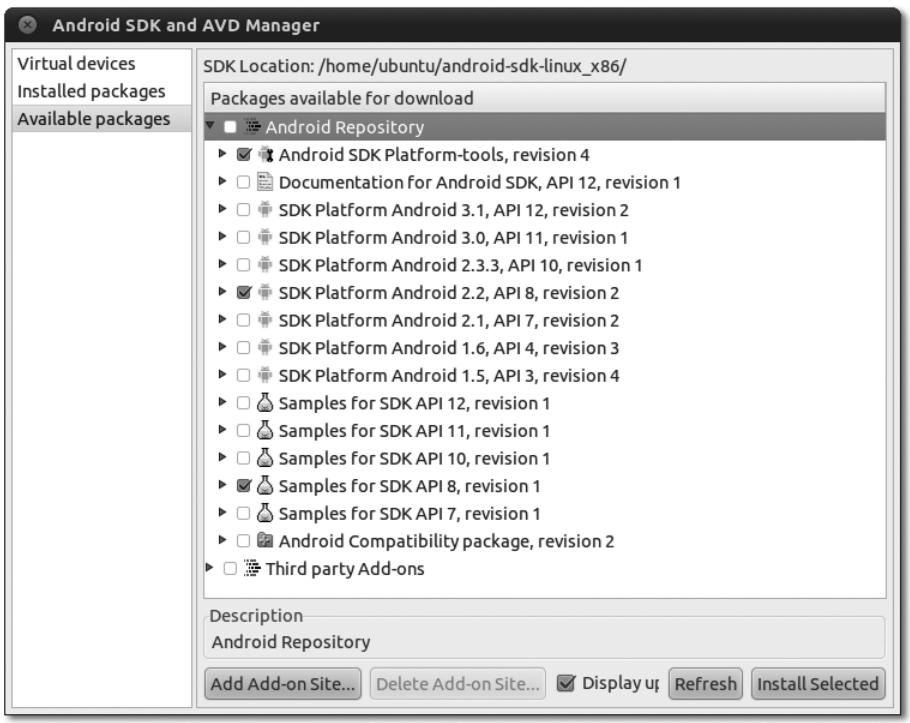



**Abbildung 3.10** Der Android SDK and AVD Manager bietet mehr an, als Sie brauchen. Setzen Sie die Häkchen wie in diesem Bild.

Erneut müssen Sie Lizenzbedingungen akzeptieren, dann erfolgen Download und Installation. Zum Schluss beantworten Sie die Frage, ob ADB (Android Debug Bridge) neu gestartet werden soll, mit JA. Was sich hinter ADB verbirgt, muss Sie im Moment nicht ablenken.

Unter INSTALLED PACKAGES können Sie das Resultat dieser Aktion begutachten (Abbildung 3.11).

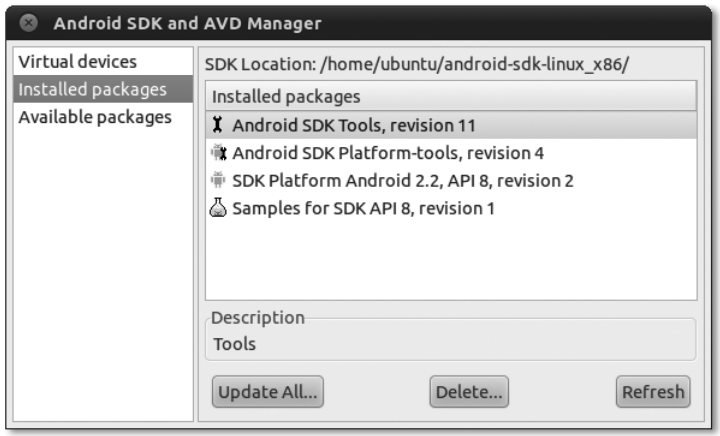



**Abbildung 3.11** Der Erfolg der Installation ist übersichtlich.

## 3.8 Ein virtuelles Gerät erzeugen

Als letzten Schritt fügen Sie ein virtuelles Gerät (AVD) hinzu. Diese virtuellen Geräte benötigen Sie, um Apps auf einem Emulator laufen zu lassen. Android ist nicht gleich Android: Es gibt unterschiedliche Betriebssystemversionen, und am Ende des Entwicklungszyklus sollten Sie Ihre App auf jeder einzelnen testen und nicht nur auf Ihrem eigenen Handy. Sie benötigen nicht zwingend einen Emulator, um dieses Buch durchzuarbeiten, aber es ist manchmal hilfreich.

Wählen Sie im Android SDK and AVD Manager links VIRTUAL DEVICES. Sie sehen den lapidaren Hinweis, dass kein AVD vorhanden ist. Klicken Sie auf NEW..., um das zu ändern (Abbildung 3.12).

Erfinden Sie einen Namen für Ihr virtuelles Gerät, verzichten Sie dabei jedoch auf Leerzeichen oder Umlaute. Ich habe in Abbildung 3.12 z. B. MEINGERAET gewählt. Wählen Sie als TARGET das einzig Mögliche: ANDROID 2.2. Wenn Sie mehr als eine Plattform installiert haben, also unterschiedliche Android-Versionen, können Sie hier auswählen, mit welcher das virtuelle Gerät arbeitet.


**Abbildung 3.12** Das erste AVD entsteht.

Ihr neues Gerät erhält eine (virtuelle) Speicherkarte, da ohne SD-Karte einige Dinge nicht erwartungsgemäß funktionieren. 32 Megabyte genügen, um eine normale App laufen zu lassen. Nur wenn Sie mit Musik-, Bild- oder Videodaten arbeiten wollen, brauchen Sie mehr Platz.

Wählen Sie als SKIN HVGA aus. HVGA steht für **Half Size VGA**, ist also halb so groß wie die Standard-VGA-Auflösung von 640 x 480 Pixeln: 480 x 320. Im Hochformat entspricht das einer Breite von 320 Bildpunkten. Viele ältere Android-Geräte besitzen Displays mit einer HVGA-Auflösung, und zum Testen genügt eine solche Größe ohne Weiteres. Sie können jederzeit weitere virtuelle Geräte anlegen oder vorhandene löschen. Beachten Sie aber, dass das dem umweltgerechten Entsorgen eines Handys gleichkommt: Alle darauf gespeicherten Daten gehen verloren.

Es wird Zeit, einen Blick auf ein emuliertes Android-Gerät zu werfen. Wählen Sie daher Ihr gerade angelegtes AVD aus, und klicken Sie auf START (Abbildung 3.13).

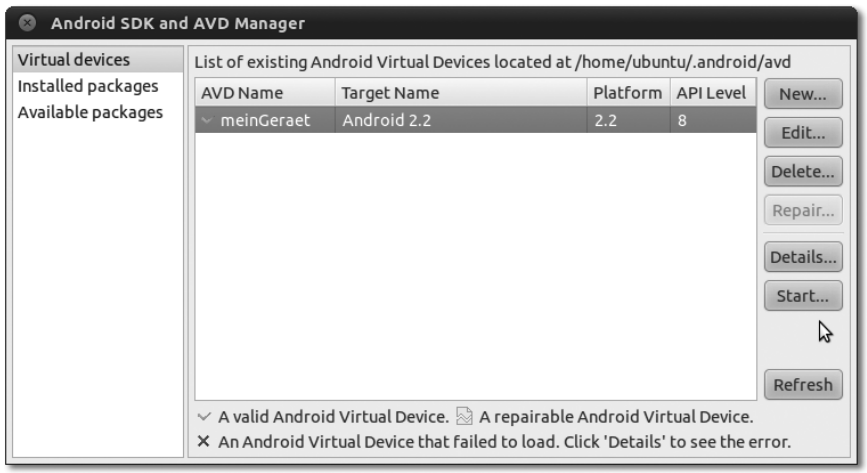



**Abbildung 3.13** Vorläufig besitzen Sie lediglich ein virtuelles Gerät, dessen Start nur einen Klick entfernt ist.

Sie werden nach Launch Options gefragt, aber für den Anfang können Sie die Vorgaben übernehmen, um endlich den Emulator zu starten. Wie auch bei einem echten Gerät dauert das eine ganze Weile – je nach Leistungsfähigkeit Ihres PCs. Am Ende steht jedenfalls ein laufendes Android-System in einem schmucklosen Fenster (Abbildung 3.14). Telefonieren können Sie damit zwar nicht – aber das ist ohnehin nicht der Hauptzweck eines Smartphones, richtig?


**Abbildung 3.14** Und täglich grüßt der grüne Roboter. Sie werden das Einführungs-Widget los, wenn Sie es mit der Maus nach unten in die Mitte des emulierten Bildschirms ziehen.

Wenn Sie möchten, können Sie jetzt dem Android-System auf dem Emulator die deutsche Sprache beibringen. Klicken Sie unten auf das Symbol mit den 16 kleinen Quadraten, um das Hauptmenü zu öffnen. Suchen Sie das Zahnräder-Icon mit der Bezeichnung Custom Locale. Scrollen Sie durch die Liste, bis Sie de German finden. Durch einen langen Klick und das Bestätigen einer Sicherheitsabfrage ändern Sie die Sprache des ganzen Emulators.

Sie können den Emulator von Eclipse aus steuern, Screenshots anfertigen und verfolgen, was auf dem virtuellen Gerät alles passiert.

Wählen Sie in Eclipse den Menüpunkt Window • Show View • Other..., dann tippen Sie in das obere Textfeld »devices«. Daraufhin ist nur noch der Eintrag des Devices-Views im unteren Bereich zu sehen. Doppelklicken Sie darauf. Der neue View öffnet sich und sollte bei laufendem Emulator einen Eintrag aufweisen, der aus einem Handy-Icon und einer Codenummer besteht. Darüber hinaus wird die laufende Android-Version angezeigt. Das Device lässt sich aufklappen, und darunter erscheinen die auf dem Gerät laufenden Prozesse.

## 3.9 Eclipse mit dem Handy verbinden

Um Ihr Handy mit Eclipse zu verbinden, öffnen Sie zunächst dessen Einstellungen-App über das Hauptmenü. Suchen Sie dann den Menüpunkt Anwendungen, gefolgt von Entwicklung. Schalten Sie das USB-Debugging an, indem Sie ein Häkchen setzen. Sorgen Sie mit einem zweiten Häkchen dafür, dass das Handy nicht einschläft, während es am USB-Kabel hängt (Abbildung 3.15).


**Abbildung 3.15** USB-Debugging vereinfacht das Entwicklerleben.

Schließen Sie als Nächstes Ihr Handy mit dem üblicherweise mitgelieferten Kabel an eine USB-Schnittstelle Ihres Rechners an. Je nach Android-Version müssen Sie anschließend noch einmal die Verbindung quittieren. Das Resultat ist, dass im Devices-View von Eclipse ein weiterer Eintrag auftaucht, nämlich der für das gerade angeschlossene Gerät.

Klicken Sie zum Spaß einmal innerhalb des Devices-Views auf das Kamera-Icon rechts oben. Sofort erscheint ein Fenster mit einem Foto, das Pixel für Pixel dem aktuellen Bildschirminhalt Ihres Handys entspricht. Dieses Bild können Sie speichern oder mit Copy in die Zwischenablage befördern, um es mit anderen Programmen weiterzuverwenden. Die Screenshot-Funktion wird sich noch als nützlich erweisen.

Das Bild von der Digicam können Sie allerdings auf diese Weise nicht fotografieren.

Den weitaus größten Vorteil aber werden Sie kennenlernen, wenn Sie Ihre erste kleine App erstellt haben: Per Knopfdruck wird Eclipse sie auf Ihrem Handy installieren und dort starten. Mehr noch: Sie werden bei Bedarf den Programmablauf Schritt für Schritt verfolgen können, was bei der Fehlersuche ungemein hilfreich ist. Und Fehlersuche – so traurig es klingt – ist eine der Arbeiten, mit denen Programmierer deutlich mehr Zeit verbringen als mit ihren Freundinnen.

## 3.10 Was tun, wenn mein Eclipse verrücktspielt?

Während Sie sich in die Entwicklung mit Eclipse einarbeiten, wird es zwangsläufig passieren, dass Sie versehentlich eine verkehrte Tastenkombination drücken oder auf das falsche Icon klicken. Es gibt eine ganze Reihe von Fällen, in denen Sie nicht auf Anhieb verstehen, was Eclipse von Ihnen will. Ich kann leider nicht alle denkbaren Fälle vorhersehen, aber auf ein paar häufig vorkommende Probleme werde ich eingehen.

### Unerklärliche Unterstreichungen

Wenn Sie Pech haben, ist Ihr Eclipse der Meinung, einen veralteten Java-Standard benutzen zu müssen, nämlich Java 1.5. Es gibt wenige, aber entscheidende Unterschiede zwischen Java 1.5 und Java 1.6, auf dem Android basiert. Die Folge sind beispielsweise rot unterstrichene Methodennamen. Prüfen Sie zur Sicherheit, ob Ihr Eclipse auf die richtige Java-Version geeicht ist. Öffnen Sie den Preferences-Dialog über das Window-Menü. Suchen Sie die Einstellungsseite für die Versionskompatibilität des Java-Compilers, und stellen Sie sicher, dass alles auf Java 1.6 steht (Abbildung 3.16).

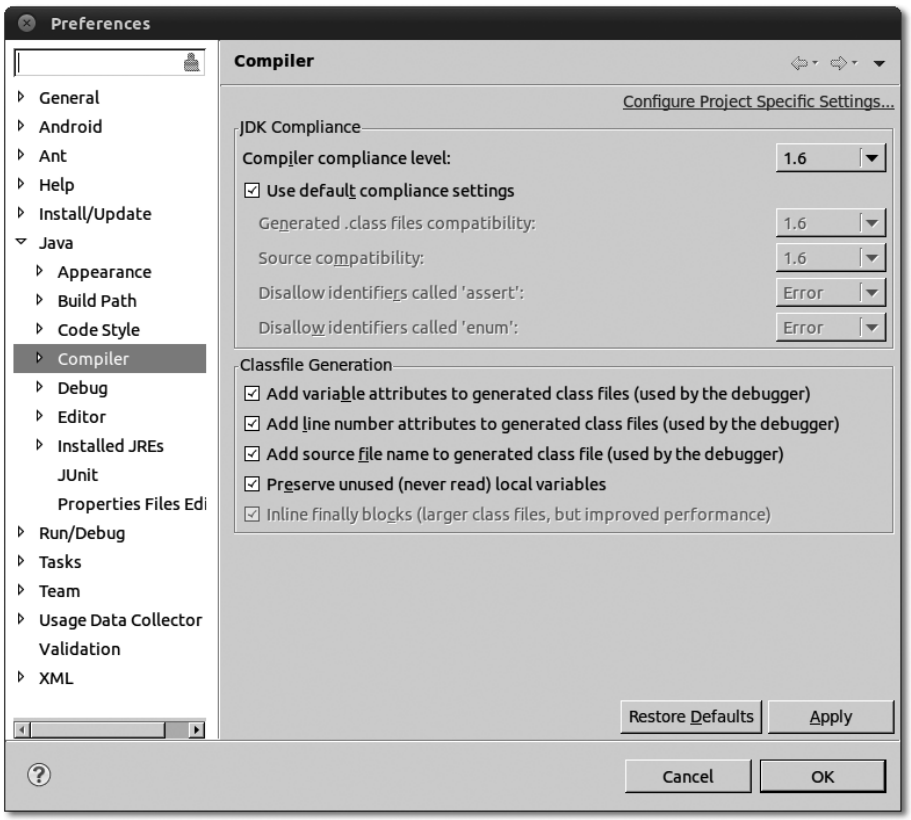



**Abbildung 3.16** Stellen Sie die Java-Kompatibilität auf Version 1.6.

Dieselbe Einstellung gibt es noch einmal für jedes einzelne Projekt. Wählen Sie im Menü PROJECT • PROPERTIES. Dort sollte ENABLE PROJECT SPECIFIC SETTINGS auf der Seite für den Compiler ausgeschaltet sein, damit die gerade vorgenommenen Einstellungen gelten.

## Ein Handy namens Fragezeichen

Wenn Sie Ihr Handy per USB an den Rechner koppeln, taucht ein Eintrag in der Liste der Geräte in Eclipse auf. Anstelle des Namens Ihres Handys sehen Sie dort unter Umständen bloß Fragezeichen.

Das passiert, wenn das ADT nicht genug Zugriffsrechte hat, um die USB-Schnittstelle zu steuern. Üblicherweise sehen Sie diese Fehler unter Linux. Geben Sie folgende Kommandozeilen-Befehle ein, um die Android Debug Bridge mit Root-Rechten zu starten:

```
sudo adb kill-server
sudo adb start-server
```

### Eclipse hängt sich auf

Es ist unerfreulich, aber wahr: Manchmal tut Eclipse einfach gar nichts mehr. Es ist dermaßen mit sich selbst beschäftigt, dass Sie manchmal mit dem ganzen Rechner nicht mehr arbeiten können. Hauptsächlich ist es die Garbage Collection, die von Zeit zu Zeit diese Blockaden verursacht.

Natürlich können Sie Eclipse jetzt abschießen und neu starten – oder Sie drehen eine Runde durch die Wohnung. Man soll ja eh nicht zu lange ununterbrochen auf einem Stuhl sitzen. Nach etwa 30 bis 60 Sekunden ist Eclipse wieder einsatzbereit, ohne dass Sie eingreifen müssen. Weiter geht's!

### Eclipse findet Resource-Dateien nicht

Wenn Sie Grafiken oder Sounds außerhalb von Eclipse in das *res*-Verzeichnis Ihres Projekts legen, merkt die Entwicklungsumgebung das nicht. Sie müssen explizit das Projektverzeichnis auf den aktuellen Stand bringen, indem Sie den RES-Knoten anklicken und im Kontextmenü REFRESH wählen. Dann erscheinen auch die vermissten Dateien, und der Android Resource Manager wird die nötigen Einträge in die Klasse `R` schreiben.

*»Wir irren uns nie.«*
*(HAL 9000)*

# 4 Die erste App

Software installieren, Java-Crashkurs ... Jetzt wird es Zeit für Ihre erste App. Starten Sie Eclipse, schließen Sie Ihr Handy an, stellen Sie Kaffee (oder Tee) und Kekse bereit. Fertig? Auf in den Kampf!

## 4.1 Sag »Hallo«, Android!

Üblicherweise ist das erste Programm, das Sie in Lehrbüchern kennenlernen, eines, das den Text »Hallo, Welt« auf den Bildschirm schreibt. Mit etwas Glück lässt es Sie die Mehrwertsteuer berechnen oder D-Mark in Gulden umrechnen.

Aus Sicht eines Smartphones kommt das einer tödlichen Beleidigung ziemlich nahe, finden Sie nicht? Daher werden wir eine standesgemäße App vorziehen. Als kleine Vorbereitung, die schon ziemlich viel verrät, öffnen Sie bitte den Android Market und suchen nach »Text to Speech«. Installieren Sie die App, falls Sie sie noch nicht auf Ihrem Gerät haben. Sie stellt, wie Sie unschwer erraten können, Sprachausgabe-Funktionen bereit. Auf dem Emulator funktioniert das leider nicht ohne Weiteres, halten Sie daher Ihr Handy samt USB-Kabel bereit.

Jede App entspricht im Arbeitsbereich von Eclipse einem **Project**. Als ersten Schritt legen Sie ein neues Projekt an. Dank des installierten ADT gibt es die Möglichkeit, gleich ein Android-Projekt mit den nötigen Voreinstellungen anzulegen. Wählen Sie im Eclipse-Menü NEW • OTHER, oder drücken Sie Strg + N.

### Ein neues Android-Projekt erstellen

Zum Anlegen von neuen Projekten oder Dateien gibt es Wizards. Wir benötigen den Wizard mit dem Namen ANDROID PROJECT. Es genügt, wenn Sie die ersten drei Buchstaben in das obere Textfeld eintippen: Daraufhin erscheinen darunter nur noch die Wizards mit dazu passenden Namen. Einer davon ist der gesuchte Wizard. Doppelklicken Sie darauf, um ihn zu starten (Abbildung 4.1).

Geben Sie dem neuen Projekt den Namen SAGHALLO. Wählen Sie in der Liste der Build Targets ANDROID 2.2 aus. Sie erinnern sich sicher, dass Sie diese SDK-Version zuvor mit dem SDK and AVD Manager installiert haben.

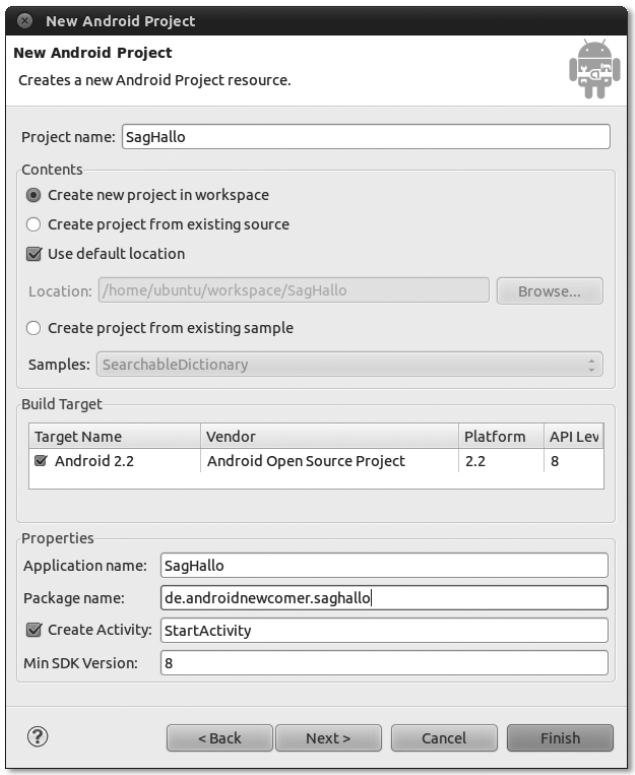



**Abbildung 4.1** Der Wizard namens »New Android Project« erzeugt alle Dateien, die wir brauchen.

Auch als Application Name tragen Sie SagHallo ein. Der Package Name bleibt Ihnen überlassen, ich wähle in allen Beispielen `de.androidnewcomer` und hänge den Projektnamen in Kleinbuchstaben an, in diesem Fall heißt mein Package also: `de.androidnewcomer.saghallo`.

Lassen Sie den Wizard auch gleich eine Activity erzeugen. Activities sind Klassen, die jeweils App-Bildschirme verwalten, und ohne macht eine App nicht viel her. Nennen Sie die Activity `StartActivity`.

Es ist eine Konvention, den Namen jeder `Activity`-Klasse mit diesem Begriff enden zu lassen. Sie werden noch sehen, dass Activities entscheidende Bestandteile von Android-Apps sind, daher sollten die Klassen auf den ersten Blick als Activities erkennbar sein.

Tragen Sie zum Schluss als Min SDK Version eine »8« ein. Diese Versionsnummer entspricht üblicherweise der Spalte API Level des ausgewählten Build Targets. Damit bestimmen Sie, dass Ihre App nur auf Geräten ab Android 2.2 läuft.

Falls Sie ein älteres Handy haben sollten, müssen Sie hier die passende Version eingeben. Android 1.6 entspricht beispielsweise API-Level 4. Laden Sie sich mit dem AVD Manager die passende Umgebung herunter, falls nötig. Dann sehen Sie auch die zugehörige API-Level-Version.

Klicken Sie auf FINISH.

Die App, die der Wizard für uns erzeugt hat, kann natürlich noch nicht sprechen. Ihnen bleibt daher nichts anderes übrig, als es ihr beizubringen – indem Sie nun feierlich die ersten eigenen Java-Zeilen hinzufügen.

## Die StartActivity

Klappen Sie im Package Explorer das dort entstandene Icon auf, das Ihr Android-Projekt **SagHallo** repräsentiert. Von den vielen Icons, die daraufhin auftauchen, ignorieren Sie zunächst alle bis auf das, neben dem SRC (Abkürzung für **Source Code,** also Quellcode) steht. In diesem Verzeichnis befinden sich alle Java-Dateien des Projekts. Der Wizard hat für den Anfang genau eine erzeugt, sie heißt *StartActivity.java* und befindet sich im Package `de.androidnewcomer.saghallo`, genau wie gewünscht.

Öffnen Sie das Package, und doppelklicken Sie auf *StartActivity.java*. Daraufhin zeigt Eclipse Ihnen im großen Fenster den vom Wizard erzeugten Java-Code (Abbildung 4.2).

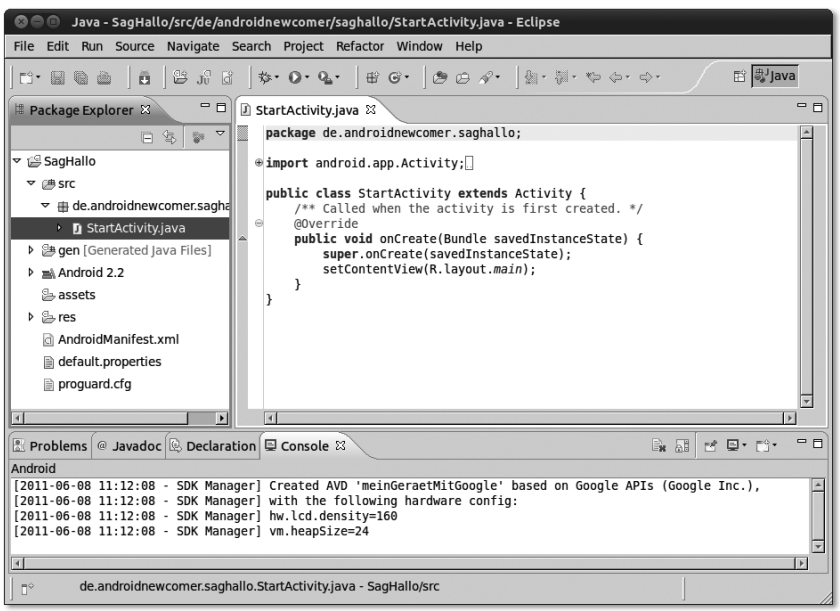



**Abbildung 4.2** Der Wizard hat genau eine Java-Datei erzeugt.

Sie sehen, dass die Klasse `StartActivity` von einer Klasse namens `Activity` erbt und genau eine Methode enthält: `onCreate()`. Die Annotation `@Override` zeigt Ihnen, dass `onCreate()` offenbar eine gleichnamige Methode in der Elternklasse `Activity` überschreibt.

Wie der ebenfalls automatisch erzeugte Quellcodekommentar in freundlichem Blau erläutert, wird diese Methode beim ersten Start automatisch aufgerufen. Darum kümmert sich das Android-System ohne unser Zutun, nachdem es ein Objekt der Klasse erzeugt hat.

**Kommentare**

Es gibt zwei Möglichkeiten, Kommentare im Programmcode zu kennzeichnen, damit der Java-Compiler nicht versucht, den Text zu übersetzen.

Die eine Schreibweise schließt den Kommentar in Schrägstriche und Sterne ein:

```
/* Dies ist ein Kommentar.
   Er kann sich über mehrere Zeilen erstrecken. */
```

Wie Sie sehen, können Sie auf diese Weise längere Kommentare schreiben. Oft verwenden Programmierer diese Notation, um kurze Codeschnipsel *auszukommentieren*.

```
if( bedingung1  /* && bedingung2 */ ) {
```

Diese Strategie dient dazu, auszuprobieren, wie sich ein Programm ohne einen bestimmten Teil verhält.

Die zweite Möglichkeit, Kommentare im Programmcode zu kennzeichnen, ist der doppelte Schrägstrich `//`.

```
Auto cabrio = new Auto(); // erzeugt mein neues Auto
```

Der Java-Compiler ignoriert alles, was hinter dem `//` steht, bis zum Ende der Zeile. Folglich können Sie damit keine einzelnen Elemente innerhalb einer Zeile auskommentieren, ebenso erfordert jede weitere Kommentarzeile einen weiteren einleitenden `//`.

Der Doppelschrägstrich wird gern am Zeilenanfang verwendet, um eine ganze Zeile auszukommentieren. Eclipse hilft Ihnen sogar dabei: Wenn Sie mehrere Zeilen markieren und Strg + ⇧ + 7 drücken (also sozusagen Strg + /), werden alle Zeilen mit `//` auskommentiert oder, wenn sie schon auskommentiert sind, wieder einkommentiert (d. h. die `//` entfernt). Natürlich klappt das auch über das Menü: Wählen Sie Source • Toggle Comment.

Eclipse hebt Kommentare zwar farblich hervor, aber allzu viele kleine Kommentare verbessern nicht gerade die Übersicht.

Die Faustregel für Kommentare lautet: Schreiben Sie welche, wenn Sie es für möglich halten, dass Sie oder andere Programmierer eine Stelle sonst nicht auf Anhieb verstehen. Und überschätzen Sie dabei niemanden …

Derzeit erledigt die `onCreate()`-Methode zwei Dinge: Erstens ruft sie die gleichnamige Methode der Elternklasse auf. Damit sie sich nicht selbst aufruft, steht

`super` davor. Selbstverständlich besitzt auch die Elternklasse eine Methode namens `onCreate()`, und sie erledigt wichtige organisatorische Aufgaben. Deshalb muss sie unbedingt aufgerufen werden.

Die zweite Codezeile in der Methode `onCreate()` ruft die Methode `setContentView()` auf. Da diese Methode offensichtlich nicht in `StartActivity` zu finden ist, können Sie davon ausgehen, dass sie in einer Elternklasse definiert ist. Die Methode erhält als Parameter einen Wert, über den später noch zu sprechen sein wird. Für den Moment genügt es, zu wissen, dass diese Zeile dafür sorgt, dass ein anderswo unter dem Namen `main` definierter Bildschirminhalt angezeigt wird.

**Zur Erinnerung: Private Methoden**

In einer Methode können Sie alle Methoden derselben Klasse aufrufen, außerdem alle Methoden der Elternklasse, die nicht mit dem Modifizierer `private` Ihren Blicken entzogen sind:

```
class Elternklasse {
   private void eineMethode() {
      ...
   }
}
...
class Kindklasse extends Elternklasse {
   public void testMethode() {
      eineMethode(); // Fehler
   }
}
```

Die Programmierer der Klasse `Activity` stellen Ihnen eine Menge hilfreicher Methoden zur Verfügung, die Sie in eigenen von `Activity` abgeleiteten Klassen verwenden können.

Lassen Sie uns nun die `StartActivity` um die gewünschte Sprachausgabe erweitern. Das wird dann dazu führen, dass die App beim ersten Start zu uns spricht.

Fügen Sie zunächst der Klasse ein Attribut hinzu:

```
private TextToSpeech tts;
```

Diese Eingabe quittiert Eclipse mit einer roten Unterstreichung, weil `TextToSpeech` unbekannt ist. Rote Unterstreichung heißt: Syntaxfehler, der Compiler kann dies nicht übersetzen, folglich kann er kein lauffähiges Programm erzeugen. Also müssen Sie `TextToSpeech` importieren.

Die gesuchte Klasse befindet sich in einem Paket des Android SDK, daher müssen Sie sie importieren. Glücklicherweise nimmt Ihnen Eclipse diese Arbeit ab, weil es im Gegensatz zu Ihnen schnell nachschauen kann, welches das zu importie-

rende Paket ist. Drücken Sie ⇧ + Strg + O (O steht für Organize Imports, Sie finden dieselbe Funktion im Menü unter Source • Organize Imports). Eclipse fügt jetzt automatisch eine `import`-Anweisung hinzu. Sie sehen nicht viel davon, weil der Editor diese selten benötigten Zeilen der Übersicht halber einklappt. Klicken Sie auf das Plus-Icon in der vertikalen Leiste am linken Rand des Edit-Fensters, um die eingeklappten Zeilen zu sehen, und noch einmal, um sie wieder loszuwerden.

Das Objekt `tts` (Abkürzung für **Text to Speech**) verwaltet die Sprachausgabe. Natürlich tut so eine Attribut-Deklaration rein gar nichts, also müssen Sie zunächst ein Objekt erzeugen. Erweitern Sie die Methode `onCreate()` am Ende, also hinter dem `setContentView()`-Aufruf, um folgende Zeile:

```
tts = new TextToSpeech(this, this);
```

Dies erzeugt ein neues `TextToSpeech`-Objekt und weist es dem Attribut `tts` zu (`this` steht für »das aktuelle Objekt«). Bis auf eine Kleinigkeit könnten Sie das `tts`-Objekt nun verwenden, um dem Handy eine freundliche Stimme zu entlocken:

```
tts.speak("Hallo!", TextToSpeech.QUEUE_FLUSH, null);
```

Während die Bedeutung des ersten Parameters dieses Methodenaufrufs offensichtlich ist, ignorieren Sie die beiden anderen vorläufig, sie sind nicht von Bedeutung.

Leider gibt es einen kleinen Haken: Da Sprachausgabe auch im 21. Jahrhundert nicht ganz trivial ist, muss Android sie zuerst initialisieren. Das kann ein paar Sekunden dauern, daher dürfen Sie nicht davon ausgehen, dass die Sprachausgabe sofort in der nächsten Zeile funktioniert. Aber woher erfahren wir, wann es so weit ist?

Freundlicherweise gibt `TextToSpeech` Ihnen Bescheid, wenn es sprechbereit ist, indem es eine Methode namens `onInit()` aufruft. Diese Methode fügen Sie jetzt der Klasse `StartActivity` hinzu. Als einzige Zeile fügen Sie die eigentliche Sprachausgabe ein:

```
@Override
public void onInit(int status) {
   tts.speak("Hallo!", TextToSpeech.QUEUE_FLUSH, null);
}
```

Erneut unterstreicht Eclipse einige Stellen. Ergänzen Sie die Definition der Klasse wie folgt:

```
public class SagHallo extends Activity implements OnInitListener
```

Dies verknüpft die Methode `onInit()` mit der Fertig-Meldung von `TextToSpeech`.

### Interfaces implementieren

Der `OnInitListener` ist ein Interface (*Schnittstelle*). Interfaces ähneln Klassen sehr stark. Allerdings besitzen sie lediglich leere Methodenrümpfe:

```
interface OnInitListener {
  void onInit();
}
```

Eine andere Klasse kann nun ein oder mehrere Interfaces implementieren. Dazu dient das Schlüsselwort `implements`. Die Folge ist, dass die Klasse alle Methoden, die im Interface definiert sind, enthalten muss.

Interfaces trennen die Funktionen, die Klassen anbieten können, von deren genauer Implementierung. So ist es der `TextToSpeech`-Klasse völlig egal, was Sie in der Methode `onInit()` tun – Hauptsache, Sie bieten ihr irgendein Objekt, das diese Methode implementiert. Deshalb definiert der Konstruktor `TextToSpeech()` als zweiten Parameter keine Klasse, sondern ein Interface – `OnInitListener` eben. Indem Sie `this` (also das aktuelle `StartActivity`-Objekt) übergeben und `StartActivity` das gewünschte Interface samt Methode `onInit()` verpassen, stellen Sie den Konstruktor zufrieden (Abbildung 4.3).

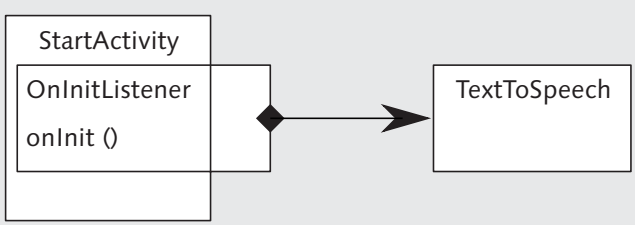



**Abbildung 4.3** »TextToSpeech« interessiert sich nur für das Interface »OnInitListener«. Dass es in der Klasse »StartActivity« implementiert ist, ist ihr ziemlich egal.

Interfaces sind eine sehr wichtige Programmierstrategie, die auch in diesem Buch häufig vorkommt.

`TextToSpeech.QUEUE_FLUSH` ist eine Konstante, über deren genaue Bedeutung Sie sich im Moment keine Gedanken machen müssen.

Lassen Sie Eclipse einmal mehr Imports organisieren (Strg + ⇧ + O), und speichern Sie mit Strg + S. Das wird alle roten Icons und Unterstreichungen aus der Welt schaffen. Bevor Sie Ihre App zum ersten Mal starten, vergewissern Sie sich, dass alles aussieht wie in Abbildung 4.4.

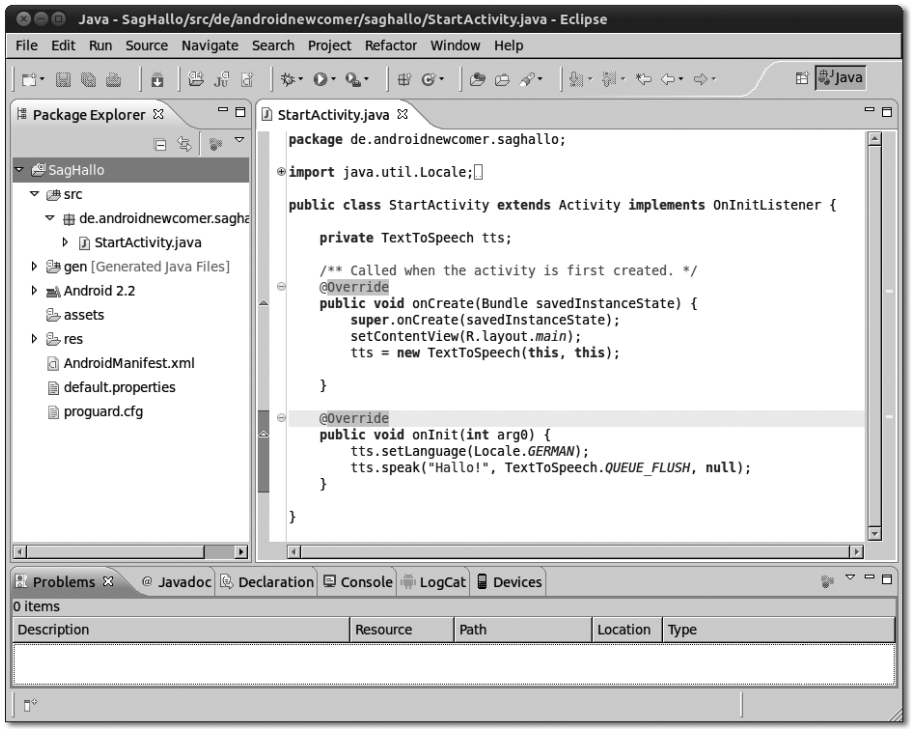



**Abbildung 4.4** Der Code der sprechenden App passt auf eine Bildschirmseite.

## Der erste Start

Schließen Sie Ihr Handy per USB-Kabel an, und warten Sie einen Moment, bis es im DEVICES-View aufgelistet wird. Dann wird es spannend! Klicken Sie mit rechts auf das Haupt-Icon Ihres Projekts SAGHALLO. Wählen Sie im Popup-Menü RUN AS..., gefolgt von ANDROID APPLICATION. Ohne Ihr Zutun hat Eclipse längst ein handliches Päckchen aus Ihrer App gebaut, das *SagHallo.apk* heißt. Sie finden es im Unterverzeichnis *bin* in Ihrem Projektordener, falls Sie es von Hand installieren möchten. Für den Moment aber überlassen wir alles Eclipse. Verfolgen Sie im CONSOLE-View am unteren Rand des Eclipse-Fensters, wie die Entwicklungsumgebung nacheinander das *SagHallo.apk* auf Ihr Handy schiebt, installiert und startet.

Sekunden später wird der Bildschirm Ihres Handys schwarz, ein Schriftzug erscheint – und die freundliche Frauenstimme sagt »Hallo«.

Ihre erste App funktioniert!

Experimentieren Sie, indem Sie im Programmcode das Wort »Hallo« durch etwas Beliebiges ersetzen und die App erneut starten. Sie können Strg + F11 drücken, um das zuletzt gestartete Projekt erneut laufen zu lassen, oder im Startmenü des Handys das Icon Ihrer App antippen, um sich dasselbe noch mal anzuhören. Außerdem finden Sie in Eclipse ein rundes, grünes Icon mit einem weißen Dreieck, das ebenfalls zum Programmstart dient.

Halten Sie sich vor Augen, was beim Start der App geschieht:

- Wenn Sie das Icon antippen oder Eclipse die App ferngesteuert startet, erzeugt Android ein Objekt der Klasse `StartActivity`, d. h., irgendwo wird `new StartActivity()` ausgeführt.
- Anschließend ruft Android die Methode `onCreate()` des erzeugten Activity-Objekts auf.
- Ihr Code erzeugt ein neues `TextToSpeech`-Objekt namens `tts`.
- Sobald `tts` sprechbereit ist, ruft es die Methode `onInit()` in Ihrem Activity-Objekt auf.
- Ihr Code in der `onInit()`-Methode ruft `tts.speak()` auf, um das Handy zum Sprechen zu bringen.

Es geschieht also einiges zwischen Start und Sprechen, allerdings ist nichts davon besonders kompliziert, noch mussten Sie anspruchsvollen Programmcode schreiben. Wie Sie an vielen Beispielen sehen werden, kommt es im Leben eines Programmierers sehr oft hauptsächlich darauf an, an der richtigen Stelle die richtige Methode mit den richtigen Parametern aufzurufen – längst vorhandener und von anderen Entwicklern geschriebener Programmcode erledigt den anstrengenden Rest.

Sie werden beim Ausprobieren verschiedener Sätze hören, dass die künstliche Stimme mit einigen Sprachwendungen deutliche Schwierigkeiten hat. Zum Teil liegt das daran, dass wir der digitalen Dame nicht erklärt haben, dass wir von ihr ordentliches Deutsch erwarten. Möglicherweise geht sie davon aus, dass wir Amerikaner sind, wie ihre Programmierer. Fügen Sie daher in die `onInit()`-Methode vor der Sprachausgabe-Zeile eine weitere ein:

```
tts.setLanguage(Locale.GERMAN);
```

Dass Sie danach mal wieder ⇧ + Strg + O drücken müssen, muss ich Ihnen sicher nicht mehr erklären, und in Zukunft werde ich mir diesen Hinweis sparen.

**Automatisch organisieren**

Dass Eclipse eine mächtige Entwicklungsumgebung ist, die Ihnen eine Menge Handarbeit abnimmt, haben Sie bereits bemerkt. Wenn nur dieses ständige ⇧ + Strg + O nicht wäre!

Wenn Sie Angst vor Abnutzung Ihrer Tasten haben oder einfach eine Menge Zeit sparen möchten, können Sie Eclipse dazu überreden, bei jedem Abspeichern einer Datei diese Arbeit für Sie zu erledigen – sowie eine Reihe anderer Arbeiten.

Öffnen Sie die zugehörige Einstellungsseite über das Menü WINDOW • PREFERENCES • JAVA • EDITOR • SAVE ACTIONS. Schalten Sie dort den Hauptschalter ein, und setzen Sie einen Haken bei ORGANIZE IMPORTS.

Wie Sie sehen, können Sie Eclipse beim Speichern weitere automatische Aktionen befehlen, allerdings rate ich Ihnen im Moment davon ab, weil sie geeignet sind, Sie zu irritieren. Natürlich können Sie jederzeit damit experimentieren; beachten Sie jedoch, dass die Funktionsfülle von Eclipse nicht nur ein Segen ist: Wenn Sie versehentlich den falschen Haken gesetzt haben, dauert es unter Umständen eine Weile, bis Sie die richtige Stelle wiederfinden, um ihn loszuwerden.

Experimentieren Sie nun weiter mit verschiedenen Sätzen. Übrigens können Sie die App (ohne allerdings den Text ändern zu können) jederzeit auch vom Handy aus starten. Sie finden ein weiß-grünes Standard-Icon in Ihrem Hauptmenü. Die installierte App bleibt auf Ihrem Handy, bis Sie sie deinstallieren oder durch eine neue Version ersetzen. Selbstverständlich können Sie jederzeit das USB-Kabel entfernen, die App bleibt auf dem Gerät.

## 4.2 Bestandteile einer Android-App

Natürlich ist es fürchterlich umständlich, jedes Mal die App neu zu installieren, obwohl Sie nur die zu sprechenden Worte geändert haben. Ich sehe Ihnen an der Nasenspitze an, dass Sie sich jetzt ein Textfeld auf dem Handy-Bildschirm wünschen, in das Sie irgendetwas eintippen können, sowie einen Button, der diesen Text in Sprache verwandelt. Allerdings genügt es dazu nicht, die eine Ihnen nun bestens bekannte Java-Datei zu bearbeiten. Deshalb verschaffe ich Ihnen jetzt einen Überblick über die restlichen Komponenten, aus denen die App besteht.

Klappen Sie alle Knoten unterhalb von SAGHALLO im Package Explorer auf. Doppelklicken Sie dann auf die Datei *AndroidManifest.xml*, die sich fast ganz unten in der langen Liste befindet. Das **Android-Manifest** ist die zentrale Beschreibung Ihrer App, daher werden wir uns an ihr orientieren, wenn wir jetzt die einzelnen Komponenten in Augenschein nehmen (Abbildung 4.5).

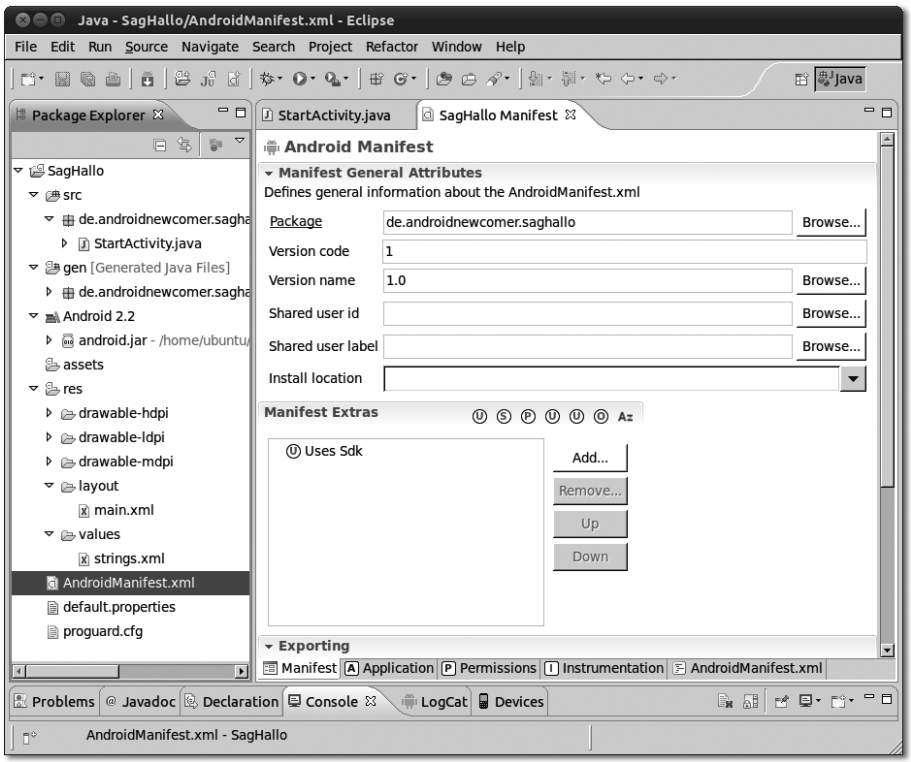



**Abbildung 4.5** Das Android-Manifest ist die Schaltzentrale jeder App.

## Versionsnummern

Jede App besitzt einen (numerischen) Versionscode. Um nicht unnötig auf Verwirrung zu verzichten, haben die Android-Erfinder zusätzlich einen Versionsnamen erfunden, der nicht nur Ziffern oder Buchstaben enthalten kann, sondern auch etwas völlig anderes als den Versions*code*.

Wenn Sie Ihre Apps später im Android Market veröffentlichen, spielt der Versionscode eine entscheidende Rolle. Sie können nämlich immer nur eine neue Variante Ihrer App veröffentlichen, wenn diese einen höheren Versionscode hat als die vorherige. Erhöhen Sie also den Code immer mindestens um 1, wenn Sie eine neue Veröffentlichung planen, selbst wenn Sie nur einen kleinen Fehler behoben haben.

Üblicherweise sind Versionsnummern aber keine einfachen Zahlen, sondern solche mit mindestens einem Dezimalpunkt: 1.0 oder 3.0.1. Solche Nummern können Sie nicht als Versionscode verwenden, weil dort kein Punkt oder Komma zulässig ist.

Sie können jedoch sinnvolle Versionscodes erhalten, indem Sie die Punkte einfach weglassen. Aus Version 0.01 wird 001 (also 1), 0.99 wird 99, und Version 1.00 erhält den Code 100. Sie können sogar jederzeit von drei- auf vierstellige Codes umsteigen, da 1003 (Version 1.003) größer ist als 100 (Version 1.00). Nur umgekehrt funktioniert das freilich nicht.

Der Versionsname ist ein beliebiger Text, den der Android Market Benutzern anzeigt. Geben Sie hier dasselbe ein wie beim Code, höchstens noch mit den Punkten, das ist am einfachsten und vermeidet Verwirrung.

### Activities anmelden

Apps mit nur einer Activity sind meist ziemlich langweilig, daher werden Sie früher oder später weitere Activities benötigen. Es genügt allerdings nicht, einfach die betreffenden Klassen zu schreiben. Zusätzlich müssen Sie jede dieser Klassen im Android-Manifest anmelden.

Wählen Sie die zweite Registerkarte namens Application aus. Hier können Sie eine ganze Menge Vorgaben einstellen, die meisten sind jedoch selten von Bedeutung. Entscheidend ist die Liste links unten, die mit Application Nodes überschrieben ist. Derzeit finden Sie dort lediglich Ihre `StartActivity`. Mit dem Button Add... können Sie Activities hinzufügen – allerdings müssen Sie die zugehörigen Klassen zuvor selbst schreiben, diese Arbeit nimmt Ihnen dieses Fenster nicht ab. Mit dem Browse-Button durchsuchen Sie Ihr Projekt nach `Activity`-Klassen. Die weiteren Eingabefelder können Sie in den meisten Fällen leer lassen.

Entfalten Sie die vorhandene `StartActivity` und die darunterliegenden Knoten (Application Nodes). Sie sehen, dass der `StartActivity` ein sogenannter **Intent Filter** zugewiesen ist (Abbildung 4.6).

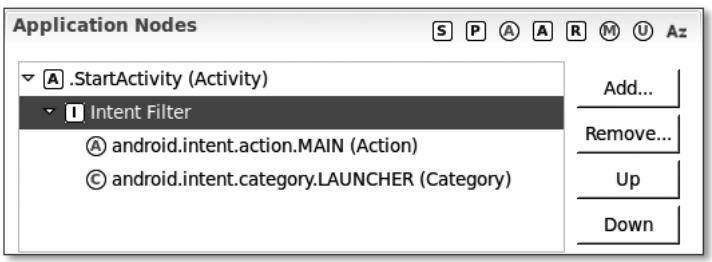



**Abbildung 4.6** Der Intent Filter der Klasse »StartActivity« ist für den Start der App zuständig.

Wozu dient dieser Intent Filter?

Stellen Sie sich vor, Ihre App würde aus mehr als einer Activity bestehen, und zu jeder würde ein Screen gehören. Wenn ein Benutzer das Icon Ihrer App antippt, um sie zu starten, welche Activity soll Android dann starten?

Die Antwort: Das System sucht diejenige Activity in Ihrer App, die einen Intent Filter mit der speziellen **Action** namens `android.intent.action.main` besitzt. Die Activity, die im Manifest mit dieser Action versehen ist, wird gestartet und ist damit die Einstiegs-Activity Ihrer App.

Ihre `StartActivity` besitzt aber offensichtlich noch einen zweiten Intent Filter: `android.intent.category.LAUNCHER`. Dieser Filter sorgt gemeinsam mit dem ersten dafür, dass der Launcher, also das Android-Hauptmenü, ein Icon zum Starten der Activity anzeigt. Wenn Sie zwei Activities anmelden und beide mit diesem Filter versehen, erhalten Sie zwei Icons im Launcher.

Mit dem Intent Filter lässt sich noch eine ganze Menge mehr anstellen: Sie erklären dem Android-System, welche Fähigkeiten eine Activity besitzt. So kann eine Activity verkünden, dass sie Bilder entgegennimmt, beispielsweise, um sie zu verändern und anzuzeigen.

**Drehwurm vermeiden**

Wenn Sie Ihr Handy drehen, passen sich die meisten Apps automatisch an die neue Bildschirmausrichtung an – eine wunderschöne Sache, wenn da nicht die Risiken und Nebenwirkungen wären. Es ist nicht nur mit zusätzlichem Aufwand verbunden, beide Ausrichtungen ordentlich zu unterstützen, die App muss auch jederzeit damit rechnen, »umgekippt« zu werden.

Stellen Sie sich vor, das geschieht mitten im Spiel! Allein schon weil der Bildaufbau die eine oder andere Sekunde in Anspruch nimmt, kann das Ihre App gehörig durcheinanderbringen.

Sie können eine Activity anweisen, nur in einer bestimmten Bildschirmausrichtung dargestellt zu werden. Dreht der Nutzer sein Gerät, ändert sich nichts, der Bildschirm bleibt, wie er ist. Probleme sind so ausgeschlossen, und Sie müssen nur das Layout für die gewünschte Ausrichtung berücksichtigen.

Stellen Sie für jede Activity im Android-Manifest das Attribut SCREEN ORIENTATION auf den gewünschten Wert PORTRAIT oder LANDSCAPE.

## Permissions

Immer wenn Sie eine App aus dem Android Market installieren, bekommen Sie eine Liste von Erlaubnissen (Permissions) angezeigt, die Sie quittieren müssen, damit die App funktioniert.

Das Android-System gewährt Apps nämlich nur jene Rechte, die Sie ihnen erteilen. Eine App kann also nur dann auf das Internet zugreifen (und damit möglicherweise Kosten für Datenverkehr verursachen), wenn Sie es ihr erlaubt haben. Es sind freilich nicht nur Browser und Multiplayer-Spiele, die die Internet-Erlaubnis benötigen – auch jede werbefinanzierte App kommt nicht ohne aus, denn woher sollte sie sonst die hübschen Kleinanzeigen holen, die sie ständig anzeigt?

Wenn Ihre App Android-Funktionen verwendet, die einer Erlaubnis durch den Benutzer bedürfen, müssen Sie das explizit im Manifest mithilfe eines Uses-Permission-Eintrags vermerken. Aktivieren Sie die Registerkarte Permissions Ihres Manifests, fügen Sie testweise mit dem Add-Button einen Uses-Permission-Eintrag hinzu, und klappen Sie rechts die Liste der infrage kommenden Permissions auf (Abbildung 4.7).

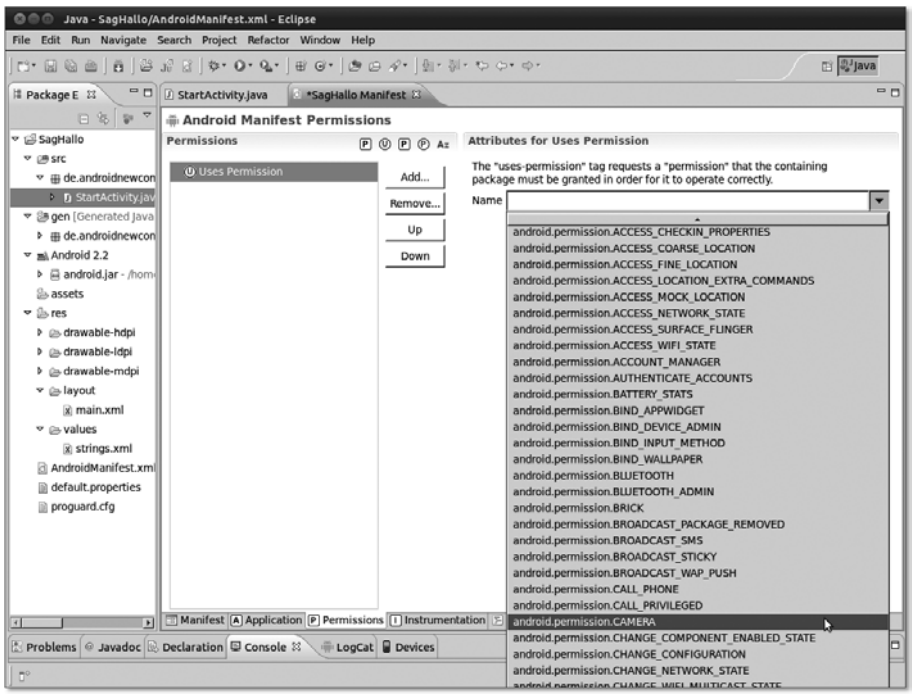



**Abbildung 4.7** Egal, ob Bluetooth, Internet, Kamera – für die meisten Funktionen muss Ihre App explizit eine Erlaubnis einholen.

Die meisten Permissions sind selbsterklärend. Wenn Sie die lange Liste überfliegen, sehen Sie Einträge wie `android.permission.CAMERA`, der den Zugriff auf die eingebaute Kamera erlaubt (merken Sie sich das schon mal für später), oder

`android.permission.BIND_WALLPAPER`, um einen animierten Bildschirmhintergrund für ein Handy anzubieten.

Sowohl dieses Buch als auch die Android-Dokumentation weisen Sie explizit darauf hin, wenn Sie zur Nutzung bestimmter Funktionen eine Erlaubnis beim Benutzer einholen müssen. Falls Sie es doch einmal vergessen, machen Sie sich keine Sorgen: Ihre App wird das Versäumnis in den meisten Fällen mit einem ordentlichen Absturz quittieren und dabei auf die fehlende Erlaubnis hinweisen. Keinesfalls aber ist der gewünschte Dienst ohne die Erlaubnis verfügbar: Bei Android werden Privatsphäre und Sicherheit großgeschrieben.

Dass die meisten Benutzer die von einer App gewünschten Erlaubnisse üblicherweise ohne groß nachzudenken in Sekundenschnelle erteilen – selbst wenn sie »kostenpflichtige SMS versenden« oder »Ihren genauen Standort ermitteln« heißen – das steht auf einem anderen Blatt. Die Missachtung der simplen Regel »erst denken, dann klicken« hat schon so manchem Benutzer einen Virus, einen Datenverlust oder eine hohe Handyrechnung beschert.

## Ressourcen

Eine App besteht nicht nur aus Java-Klassen und dem Android-Manifest. Bildschirmlayouts, Grafiken und Texte gehören ebenfalls dazu. Der Oberbegriff für solche Dateien lautet **Ressourcen** (engl. Resources**).** In Ihrem Projekt SagHallo hat der Android-Wizard beim Anlegen einen Verzeichnisbaum res erzeugt, in dem sich alle Ressourcen befinden müssen.

Klappen Sie alle Knoten des Verzeichnisses res im Package Explorer von Eclipse auf (Abbildung 4.8). Was Sie sehen, entspricht gleichnamigen Dateien auf Ihrer Festplatte, die Sie hier zum einfachen, direkten Zugriff vorfinden.

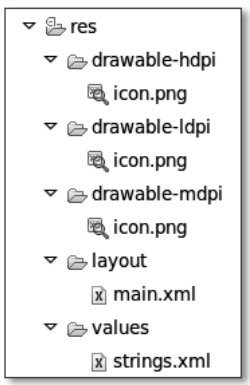



**Abbildung 4.8** Im Ressourcen-Baum des Projekts sind unterschiedliche Dateien einsortiert.

In den ersten drei Unterverzeichnissen geht es recht langweilig zu. Doppelklicken Sie ruhig auf die drei Dateien namens *icon.png*. Sie sehen alle gleich aus – es handelt sich um das Standard-App-Icon, das Sie bereits aus dem Startmenü Ihres Handys kennen. Beim genaueren Hinsehen werden Sie jedoch feststellen, dass die Bilder des grünen Roboters in unterschiedlichen Größen vorliegen: von 72 mal 72 über 48 mal 48 bis 36 mal 36 Pixel.

Die verschiedenen Bilder sind für Bildschirme mit unterschiedlichen Auflösungen vorgesehen. Auf älteren Geräten mit geringer Auflösung verwendet Android die Grafiken aus dem Verzeichnis *drawable-ldpi*. Dabei steht **ldpi** für **low dots per inch**, also geringe Auflösung. Sie erraten sicher leicht, dass **mdpi** für mittlere und **hdpi** für hohe Auflösung steht. Unterstützt wird übrigens auch noch **xhdpi** für extrem hoch aufgelöste Grafiken, allerdings legt der Android-Wizard ein zugehöriges Verzeichnis nicht von alleine an. Die unterschiedlich großen Grafiken erlauben es Android also, auf verschieden hoch aufgelösten Bildschirmen das Icon trotzdem in etwa der gleichen Größe darzustellen.

Darüber hinaus können Sie ein Verzeichnis *drawable* ohne Postfix anlegen. Darin sucht Android nach Grafiken, wenn in den anderen Verzeichnissen nichts Passendes zu finden ist, und skaliert sie entsprechend. Indem Sie bis zu fünf *drawable*-Verzeichnisse für Grafiken verwenden, können Sie Ihre App für einen ganzen Zoo von Geräten optimieren.

Diese Methode hat einen entscheidenden Vorteil, aber auch einen erheblichen Nachteil. Der Vorteil ist, dass Android-Geräte die für die vorhandene Auflösung nötigen Grafiken nicht selbst berechnen müssen (denn theoretisch genügt eine einzige Grafik). Auf schwach motorisierten Geräten ist das durchaus relevant. Der offensichtliche Nachteil: Sie müssen jede Grafik, die in Ihrer App vorkommen soll, in dreifacher Ausfertigung erzeugen. Das erinnert ein wenig an preußische Bürokratie – und angesichts immer stärkerer Geräte können Sie sich den Aufwand tatsächlich sparen. Erzeugen Sie einfach alle Grafiken in mittlerer Auflösung, legen Sie sie in ein neues Verzeichnis *drawable*, und löschen Sie die anderen.

Grafikdateien dürfen die Endungen *.png*, *.jpg* oder *.gif* besitzen, alle Buchstaben müssen kleingeschrieben werden, Leerzeichen sind nicht erlaubt. Ich empfehle Ihnen für grafische Elemente (ausgenommen Fotos) das PNG-Format (**Portable Network Graphics**), weil es effektvolle Transparenz-Effekte bei vertretbarer Größe ermöglicht. Passende Bilder können Sie mit allen gängigen Grafikprogrammen erzeugen. An kostenloser Software stehen beispielsweise **GIMP** und **Inkscape** zur Verfügung (auch auf der Buch-DVD).

### Generierte Dateien

Wenn Sie in einer App eine der Ressourcen wie Grafiken oder Layouts verwenden möchten, müssen Sie sie referenzieren. Sie benötigen also irgendein Identifikationsmerkmal, um Android mitzuteilen, welche Grafik Sie gerade benötigen.

Nun wäre eine naheliegende Möglichkeit, den Dateinamen zu verwenden. Nehmen wir an, Sie speichern eine Grafik unter dem Namen *mein_cabrio.png* und eine andere unter *tomate.png*. Dann könnten Sie die Grafik mit folgendem (fiktiven) Code darstellen:

```
bild_von_cabrio = loadImage("mein_cabrio.png");
bild_von_tomate = loadImage("tamote.png");
```

Diese Vorgehensweise hat einen entscheidenden Nachteil. Was passiert, wenn Sie sich beim Dateinamen vertippen? Wie Sie sehen, ist mir das in der zweiten Zeile bereits passiert. Verflixt.

Zunächst einmal passiert nichts weiter: Eclipse akzeptiert Ihren Code, Java kompiliert ihn, die App startet, und Sie sehen an der erwarteten Stelle Ihr Cabrio, aber anstelle einer Tomate nur gähnende Leere – wenn Sie Glück haben. Wenn Sie Pech haben, produziert Ihre App einen ordentlichen Crash, der erst bei genauem Hinsehen die Ursache offenbart: einen Vertipper in einem String.

Da Vertippen genauso untrennbar zum Entwicklerleben gehört wie Kaffee- oder Teekochen, haben sich die Android-Macher eine Lösung ausgedacht, die Ihnen das Leben deutlich leichter macht. Sie sorgt dafür, dass Tippfehler sofort auffallen, rot unterstrichen werden und die App gar nicht erst kompiliert und gestartet wird. Die Fehlerbehebung geht auf diese Weise viel schneller. Es dauert nur Sekunden, da Sie nicht extra die App kompilieren, auf das Gerät laden und starten müssen, von der Suche nach dem Vertipper ganz abgesehen.

Das kann offensichtlich nur funktionieren, wenn aus dem Dateinamen irgendwie ein Java-Objekt wird, denn nur dann kann der Compiler beurteilen, ob ein Objekt dieses Namens existiert oder nicht. Woher aber kommt dieses Objekt?

Um das zu verstehen, werfen wir einem Blick auf den Mechanismus, mit dem das Android Development Tool für Eclipse diesen Trick bewerkstelligt. Öffnen Sie die Projekteigenschaften über das Menü PROJECT • PROPERTIES. Wählen Sie im Einstellungsbaum links den Knoten BUILDERS aus, um zu sehen, welche Module Eclipse aufruft, wenn es im Hintergrund Ihre Eingaben in ausführbaren Java-Code verwandelt (Abbildung 4.9).

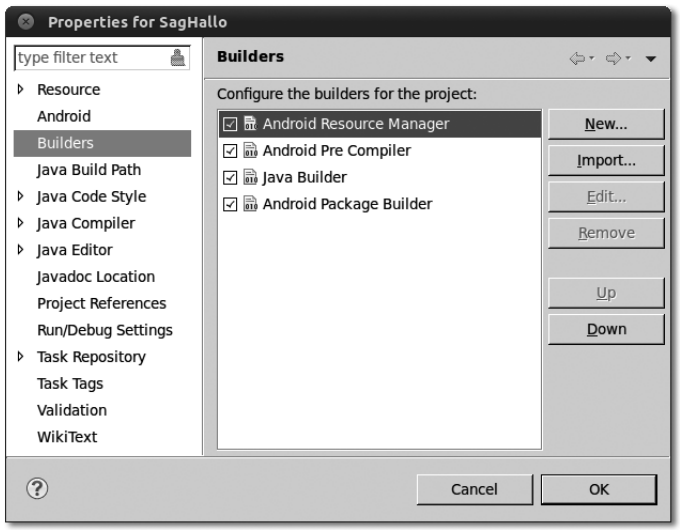



**Abbildung 4.9** Das ADT setzt zwei Android-spezifische Builder in Ihrem Projekt ein.

Immer wenn Sie an Ihrem Android-Projekt arbeiten, führt Eclipse der Reihe nach im Hintergrund die vier eingetragenen **Builder** aus. Der **Java Builder** kompiliert den Programmcode (wenn er es nicht kann, unterstreicht er Fehler in roter Farbe), der Rest ist spezifisch für Android-Projekte. Der **Package Builder** erzeugt das APK, also das App-Paket, das später auf dem Handy oder Emulator ausgeführt wird. Um Resource-Dateien kümmert sich der **Android Resource Manager** – und zwar noch *vor* dem **Java Builder**.

Der Resource Manager hat die Aufgabe, jegliche Änderung an Dateien im *res*-Verzeichnis zu überwachen. Er erstellt daraus Java-Programmcode, der für jede Grafik ein eindeutiges Objekt erzeugt. Glücklicherweise lässt sich der Resource Manager dabei ohne Weiteres auf die Finger schauen. Öffnen Sie im Package Explorer das Verzeichnis *gen*. Darin finden Sie ein Java-Package, das genau eine Datei enthält: *R.java*. Doppelklicken Sie auf diese Datei, um einen Blick darauf zu werfen:

```
/* AUTO-GENERATED FILE.  DO NOT MODIFY.
 *
 * This class was automatically generated by the
 * aapt tool from the resource data it found.  It
 * should not be modified by hand.
 */

package de.androidnewcomer.saghallo;
```

```
public final class R {
    public static final class attr {
    }
    public static final class drawable {
        public static final int icon=0x7f020000;
    }
    public static final class layout {
        public static final int main=0x7f030000;
    }
    public static final class string {
        public static final int app_name=0x7f040001;
        public static final int hello=0x7f040000;
    }
}
```

Diese Klasse mit dem handlichen Namen `R` enthält eine Reihe untergeordneter Klassen, die wiederum eine oder mehrere kryptische Konstanten enthalten – oder gar nichts, wie im Fall der Klasse `attr`.

**Der Modifzierer »final«**

Während Sie die Modifizierer `public` und `static` bereits kennen, sehen Sie in *R.java* zum ersten Mal einen weiteren Modifizierer, nämlich `final`.

Dieses `final` verhindert mehr als eine Zuweisung an das betreffende Attribut und wird daher gern für Konstanten verwendet, deren Wert sich ohnehin nie ändert.

Theoretisch könnte man auf das `final` verzichten. Allerdings sind die Attribute in `R.java` alle `public`, sie dürften also von anderen Klassen prinzipiell verändert werden. Da Sie aber darauf vertrauen möchten, dass wirklich das Layout *main* angezeigt wird, wenn Sie es referenzieren, und nicht das Layout eines, sagen wir, Trojaners zum Klauen von Kreditkartennummern, ist es sicherer, ohnehin konstante Werte als `final` zu deklarieren.

Nicht nur beim Programmablauf, sondern schon beim Kompilieren prüft Java die Verwendung eines `final`-Objekts und quittiert Zuweisungsversuche mit einem Fehler:

```
final int konstante = 1;
konstante=2; // Compilerfehler
```

Vergleichen Sie die fett hervorgehobenen Klassennamen und Attribute mit dem Inhalt des Verzeichnisses *res*. Sie stellen fest, dass es für jede Datei in einem Unterverzeichnis von *res* einen passenden Eintrag in einer Unterklasse von `R` gibt. Im Fall von *drawable* gibt es zwar mehrere Verzeichnisse, aber wie Sie bereits wissen, enthalten die bloß Grafikdateien mit unterschiedlichen Auflösungen derselben Bilder. Daher genügt eine Klasse `drawable` in `R`.

Die auf den ersten Blick kryptischen Zahlenkombinationen hinter den =-Zeichen haben einen großen Vorteil: Sie müssen sich nicht darum kümmern. Merken Sie sich lediglich für später, dass das Präfix `0x` eine Hexadezimalzahl einleitet, die nicht besonders geheimnisvoll ist, sondern lediglich eine Krücke für Programmierer, weil Computer nicht gerne mit zehn Fingern rechnen, sondern lieber mit sechzehn.

Entscheidend ist, dass Sie dank der automatisch erzeugten Klasse `R` Ressourcen mit Java-Bezeichnern referenzieren können. Beispielsweise hört das App-Icon auf `R.drawable.icon`. Legen Sie also Ihre Bilder *mein_cabrio.png* und *tomate.png* in eines der *drawable*-Verzeichnisse, erzeugt der Android Resource Manager automatisch passende Einträge in *R.java*, die Sie direkt verwenden können:

```
image1 = R.drawable.mein_cabrio;

image2 = R.drawable.tamote; // wird rot unterstrichen
```

Wenn Sie sich nun vertippen, unterstreicht Eclipse sofort die fehlerhafte Zeile, und Sie wissen, woran Sie sind. Mehr noch: Mit nahezu magischen Mitteln bietet Eclipse Ihnen bereits beim Eintippen des ersten Buchstabens des Worts **tomate** an, den Rest selbst zu vervollständigen. Wenn sich die Autovervollständigung nicht von allein öffnet, drücken Sie Strg + Leertaste, um sie aufzurufen.

**Zur Erinnerung: Kommentare**

Wenn Sie guten Java-Programmcode schreiben, dann kann jeder andere Programmierer ihn ohne Weiteres lesen und seine Bedeutung verstehen.

Das ist allerdings ein weit verbreiteter Irrtum.

In der Realität verfügt nicht jeder Entwickler über den gleichen Ausbildungsstand. Manchmal würde er selbst etwas ganz anders programmieren, oder er hat Tomaten auf den Augen und braucht einen Gedankenanstoß, um Code zu erfassen. Deshalb sind Kommentare im Quelltext sehr wichtig. Kommentare schreiben Sie hinter zwei Schrägstriche, wenn sie nicht länger als eine Zeile sind:

```
Auto c = new Auto(); // c bedeutet Cabrio
```

Verwenden Sie Schrägstriche und Sterne für Kommentare, die länger sind als eine Zeile:

```
class Auto {
/* Diese Klasse stellt ein Auto dar.
Derzeit ist sie unfertig. */
}
```

Wichtig ist aber nicht nur, *wie* Sie Kommentare schreiben, sondern auch *was* Sie kommentieren. Das Cabrio-Beispiel stellt einen wirklich überflüssigen Kommentar dar: Hätte der Programmierer sein Objekt einfach `cabrio` genannt statt bloß `c`, wäre

der Kommentar überflüssig und auch der folgende Programmcode sicher verständlicher. Im zweiten Beispiel ist immerhin die zweite Zeile relevant, weil sie einem Leser die Frage beantwortet, warum die Klasse so leer daherkommt.

Die Faustregel lautet: Kommentieren Sie nur dort, wo selbst der selbsterklärendste Programmcode nicht auf Anhieb verständlich ist. Ansonsten schreiben Sie einfach selbsterklärenden Programmcode!

Beachten Sie, dass die Dateiendungen grundsätzlich weggelassen werden. Daraus folgt, dass Sie keine zwei gleichnamigen Bilder wie zum Beispiel *tomate.png* und *tomate.jpg* verwenden können. Aber keine Sorge: Wenn Sie das versuchen, macht Sie der Resource Manager in freundlichem Rot darauf aufmerksam.

Neben den Grafiken in *drawable* beobachtet der Resource Manager offenbar Layouts (zu denen kommen wir gleich), Strings und Attribute – aber noch einiges mehr, das in der aktuellen *R.java* nicht auftaucht, weil es im Projekt SagHallo nicht vorkommt. In späteren Kapiteln ändert sich diese Lage, aber Sie werden die erzeugten Referenzen stets verwenden, ohne über den im Hintergrund werkelnden Resource Manager nachdenken zu müssen.

## 4.3 Benutzeroberflächen bauen

Zugegebenermaßen sieht Ihre erste App nicht so beeindruckend aus wie Angry Birds oder Google Maps. Allerdings haben wir uns bisher auf Sprachausgabe konzentriert, nicht auf Bildschirmdarstellung.

Sie ahnen bereits, dass Sie einen Blick in die Ressourcen des Projekts werfen müssen, um mehr über die Gestaltung von Benutzeroberflächen herauszufinden. Immerhin hat Ihre SagHallo-App bereits eine, die bloß schwarz ist und einen relativ sinnlosen Spruch ausgibt.

### Layout bearbeiten

Machen Sie sich auf die Suche nach der zuständigen Komponente. Öffnen Sie im Package Explorer die Datei *main.xml* im Verzeichnis *layout* durch einen Doppelklick. Es öffnet sich der Layout-Editor, der (manchmal nach kurzer Ladezeit) das Layout und eine Reihe von Bedienelementen drum herum anzeigt (Abbildung 4.10).

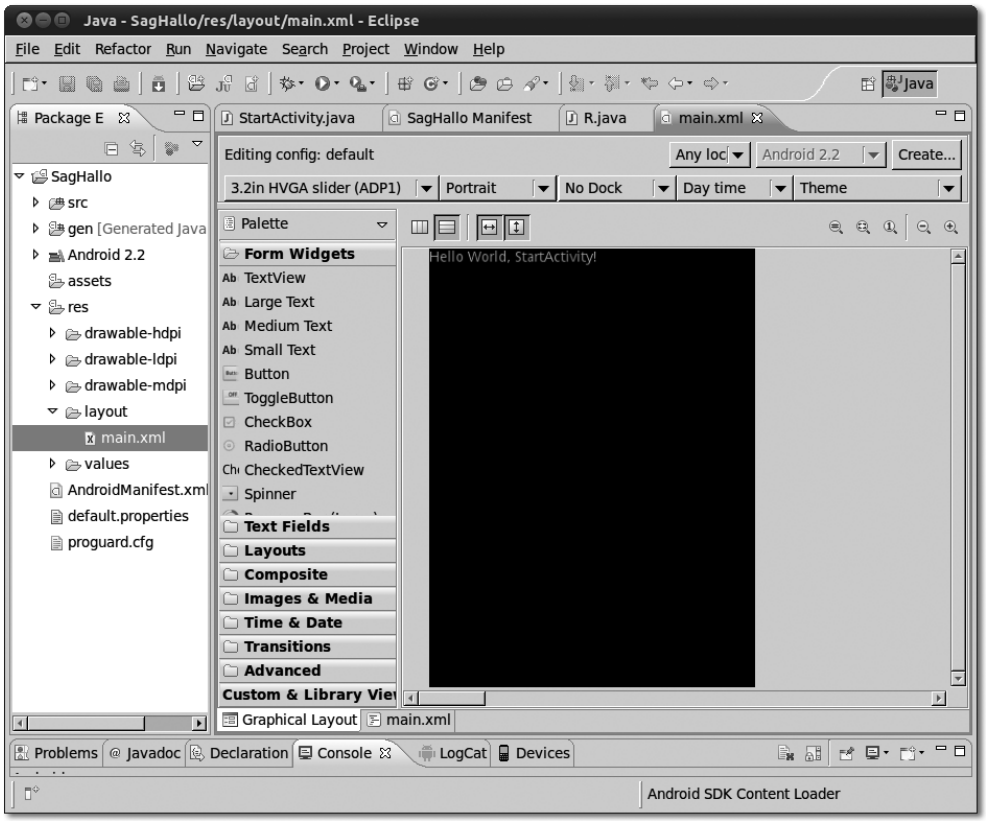



**Abbildung 4.10** Der Layout-Editor sieht Schwarz. Noch.

Warum aber zeigt Ihre App genau dieses Layout nach dem Start an? Sie ahnen, dass die App das nicht durch Telepathie schafft, sondern ganz profan die passende Methode aufrufen muss.

Rufen Sie sich eine bestimmte Zeile aus der `StartActivity` in Erinnerung:

```
setContentView(R.layout.main);
```

Im Zusammenhang mit dem letzten Kapitel über die generierte Klasse `R` erklärt sich diese Zeile jetzt fast von selbst: Als Ansicht (**View**) für die `StartActivity` wird hier die Datei *main.xml* im Verzeichnis *layout* festgelegt. Der Resource Manager hat diese Datei als fehlerfreie Layout-Beschreibung identifiziert und das passende Attribut `layout.main` in der Klasse `R` angelegt. Beachten Sie, dass wie bei Grafiken auch hier die Dateiendung *.xml* entfällt.

Dass hinter `R.layout.main` irgendeine Hexadezimalzahl steckt, muss uns nicht interessieren. Wenn Sie beispielsweise eine weitere Layout-Datei namens *my_nice_*

*view.xml* anlegen, könnten Sie sich diese mit einer einfachen Änderung anstelle der langweiligen *main.xml* anzeigen lassen:

```
setContentView(R.layout.my_nice_view);
```

Als Sie Ihr Layout gespeichert haben, hat der Resource Manager in der Klasse `R` das Attribut `layout.my_nice_view` erzeugt, und zwar mit einem anderen Hexadezimalwert als bei `layout.main`.

Sie ahnen schon, dass Sie auf diese Weise zwischen verschiedenen Bildschirmdarstellungen hin- und herschalten können, allerdings benötigen wir diese Funktion erst später. Lassen Sie uns zuerst einen genaueren Blick auf die Möglichkeiten des Layout-Editors werfen. Es ist wichtig, einen Überblick über die existierenden visuellen Komponenten zu erhalten, damit Sie wissen, wie Sie eine Benutzeroberfläche zusammenstellen können.

Klicken Sie sich durch die verschiedenen Rubriken in der Palette links von der Vorschau. Sie sehen da zunächst die Form Widgets, also Eingabeelemente wie Button und CheckBox, aber auch ProgressBar und einfache TextViews (es wird Sie nicht überraschen, dass der graue Text in der Vorschau ein solcher TextView ist). All diese Widgets haben Sie schon an allen möglichen Stellen in Android-Apps kennengelernt. Hier warten sie darauf, in ein Layout eingebaut zu werden.

Sie haben sicher noch im Hinterkopf, dass unser momentanes Ziel ist, der Sprachausgabe-App ein Texteingabefeld zu verpassen, damit Sie nicht für jede Änderung des zu sprechenden Satzes das ganze Programm neu übersetzen müssen. Leider finden Sie das gewünschte Eingabefeld nicht in der Rubrik Form Widgets – schalten Sie um auf Text Fields, um es zu sehen.

Dort bietet der Editor Ihnen eine ganze Reihe unterschiedlicher Textfelder (Edit-Text) an, zum Beispiel solche, in die der Benutzer ausschließlich Zahlen eingeben kann (Number), Passwort-Felder oder Felder für E-Mail-Adressen. Die Darstellung dieser Felder unterscheidet sich in der App nur geringfügig, aber entscheidend: Passwort-Felder verstecken Ihre Eingaben durch Sternchen, bei E-Mail-Adressfeldern zeigt die eingeblendete Bildschirmtastatur auf jeden Fall das @-Zeichen.

Für die Sprachausgabe-Eingabe genügt das einfachste Textfeld namens Plain Text. Ziehen Sie es mit der Maus aus der Palette hinüber in die Vorschau, sodass es unterhalb des vorhandenen Standardsatzes liegt.

Um die Sprachausgabe auszulösen, fehlt noch ein Button. Also schalten Sie wieder zurück auf die Rubrik Form Widgets. Ziehen Sie einen Button hinüber in die Vorschau, und legen Sie ihn unterhalb des Textfeldes ab (Abbildung 4.11).

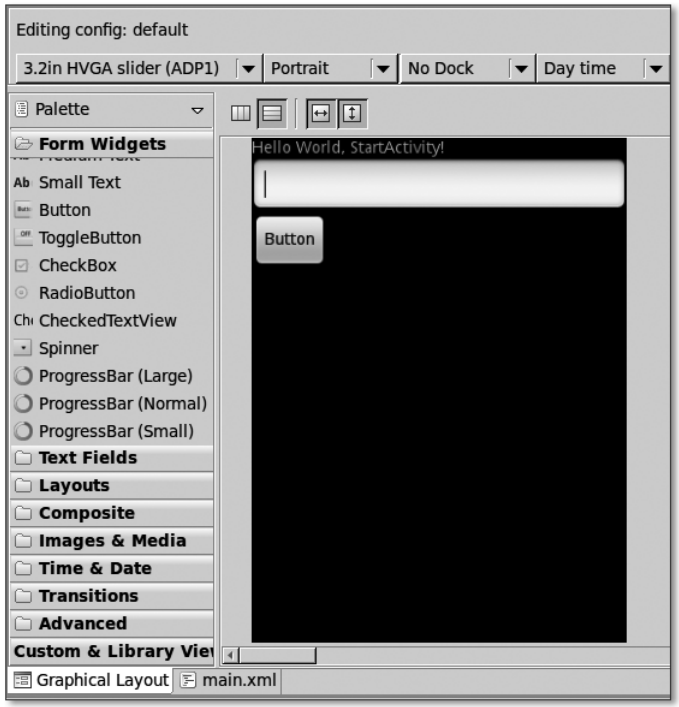



**Abbildung 4.11** Textfeld und Button bereichern das »main«-Layout.

Speichern Sie Ihre Änderungen mit Strg + S. Sie können die App jetzt noch einmal starten, um das neue Layout auf dem Handy zu begutachten. Zwar können Sie einen beliebigen Text in das neue Textfeld eingeben, aber der Button tut rein gar nichts – kein Wunder, wir haben ihm auch noch nichts dergleichen aufgetragen.

Das heben wir uns für den nächsten Abschnitt auf. Lassen Sie uns zunächst die im Moment recht unpassende Beschriftung ändern.

Klicken Sie mit der rechten Maustaste auf den Text »Hello World, StartActivity!«, und wählen Sie im Popup-Menü Edit Text. Daraufhin erscheint der Resource Chooser (Abbildung 4.12).

Vielleicht haben Sie erwartet, einfach den sichtbaren Text ändern zu können, aber die Sache gestaltet sich ein klein wenig komplizierter.

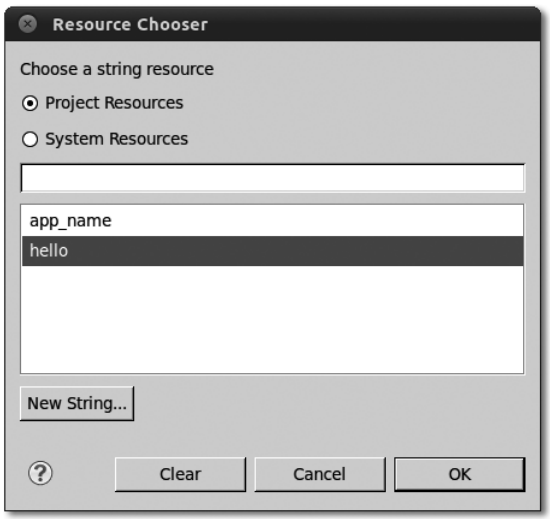



**Abbildung 4.12** Der Resource Chooser erlaubt es, einen vorhandenen String auszuwählen oder einen neuen zu erzeugen.

## String-Ressourcen

Immer wenn es um Texte in Benutzeroberflächen geht, kommen die String-Ressourcen ins Spiel. Denn der darzustellende Text steht nicht direkt in der Layout-Datei, sondern anderswo – im Layout steht nur ein Verweis. Das hat den Vorteil, dass gleiche Texte in der App nicht mehrfach vorhanden sind, sondern nur einmal. Es existiert bloß überall, wo der Text benötigt wird, der gleiche Verweis. Das spart Speicher, der auf Smartphones trotz allen Fortschritts immer noch knapp ist. Ein anderer Vorteil ist die leichte Übersetzbarkeit. Denn alle Strings für eine Sprache stehen in einer einzigen Datei. Schicken Sie diese Datei einem Übersetzer, speichern Sie sie zusätzlich in Ihrem Projekt an der richtigen Stelle, und schon ist Ihre App mehrsprachig. Android wählt beim Start der App automatisch die richtige String-Datei und löst die Verweise korrekt auf.

Im Fall von »Hello World ...« ist der Name des Verweises offenbar `hello`. Lassen Sie uns für die richtige Beschriftung einen neuen Verweis samt String anlegen. Klicken Sie auf den Button New String. Es erscheint ein Dialog, der Ihnen beim Anlegen eines neuen Strings hilft. Das wirkt auf den ersten Blick komplizierter, als es ist, denn auch dieser Dialog bringt einige Funktionen mit, die Sie im Augenblick nicht benötigen. Geben Sie in das erste Textfeld den Text ein, der später angezeigt werden soll, und darunter den Namen des neuen Verweises, so wie in Abbildung 4.13.

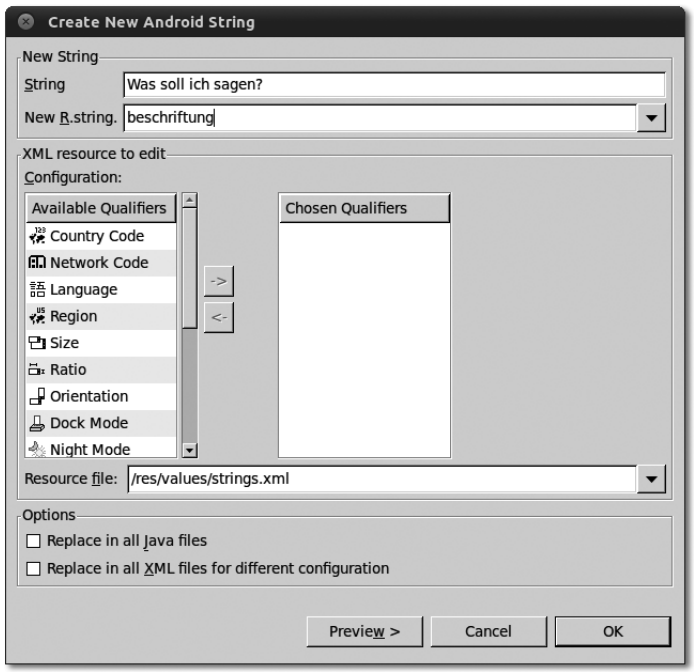



**Abbildung 4.13** Ein neuer Android-String erblickt das Licht der App.

Sobald Sie den Dialog mit dem OK-Button schließen, gelangen Sie zurück zum Resource Chooser, in dem der neue Verweis bereits ausgewählt ist. Sie müssen nur noch mit OK bestätigen, und in der Vorschau erscheint der neue Text.

Wiederholen Sie die Prozedur für den hinzugefügten Button, indem Sie mit der rechten Maustaste klicken und wieder Edit Text auswählen. Erzeugen Sie einen weiteren String wie zum Beispiel »Sag es« mit einem Verweis namens `sag_es`. Als Verweis können Sie nur gültige Java-Bezeichner eingeben.

Wenn Sie Ihre Arbeit mit Strg + S gespeichert haben, können Sie die App ein weiteres Mal starten, um sich davon zu überzeugen, dass sie wie erwartet ausschaut.

Doch was haben Eclipse und Android im Hintergrund alles getan, während Sie an diversen Dialogen hantiert haben?

Um diese Frage aufzuklären, öffnen Sie erneut die generierte Datei *R.java*. Sie sehen jetzt eine ganze Reihe neuer Einträge (hier fett gedruckt).

```
public final class R {
    public static final class attr {
    }
    public static final class drawable {
        public static final int icon=0x7f020000;
    }
    public static final class id {
        public static final int button1=0x7f050001;
        public static final int editText1=0x7f050000;
    }
    public static final class layout {
        public static final int main=0x7f030000;
    }
    public static final class string {
        public static final int app_name=0x7f040001;
        public static final int beschriftung=0x7f040002;
        public static final int hello=0x7f040000;
        public static final int sag_es=0x7f040003;
    }
}
```

Sie sehen zunächst, dass die beiden neu eingefügten visuellen Komponenten je eine `id`-Konstante erhalten haben. Jede Komponente, die später im Programmcode in irgendeiner Weise verwendet werden soll, benötigt eine ID. Die Beschriftung hatte von Anfang an keine ID, und wir haben ihr keine gegeben, allerdings ist es auch unwahrscheinlich, dass wir eine brauchen. Die IDs des Buttons (`button1`) und des Eingabefeldes (`editText1`) werden Sie aber auf jeden Fall demnächst verwenden.

Etwas weiter unten sind zwei String-Konstanten hinzugekommen. Das sind genau die beiden neuen Verweise, die Sie gerade mit dem Resource Chooser erzeugt haben. Sie sehen, dass diese Verweise wiederum nichts anderes sind als Zahlencodes, um die Sie sich nicht weiter kümmern müssen.

Wo aber sind die eigentlichen Texte versteckt?

Suchen Sie die Datei *strings.xml* im Verzeichnis *res/values*. Öffnen Sie diese Datei in Eclipse, und Sie erhalten eine Liste der existierenden String-Verweise. Sie können jeden der Verweise anklicken und auf der rechten Seite verändern (Abbildung 4.14).

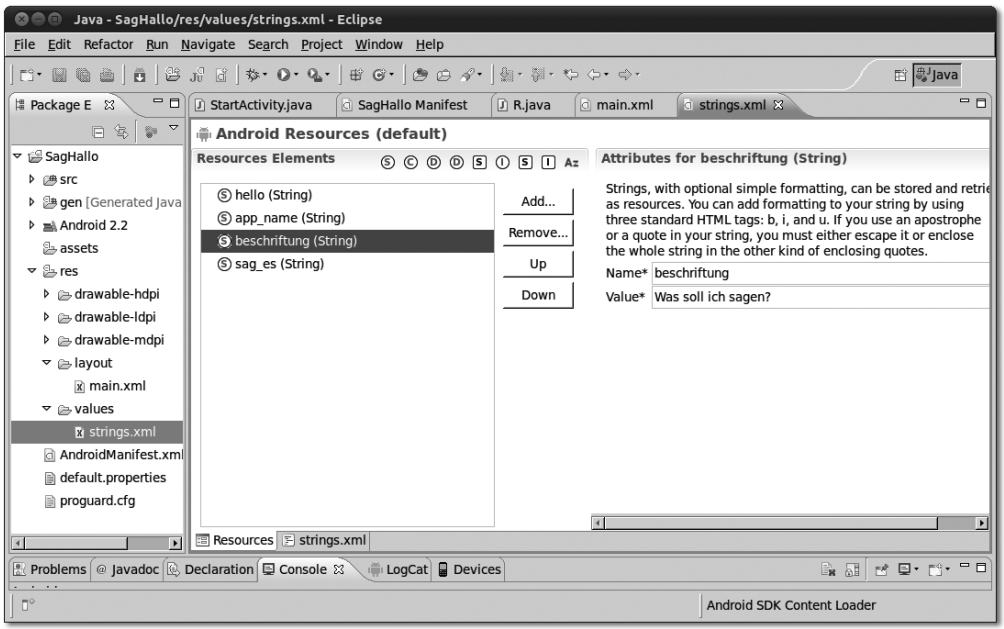



**Abbildung 4.14** Die Strings sind in der Datei »strings.xml« versteckt.

### XML

Wenn Sie lieber mit der Tastatur als mit der Maus arbeiten, können Sie sich jederzeit die Quelltexte der Android-Ressourcen anschauen. Klicken Sie beispielsweise auf die Registerkarte, die mit STRINGS.XML beschriftet ist, um zu sehen, wie Android die Strings intern verwaltet.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
 <string name="hello">Hello World, StartActivity!</string>
 <string name="app_name">SagHallo</string>
 <string name="beschriftung">Was soll ich sagen?</string>
 <string name="sag_es">Sag es</string>
</resources>
```

Sie können ohne Weiteres direkt in dieser Datei Änderungen vornehmen, wenn es Ihnen zu umständlich ist, sich mit der Maus durch eine lange Liste von String-Verweisen zu scrollen. Am Ende ist es Geschmackssache – vergessen Sie aber nicht, dass Sie in einer XML-Datei leichter etwas kaputtmachen können als in einer komfortablen Benutzeroberfläche.

Bevor Sie jetzt dem Button beibringen, für die gewünschte Sprachausgabe zu sorgen, begleiten Sie mich auf eine Besichtigungstour durch die visuellen Komponenten, die Android anbietet.

## Layout-Komponenten

Bislang haben Sie dem Layout *main.xml* zwei visuelle Komponenten auf die einfachstmögliche Weise hinzugefügt. Die Positionen, die Ihre insgesamt drei Komponenten (TextView, EditText und Button) auf dem Screen einnehmen, erinnern an eine Lasagne: Sie stehen in Schichten übereinander, der Button ist ganz unten, und der »Was soll ich sagen?«-Schriftzug bildet ganz oben den Käse.

Lassen Sie uns identifizieren, wer für diese vertikale Anordnung verantwortlich ist. Schalten Sie in Eclipse den **View** namens Outline zu. Wählen Sie dazu Window • Show View • Outline. Dies bereichert die Eclipse-Oberfläche um eine Ansicht der Baumstruktur Ihres Layouts (Abbildung 4.15). Soweit Komponenten über eine ID verfügen (wie `editText1` und `button1`), wird diese angezeigt, ansonsten nur der Typ der Komponente (z. B. TextView**).**

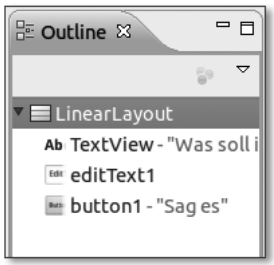



**Abbildung 4.15** Der »Outline«-View zeigt das Layout als Baumstruktur.

Sie sehen, dass Ihre drei Komponenten in der erwarteten Reihenfolge unter einem Elternknoten vom Typ LinearLayout angeordnet sind. Dabei handelt es sich um eine Layout-Komponente, die selbst unsichtbar bleibt, aber einen bestimmten freien Bereich zur Verfügung stellt, den andere Komponenten bevölkern können. In diesem Fall füllt das LinearLayout den gesamten Bildschirm aus.

Die Tatsache, dass dieses bestimmte LinearLayout die darunter einsortierten Komponenten untereinander anzeigt, ist natürlich konfigurierbar. Öffnen Sie einen weiteren View, diesmal Properties. Sie finden ihn im Menü unter Window • Show View • Other... • Properties. Am besten klinken Sie diese Ansicht in Eclipse unterhalb der zuvor geöffneten Outline an. Klicken Sie dann den LinearLayout-Knoten an, sodass sich alle Properties dieser Komponente offenbaren. Sie werden sehen, dass es eine ganze Menge Properties gibt, von denen aber die wenigsten gefüllt sind. Es würde den Rahmen dieses Abschnitts sprengen, alle Properties zu erläutern; ich werde bis auf zwei Ausnahmen jeweils die relevanten vorstellen, wenn sie benötigt werden.

Suchen Sie in der Liste der Properties nach GRAVITY. Wählen Sie in der Spalte VALUE den Wert `center`, und drücken Sie ↵. Sie haben damit dem LinearLayout befohlen, seinen Inhalt zentriert anzuordnen, und zwar sowohl vertikal als auch horizontal.

**Links, rechts, zentriert ...**

Sie können die Ausrichtung von Komponenten mit der GRAVITY-Eigenschaft eines Layouts unabhängig voneinander horizontal und vertikal einstellen. Wie Sie bereits gesehen haben, bewirkt `center` eine horizontal und vertikal zentrierte Anordnung. Wollen Sie nur horizontal zentrieren, verwenden Sie `center_horizontal`. Sie erraten leicht, dass es außerdem den möglichen Wert `center_vertical` gibt.

Abgesehen vom Standard `left`, gibt es natürlich `right`, außerdem `top` und `bottom`.

Wenn Sie vertikal und horizontal unabhängig voneinander ausrichten möchten, können Sie zwei Werte mit dem `|` kombinieren:

`center_horizontal | top` richtet die Elemente beispielsweise horizontal zentriert am oberen Rand aus.

Außerdem gibt es `fill_horizontal` und `fill_vertical` – seien Sie vorsichtig mit diesen Werten. Wie man sie genau einsetzt, erkläre ich später.

Bedauerlicherweise verhält sich Ihr TextView nicht wie erwartet. Wenn Sie ihn anklicken, sehen Sie es am blauen Markierungsrahmen: Er füllt die gesamte Breite des Layouts aus. Der eigentliche Text ist innerhalb des TextViews linksbündig angeordnet, denn Sie haben ihm nichts Gegenteiliges aufgetragen.

Es gibt zwei Möglichkeiten, für Abhilfe zu sorgen. Ändern Sie die Breite des TextViews auf die minimale Größe. Bei dieser Gelegenheit zeige ich Ihnen eine zweite Möglichkeit, ein Property zu ändern – manchmal geht's so sogar schneller. Klicken Sie mit der rechten Maustaste auf den TextView, und schalten Sie im Popup-Menü die LAYOUT WIDTH von `match_parent` auf `wrap_content` um. Die blaue Umrandung nimmt dann sofort die kleinste Größe ein, gleichzeitig greift die gewünschte zentrierte Ausrichtung.

Die Breitenangabe `wrap_content` sorgt grundsätzlich dafür, dass die Komponente nur so breit ist, dass ihr Inhalt vollständig angezeigt wird. Im Gegensatz dazu bedeutet `match_parent`, dass die Breite bis auf jene der jeweils übergeordneten Komponente ausgedehnt wird. Es wird Sie kaum überraschen, wenn ich Ihnen verrate, dass dasselbe analog für Höhenangaben gilt. Bei der Arbeit an Layouts ist es oft erforderlich, Breite und Höhe auf diese Weise umzuschalten, daher stellt Ihnen das Android Development Tool zwei kleine Buttons oberhalb des Vorschaubereichs zur Verfügung, mit denen Sie zwischen den beiden Optionen hin- und herschalten können.

Experimentieren Sie ruhig, indem Sie beispielsweise die Breite des Eingabefelds `editText1` auf minimale Breite stellen (machen Sie das anschließend bitte wieder rückgängig) oder indem Sie den `button1` auf maximale Breite ausdehnen (das können Sie so lassen). Das Resultat können Sie natürlich gleich auf dem Handy ausprobieren, indem Sie die App einmal mehr starten.

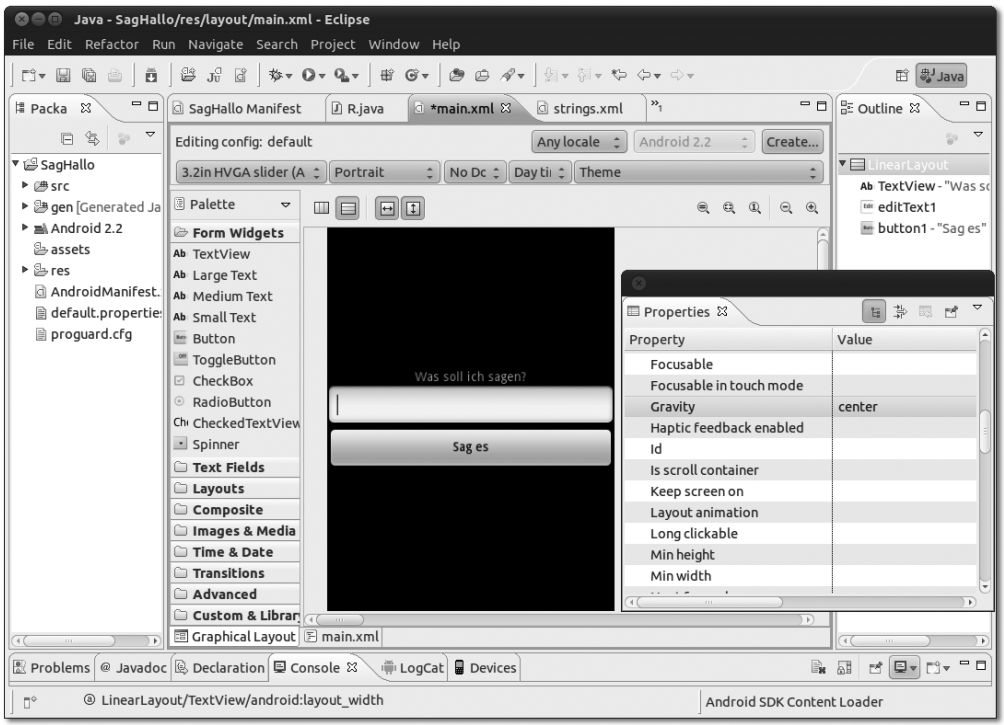



**Abbildung 4.16** Der Properties-View ist die Schaltzentrale der Layout-Komponenten.

Um komplexe Layouts zu basteln, werden Sie später haufenweise LinearLayouts (und andere) ineinander verschachteln. Halten Sie sich vor Augen, warum Sie hier nicht mit numerischen Koordinaten arbeiten, um Komponenten zu positionieren: Android-Geräte haben unterschiedlich hohe Auflösungen, und Ihre App soll möglichst ohne spezielle Vorkehrungen auf möglichst vielen Geräten akzeptabel aussehen – noch dazu im Hochkant- und Breitbild-Format.

**Vorschauvarianten**

Sie haben sicher bereits die zahlreichen Dropdown-Menüs oberhalb der Layout-Vorschau bemerkt. Dabei handelt es sich um eine vereinfachte Simulation verschiedener Geräte-Displays.

**Vorschauvarianten**

Das erste Dropdown-Menü schaltet zwischen verschieden großen Displays um. Versäumen Sie nicht, Ihre Layouts schon hier auf dem kleinsten Screen (2,7 Zoll QVGA) zu begutachten – wenn Sie ungünstig gestalten, kann es leicht passieren, dass auf so kleinen Displays Komponenten gar nicht sichtbar sind! Im schlimmsten Fall wird Ihre App so auf kleinen Geräten unbenutzbar.

Das zweite Dropdown-Menü schaltet zwischen Hochkant- und Breitbild-Ansicht um (Portrait/Landscape). Oft haben Sie keine Chance, ein anspruchsvolleres Layout für beide Modi ansehnlich zu gestalten. Daher müssen Sie entweder zwei separate Layout-Dateien erzeugen oder Android im Manifest befehlen, eine Activity ausschließlich in einem der beiden Modi zu betreiben.

Spielen Sie mit den anderen Dropdowns; beachten Sie dabei auch die Möglichkeit, an dieser Stelle die Lokalisierung (Locale) der Vorschau umzuschalten. Auf diese Weise können Sie prüfen, ob Strings in der richtigen Sprache erscheinen.

Nutzen Sie die Vorschaukonfiguration, damit Sie nicht für alle möglichen Kombinationen virtuelle Geräte erstellen müssen, denn der Test in X Emulatoren dauert natürlich viel länger als die Umschaltung hier in der Vorschau.

Klappen Sie in der Palette die Layouts-Rubrik auf. Abgesehen vom horizontalen und vertikalen LinearLayout, finden Sie hier noch andere Layout-Komponenten; allerdings ist das LinearLayout bei Weitem das wichtigste. Auf die anderen gehe ich jeweils ein, wenn sie benötigt werden.

### Weitere visuelle Komponenten

Die noch nicht angetasteten Rubriken der Palette verbirgt eine Reihe spannender und mächtiger Komponenten. Im Detail werden wir sie dort besprechen, wo sie zum ersten Mal benötigt werden. Für den Moment genügt ein kurzer Überblick.

- `ListView`: Immer wenn Sie mit dem Finger durch eine Liste gleichartiger Komponenten scrollen, haben Sie einen ListView vor sich.
- `ScrollView`/`HorizontalScrollView`: Nicht genug Platz auf dem Screen? Der ScrollView bietet einen scrollbaren Bereich beliebiger Höhe, den Sie meist mit einem vertikalen LinearLayout ausfüllen, weil der ScrollView selbst nur ein Element enthalten darf. Der HorizontalScrollView funktioniert analog.
- `WebView`: Zeigen Sie eine Webseite ohne externen Browser direkt in Ihrer App, indem Sie einen WebView einbauen.
- `ImageView`: Bilder beleben jede App. Der ImageView zeigt Bilder aus den Ressourcen oder anderen Quellen an.
- `ImageButton`: Verschönern Sie Buttons, indem Sie mit ImageButtons Text und Bild kombinieren.

- `VideoView`: Vermutlich haben Sie ohnehin die Absicht, für Ihr nächstes Game einen Trailer zu produzieren. Sie können ihn mit dem VideoView ins Spiel einbinden.
- `TimePicker`/`DatePicker`: Apps, die mit Zeit oder Datum hantieren, freuen sich, dass Android fertige Komponenten zur Verfügung stellt, die sich auch mit Tricks wie Schaltjahren und Sommerzeit bestens auskennen.
- `SurfaceView`: Unter den zahlreichen Komponenten, die in der Rubrik mit dem vielversprechenden Namen ADVANCED auf ihre Verwendung warten, nenne ich zunächst den `SurfaceView`, weil er unter anderem dazu dienen kann, ein Live-Bild von der eingebauten Kamera darzustellen. Wir werden später in diesem Buch darauf zurückkommen.

Soweit die verfügbaren Komponenten nicht in diesem Buch besprochen werden, finden Sie mehr oder weniger ausführliche Beschreibungen in der Dokumentation unter *http://developer.android.com* sowie Beispiele im Android SDK.

## 4.4 Buttons mit Funktion

Sie haben ein ansehnliches Layout gebastelt und damit den ersten Schritt getan, damit die Sprachausgabe-App nicht immer bloß dasselbe sagt. Lediglich der neue Button tut noch nicht, was er soll. Es wird Zeit, dies zu ändern.

### Der OnClickListener

Öffnen Sie erneut die Datei *StartActivity.java*, denn ohne Java-Code können wir den Sprech-Button nicht zur Mitarbeit überreden.

Halten Sie sich die nötige Änderung im Ablauf der App vor Augen: Bisher initialisiert die App beim Starten die Sprachausgabe und spricht sofort danach den festgelegten Text.

Die gewünschte neue Verhaltensweise sieht wie folgt aus:

- Sprachausgabe initialisieren
- danach darauf warten, dass der Benutzer den Button anklickt
- den Text aus dem Eingabefeld sprechen

Der zweite Schritt ist hierbei der entscheidende, denn die App selbst bestimmt nicht mehr, *wann* die Sprachausgabe erfolgt – der Benutzer tut das. Aber wie erfährt die App, wann der Benutzer auf den Button klickt?

Erinnern Sie sich an die Funktionsweise der Sprachausgabe-Engine: Sie teilt der App mit, wann sie mit der Initialisierung fertig ist, denn erst dann darf die

Methode `speak()` aufgerufen werden. Dazu haben Sie eine Methode `onInit()` in die `StartActivity` eingebaut, die von der Sprachausgabe-Engine im richtigen Moment aufgerufen wurde.

Das Warten auf den Knopfdruck funktioniert analog: Sie teilen dem Button mit, dass er einen `OnClickListener` verwenden soll. Dabei handelt es sich um ein Interface, das die `StartActivity` implementieren muss. Das bedeutet zwei Dinge. Erstens erklärt die Klasse `StartActivity`, dass Sie das Interface `OnClickListener` implementiert:

```
public class StartActivity extends Activity implements OnInitListener,
OnClickListener {
...
}
```

Beim obligatorischen Organisieren des Imports fragt Eclipse diesmal bei Ihnen nach, weil es zwei Interfaces gibt, die `OnClickListener` heißen. Wählen Sie `android.view.View.OnClickListener`.

Das Interface `OnClickListener` definiert eine Methode namens `onClick()`, die Ihre Klasse `StartActivity` jetzt implementieren muss, um der `implements`-Erklärung zu genügen. Verschieben Sie die Zeile mit der Sprachausgabe aus der `onInit()`-Methode in die neue `onClick()`-Methode:

```
@Override
public void onClick(View view) {
   tts.speak("Hallo!", TextToSpeech.QUEUE_FLUSH, null);
}
```

Prüfen Sie an dieser Stelle, ob Eclipse keine Stellen rot markiert, ansonsten ist der Code nicht kompilierbar, weil Sie irgendwo einen Fehler eingebaut haben.

**Zur Erinnerung: Interfaces**

Indem die Klasse `StartActivity` sagt, dass sie `OnClickListener` implementiert, verpflichtet sie sich dazu, die nötige Methode `onClick()` anzubieten. Das ähnelt abstrakten Methoden in einer Basisklasse, allerdings kann eine Klasse nur von einer Basisklasse erben, aber beliebig viele Interfaces implementieren.

Stellen Sie sich Interfaces vor wie Steckverbindungen an Ihrem Rechner: Wenn der PC über keinen HDMI-Ausgang verfügt, implementiert er das betreffende Interface nicht, und Sie können keinen HD-Fernseher anschließen. Die Methode »Stelle Film auf meinem Fernseher dar« ist im Rechner nicht implementiert.

Man könnte auch ein Analogon zum Thema »Schnittstellen« zwischen den Geschlechtern heranziehen, aber das erspare ich uns lieber.

Als Nächstes müssen Sie dem Button befehlen, dass er die laufende `StartActivity` als seinen `OnClickListener` verwendet, denn sonst passiert beim Knopfdruck immer noch nichts. Überlegen Sie, wo die richtige Stelle ist, um diese Verknüfung herzustellen.

In der Methode `onCreate()`? Nein, denn besonders schnelle Benutzer könnten dann den Button anklicken, bevor die Sprachausgabe initialisiert ist.

Der richtige Ort ist also die `onInit()`-Methode. Dort holen Sie sich zuerst eine Referenz auf den Button anhand dessen ID, die `button1` lautet:

```
Button button = (Button) findViewById(R.id.button1);
```

Die überaus häufig verwendete Methode `findViewById()` entstammt der Basisklasse `Activity` und beschafft Ihnen eine Referenz auf eine gewünschte Komponente anhand deren Konstante aus der automatisch generierten Klasse `R.java`. Dazu durchsucht die Methode das aktuelle Layout, das mit `setContentView()` festgelegt wurde. Beachten Sie daher, dass Sie `findViewById()` nie vor `setContentView()` aufrufen dürfen.

Vorsicht: Wenn die Methode nichts Passendes findet, knallt's. Glücklicherweise kann dank des Android Resource Managers, der Ihnen die Konstante namens `R.id.button1` zur Verfügung stellt, in diesem Fall nichts schiefgehen. Später werden Sie aber in manchen Fällen genauer darauf achten müssen, hier keine ID von nicht vorhandenen Komponenten zu übergeben.

Der in Klammern gesetzte Klassenname `(Button)` vor dem Aufruf von `findViewById()` ist ein sogenannter **expliziter** Cast. Das von der Methode `findViewById()` zurückgegebene Objekt ist nämlich leider immer von der Elternklasse `View`. Unsere App ist sich sicher, dass es sich dabei in Wirklichkeit um einen `Button` (eine Kindklasse von `View`) handelt, und wandelt das Objekt entsprechend um.

Sie haben bestimmt schon längst ⇧ + Strg + O gedrückt, um die Klasse `Button` automatisch zu importieren, stimmt's?

Befehlen Sie dem Objekt `button` nun, die laufende Activity als `OnClickListener`-Implementierung zu verwenden:

```
button.setOnClickListener(this);
```

Sie erinnern sich, dass das Schlüsselwort `this` immer eine Referenz auf das Objekt enthält, in dessen Kontext sich eine Methode befindet – in diesem Fall die laufende Activity, die die nötige Schnittstelle ja zur Verfügung stellt. Daher passt `this` als Parameter der Klasse `OnClickListener`.

Die Methode `onInit()` ist damit fertig und sieht in ihrer ganzen Schönheit wie folgt aus:

```
@Override
public void onInit(int arg0) {
   tts.setLanguage(Locale.GERMAN);
   Button button = (Button) findViewById(R.id.button1);
   button.setOnClickListener(this);
}
```

Nun fehlt nur noch eine Kleinigkeit: Anstelle des fest eingebauten Strings `"Hallo!"` soll die `onClick()`-Methode den Inhalt des Eingabefeldes verwenden. Um an den Inhalt des Feldes zu kommen, benötigen Sie zunächst eine Referenz auf das Eingabefeld selbst. Das funktioniert ähnlich wie beim Button:

```
EditText editText = (EditText)findViewById(R.id.editText1);
```

**Content Assist**

Sie werden beim Eintippen feststellen, dass das neunmalkluge Eclipse oft schon nach einem Buchstaben errät, was Sie wollen. In Wirklichkeit müssen Sie anstelle von `findViewById` nur bis zum ersten V tippen, gefolgt von Strg + Leertaste – Eclipse weiß, dass es nur eine an dieser Stelle verfügbare Funktion gibt, deren Name mit `findV` beginnt, und ergänzt den Rest automatisch.

Das gilt auch für den Parameter in Klammern: Sie erinnern sich nicht mehr genau, wie die ID der EditText-Komponente heißt? Kein Problem, tippen Sie lediglich `R.id.`, und drücken Sie Strg + Leertaste. Eclipse bietet Ihnen die verfügbaren Konstanten zur Auswahl an, und Sie wählen mit den Pfeiltasten und ↵ die gewünschte aus.

Sie sehen: Content Assist spart Ihnen viele Tastendrücke, wenn Sie sich einmal daran gewöhnt haben.

Der letzte Schritt: Ersetzen Sie `"Hallo!"` durch den Inhalt des Eingabefeldes. Den ermittelt die Methode `editText.getText()`. Leider handelt es sich nicht um ein `String`-Objekt, das die `speak()`-Methode sich wünscht, sondern um ein Editable. Das Editable wiederum besitzt eine Methode `toString()`, deren Name bereits verrät, was sie kann.

Der Einfachheit halber verketten Sie die Methodenaufrufe und reichen das Resultat an die Sprachausgabe weiter. Die `onClick()`-Methode ist damit fertig und sieht wie folgt aus:

```
@Override
public void onClick(View view) {
   EditText et = (EditText) findViewById(R.id.editText1);
   tts.speak(et.getText().toString(),
```

```
        TextToSpeech.QUEUE_FLUSH, null);
}
```

Wenn Eclipse jetzt nichts zu meckern hat, also keine roten Markierungen erscheinen, schließen Sie Ihr Handy an und starten die App mit Strg + F11.

Viel Spaß beim Testen!

## 4.5 Eine App installieren

Bisher haben Sie einen sehr einfachen Weg gewählt, Ihre App auf dem Handy zu installieren. Eclipse und das ADT haben Ihnen die Arbeit abgenommen. Wie aber können Sie die App einem Freund geben, der kein Eclipse hat? Was ist der saubere Weg, eine App weiterzugeben?

### Start mit ADT

Wenn Sie beim Start Ihrer App aus Eclipse heraus den Console-View aktivieren, sehen Sie, welche Schritte das ADT genau vornimmt, um Ihre App zu installieren.

Folgende Ausgaben sehen Sie dort unter anderem, alle versehen mit einem Zeitstempel, den ich der Übersicht halber weglasse:

```
Android Launch!
adb is running normally.
Uploading SagHallo.apk onto device '55567234411'
Installing SagHallo.apk...
Success!
Starting activity ...StartActivity on device 55567234411
```

Sie sehen, dass ADT zunächst die fertig verpackte App in Form der Datei *SagHallo.apk* auf das Gerät hochlädt. Anschließend wird das APK installiert und im letzten Schritt die richtige Activity gestartet.

Diese Schritte können Sie auch von Hand ausführen.

### Installieren per USB

Da das ADT die Datei *SagHallo.apk* hochlädt, muss sie sich irgendwo auf Ihrer Festplatte befinden. In der Tat finden Sie sie im Unterordner *bin* innerhalb Ihres Projektverzeichnisses.

Diese Datei können Sie nach Belieben an eine andere Stelle kopieren, per E-Mail an Freunde verschicken oder irgendwo zum Download bereitstellen.

Kopieren Sie die Datei beispielsweise auf die SD-Karte Ihres Handys, indem Sie die Karte per Kartenleser mit Ihrem PC verbinden. Falls Sie ein passendes Datenkabel haben, können Sie die meisten Geräte als externe Datenträger an Ihren PC stöpseln und die APK-Datei daraufkopieren.

Auf dem Handy müssen Sie schließlich nur noch die APK-Datei öffnen. Falls Sie in den Android-Einstellungen die Installation aus beliebigen Quellen erlaubt haben, geht der Rest nach einer Bestätigung per OK-Button automatisch.

**Dateimanager**

Zahlreiche Android-Smartphones werden vom Hersteller nicht mit einem Dateimanager ausgeliefert. Den benötigen Sie aber, um beispielsweise eine APK-Datei auf der SD-Karte auszuwählen und zu öffnen.

Aus diesem Grund bietet der Android Market eine ganze Reihe kostenloser Dateimanager an. Beliebt ist beispielsweise der *Astro-Datei-Manager*, dessen Gratis-Version allerdings Anzeigen einblendet. Eine werbefreie Alternative hört auf den schlichten Namen *Explorer* und ist deshalb im Market nicht allzu leicht zu finden. Geben Sie als Suchbegriff zusätzlich den Hersteller *Speed Software* an, um diese App zu finden.

Und behalten Sie im Hinterkopf, Ihre eigenen Apps immer mit individuellen Namen zu versehen, damit jemand, der danach sucht, sie auch sofort findet ...

## Installieren mit ADB

Falls Sie eine Vorliebe für die Arbeit an der Kommandozeile haben, bietet Ihnen das Android SDK ein mächtiges Instrument: die Android Debug Bridge, kurz: **ADB**.

Probieren Sie als Erstes aus, ob ADB korrekt funktioniert, indem Sie ein Terminal (unter Windows: Eingabeaufforderung) öffnen und `adb`, gefolgt von ↵ eintippen. Falls dann keine ausführliche Bedienungsanleitung erscheint, müssen Sie dafür sorgen, dass Ihr System das ADB findet. Wie Sie unter Windows den Ordner *platform-tools* in Ihrem Android-SDK-Verzeichnis zur `Path`-Variablen hinzufügen, wurde bereits in Abschnitt 3.2, »JDK installieren«, anhand des JDK erläutert.

Wenn Sie ADB zur Mitarbeit überredet haben, starten Sie einen Emulator oder verbinden Ihr Android-Gerät per USB und versuchen als Nächstes:

```
adb devices
```

Sie erhalten eine Liste der angeschlossenen Geräte, die Sie nunmehr mit ADB kontrollieren können. Die Gerätenamen sind erwartungsgemäß dieselben, die Sie auch in Eclipse sehen können, wenn Sie die Geräte-Ansicht öffnen – im

Grunde tut das ADT unter Eclipse nichts anderes als Sie: Es ruft im Hintergrund ADB auf.

Wechseln Sie als Nächstes mit dem Befehl `cd <Verzeichnis>` in Ihr Projektverzeichnis. Rufen Sie dann auf:

```
adb install bin/SagHallo.apk
```

Falls Sie die Fehlermeldung »already exists« erhalten, haben Sie etwas über die Android-Versionsverwaltung gelernt: Zwei APKs mit gleichem Versionscode werden als identisch behandelt – folglich hält Android die Installation für überflüssig. Ergänzen Sie einfach den Parameter `-r`, um diese Beschränkung zu umgehen.

```
adb install -r bin/SagHallo.apk
```

Falls Ihre App über Einstellungen oder andere Daten verfügt (was bei SagHallo nicht der Fall ist), bleiben diese erhalten.

Wenn Sie Ihre App deinstallieren möchten, verwenden Sie:

```
adb uninstall de.androidnewcomer.saghallo
```

Da Android nach erfolgter Installation piepegal ist, wie die verwendete APK-Datei hieß, müssen Sie die gewünschte App jetzt anhand ihres individuellen Paketnamens identifizieren.

Vorsicht: Dieser Befehl löscht eventuell vorhandene Daten der App, genau wie die Deinstallation über die Android-App-Verwaltung direkt am Gerät. Um die Daten zu behalten, ergänzen Sie den Parameter `-k`:

```
adb uninstall -k de.androidnewcomer.saghallo
```

Natürlich kann ADB noch viel mehr. Das meiste davon lässt sich bequemer über Eclipse und ADT erledigen, aber nicht alles. Wir werden noch darauf zurückkommen.

### Drahtlos installieren

Streng genommen, kommt es einer Beleidigung nahe, ein Smartphone, dessen Domäne die drahtlose Kommunikation ist, mit einem Kabel zu belästigen.

Wenn Ihr PC und Ihr Handy über WLAN im gleichen lokalen Netzwerk angemeldet sind, können Sie beispielsweise das Verzeichnis *bin* freigeben. Auf eine solche Windows-Ordnerfreigabe kann ein Smartphone mittels SMB-Protokoll zugreifen. Verwenden Sie dazu eine App wie den SharesFinder oder AndSMB.

Falls Sie auf Ihrem PC einen Webserver wie Apache betreiben, können Sie das Verzeichnis *bin* unter einer bestimmten URL einbinden oder das APK in einen vom Webserver erreichbaren Ordner kopieren. Dann können Sie den Browser Ihres Handys auf die richtige URL richten, um das APK herunterzuladen und zu installieren. Die Installation und den Betrieb eines lokalen Webservers zu erläutern würde den Rahmen dieses Kapitels freilich sprengen. Wenn Sie keinen solchen Webserver auf Ihrem PC haben, gibt es aber noch mindestens zwei weitere Optionen.

Falls Sie über eigenen Webspace, z. B. eine private Homepage, verfügen, können Sie das APK einfach dorthin hochladen. Sie müssen nicht einmal einen Link auf das APK hinterlegen; es genügt, wenn Sie es vom Handy aus mit der direkten URL (z. B. *http://meine-homepage.de/SagHallo.apk*) herunterladen. Dies hat im Gegensatz zu den zuvor genannten Methoden sogar den Vorteil, dass es auch funktioniert, wenn Ihr Android sich nicht im gleichen Netzwerk befindet wie Ihr PC, denn in diesem Fall werden die Daten über das Internet übertragen – egal, ob per LAN, WLAN oder Mobilfunk.

Eine weitere Alternative ist ein Cloud-Dienstleister wie z. B. Dropbox (*http://dropbox.com*). Dort bekommen Sie kostenlosen Speicher, der permament über das Netz mit einem bestimmten Verzeichnis auf Ihrer Festplatte synchronisiert wird. Sie können Ihr APK einfach in diesen Ordner legen, einen Moment warten, und die Datei vom Smartphone aus runterladen – denn selbstverständlich existiert für Android eine Dropbox-App.

Wenn Ihnen all das noch nicht reicht, schicken Sie das APK einfach als Anhang einer E-Mail an sich selbst (oder einen Freund). Auch so lässt sich Ihre App leicht installieren.

Bevor Sie sich darüber freuen, wie einfach der Umgang mit der APK-Datei ist, verrate ich Ihnen den Haken an der Sache: Diese APK-Datei ist mit einem Debug-Zertifikat digital signiert. Der Android Market verlangt allerdings, dass Apps mit einem anderen als einem Debug-Zertifikat signiert werden. Bevor Sie also Ihre App im Android Market veröffentlichen können, müssen Sie noch eine kleine Hürde überwinden. Darauf komme ich im Abschnitt »Zertifikat erstellen« zurück.

**Das Verfallsdatum des Debug-Zertifikats**

Falls Sie schon einmal mit digitalen Zertifikaten zu tun hatten – sie kommen ja nicht nur in der Android-Entwicklung vor – wissen Sie, dass jedes Zertifikat einen Gültigkeitszeitraum eincodiert hat. Einfach ausgedrückt: Jedes Zertifikat verfällt an einem bestimmten Tag zu einer bestimmten Uhrzeit.

Das gilt auch für das Debug-Zertifikat Ihrer Entwicklungsumgebung. Da es automatisch erzeugt wird, haben Sie leider keinen Einfluss darauf, wie lange es gültig ist. Der Standard ist leider nur ein Jahr.

Das bedeutet, dass Sie zwangsläufig ein Jahr nach dem ersten Bau einer App eine obskure Fehlermeldung erhalten werden, die Sie ungemein nerven würde, es sei denn, Sie haben diesen Hinweis gelesen.

Um das Verfallsdatum Ihres Zertifikats zu ermitteln, holen Sie sich am besten ein kleines Java-Programm namens *KeyToolIUI* aus dem Netz (*http://code.google.com/p/keytool-iui/*). Es lässt sich direkt von der Webseite per Java WebStart in Gang bringen.

Wählen Sie im Menü VIEW • FILE • KEYSTORE • JKS KEYSTORE, und öffnen Sie die Datei *debug.keystore* auf Ihrem Rechner. Sie finden sie unter Windows im Verzeichnis *Benutzer\(Ihr Benutzername)\.android* und unter Linux im Ordner *~/.android*.

Das streng geheime Passwort lautet `android`.

Der Keystore enthält erwartungsgemäß nur einen Eintrag, nämlich *androiddebugkey*. Klicken Sie mit der rechten Maustaste, und wählen Sie VIEW CERTIFICATE CHAIN, um sich die Details des Debug-Zertifikats anzusehen (Abbildung 4.17). Eines davon ist das Verfallsdatum (*valid until*).

Machen Sie sich einen Kalendereintrag: An diesem Tag werden Sie ein neues Debug-Zertifikat benötigen. Löschen Sie dazu einfach die Datei *debug.keystore*, das ADT erzeugt automatisch eine neue – mit einem Zertifikat, das erneut genau ein Jahr gilt.

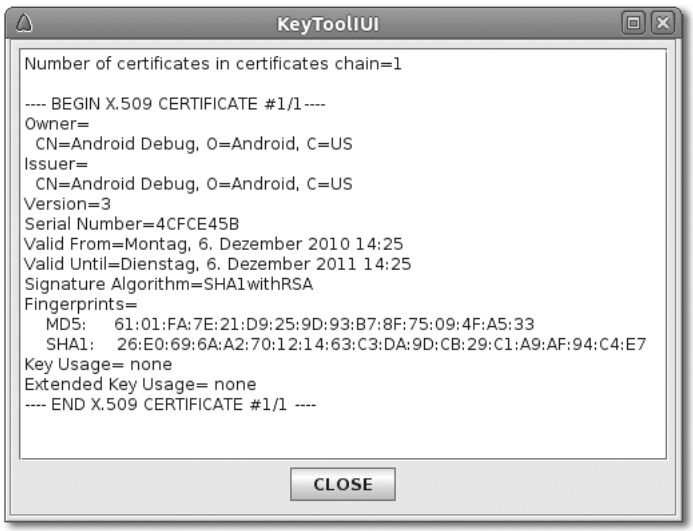



**Abbildung 4.17** Die Details des Debug-Zertifikats verraten, wann es abläuft – dieses hier zufälligerweise am Nikolaustag.

**Zwei Zertifikate, eine App?**

Zur Sicherheit erlaubt Android nicht, eine App, die mit dem Zertifikat A signiert ist, mit einer identischen App (d. h. mit dem gleichen Paketnamen) zu überschreiben, die mit dem Zertifikat B signiert ist.

Falls Sie also Ihre App mit Debug-Zertifikat installiert haben, können Sie sie nicht mit jener überschreiben, die Sie mit Ihrem endgültigen Zertifikat signiert haben. Sie müssen vorher deinstallieren.

Dasselbe Problem tritt sogar mit zwei verschiedenen Debug-Zertifikaten auf. Falls Sie mit zwei Eclipse-Installationen auf zwei Rechnern arbeiten, kann die vom ersten Eclipse erzeugte APK nicht mit der aus dem anderen überschrieben werden, weil jede beim ersten Benutzen ein eigenes Debug-Zertifikat generiert hat.

*»Eigentlich mag ich gar keine Pixel.«*
*(Pac Man)*

# 5 Ein Spiel entwickeln

Wenn Sie gelegentlich mit öffentlichen Verkehrsmitteln unterwegs sind, wird Ihnen die wachsende Anzahl Pendler aufgefallen sein, die konzentriert auf ihr Handy starren, anstatt die schöne Aussicht zu genießen.

Manch einer liest seine E-Mails oder ist in irgendeinem sozialen Netzwerk unterwegs, andere mögen tatsächlich noch etwas Antikes wie eine SMS verfassen – aber eine ganze Reihe Menschen nutzt die Zeit im Zug, um zu spielen.

Smartphones machen es möglich. Nicht nur dem Anwender, sondern auch dem Entwickler machen Spiele mehr Spaß. Aus diesem Grund lernen Sie die weiteren Schritte der Android-Programmierung anhand eines Spiels. Wir beginnen mit einem recht simplen Game, aber in jedem Kapitel fügen Sie weitere Funktionen hinzu, bis das Resultat Ihre Mitmenschen in Staunen versetzt. Jedenfalls wenn sie hören, dass Sie es programmiert haben, obwohl Sie vor ein paar Tagen vielleicht noch gar kein Java konnten.

## 5.1 Wie viele Stechmücken kann man in einer Minute fangen?

Gehören Sie auch zu den bevorzugten Getränken der Insektenfamilie namens Culicidae? Meine Frau hat gut reden, wenn ich mitten in der Nacht das Licht anschalte, um auf die Jagd nach einem summenden Vampir zu gehen: *Ihr* Blut schmeckt den Mistviechern ja nicht.

Es wird Zeit für eine fürchterliche Rache. Millionen Mücken sollen künftig zerquetscht werden, um der Gerechtigkeit Genüge zu tun. Und zwar auf den Bildschirmen Ihrer Handys, liebe Leser. Gut, die Mücken sind nicht echt, die App ist nur ein Spiel. Aber es gibt mir Genugtuung.

### Der Plan

Wie soll das Spiel funktionieren?

Stellen Sie sich vor, dass Sie auf dem Handy-Bildschirm kleine Bilder von Mücken sehen. Sie tauchen auf und verschwinden, und es sind viele. Treffen Sie

eine mit dem Finger, bevor sie verschwindet, erhalten Sie einen Punkt. Sobald Sie eine geforderte Anzahl Punkte erreicht haben, ist die Runde vorbei – spätestens aber nach einer Minute. In dem Fall heißt es »Game Over«: Sie haben verloren. Ansonsten geht das Spiel mit der nächsten, schwierigeren Runde weiter.

Vermutlich kommt Ihnen jetzt eine ganze Menge Ideen: Die Mücken könnten sich bewegen, man könnte einen Highscore speichern, oder jedes zerquetschte Insekt könnte ein lustiges Geräusch von sich geben. Solche Ideen sind wunderbar – machen Sie sich Notizen für später. Denn die Grundversion des Spiels zu bauen ist als erster Schritt Herausforderung genug.

Halten Sie sich vor Augen, welche Fähigkeiten die Mücken-App haben muss:

- zwei verschiedene Layouts: ein Startbildschirm und das eigentliche Spiel
- Bilder von Mücken an zufälligen Stellen anzeigen und verschwinden lassen
- den berührungsempfindlichen Bildschirm verwenden, um festzustellen, ob der Spieler eine Mücke getroffen hat
- eine Zeitanzeige rückwärts laufen lassen bis zum »Game Over«

Das klingt überschaubar, nicht wahr? Es ist eine wichtige Regel für Projekte in der Informationstechnologie, größere Aufgaben in kleinere zu unterteilen und jene der Reihe nach anzugehen. Nehmen Sie sich am Anfang nie zu viel vor, denn kleine Schritte bringen schneller Erfolge, und Sie verlieren nicht so leicht den Überblick.

Bereit? Möge die Jagd beginnen.

## Das Projekt erzeugen

Als ersten Schritt legen Sie in Eclipse ein neues Android-Projekt an. Schließen Sie alle anderen Projekte im Workspace, indem Sie das Kontextmenü mit der rechten Maustaste öffnen und Close wählen.

Starten Sie den Wizard für ein neues Android-Projekt (Strg + N). Nennen Sie das Projekt nicht `Mückenmassaker`, denn ein Umlaut ist an dieser Stelle unerwünscht. `Mueckenmassaker` oder, falls es Ihnen aus irgendeinem Grund lieber ist, `Mueckenfang` funktionieren.

Wählen Sie einen passenden Package-Namen, und lassen Sie sich eine Activity erstellen. Verwenden Sie als Build Target und Min SDK Version die 8, also Android 2.2, es sei denn, Sie haben ein älteres Smartphone (Abbildung 5.1).

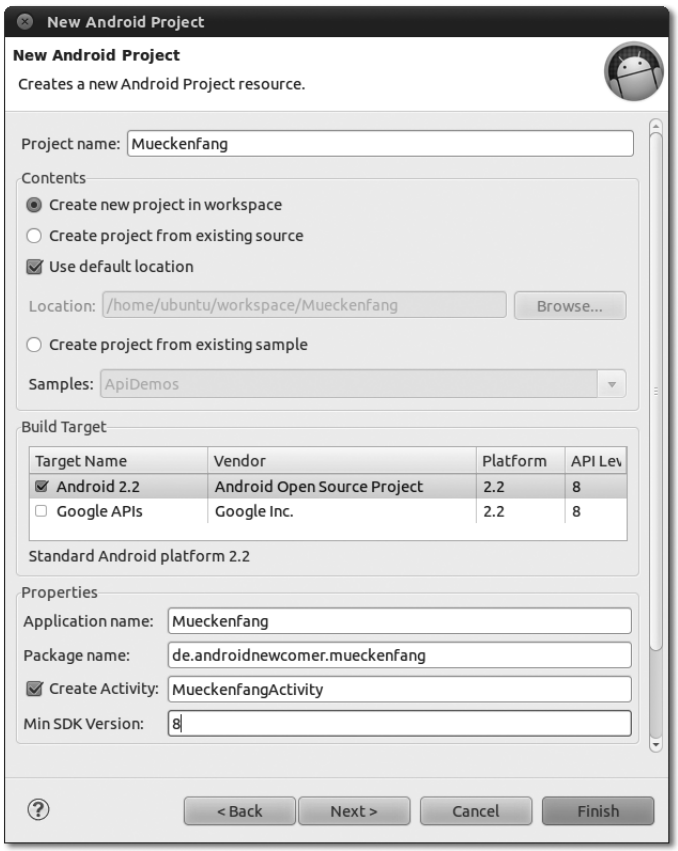



**Abbildung 5.1** Der Mückenfang beginnt, bloß ohne Umlaute.

Werfen Sie einen Blick auf die Dateien, die der Wizard erzeugt hat. Wie schon bei Ihrer ersten App finden Sie den Quellcode Ihrer Activity, das Android-Manifest und ein Standard-Icon im *res*-Verzeichnis.

## Layouts vorbereiten

Der Android-Wizard hat unter anderem eine Layout-Datei *main.xml* erzeugt. Dieses Layout wird später den Startbildschirm des Spiels definieren.

Für das eigentliche Spiel benötigen Sie ein zweites Layout.

Doppelklicken Sie auf *main.xml*, und erzeugen Sie eine Kopie, indem Sie die Datei mit FILE • SAVE AS unter einem neuen Namen speichern: *game.xml*. Das geht schneller als mit dem zugehörigen Wizard, der natürlich ebenfalls zur Verfügung steht.

Löschen Sie den TextView, damit die Layouts auf den ersten Blick unterscheidbar sind. Das endgültige Layout werden Sie zu einem späteren Zeitpunkt erstellen.

Öffnen Sie dann erneut das Layout *main.xml*, und fügen Sie einen Button mit der Aufschrift START hinzu (siehe Abschnitt 4.3, »Benutzeroberflächen bauen«). Da Sie dabei ohnehin die *strings.xml* bearbeiten müssen, ändern Sie den »Hello«-Text in »Willkommen beim Mückenfang«.

## Die GameActivity

Aus dem Spielkonzept geht hervor, dass Sie zwei Screens benötigen: Da ist zunächst der Hauptbildschirm, der den Spieler begrüßt, ihm vielleicht das Spiel erklärt und natürlich einen START-Button bietet. Verwenden Sie die vom Wizard erzeugte `MueckenfangActivity` als Basis dafür, denn diese Activity ist im Android-Manifest bereits als Start-Activity definiert. Das ist genau das gewünschte Verhalten: Wenn der Spieler das App-Icon antippt, gelangt er in den Hauptbildschirm.

Der zweite Screen wird das eigentliche Spiel darstellen. Dazu benötigen Sie eine zweite Activity, für die sich der Name `GameActivity` anbietet. Drücken Sie Strg + N, und wählen Sie den Wizard zum Erzeugen einer neuen Java-Klasse (Abbildung 5.2).


**Abbildung 5.2** Reichhaltiges Wizard-Angebot …

Der Wizard zeigt Ihnen einen Dialog, in dem Sie die wichtigsten Eckdaten der gewünschten Klassen eintragen können. Dazu gehört an erster Stelle der Name, in diesem Fall `GameActivity`. Achten Sie darauf, dass der richtige Package-Name vorausgefüllt ist.

Tragen Sie als Basisklasse für Activities immer `android.app.Activity` ein. An diesem Textfeld signalisiert die kleine Glühbirne, dass Eclipse bereit ist, Sie bei der Eingabe zu unterstützen. Tippen Sie »Activity«, gefolgt von Strg + Leertaste, und Sie sparen dank Content Assist Tipparbeit (Abbildung 5.3). Beenden Sie den Wizard mit FINISH.


**Abbildung 5.3** Der Wizard erzeugt die »GameActivity« mit wenigen Klicks.

Das Resultat zeigt Ihnen Eclipse sofort an: Den Quellcode Ihrer neuen Activity. Sie hätten diesen Java-Code auch höchstpersönlich eintippen können, und viele Programmierer ziehen das durchaus vor – es ist Geschmackssache. Die Hauptsache ist, dass Sie die verschiedenen Wege kennen, die zu einem bestimmten Resultat führen. Welcher Ihnen am meisten liegt, können Sie selbst entscheiden.

Sorgen Sie nun dafür, dass die `GameActivity` das richtige Layout anzeigt. Am einfachen bekommen Sie das hin, indem Sie den betreffenden Code aus der `MueckenfangActivity` übernehmen: Öffnen Sie also diese Klasse, und kopieren Sie die `onCreate()`-Methode hinüber in die `GameActivity`. Übergeben Sie dann anstelle von `R.layout.main` an `setContentView()` `R.layout.game`.

Der Code sieht dann wie folgt aus:

```
package de.androidnewcomer.mueckenfang;
import android.app.Activity;
import android.os.Bundle;
public class GameActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.game);
    }
}
```

Im Layout *main.xml* haben Sie bereits einen Button eingefügt, und wie Sie den Klick behandeln, wissen Sie schon aus dem Abschnitt »Der OnClickListener«. Fügen Sie also eine Methode `onClickListener()` in die Klasse `MueckenfangActivity` ein, und lassen Sie sie das Interface `OnClickListener` implementieren:

```
public class MueckenfangActivity extends Activity implements
OnClickListener {
   …
   @Override
   public void onClick(View v) {
   }
}
```

Sorgen Sie nun dafür, dass diese Methode die `GameActivity` startet, indem Sie darin die Methode `startActivity()` aufrufen:

```
startActivity(new Intent(this, GameActivity.class));
```

Das `Intent`-Objekt, das Sie als Parameter übergeben müssen, definiert, welche Klasse Android für die zu startende Activity verwenden soll. Mit solchen `Intent`-Objekten lässt sich noch eine ganze Menge mehr anstellen – ich werde später darauf zurückkommen.

Beachten Sie, dass dieser Aufruf zwar die `GameActivity` startet, aber die aktuelle Activity *nicht beendet*. Sie können daher später mit der Zurück-Taste am Gerät von der `GameActivity` in die `MueckenfangActivity` zurückkehren. Es gibt durchaus die Möglichkeit, eine Activity zu beenden (mit der Methode `finish()`), allerdings ist das Standard-Verhalten im Sinne unseres Spiels.

Verbinden Sie als Nächstes in der `onCreate()`-Methode den Button mit der Activity (siehe ebenfalls den Abschnitt zum OnClickListener):

```
Button button = (Button) findViewById(R.id.button1);
button.setOnClickListener(this);
```

Der Code der `MueckenfangActivity` sieht damit wie folgt aus:

```
package de.androidnewcomer.mueckenfang;
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;

public class MueckenfangActivity extends Activity implements
OnClickListener {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        Button button = (Button) findViewById(R.id.button1);
        button.setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {
           startActivity(new Intent(this,GameActivity.class));
    }
}
```

Melden Sie zum Schluss die neue `GameActivity` im Android-Manifest an (siehe Abschnitt »Activities anmelden«).

Prüfen Sie, ob irgendwo Tippfehler rote Fehler-Symbole verursachen. Wenn nicht, starten Sie die App mit Rechtsklick auf das Projekt und Run As... Android Application – entweder im Emulator oder auf einem angeschlossenen Smartphone.

Bislang ist von einem Spiel nichts zu sehen: Sie können mit dem Start-Button zum leeren Game-Screen wechseln und mit der Back-Taste wieder zurück. Es wird Zeit, die Schwärze der vorgefertigten Layouts gegen etwas Ansehnlicheres auszutauschen: Grafiken.

## 5.2 Grafiken einbinden

Die grafischen Elemente der ersten Version der Mückenjagd sind überschaubar: Sie benötigen offensichtlich ein Bild von einer Mücke.

## Die Mücke und der Rest der Welt

Je nach Zeichentalent oder Ihrer Geduld, Insekten mit Makroobjektiven aufzulauern, können Sie unterschiedliche Grafiken verwenden. Kommen Sie bitte nicht auf die Idee, das nächstbeste Bild aus der Google-Bildersuche zu kopieren: Jene Bilder sind fast immer urheberrechtlich geschützt. Für die Mückenjagd ist das weniger relevant, weil Sie die kaum im Android Market veröffentlichen werden, aber bei eigenen Spielideen müssen Sie in dieser Hinsicht vorsichtig sein.

Wenn Sie Ihr Zeichentalent ausprobieren wollen, empfehle ich Ihnen das Vektorgrafikprogramm Inkscape, das für alle Betriebssysteme frei verfügbar und auf der Buch-DVD enthalten ist. Mit wenigen Mausklicks zeichnen Sie die wesentlichen Körperteile des fraglichen Insekts. Speichern Sie die Grafik nicht nur im Vektorformat SVG, sondern exportieren Sie sie auch als PNG in einer Auflösung von etwa 50 mal 50 Pixeln. Alternativ verwenden Sie meinen Entwurf, den Sie auf der DVD im Projektverzeichnis finden. Übrigens empfehle ich Ihnen, für nebenbei anfallende Dateien wie die Vektordatei einen Ordner *verschiedenes* im Projektverzeichnis zu erstellen. Auf diese Weise haben Sie immer alle relevanten Dateien griffbereit.

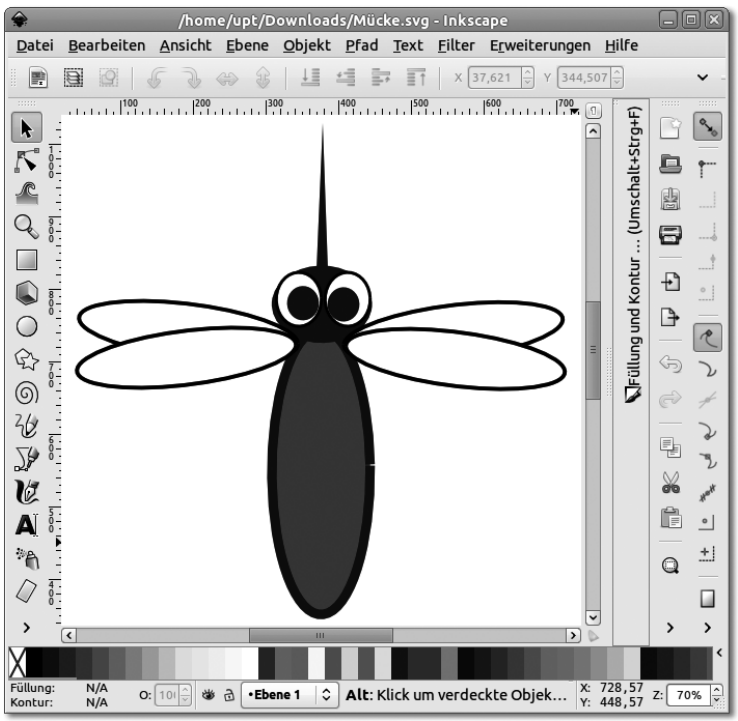



**Abbildung 5.4** Entwerfen Sie Ihre Mücke zum Beispiel mit Inkscape, wenn Sie mit dem Makroobjektiv keinen Erfolg haben.

Speichern Sie die Pixeldatei *muecke.png* (ohne Umlaute!) im Verzeichnis *drawable-mdpi*. Vermutlich müssen Sie in Eclipse einmal das Verzeichnis aktualisieren (F5), damit die PNG-Datei auftaucht. Benennen Sie *drawable-mdpi* in *drawable* um, denn vorerst kümmern wir uns nicht um niedrige oder hohe Bildschirmauflösungen.

Löschen Sie die anderen *drawable*-Verzeichnisse, und löschen Sie außerdem die Datei *icon.png*, die das weiße Standard-Icon enthält. Legen Sie stattdessen eine Kopie der Mücke unter diesem Namen an die gleiche Stelle. Klicken Sie dazu *muecke.png* an, und drücken Sie Strg + C zum Kopieren und Strg + V zum Einfügen. Eclipse bittet Sie automatisch, einen anderen Dateinamen zu vergeben: Geben Sie dann »icon.png« ein.

Mücken zu fangen macht auf einem nachtschwarzen Bildschirm wenig Spaß. Natürlich könnten Sie auch für den Hintergrund eine Zeichnung verwenden. Alternativ schießen Sie einfach ein Foto mit der ins Handy eingebauten Kamera (hochkant). Skalieren Sie das Foto auf 640 Pixel Breite und 800 oder 854 Pixel Höhe (je nach Seitenverhältnis Ihrer Kamera und Auflösung Ihres Handy-Bildschirms). Speichern Sie das Bild als *hintergrund.jpg* im *drawable*-Verzeichnis.

## Grafiken einbinden

Nun sind die Hintergrundgrafik sowie die Mücke Teil Ihres Projekts. Im nächsten Schritt werden Sie die Bilder in die Layouts einbauen. Beginnen Sie mit der Datei *main.xml*: Öffnen Sie das Layout, dann klicken Sie mit rechts in den leeren, schwarzen Hintergrundbereich. Wählen Sie im Popup-Menü PROPERTIES • BACKGROUND..., und es erscheint der Reference Chooser, der unter anderem die *drawable*-Ressourcen, die dem Projekt bekannt sind, auflistet (Abbildung 5.5).

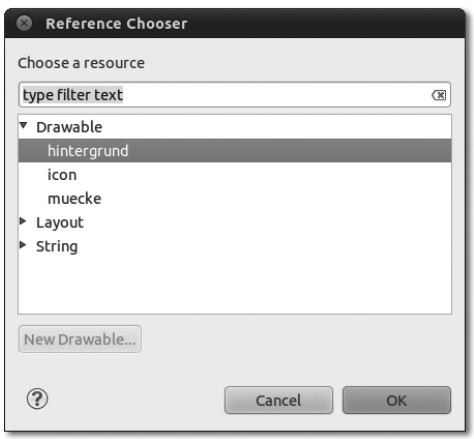



**Abbildung 5.5** Im Reference Chooser wählen Sie die richtige Grafik aus.

In diesem Fall ist HINTERGRUND die offensichtlich richtige Wahl. Fertig: In der Vorschau ist Ihr Foto zu sehen.

Fügen Sie als Nächstes aus der Palette einen ImageView zwischen TextView und START-Button ein. Diesmal präsentiert Ihnen Eclipse automatisch den Resource Chooser, in dem Sie die Mücke auswählen. Verschönern Sie den Bildschirm weiter, indem Sie dem Hintergrund die Gravity `center` befehlen, die Breite des TextViews auf `wrap_content` ändern und die TextSize auf `20sp` erhöhen. Dabei steht `sp` für **scale-independant pixels**. Diese Einheit sorgt dafür, dass die Schrift auf allen Geräten im Verhältnis zu anderen Elementen gleich groß ist – egal, wie hoch die physikalische Auflösung des Bildschirms ist, außerdem wird die vom Benutzer gewählte Fontgröße berücksichtigt. Versehen Sie schließlich den ImageView mit einem Padding von etwa `20dp`, um Abstand zwischen der Mücke und den restlichen Elementen zu schaffen.

Der Standard-Bildschirm hat eine Breite von 320 Bildpunkten, daher ergibt ein Rand von 20 ein brauchbares Erscheinungsbild. Die Einheit `dp` steht für **device independant pixels** und ähnelt der Einheit `sp`, berücksichtigt allerdings nicht die benutzerspezifische Fontgröße. Folglich verwenden Sie `sp` immer bei TextSize-Angaben und `dp` bei allen anderen. All diese Einstellungen nehmen Sie entweder über das Kontextmenü oder über den PROPERTIES-View vor.

Experimentieren Sie mit den verschiedenen Einstellungen, bis Ihnen der Screen gefällt (Abbildung 5.6).

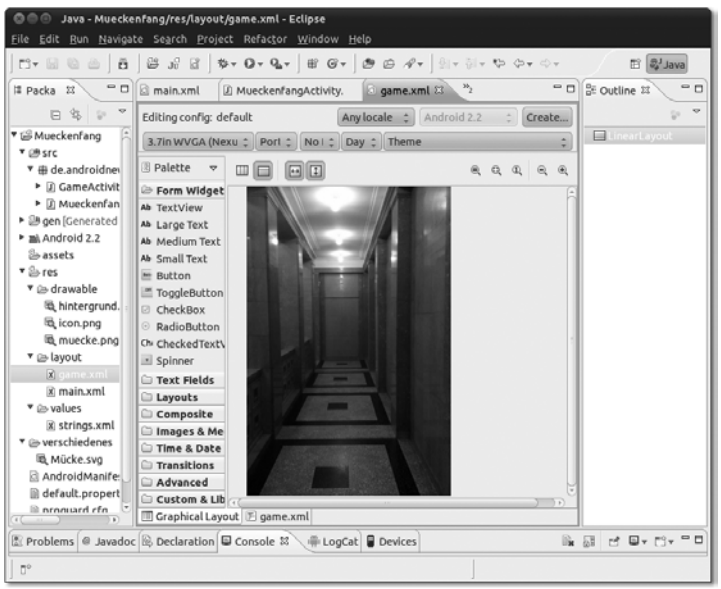



**Abbildung 5.6** Basteln Sie bitte ein hübscheres Startscreen-Layout als ich.

Zum Schluss ordnen Sie dem Layout *game.xml* ebenfalls Ihr Hintergrundbild zu. Wenn Sie möchten, können Sie natürlich zwei verschiedene Bilder verwenden, die Sie mit unterschiedlichen Dateinamen versehen.

Probieren Sie Ihre App auf dem Handy aus, und kochen Sie sich frischen Kaffee oder Tee, bevor die Arbeit am eigentlichen Spiel beginnt.

## 5.3 Die Game Engine

Vielleicht spielen Sie gelegentlich Brettspiele. Egal, ob Schach, Siedler von Catan oder Monopoly: Alle Spiele haben eine wichtige Gemeinsamkeit, nämlich Spielregeln. Üblicherweise sind sie in Schriftform beigelegt, oder alle Teilnehmer haben sie im Kopf, weil sie übersichtlich und leicht zu behalten sind (im Fall von Schach). Ohne Regeln wäre ein Spiel sinnlos.

Bei Brettspielen sind die Mitspieler selbst dafür zuständig, auf die Einhaltung der Regeln zu achten. Bei Computerspielen funktioniert das nicht, weil ein Teil der Spielfunktionen (manchmal sogar ganze Mitspieler) aus Software bestehen. Deshalb ist es von entscheidender Bedeutung, dass die Spielregeln vollständig in Programmcode vorhanden sind.

Da der Computer üblicherweise auch noch die Darstellung der Spielutensilien übernimmt, ist es nötig, die Regeln und die Bildschirmausgabe zu koppeln.

Um all das kümmert sich eine Game Engine.

### Aufbau einer Game Engine

Auf den ersten Blick mag es sinnvoll erscheinen, die Game Engine als eigene Java-Klasse zu implementieren. Je nach Komplexität eines Spiels genügt allerdings eine einzige Klasse nicht – Sie benötigen ein ganzes Package.

Bei einfachen Android-Spielen wie der Mückenjagd, die noch dazu sehr eng an die grafische Darstellung gebunden sind, ist es oft möglich, die Game Engine in einer `Activity`-Klasse unterzubringen. In unserem Fall wäre das die Klasse `GameActivity`.

Welche Komponenten benötigt eine Game Engine?

Überlegen Sie zunächst, welche Attribute nötig sind, um den jeweils aktuellen Zustand des Spiels zu beschreiben:

- Nummer der laufenden Runde (beginnend mit 1)
- Anzahl zu fangender Mücken in der laufenden Runde

- Anzahl schon gefangener Mücken in der laufenden Runde
- verbleibende Zeit für die laufende Runde (zu Beginn jeder Runde beginnend bei 60 Sekunden)
- Anzahl erzielter Punkte

Halten Sie sich vor Augen, dass Sie vermutlich jedes dieser Attribute in Ihrer Klasse wiederfinden werden.

Schließlich überlegen Sie, welche Methoden erforderlich sind, um Spielereignisse und Aktionen des Spielers auszuführen:

- ein neues Spiel starten
- eine neue Runde starten
- den Bildschirm aktualisieren
- die verbleibende Zeit herunterzählen
- prüfen, ob das Spiel vorbei ist
- eine Mücke anzeigen
- eine Mücke verschwinden lassen
- das Treffen einer Mücke mit dem Finger verarbeiten
- »Game Over«

Sie können sich schon denken, dass für jeden Punkt in dieser Liste eine Methode in Ihrer Game Engine erforderlich ist. Auf den ersten Blick sieht das nach einer ganzen Menge Arbeit aus für ein so einfaches Spiel, und damit liegen Sie nicht ganz falsch.

Bedenken Sie jedoch, dass Ihrem Spielcomputer selbst die simpelsten und selbstverständlichsten Regeln (treffe ich eine Mücke, verschwindet sie, und ich erhalte einen Punkt) fremd sind. Sie müssen jede Kleinigkeit explizit programmieren. Dass Sie sich zum jetzigen Zeitpunkt bereits eine Menge Gedanken gemacht haben, wird Ihnen beim Programmieren viel Zeit sparen. Denn die meisten Programmzeilen werden sich fast von allein ergeben. Lassen Sie uns zunächst aufschlüsseln, was in jeder der Methoden geschehen muss.

### Ein neues Spiel starten

Die Methode zum Start eines neuen Spiels wird offensichtlich aufgerufen werden, wenn der Benutzer auf den START-Button drückt. Überlegen Sie, welche Attribute gesetzt werden müssen:

- laufende Runde = 0 (Sie werden gleich sehen, warum 0 und nicht 1)
- Anzahl erzielter Punkte = 0
- eine neue Runde starten

Beachten Sie, dass »eine neue Runde starten« für jede Runde gleichermaßen funktionieren soll. Es gibt keine separate Methode »erste Runde starten«.

Folglich sieht die Methode zum Starten eines neuen Spiels sehr übersichtlich aus:

```
private void spielStarten() {
   spielLaeuft = true;
   runde = 0;
   punkte = 0;
   starteRunde();
}
```

Sie vermissen vielleicht die anderen Attribute. Aber um die kümmert sich die nächste Methode. Überlegen Sie immer genau, an welcher Stelle eine Aktion auszuführen ist. Oft können Sie so redundanten Programmcode vermeiden. Beispielsweise wäre es nicht falsch, in dieser Methode die Anzahl schon gefangener Mücken auf 0 zu setzen. Da dies aber in jeder Runde geschehen muss und nicht bloß am Anfang des Spiels, genügt es, den Code in die Rundenstart-Methode zu schreiben. Und zu der kommen wir als Nächstes.

### Eine Runde starten

Welche Aktionen sind beim Start einer Runde nötig? Beachten Sie, dass diese Methode sowohl für die *erste* als auch für jede weitere Runde funktionieren muss, und zwar möglichst ohne komplizierte Spezialbehandlung:

- Nummer der laufenden Runde um 1 erhöhen (Jetzt verstehen Sie, warum dieses Attribut beim Spielstart auf 0 gesetzt wird, nicht wahr? Halten Sie sich vor Augen, dass diese einfache Zeile dank dieses Tricks in jeder weiteren Runde gleichermaßen funktioniert!)
- Anzahl der zu fangenden Mücken in dieser Runde auf einen bestimmten Wert setzen, der in jeder Runde immer größer wird. Beispiel: 10, 20, 30, … also das Zehnfache der Nummer der Runde
- Anzahl der schon gefangenen Mücken in dieser Runde = 0
- verbleibende Zeit für die laufende Runde = 60 Sekunden
- den Bildschirm aktualisieren

Auch in dieser Methode finden Sie keine höhere Magie. Je komplizierter ein Spiel ist, umso kniffliger ist es allerdings, sich die richtigen Operationen zu überlegen. Manchmal liegen Sie mit Ihrem ersten Versuch schief. Das macht nichts, denn im Gegensatz zu einem Brettspiel, das Sie vielleicht plötzlich mit Flughäfen anstelle von Bahnhöfen bedrucken müssten, bedarf es nur weniger Änderungen am Programmcode, um ein ganz unterschiedliches Verhalten des Spiels zu erreichen.

Die Methode wird wie folgt aussehen:

```
private void starteRunde() {
  runde = runde +1;
   muecken = runde * 10;
   gefangeneMuecken = 0;
   zeit = 60;
   bildschirmAktualisieren();
}
```

Sie sehen, dass diese Methode die erste ist, die mit dem Bildschirm interagiert. Werfen wir als Nächtes einen genaueren Blick darauf, was der Spieler zu sehen bekommt.

### Den Bildschirm aktualisieren

Schließen Sie die Augen (oder starren Sie auf ein leeres Blatt Papier), um sich vorzustellen, wie der Spielbildschirm aussehen soll.

Natürlich nimmt die Fläche, auf der die Mücken erscheinen, den größten Raum ein. Davon abgesehen, möchte der Spieler aber ständig einige Informationen sehen können:

- aktuelle Punktzahl
- Nummer der aktuellen Runde
- Anzahl gefangener und noch zu fangender Mücken
- verbleibende Zeit

Entscheiden Sie für jede der Informationen, wie wichtig sie ist und welches der beste Weg ist, sie dem Spieler zu vermitteln. Beispielsweise ist eine numerische Anzeige der verbleibenden Zeit gut und schön, aber im Eifer des Spiels alleine nicht günstig. Viel praktischer ist ein Balken, der immer kürzer wird, bis die Zeit abgelaufen ist.

Ähnliches gilt für die Anzahl zu fangender Mücken: In einem Spiel, in dem es auf Tempo ankommt, sollte der Spieler keine Ziffern ablesen müssen. Wählen Sie also auch hier einen zusätzlichen Balken: Immer wenn eine Mücke gefangen wird, verlängert sich der Balken, bis er bei erfolgreichem Beenden der Runde die volle Bildschirmbreite erreicht hat.

Für den Anfang positionieren wir beide Balken vor unserem geistigen Game-Design-Auge am unteren Bildschirmrand, aber in verschiedenen Farben. Die aktuelle Punktzahl und die laufende Runde können prima am oberen Rand in der linken und der rechten Ecke erscheinen.

Aufgabe der Methode wird es also sein, die korrekten Zahlen in die Layout-Elemente einzutragen und die Länge der Balken richtig zu setzen.

Schreiten Sie zur Tat, und fügen Sie die nötigen Elemente in das Layout *game.xml* ein.

Derzeit besteht das Layout lediglich aus einem LinearLayout-Element mit vertikaler Aufteilung. Das ist ein brauchbarer Ausgangspunkt, aber um das gewünschte Design zu erhalten, müssen Sie weitere Layout-Elemente verschachteln.

Die obere Punkteleiste soll eine Anzeige links und eine rechts enthalten. Das entspricht zwei TextView-Elementen, wobei das eine eine linksseitige Layout Gravity (nicht gravity!) erhält und das andere eine rechtsseitige. Verwenden Sie ein FrameLayout, um die beiden TextViews zu umschließen, ohne dass sie einander in die Quere kommen.

Ziehen Sie als Erstes ein FrameLayout aus der Palette, und es wird sich am oberen Rand des Bildschirms anordnen. Pflanzen Sie zwei TextViews mit großer Schrift (Large Text) hinein, ändern Sie deren IDs mit dem Kontextmenü Edit ID... auf `points` bzw. `round`, und setzen Sie bei dem einen das Property Layout Gravity auf `right`. Sie müssen keine Strings für diese TextViews erzeugen, denn die richtigen Zahlenwerte schreibt die Methode Bildschirm aktualisieren später einfach direkt hinein. Setzen Sie schließlich den Text Style auf `bold`.

Ändern Sie die Farbe des Textes (Property TextColor), sodass sie zu Ihrem Bildschirmhintergrund passt. Leider müssen Sie zuerst die gewünschten Farben in einer neuen Datei definieren, um sie im Kontextmenü auswählen zu können. Die schnelle Lösung ist an dieser Stelle der Properties-View. Dort können Sie als Text Color einfach einen Hexadezimal-Wert (wie in HTML) eintragen, beispielsweise `#00FF00` für Grün, `#FF0000` für Rot oder `#0000FF` für Blau.

**Farbressourcen**

Sie werden oft in Apps dieselbe Farbe an mehreren Stellen verwenden wollen. Was geschieht, wenn Sie feststellen, dass Pink doch nicht die richtige Wahl war? Sie müssen in jedem einzelnen Element den Farbwert ändern.

Auf den ersten Blick umständlicher, am Ende aber wesentlich effizienter ist der Weg über eine Datei mit Farbressourcen. Darin definieren Sie Platzhalter für Farben, die von Layout-Elementen referenziert werden. Ändern Sie die Farbe dann nur noch in der Farbendatei, und alle Elemente übernehmen das automatisch.

Erzeugen Sie eine neue Farbendatei, indem Sie Strg + N drücken und den Wizard New Android XML File auswählen. Geben Sie darin als Dateinamen *colors.xml* an, und wählen Sie bei Type of Resource bitte Values (nicht Color List!).

Mit dem Button ADD können Sie leicht Farben hinzufügen. Jeder Eintrag besteht aus einem Platzhalternamen für eine Farbe und einem HTML-Farbwert. Beachten Sie, dass für Platzhalter die üblichen Regeln gelten: keine Leerzeichen, keine Umlaute oder Sonderzeichen. Lediglich Ziffern und der Unterstrich sind erlaubt. Am besten verwenden Sie nur Kleinbuchstaben (Abbildung 5.7).

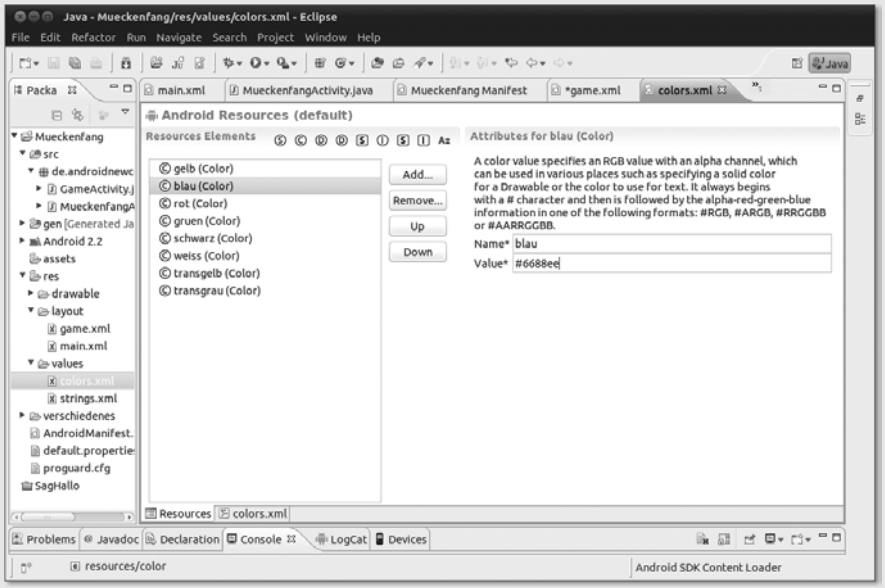



**Abbildung 5.7** Erstellen Sie eine Resource-Datei für Farben.

Noch ein Wort zu den HTML-Farbcodes: Darin stehen jeweils zwei hexadezimale Ziffern für eine der Grundfarben Rot, Grün und Blau. Ich kann Ihnen an dieser Stelle das hexadezimale Zahlensystem nicht erklären, aber es gibt im Netz (z. B. *http://colorblender.com*) und in Programmen wie Inkscape, GIMP (beide auf der Buch-DVD) oder Photoshop einfache Möglichkeiten, den HTML-Code einer Farbe zu ermitteln.

Grundsätzlich sind auch achtstellige Farbcodes erlaubt. Die beiden zusätzlichen Ziffern stehen vor den anderen sechs und bestimmen die Alpha-Transparenz, wobei `00` für Unsichtbarkeit und `FF` für Sichtbarkeit steht. Mittlere Werte wie `88` erzeugen einen hübschen halb transparenten Effekt.

Sobald Sie die Datei *colors.xml* gespeichert haben, stehen Ihnen die eingetragenen Platzhalter im Resource Chooser zur Auswahl zur Verfügung, etwa wenn Sie ein Property wie TEXT COLOR über das Kontextmenü bearbeiten.

Bevor Sie sich um die Balken kümmern können, die am unteren Ende des Bildschirms erscheinen sollen, steht das eigentliche Spielfeld auf dem Programm, weil das große LinearLayout-Element seine Kindelemente vertikal übereinander anordnet. In der Reihenfolge von oben nach unten ist nach den Elementen am oberen Rand das Spielfeld an der Reihe.

Fügen Sie dem Wurzel-LinearLayout dazu ein FrameLayout aus der Rubrik LAYOUTS hinzu, und stellen Sie das Property LAYOUT WEIGHT auf den Wert 1 (Sie können dazu auch das Icon mit dem stilisierten Portrait verwenden). Verpassen Sie diesem Element die ID `spielbereich`, denn dort werden später die Mücken erscheinen.

Kommen wir also zum unteren Bildschirmbereich. Dort positionieren Sie ein vertikales LinearLayout, das alle Balken und deren Beschriftungen übereinander darstellen wird. Fügen Sie zwei FrameLayouts hinzu. Jeder erhält ein weiteres, inneres FrameLayout, das wir als Balken zweckentfremden (man könnte auch ein anderes Element verwenden). Stellen Sie die Properties wie folgt ein:

- LAYOUT GRAVITY = `center_vertical`
- LAYOUT WIDTH = `50dip` (diese Breite wird später vom Spiel verändert)
- LAYOUT HEIGHT = `5dip`
- BACKGROUND = eine Farbe Ihrer Wahl

Legen Sie als ID des Balkens `bar_hits` bzw. `bar_time` fest.

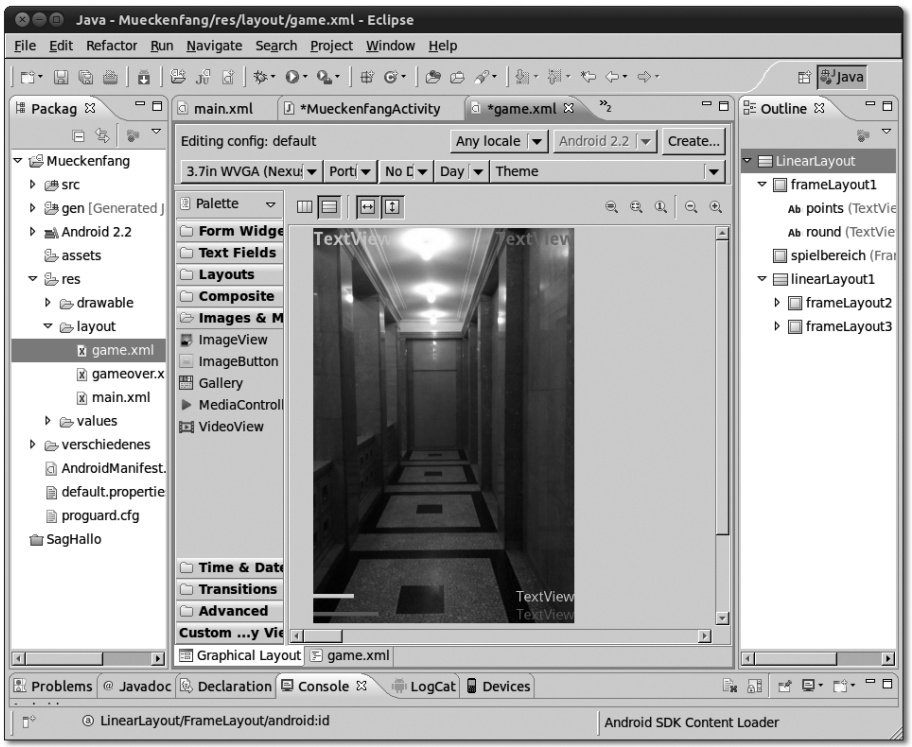



**Abbildung 5.8** Achten Sie darauf, dass die Elemente im »Outline« hierarchisch korrekt angeordnet sind.

Fügen Sie schließlich in jedes der beiden FrameLayouts einen TextView mit Layout Gravity `right` ein, um einen Zahlenwert anzuzeigen. Setzen Sie deren IDs auf `hits` beziehungsweise `time`, und verpassen Sie Ihnen die passende Text Color.

Puh. Geschafft: Wenn Ihr *game.xml*-Layout jetzt in etwa so aussieht wie meines (Abbildung 5.8), können Sie den nächsten Schritt in Angriff nehmen.

Genug Layout-Gefummel; lassen Sie uns die Methode `bildschirmAktualisieren()` schreiben, die die Elemente mit den richtigen Werten füllt.

Für die Punktzahl sieht das wie folgt aus:

```
private void bildschirmAktualisieren() {
   TextView tvPunkte = (TextView)findViewById(R.id.points);
   tvPunkte.setText(Integer.toString(punkte));
}
```

Wie üblich holen Sie sich also eine Referenz zum betreffenden View, in diesem Fall einem TextView. Dessen Objektname (`tvPunktzahl`) ist beliebig; ich setze hier eine Abkürzung der zugehörigen Klasse davor, um nicht einem gleichnamigen Attribut in die Quere zu kommen.

Wichtig ist die explizite Umwandlung des `int`-Attributs `punkte` in einen String, denn ein TextView stellt immer Strings dar, selbst wenn die nur aus Ziffern bestehen.

Füllen Sie in derselben Methode die Runde analog:

```
TextView tvRunde = (TextView)findViewById(R.id.round);
tvRunde.setText(Integer.toString(runde));
```

Die Zahl getroffener Mücken und die Restzeit funktionieren auf dieselbe Weise.

Spannender werden die beiden Balken. Holen Sie sich zunächst die zugehörigen Objekte:

```
FrameLayout flTreffer = (FrameLayout)findViewById(R.id.bar_hits);
FrameLayout flZeit = (FrameLayout)findViewById(R.id.bar_time);
```

Offensichtlich besteht die Aufgabe jetzt darin, den beiden FrameLayouts die richtige Breite zu verpassen. Dazu müssen Sie allerdings ein wenig rechnen, denn die Maße sind in Bildschirmpixeln anzugeben, nicht in **device independant pixel** (`dp` oder `dip`) wie im Layout-Editor.

Um die Maße der Balken in Bildschirmpixeln zu ermitteln, müssen Sie die gewünschte dp-Breite mit der Pixeldichte des Bildschirms multiplizieren.

Sie ermitteln diesen Maßstab aus dem `DisplayMetrics`-Objekt Ihrer App wie folgt:

```
massstab = getResources().getDisplayMetrics().density;
```

Da Sie diesen konstanten Wert an einer anderen Stelle noch einmal benötigen werden, schreiben Sie ihn in der `onCreate()`-Methode in ein `final`-Feld der Activity:

```
private float massstab;
```

Die Breite ist Teil der sogenannten **Layout-Parameter** des Views. Holen Sie sich das zugehörige `LayoutParams`-Objekt:

```
LayoutParams lpTreffer = flTreffer.getLayoutParams();
```

Achten Sie beim Organisieren der Imports darauf, dass Sie an dieser Stelle `ViewGroup.LayoutParams` erwischen und keine der anderen existierenden Varianten.

Ändern Sie nun die Breite auf einen geeigneten Wert:

```
lpTreffer.width = Math.round( massstab * 300 *
Math.min( gefangeneMuecken,muecken) / muecken );
```

Die statische Methode `round()` aus der Klasse `Math` liefert als Rückgabewert einen `long`, das Resultat der Multiplikation des `float massstab` mit den anderen Werten ergibt wiederum einen `float`. Das Ergebnis hat immer den genauesten Typ, wenn Sie unterschiedliche Typen in eine Formel stecken. `Math.min()` ermittelt den kleineren der beiden Werte in Klammern und verhindert so, dass der Balken je zu lang wird.

An der Formel in Klammern können Sie ablesen, dass der Balken anfangs die Länge 0 hat (weil `gefangeneMuecken` anfangs 0 ist), und wenn alle oder mehr Mücken gefangen wurden, 300 dip. Da der ganze Bildschirm 320 dip Platz beitet, bleibt rechts noch genügend Raum für die numerische Anzeige.

**Zur Erinnerung: Statische Methoden**

Methoden, die weder auf Attribute eines Objekts zugreifen noch auf andere Methoden, die das tun, funktionieren unabhängig vom Objekt.

Sie können sie als `static` deklarieren und ohne vorhandenes Objekt verwenden – es genügt, anstelle eines Objektnamens den Klassennamen voranzusetzen. Eclipse kennzeichnet statische Methoden, indem es ihre Namen kursiv schreibt.

Häufigster Einsatzfall für statische Methoden sind Hilfsfunktionen, die mehrere Rechenschritte zusammenfassen und dabei nur auf einfache Eingabewerte angewiesen sind:

```
class Math {
   public static int min(int a, int b) {
      if(a<b) {
         return a;
      }
      return b;
   }
}
```

So ähnlich könnte die Methode `Math.min()` implementiert sein.

Nun zum finalen Countdown: Bei 60 Sekunden Gesamtzeit pro Runde entspricht jede Sekunde 5 dip Balkenbreite (300:60=5). In Java-Code sieht das dann so aus:

```
LayoutParams lpZeit = flZeit.getLayoutParams();
lpZeit.width = Math.round( massstab * zeit * 300 / 60 );
```

(Ich habe absichtlich 300/60 statt 5 hingeschrieben, Sie werden später sehen, warum.)

Natürlich wollen Sie den Bildschirm nicht nur beim Start einer Runde aktualisieren, sondern mindestens jede Sekunde. Und das bringt uns zum nächsten Thema.

### Die verbleibende Zeit herunterzählen

Wir werden später dafür sorgen, dass die Methode zum Herunterzählen der Zeit automatisch einmal pro Sekunde aufgerufen wird. Welche Aktionen muss diese Methode dann durchführen?

- die verbleibende Zeit um 1 verringern
- manchmal eine neue Mücke anzeigen
- falls eine Mücke lange genug angezeigt wurde, die Mücke entfernen
- den Bildschirm aktualisieren
- prüfen, ob die Runde vorbei ist
- prüfen, ob das Spiel vorbei ist

Hier geschieht eine ganze Menge mehr als nur ein Countdown. Für das Anzeigen und Entfernen einer Mücke könnte man eine eigene Methode erfinden – das wäre nicht falsch, aber auch jene Methode müsste dann in regelmäßigen Abständen aufgerufen werden. Wir erledigen lieber alles an zentraler Stelle.

Bleiben zwei Details zu klären:

Erstens: Wann verschwindet eine Mücke? Dazu werden wir jeder Mücke ihr »Geburtsdatum« mitgeben, um ihr »Alter« berechnen zu können. Überschreitet sie ein Alter von z. B. zwei Sekunden, fliegt sie mit dem Blut des Spielers hinfort.

Zweitens: das »Manchmal«. Wann genau muss eine neue Mücke erscheinen? Das ist die kniffligere Frage.

Die Mücken sollen nicht in kalkulierbaren Abständen, sondern zufällig erscheinen. Deshalb müssen wir mit Wahrscheinlichkeiten arbeiten. Wie groß ist also in jeder Sekunde die Wahrscheinlichkeit, dass wir eine Mücke anzeigen müssen?

*Die Gesamtanzahl Mücken geteilt durch die Dauer der Runde ist die gewünschte Wahrscheinlichkeit.*

Überzeugen Sie sich anhand von fünf Beispielen von der Richtigkeit dieser Überlegung:

- Bei zehn Mücken ist die Wahrscheinlichkeit 10/60, das ist ein Sechstel, also 16,7 %. Das entspricht der Wahrscheinlichkeit, mit einem sechsseitigen Würfel eine bestimmte Zahl zu werfen. Probieren Sie es aus: Würfeln Sie 60-mal. Sie werden ungefähr 10-mal sechs Augen werfen. Da unsere Methode jede Sekunde einmal aufgerufen wird, insgesamt also 60-mal, entspricht das genau unserem Ziel.
- Bei 20 Mücken ist die Wahrscheinlichkeit 20/60, also ein Drittel. Bei etwa jedem dritten Aufruf wird unsere Methode also eine Mücke erzeugen, das entspricht bei 60 Aufrufen den gewünschten 20 Stück.
- Bei 60 Mücken ist die Wahrscheinlichkeit 60/60, also 100 %. Bei jedem Aufruf wird eine Mücke erscheinen, macht 60 Stück.
- Bei 90 Mücken ist die Wahrscheinlichkeit 90/60, also 150 %. Wir müssen auf jeden Fall jede Sekunde eine Mücke zeigen und bei jedem zweiten Aufruf eine zweite.

Wie viele Mücken zeigen wir denn nun in jeder Runde an?

Auf diese Frage gibt es unterschiedliche Antworten, die eng mit dem Schwierigkeitsgrad des Spiels verknüpft sind. Nehmen wir zunächst den einfachsten Fall: Die Anzahl der Mücken, die der Spieler in einer Runde treffen muss, ist die vorgegebene Siegbedingung. Wir erlauben ihm Fehlschläge und zeigen daher 50 % mehr Mücken an, weil der Zufallsgenerator sonst manchmal nicht genug erzeugt.

Sie sehen, dass Sie beim Bau einer Game Engine um ein bisschen Mathematik nicht herumkommen. Gerade beim Umgang mit dem Zufallsgenerator, der uns bevorsteht, ist außerdem große Vorsicht geboten, denn ein anständiges Gefühl für Wahrscheinlichkeiten liegt uns nicht in den Genen. Sonst würde nämlich kein Mensch Lotto spielen.

**Spezialfälle**

Ist Ihnen aufgefallen, dass eine Situation eintreten könnte, in der ein Spieler eine Runde unmöglich gewinnen kann, wenn die Mücken in immer gleichen Zeitintervallen erscheinen? Wurden beispielsweise erst fünf Mücken getroffen, verlangt werden aber insgesamt zehn, und es ist nur noch Zeit, um zwei anzuzeigen, ist die Runde unmöglich zu schaffen. Solche Sonderfälle können bei Spielregeln leicht auftreten, und manchmal dauert es eine Weile, bis man darauf kommt. Je früher man sich eine bessere Regel überlegt, desto besser.

Es gibt zwei Möglichkeiten, mit dem genannten Fall umzugehen:

a) mehr Mücken anzeigen

b) sofortiges »Game Over«

Solche Spezialfälle können Programmcode sehr kompliziert machen, und deshalb werden wir das für den Moment außen vor lassen. Wer nicht genug Mücken trifft, muss die Runde also zu Ende spielen, obwohl er vielleicht keine Chance mehr hat, sie zu gewinnen.

Greifen wir nun zum digitalen Würfel, einen sogenannten **Zufallsgenerator**. Auch dafür bietet Java selbstverständlich eine passende Klasse: `Random`.

Um einen Zufallsgenerator als privates Attribut einer Activity zu erzeugen, schreiben Sie einfach:

```
private Random zufallsgenerator = new Random();
```

Ein solcher Generator ist nicht ganz so zufällig wie die Lottozahlen, aber für die meisten Zwecke ausreichend. Er liefert beispielsweise Kommazahlen von 0 bis 1 (aber nie genau 1):

```
float zufallszahl = zufallsgenerator.nextFloat();
```

Die Wahrscheinlichkeit, dass eine solche Zufallszahl kleiner als ein bestimmter Prozentwert ist, entspricht genau diesem Prozentwert.

Sprich: Die Bedingung (`zufallszahl < 0.5`) trifft auf etwa 50 % der Zufallszahlen zu, (`zufallszahl < 1.0`) immer (100 %) und (`zufallszahl < 0.0`) nie (0 %). Das bedeutet für die neuen Mücken folgende einfache Bedingung:

```
if ( zufallszahl < muecken*1.5/60 ) {
   eineMueckeAnzeigen();
}
```

Die Multiplikation mit 1,5 entspricht der gewünschten Zugabe von 50 %, die Division durch 60 ist die zeitliche Skalierung.

Hieraus können Sie sich ausrechnen, dass die Bedingung ab 60/1,5 gleich 40 Mücken immer erfüllt ist. Ab Runde 4 zeigen wir also schon in jeder Sekunde eine neue Mücke an.

In dem Fall müssen wir also manchmal zwei Mücken erzeugen!

Schon wird's relativ kompliziert:

```
double wahrscheinlichkeit = muecken * 1.5f / ZEITSCHEIBEN;
if ( wahrscheinlichkeit > 1 ) {
   eineMueckeAnzeigen();
   if ( zufallszahl < wahrscheinlichkeit - 1 ) {
      eineMueckeAnzeigen();
   }
} else {
   if ( zufallszahl < wahrscheinlichkeit ) {
      eineMueckeAnzeigen();
   }
}
```

Ist die berechnete Wahrscheinlichkeit größer 1, wird zunächst auf jeden Fall eine Mücke angezeigt und dann mit um 1 verringerter Wahrscheinlichkeit eine weitere. Eine Wahrscheinlichkeit von 123 % erzeugt also eine Mücke plus in 23 % der Fälle eine weitere.

Die Methode sieht also summa summarum wie folgt aus:

```
private void zeitHerunterzaehlen() {
   zeit = zeit -1;
   float zufallszahl = zufallsgenerator.nextFloat();
   double wahrscheinlichkeit = muecken * 1.5 / ZEITSCHEIBEN;
   if ( wahrscheinlichkeit > 1 ) {
      eineMueckeAnzeigen();
      if ( zufallszahl < wahrscheinlichkeit - 1 ) {
         eineMueckeAnzeigen();
      }
   } else {
      if ( zufallszahl < wahrscheinlichkeit ) {
         eineMueckeAnzeigen();
      }
   }
   mueckenVerschwinden();
   bildschirmAktualisieren();
   if(!pruefeSpielende()) {
      pruefeRundenende();
   }
}
```

Das Ende der Methode schaut kompliziert aus – daher lenke ich Ihre Aufmerksamkeit zuerst auf die dortigen Zeilen.

### Prüfen, ob das Spiel vorbei ist

Die Bedingung für »Game Over« ist ja bekannt: Wenn die Zeit in einer Runde abgelaufen ist und der Spieler nicht die geforderte Anzahl Mücken erwischt hat, hat er verloren.

Wichtig ist dabei das Wörtchen *und*: Es handelt sich um eine Verknüpfung von zwei Bedingungen. Nur wenn beide erfüllt sind, wird die Methode »Game Over« aufgerufen.

**Die Aussagenlogik des Herrn Bool**

Im Gegensatz zur Umgangssprache ist die Bedeutung der Wörtchen *und* und *oder* in allen Programmiersprachen klar und einheitlich definiert. Da jede Aussage (»Die Zeit ist abgelaufen«) nur zwei Wahrheitswerte annehmen kann (»wahr« und »falsch«), ist es leicht, alle infrage kommenden Kombinationen aufzuschreiben:

`wahr` und `wahr` = `wahr`

`wahr` und `falsch` = `falsch`

`falsch` und `wahr` = `falsch`

`falsch` und `falsch` = `falsch`

Bei *oder* sieht die Sache anders aus. Damit eine Oder-Aussage wahr ist, genügt es, wenn *eine der beiden* verknüpften Aussagen wahr ist. Auch wenn beide wahr sind, ist die Gesamtaussage wahr:

`wahr` oder `wahr` = `wahr`

`wahr` oder `falsch` = `wahr`

`falsch` oder `wahr` = `wahr`

`falsch` oder `falsch` = `falsch`

Ach ja, und der Vollständigkeit halber erwähne ich auch den *nicht*-Operator:

nicht `wahr` = `falsch`

nicht `falsch` = `wahr`

Die boolschen Operatoren schreibt man in Java mit doppelten `&`-Zeichen **(und)**, doppelten `|`-Zeichen (**oder**) und einfachem Ausrufezeichen (**nicht**):

```
if ( zeit == 0 && gefangeneMuecken < muecken ) ...
```

Wie Sie in Kürze sehen werden, ist es sehr sinnvoll, wenn diese Methode ein Ergebnis zurückgibt:

```
private boolean pruefeSpielende() {
   if ( zeit == 0 && gefangeneMuecken < muecken ) {
      gameOver();
      return true;
   }
   return false;
}
```

Anstelle von `void` wird diese Methode mit dem Rückgabewert `boolean` definiert. Also muss sie auch passende Werte zurückgeben, und dazu dient das Schlüsselwort `return`, das den Ablauf der Methode sofort beendet und den angegebenen Wert an den aufrufenden Code zurückgibt. Falls das Spiel beendet ist, rufen wir also nicht nur die noch zu schreibende Methode `gameOver()` auf, sondern geben auch den Wahrheitswert `true` zurück. Im anderen Fall ist das Ergebnis der Methode `false`. Beachten Sie, dass die letzte Zeile der Methode nicht ausgeführt wird, wenn die `if`-Bedingung erfüllt ist und das `return true` in den geschweiften Klammern die Methode vorzeitig verlässt.

Jetzt verstehen Sie auch, was am Ende von `zeitHerunterzaehlen()` geschieht:

```
if(!pruefeSpielende()) {
   pruefeRundenende();
}
```

Nur wenn das Spiel nicht beendet ist, wird überhaupt geprüft, ob die Runde zu Ende ist, denn das hätte überhaupt keinen Sinn. Mehr noch: Es wäre falsch, eine neue Runde zu beginnen, denn genau das geschieht in `pruefeRundenende()`.

**Kurz ist gut**

Sie sehen, dass diese Methode sehr wenig Code enthält. Das ist eine gute Nachricht! Je weniger Programmzeilen eine Methode umfasst, umso weniger Fehler kann sie enthalten, umso leichter verstehen sie andere Programmierer (oder Sie selbst nach einigen Monaten), umso schneller arbeitet die Java Runtime sie ab.

Als nützliche Regel hat sich eingebürgert, dass jede Methode sich *nur um eine Sache kümmern sollte*. In diesem Fall ist das die Prüfung auf das Ende des Spiels. Unterschätzen Sie nicht, wie wichtig es ist, immer den Überblick zu behalten!

Eine weitere Faustformel lautet: Wenn eine Methode nicht vollständig in Ihr Eclipse-Fenster passt, ist sie zu lang. Teilen Sie sie in mehrere Funktionen auf, oder lagern Sie einen Teil des Codes in eine eigene Methode aus – selbst wenn die nur an dieser Stelle verwendet wird.

Eclipse hilft Ihnen übrigens dabei. Wenn Sie Programmcode in eine eigene Methode auslagern wollen, markieren Sie ihn, klicken Sie mit der rechten Maustaste, und wählen Sie Refactor • Extract Method. Daraufhin prüft Eclipse, ob die Auslagerung möglich ist, und erlaubt Ihnen, einen Namen für die neue Methode festzulegen.

Es gibt eine ganze Menge weiterer Unterstützung für Refactoring, also Umbaumaßnahmen am Programmcode, die ich Ihnen jeweils bei passender Gelegenheit vorstellen werde.

### Prüfen, ob eine Runde vorbei ist

Die Methode, die das Ende der Runde erkennt, funktioniert ähnlich wie jene, die das Ende des ganzen Spiels erkennt: Wenn die Zeit abgelaufen ist, beginnt eine neue Runde.

Der Code sieht auf den ersten Blick sehr einfach aus:

```
private boolean pruefeRundenende() {
   if (zeit == 0) {
      starteRunde();
      return true;
   }
   return false;
}
```

Da wir davon ausgehen, dass diese Methode nur aufgerufen wird, wenn nicht ohnehin das ganze Spiel vorbei ist, genügt die `zeit`-Bedingung. Grundsätzlich wäre es nicht falsch, hier sicherheitshalber zu prüfen, ob der Spieler genug Mücken getroffen hat. Denn möglicherweise bauen Sie irgendwann das Spiel derart um, dass diese Methode auch unter anderen Umständen aufgerufen wird. Wir belassen es aber für den Moment bei der einfachsten Variante, um den Code übersichtlich zu halten.

### Eine Mücke anzeigen

Sie haben in das Layout *game.xml* bereits ein `FrameLayout`-Element eingefügt, in dem die Mücken erscheinen sollen. Das FrameLayout ist die einfachste Möglichkeit, um Elemente an einer beliebigen Stelle innerhalb eines rechteckigen Bereichs zu positionieren. Jedes `FrameLayout`-Element kann nämlich eine beliebige Anzahl anderer Elemente enthalten, die alle relativ zur linken oberen Ecke des FrameLayouts ausgerichtet werden. Fügen Sie probeweise im Layout-Editor einen ImageView hinzu, und verpassen Sie ihm Ihre Mücke als Drawable. Sie sehen, dass die Mücke links oben erscheint.

Ändern Sie nun den linken Rand (Layout Margin Left) auf `10dip`, und die Mücke rückt ein Stück nach rechts. Ähnlich funktioniert das für die Vertikale, indem Sie den oberen Rand ändern (Layout Margin Top).

Aber Vorsicht: Die Mücken dürfen nicht außerhalb des Bildschirms landen. Sie müssen daher herausfinden, wie breit und wie hoch das FrameLayout ist, und die möglichen Werte für die Mückenpositionen entsprechend begrenzen. Der maximale linke Rand einer Mücke entspricht der Breite des FrameLayouts abzüglich der Breite der Mücke. In der Vertikalen gilt dasselbe, wobei natürlich die Höhe zu berücksichtigen ist, nicht die Breite.

Höhe und Breite können Sie direkt beim Element erfragen. Sie holen sich zunächst das zugehörige `FrameLayout`-Objekt mit `findViewById()`:

```
FrameLayout spielbereich = (FrameLayout)findViewById(R.id.spielbereich);
```

Ermitteln Sie dann Breite und Höhe in Bildschirmpixeln, indem Sie zwei sehr einfache Methoden der Android-Klasse `View` aufrufen:

```
int breite = spielbereich.getWidth();
int hoehe = spielbereich.getHeight();
```

Da `View` eine Elternklasse von `FrameLayout` ist (wie alle sichtbaren Bildschirmelemente), stehen deren `public`-Methoden `getWidth()` und `getHeight()` (unter anderem) auch im Objekt `spielbereich` zur Verfügung.

Um die Maße der Mücke in Bildschirmpixeln zu ermitteln, müssen Sie die Originalmaße Ihrer Grafik noch mit der Pixeldichte des Bildschirms multiplizieren, weil Android alle Bilder automatisch hochskaliert, wenn der Bildschirm mehr als 320 Punkte breit ist.

Der Maßstab ist eine Kommazahl. Im Grunde möchten Sie sicher lieber mit ganzen Zahlen rechnen, also bauen Sie in die Berechnung der Mückengröße gleich eine Rundung mit ein:

```
int muecke_breite = (int) Math.round(massstab*50);
int muecke_hoehe = (int) Math.round(massstab*42);
```

50 und 42 sind die Breite und Höhe meiner Mückengrafik *muecke.png*. Falls Ihre Mücke andere Maße hat, verwenden Sie natürlich Ihre eigenen.

Die statische Methode `round()` aus der Klasse `Math` liefert als Rückgabewert einen `long`. Da wir sicher sein können, dass unsere Werte nie derart groß werden, dass wir irgendwann mit `long`-Werten hantieren müssen, führen wir eine Typenumwandlung in `int` durch. Dazu dient der vorangestellte Ausdruck `(int)` – ein Type-Casting.

**Casting**

Keine Schönheiten sind bei dieser Art Casting am Start, sondern unterschiedliche primitive Typen oder Klassen, die einander zugewiesen werden sollen:

```
long a = 100;
int x = (int) a;
```

Sie können nur Typen ineinander umwandeln, wenn sie passen. Versuchen Sie nicht, auf diese Weise einen String in einen TextView zu verwandeln – es wird schiefgehen, weil die Ausgangsklassen nicht kompatibel sind.

Unterscheiden muss man hier zwischen:

- Inkompatibilitäten, die schon beim Kompilieren auffallen und demzufolge von Eclipse rot angestrichen werden
- Inkompatibilitäten, die erst zur Laufzeit des Programms auftreten. In dem Fall bricht Java den Ablauf mit einer Exception ab.

Führen Sie nur Castings durch, wenn Sie genau wissen, was Sie tun – oder wenn Sie einen Film drehen wollen.

Da die Mücken an zufälligen Orten erscheinen sollen, benötigen Sie einen Zufallsgenerator, der die nötigen Koordinaten erzeugt. Sie haben bereits einen für das zufällige Erscheinen von Mücken, wozu also einen neuen erzeugen?

Abhängig davon, was für Zufallszahlen Sie gerade benötigen, können Sie unterschiedliche Methoden aufrufen. Für die Position der Mücke benötigen Sie ganze Zahlen zwischen 0 und einem maximalen Wert, nämlich der Breite des Spielfeldes minus der Breite der Mücke (und analog für die Höhe). Verwenden Sie dazu die Methode `nextInt()`:

```
int links = zufallsgenerator.nextInt( breite - muecke_breite );
int oben = zufallsgenerator.nextInt( hoehe - muecke_hoehe );
```

Damit liegen alle sachdienlichen Hinweise zum Positionieren der Mücke vor. Es wird Zeit, das eigentliche grafische Element zu erzeugen: den ImageView.

Bisher haben Sie alle Elemente erzeugt, indem Sie sie mithilfe des Layout-Editors in die Datei *game.xml* eingebaut haben. Natürlich tut Android beim Aufbau eines Screens nichts anderes, als anhand Ihrer Angaben bestimmte Objekte zu erzeugen und Attribute zu setzen.

Was Android kann, können Sie schon lange! Also erzeugen Sie einen ImageView für die Mücke:

```
ImageView muecke = new ImageView(this);
```

Der Konstruktor der Klasse `ImageView` erwartet als Parameter einen Context. Da jede Activity von der abstrakten Basisklasse `Context` erbt, können Sie einfach `this` übergeben und müssen sich für die Details nicht interessieren.

Verpassen Sie dem Objekt `muecke` nun die richtige Grafik:

```
muecke.setImageResource(R.drawable.muecke);
```

Diese Zeile entspricht der Auswahl des darzustellenden Bildes durch Rechtsklick im Layout-Editor.

Ich verrate Ihnen nicht zu viel aus dem übernächsten Kapitel, wenn ich Ihnen jetzt sage, dass Sie am besten gleich den `OnClickListener` der Mücke setzen, und zwar auf die `GameActivity` selbst, die dazu das `OnClickListener`-Interface implementieren muss:

```
public class GameActivity extends Activity implements OnClickListener
```

Fügen Sie vorerst eine leere `onClick()`-Methode ein, um das `OnClickListener`-Interface zu bedienen:

```
@Override
public void onClick(View v) {
}
```

Verknüpfen Sie die neue Mücke mit dieser Methode:

```
muecke.setOnClickListener(this);
```

Fast fertig. Um die Mücke an der gewünschten Stelle anzuzeigen, müssen Sie ein `LayoutParams`-Objekt mit den richtigen Werten füllen.

Erzeugen Sie also zunächst eines:

```
FrameLayout.LayoutParams params = new FrameLayout.LayoutParams(muecke_
breite,muecke_hoehe);
```

Leider gibt es verschiedene Klassen namens `LayoutParams`, die alle in andere Klassen eingebettet sind. Weiter oben haben Sie bereits `LayoutParams` aus `ViewGroup` verwendet, hier benötigen Sie `FrameLayout.LayoutParams`.

Setzen Sie nun den linken und den oberen Abstand, indem Sie die betreffenden Attribute des `LayoutParams`-Objekts füllen:

```
params.leftMargin = links;
params.topMartin = oben;
```

Schließlich setzen Sie die Gravitation auf links oben:

```
params.gravity = 51;
```

Jetzt fragen Sie sich zu Recht: Wie zum Teufel soll ein Mensch auf diese 51 kommen?

Die Antwort: gar nicht. Schreiben Sie die Zeile lieber wie folgt:

```
params.gravity = Gravity.TOP + Gravity.LEFT;
```

Diese Version können Sie auf Anhieb lesen, sie macht aber genau dasselbe. Denn um unverständliche Zahlencodes wie die 51 zu vermeiden, definiert Android Konstanten, die Sie stattdessen verwenden wollten. `Gravity.TOP` hat den Wert 48 und `Gravity.LEFT` den Wert 3. Beides müssen Sie sich natürlich nicht merken, denn die Klasse `Gravity` hält ja die simplen Konstanten bereit.

Endlich ist die Zeit gekommen, die Mücke auf den Bildschirm zu verfrachten:

```
spielbereich.addView(muecke,params);
```

Dieser Aufruf der Methode `addView()` fügt dem FrameLayout `spielbereich` den gewünschten ImageView `muecke` mit der Mücke an der zufälligen Stelle hinzu.

Eine Kleinigkeit fehlt noch. Sie erinnern sich, dass wir im Abschnitt »Die verbleibende Zeit herunterzählen« beschlossen haben, der Mücke ihr »Geburtsdatum« mitzugeben. Das ist offenbar eine spezielle Anforderung unseres Spiels, daher steht kaum zu erwarten, dass die Klasse `ImageView` über eine Methode `setBirthdate()` verfügt, die wir verwenden können.

Aber die Macher von Android haben unseren Bedarf vorausgeahnt. Sie haben eine Möglichkeit geschaffen, einem View nahezu beliebige Objekte anzukleben, die später wieder ausgelesen werden können: Tags. Sie kennen Tags (Aufkleber, Etiketten) vielleicht von Blogs, Fotoverwaltungen oder anderen Anwendungen.

Um mehrere verschiedene Tags ankleben und später wieder unterscheiden zu können, müssen wir jeweils eine ID definieren. Das darf leider nicht irgendeine Zahl sein, sondern Android fordert einen applikationsspezifischen Wert, den Sie als Ressource definieren müssen.

Legen Sie dazu mit dem Wizard New Android XML File eine neue Resource-Datei namens *ids.xml* im Verzeichnis *values* an. Fügen Sie der leeren Datei ein `Item`-Element hinzu. Geben Sie als Namen »geburtsdatum« ein, und legen Sie als Typ `id` fest.

Diese ID finden Sie dank der Hintergrundarbeit des Android Resource Managers unter dem Bezeichner `R.id.geburtsdatum` wieder. Pappen Sie nun den Geburtsdatum-Aufkleber auf die Mücke:

```
muecke.setTag(R.id.geburtsdatum, new Date());
```

Achten Sie beim Organisieren der Importe darauf, dass Sie `java.util.Date` erwischen. Immer wenn Sie ein Objekt dieser Klasse mit `new` erzeugen, merkt es sich das aktuelle Datum und die Uhrzeit. Später können Sie dieses `Date` mit dem dann aktuellen vergleichen, um zu entscheiden, ob die Mücke verschwinden muss.

Womit wir beim nächsten Thema wären.

### Eine Mücke verschwinden lassen

Genau wie das Hinzufügen einer Mücke erfolgt auch das Gegenteil abhängig von einer bestimmtem Bedingung: Wenn eine Mücke lange genug auf dem Bildschirm zu sehen war, soll sie verschwinden.

Formulieren wir diese Bedingung etwas anders, indem wir berücksichtigen, dass jede Mücke ihr eigenes Geburtsdatum kennt: Wenn eine Mücke auf dem Bildschirm ist, deren Geburtsdatum länger zurückliegt als eine bestimmte Zeitspanne, soll sie verschwinden.

Sie sehen, dass die Verschwinden-Methode dazu alle Mücken auf dem Bildschirm in Betracht ziehen muss. Da Sie alle Mücken dem `spielbereich`-Objekt hinzugefügt haben, können Sie sich darauf verlassen, dass es Ihnen alle Mücken liefern kann, ohne dass Sie noch irgendwo sonst eine Liste speichern müssen.

Die Anzahl der Mücken ist beispielsweise:

```
spielbereich.getChildCount()
```

Der Name dieser Methode der Klasse `ViewGroup` (von der `FrameLayout` erbt) spiegelt die hierarchische Struktur der Views wider. Alle Views, die der Klasse `ViewGroup` hinzugefügt wurden, heißen **Kinder** (Children).

Die Kinder des Spielbereichs (also die Mücken) sind direkt über eine fortlaufende Nummer erreichbar:

```
spielbereich.getChildAt(nummer)
```

Beachten Sie, dass Programmierer fast immer bei 0 anfangen zu zählen, nicht bei 1. Die erste Mücke bekommen Sie also mit:

```
spielbereich.getChildAt(0)
```

Die Nummer der letzten ist folglich die Anzahl minus 1. Sie ahnen vielleicht schon, dass Sie eine Laufvariable vom Typ `int` benötigen, um alle Kinder des Spiebereichs zu erwischen. Diese Variable wird bei 0 beginnen und als Höchstwert die Anzahl-1 annehmen.

Um alle Mücken der Reihe nach zu betrachten, benötigen wir eine Programmierstrategie, um gewisse Codezeilen (die Altersprüfung) mehrfach zu durchlaufen. Ein solches Konstrukt heißt **Schleife** (Loop). Es gibt verschiedene Möglichkeiten, die gewünschte Schleife zu programmieren. Wir benutzen in diesem Fall das folgende Schlüsselwort:

```
while(bedingung) {
   ...
}
```

In die runden Klammern schreiben Sie eine Bedingung. Solange diese Bedingung erfüllt ist (also den Wert `true` hat), wird der Code, der in den geschweiften Klammern steht, ausgeführt. Immer, wenn diese Kommandos ausgeführt wurden, kehrt die Ausführung zum `while` zurück und prüft erneut die Bedingung. Ist sie irgendwann `false`, wird der Schleifencode nicht noch einmal durchlaufen, sondern die Ausführung setzt dahinter wieder ein.

Schauen Sie sich die Schleife in Ruhe an:

```
int nummer = 0;
while(nummer < spielbereich.getChildCount() ) {
   ImageView muecke = (ImageView) spielbereich.getChildAt(nummer);
   nummer = nummer+1;
}
```

Achten Sie vor allem auf die `while`-Bedingung. Halten Sie sich vor Augen, dass sie wie gewünscht funktioniert, indem Sie im Kopf durchspielen, was passiert. Zunächst hat die Laufvariable `nummer` den Wert 0. Das ist kleiner als die Anzahl der Mücken, es sei denn, es gibt gerade keine. In dem Fall hat die Bedingung den Wert `false`, und die Schleife wird kein einziges Mal durchlaufen.

Gibt es Mücken, wird der Schleifencode ausgeführt. Die Methode `getChildAt(nummer)` wird die erste Mücke zurückgeben. Ähnlich wie schon bei der Verwendung von `findViewById()` müssen Sie auch hier den Rückgabewert explizit in einen ImageView umwandeln, weil Java an dieser Stelle nicht weiß, worum es sich bei dem Kind genau handelt – Sie schon.

Als derzeit letzte Zeile innerhalb der Schleife wird die Laufvariable um 1 erhöht. Zwar sieht diese Zeile aus Sicht eines Mathematikers fürchterlich falsch aus, aber wie Sie wissen, bedeutet das Zeichen = hier eine Zuweisung, keinen Vergleich.

**Abkürzungen**

Eine Zeile wie die folgende lässt sich auf mehrere Arten schreiben:

```
nummer = nummer +1;
```

Da Programmierer bekanntlich faul sind, haben sie meist keine Lust, den Bezeichner zweimal hinzuschreiben, und tippen nur:

```
nummer += 1;
```

Analog existieren auch Operatoren zum Subtrahieren, Multiplizieren und Dividieren:

```
nummer -= 1;
nummer *= 2;
nummer /= 2;
```

Und auch das ist noch nicht kurz genug. Wenn eine `int`-Variable genau um 1 erhöht werden soll, können Sie auch schreiben:

```
nummer++;
```

Hierzu allerdings eine Warnung: Kürzerer Code ist nicht immer übersichtlicher. Gerade der `++`-Operator wird gerne dermaßen in weitere Operationen verstrickt, dass man gewisse Nebenwirkungen übersieht. Wenn Sie Operatoren wie `++` oder `--` verwenden möchten, schreiben Sie sie sicherheitshalber in eine eigene Zeile, es sei denn, Sie wissen, was Sie tun.

Die Schleife zum Durchlaufen aller Mücken ist jetzt also bereit. Fehlt also nur die Altersprüfung. Ermitteln Sie zunächst das Geburtsdatum:

```
Date geburtsdatum = (Date) muecke.getTag(R.id.geburtsdatum);
```

Um jetzt nicht mit Tag, Monat, Jahr, Sekunden, Minuten, Stunden, Sommerzeit und Zeitzonen hantieren zu müssen (ja, so kompliziert ist unsere Zeitrechnung!), machen wir es uns einfach: Wir verwenden eine simple, wenngleich große Zahl, nämlich die Anzahl der Millisekunden seit dem 1.1.1970. Diesen auf den ersten Blick sinnlosen Wert können Sie leicht ermitteln, weil er ohnehin intern von Java zur Zeitrechnung eingesetzt wird:

```
geburtsdatum.getTime()
```

Subtrahieren Sie diesen Wert von der aktuellen Zeit, erhalten Sie das Alter der Mücke in Millisekunden:

```
long alter = (new Date()).getTime() - geburtsdatum.getTime();
```

Jetzt können Sie sehr leicht prüfen, ob die Mücke zu alt ist und entfernt werden muss:

```
if(alter > HOECHSTALTER_MS) {
   ...
}
```

Sie sehen, dass ich `HOECHSTALTER_MS` in Großbuchstaben geschrieben habe, wie es typisch für Konstanten ist. Denn diesen Wert möchten Sie möglicherweise irgendwann mal ändern, und dann wäre es ungünstig, irgendwo mitten im Code nach der betreffenden Zahl zu suchen. Die Konstante dagegen steht ganz oben in der Klasse und ist daher leicht zu finden. Nicht nur das: Der Name verrät ihre Bedeutung, in diesem Fall inklusive der Einheit. Denken Sie sich das `_MS` einmal weg: Erinnern Sie sich in ein paar Monaten noch daran, dass Sie hier einen Wert in Millisekunden und nicht in Sekunden oder Jahren eintragen müssen?

Setzen Sie die Konstante zunächst auf zwei Sekunden:

```
private static final long HOECHSTALTER_MS = 2000;
```

Und was ist zu tun, wenn das Alter zu groß ist? Der View `muecke` muss vom `spielbereich` entfernt werden:

```
spielbereich.removeView(muecke);
```

Allerdings offenbart diese Anweisung eine kleine Lücke im bisherigen Code. Denn durch das Löschen eines Views aus dem Spielbereich ändert sich sofort die Anzahl seiner Kinder! Nicht nur das: Falls es eine weitere Mücke gibt, rückt die in der numerischen Liste eins nach vorn. Würden wir jetzt einfach die Laufvariable `nummer` um 1 erhöhen, würde der nächste Schleifendurchlauf die *übernächste* Mücke erwischen und damit eine übersehen! Sie dürfen also nummer nur erhöhen, wenn Sie die aktuelle Mücke nicht entfernt haben, wenn sie also nicht alt genug war. Das ist genau der richtige Moment, um Ihnen das Schlüsselwort `else` näherzubringen:

```
if(alter > HOECHSTALTER_MS) {
   spielbereich.removeView(muecke);
} else {
   nummer++;
}
```

Hinter dem `else` steht ein weiterer durch geschweifte Klammern begrenzter Code-Block. Dieser wird genau dann und *nur* dann ausgeführt, wenn die `if`-Bedingung nicht wahr ist, sondern falsch.

### Das Treffen einer Mücke mit dem Finger verarbeiten

Sie haben allen Mücken bereits den richtigen `OnClickListener` verpasst und eine leere Methode `onClick()` geschrieben. Nun gilt es, auf den Fingerdruck zu reagieren.

Überlegen Sie, was beim Treffen einer Mücke alles geschehen muss:

- Anzahl getroffener Mücken um 1 erhöhen
- Punktzahl erhöhen
- den Bildschirm aktualisieren
- die Mücke entfernern

Komplikationen sind hier nicht in Sicht – Sie wissen schon ganz genau, wie Sie das alles bewerkstelligen können. Vermutlich haben Sie die nötigen vier Zeilen schon eingetippt, während Sie dies hier lesen. Daher behellige ich Sie nur mit einem kleinen Hinweis: Spieler lieben es, viele Punkte zu bekommen. Deshalb

lieben sie Flipperautomaten. Kommen Sie also nicht auf die Idee, für jede Mücke bloß einen Punkt zu verteilen – geben Sie dem Spieler gleich 100. Weder programmiertechnisch noch spieltechnisch macht es einen Unterschied – aber ein Highscore von 6.700 klingt einfach viel besser als 67!

Auch das Entfernen von Mücken kennen Sie schon – und da der Methode `onClick()` der angeklickte View (also die Mücke) freundlicherweise als Parameter übergeben wird, sieht das Resultat wirklich übersichtlich aus:

```
public void onClick(View muecke) {
  gefangeneMuecken++;
  punkte += 100;
  bildschirmAktualisieren();
  spielbereich.removeView(muecke);
}
```

### »Game Over«

Das war's! Aus und vorbei! Vor allem bedeutet das: Es dürften keine Mücken mehr erscheinen. Der weitere Aufruf der Countdown-Methode muss also unterbleiben. Außerdem zeigen wir dem Spieler den Schriftzug »Game Over« an. Wenn er drauftippt, schicken wir ihn zurück zum Hauptbildschirm.

Den Schriftzug werden wir mit einem Dialog realisieren. Das sind grafische Elemente, die im Vordergrund des aktuellen Bildschirms eingeblendet werden. Gleichzeitig setzen sie alle Funktionen, die sie verdecken, außer Kraft.

Ein solcher Dialog basiert auf einer Layout-Datei. Im vorliegenden Fall ist die nicht sonderlich kompliziert: Sie besteht aus einem halb transparenten Hintergrund und einem Game-Over-Schriftzug.

Legen Sie also ein neues Layout unter dem Namen *gameover.xml* an. Drücken Sie Strg + N, und wählen Sie den Wizard für eine neue Android-XML-Datei. Tragen Sie den Dateinamen *gameover.xml* ein, wählen Sie LAYOUT als Resource-Typ und unten im Wizard FRAMELAYOUT als Wurzelelement.

Ändern Sie das Property BACKGROUND des FrameLayouts, um es halb durchsichtig zu machen. Ein passender Farbcode ist `#88888888`.

Fügen Sie dem FrameLayout einen Large TextView hinzu, dem Sie einen neuen String mit dem Inhalt »Game Over« verpassen. Setzen Sie LAYOUT GRAVITY auf `center`, wählen Sie eine hübsche TEXTCOLOR, und setzen Sie den TEXTSTYLE auf `bold` (Abbildung 5.9).

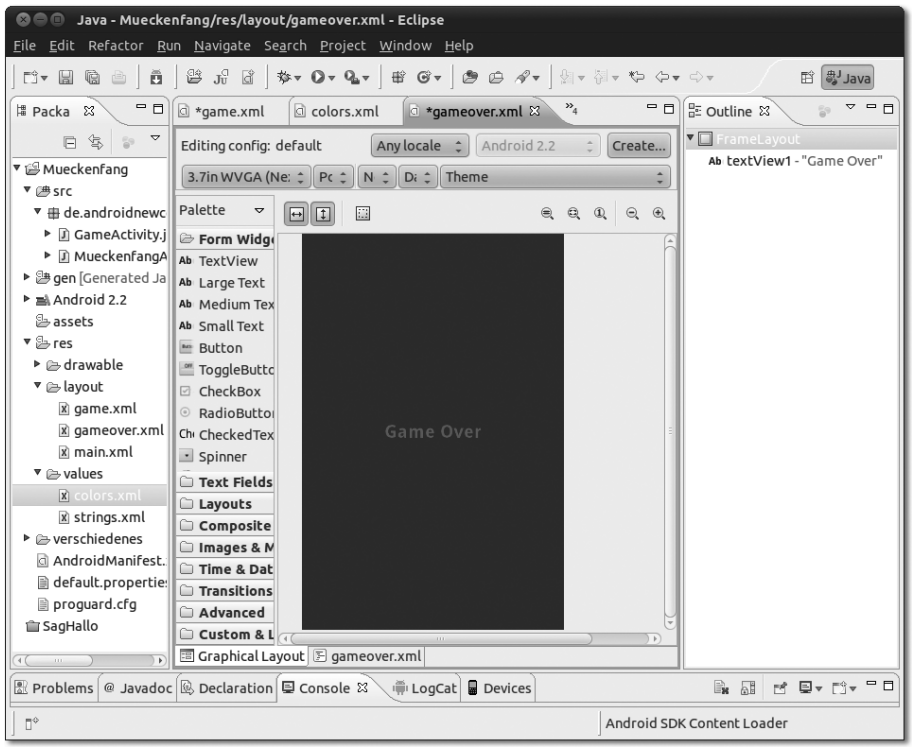



**Abbildung 5.9** Die berühmten letzten Worte

Um diesen Dialog anzuzeigen, erzeugen Sie in der Methode `gameOver()` zunächst ein passendes `Dialog`-Objekt:

```
Dialog dialog = new Dialog(this, android.R.style.
Theme_Translucent_NoTitleBar_Fullscreen);
```

Dabei müssen Sie einen Style angeben; in diesem Fall wählen wir einen ohne irgendwelche Ausschmückungen, die nur stören würden.

Damit dieser Dialog das richtige Layout anzeigt, verwenden Sie die altbekannte Methode `setContentView()`, nur eben nicht in Ihrer Activity, sondern im Dialog:

```
dialog.setContentView(R.layout.gameover);
```

Schließlich zeigen Sie den Dialog einfach an:

```
dialog.show();
```

Viel mehr über Dialoge erfahren Sie in Abschnitt 10.3, »Dialoge«.

Sie haben sicher bemerkt, dass ich Ihnen noch nicht verraten habe, wie Sie das Erscheinen weiterer Mücken unterbinden. Das hängt hauptsächlich damit zusammen, dass ich Ihnen auch noch nicht erklärt habe, wie Sie welche erscheinen lassen. Denn die zentrale Methode `zeitHerunterzaehlen()` wird ja überhaupt noch nicht aufgerufen.

Sie ahnen es schon: Dazu kommen wir als Nächstes.

### Der Handler

Nein, in dieser Überschrift fehlen keine Pünktchen auf dem a. Gemeint ist der englische Begriff **Handler**, für den es keine Übersetzung gibt, die verständlich machen würde, worum es sich eigentlich handelt. Deshalb müssen Sie sich wohl oder übel einfach den Begriff merken, und ich erkläre Ihnen jetzt, welches Mysterium sich dahinter verbirgt.

Halten Sie sich vor Augen, dass auf Ihrem Handy eine Menge Dinge quasi gleichzeitig passieren: Sie spielen ein Spiel, es erscheint eine neue E-Mail für Sie, die Bildschirmhelligkeit wird automatisch angepasst, und manchmal klingelt das Gerät sogar, weil Sie jemand anruft. Was aber tut das Android-System dabei am meisten?

Warten.

Denn nicht nur Tastendrücke werden ereignisorientiert verarbeitet, sondern im Grunde alles. Das System wartet, bis etwas geschieht, und erledigt dann die zugehörige Aufgabe, um anschließend mit dem Warten fortzufahren.

In der Steinzeit der Computerprogrammierung (also vor 20 bis 30 Jahren) sah die Lage meistens noch anders aus: Sobald man ein Programm startete, lief es, bis es fertig war, und kein anderes konnte währenddessen etwas anderes tun (ausgenommen solche mit Sondererlaubnis). Wenn ein Programm auf eine Benutzereingabe wartete, tat es das in einer tumben Endlosschleife. Während dieser nutzlos verschwendeten Zeit durfte kein anderes Programm aktiv werden. Dieses Single Processing ist längst auf dem Müllhaufen der Geschichte gelandet, genau wie Dampflokomotiven, Röhrenfernseher und Wählscheibentelefone.

Praktisch alle Systeme, egal, ob Windows, Linux, Mac oder Android, erlauben heutzutage Multitasking. Dabei dürfen beliebig viele Programme (**Tasks** oder **Processes**) gleichzeitig laufen (siehe Abbildung 5.10) – indem sie die ganze Zeit fast nichts anderes tun, als zu *warten*.

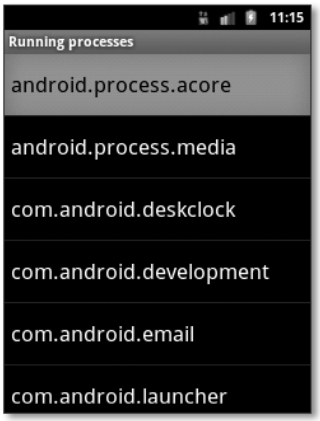



**Abbildung 5.10** Die Dev Tools – eine App, die im Emulator vorinstalliert ist – zeigen Ihnen die laufenden Prozesse. Es gibt auch eine Reihe Apps im Market, die das können.

In Android gibt es für jede App eine **Ereigniswarteschlange** (Eventqueue). Wenn mindestens ein **Ereignis** (Event) in der Warteschlange steht, für das eine App zuständig ist, erledigt sie baldmöglichst die damit verbundenen Aufgaben und schnappt sich das nächste Ereignis, falls vorhanden. Die Reaktion auf das Antippen des Bildschirms (`onClick()`) ist ein Beispiel dafür: Ihre App führt ein paar Programmzeilen aus und gibt die Kontrolle wieder ab. So wird keine Rechenzeit verschwendet.

Es ist wichtig, dass die Verarbeitung schnell geschieht, weil die App erst danach weitere Aufgaben erledigen kann, die unterdessen in der Warteschlange gelandet sind. Bildlich gesprochen: Ihre App kann nicht ohne Weiteres an mehreren Stellen gleichzeitig sein.

Wenn Sie eine Methode schreiben würden, die eine mehrere Sekunden dauernde Berechnung durchführt, dann halten Sie sich vor Augen, dass in dieser Zeit das Antippen von Bedienungselementen keine sichtbare Wirkung hat! Das ist ein derart unerwünschtes Verhalten, dass Android es gnadenlos bestraft. Sie haben vielleicht schon diese Dialoge gesehen, die Ihnen erklären, dass eine App nicht antwortet. Ich bin sicher, in solchen Fällen haben Sie schon oft entnervt den Beenden-Button angeklickt. Recht so: Es ist die verdiente Strafe für eine benutzerunfreundliche Programmierung.

Bis hierher habe ich Ihnen erklärt, dass Sie nicht einfach schreiben können:

```
while(!spielZuende) {
   zeitHerunterzaehlen();
   warteEineSekunde();
}
```

Dies ist nämlich ein Beispiel für **blockierende** Programmierung anno 1985.

Die Antwort auf die Frage, wie man so etwas denn nun im 21. Jahrhundert löst, führt uns zurück zum Handler.

Der Handler erlaubt den Zugriff auf die aktuelle Ereigniswarteschlange Ihrer App. Sie können selbst Ereignisse erzeugen und in die Warteschlange stellen. Und der Clou an der Sache ist: Dabei können Sie eine zeitliche Verzögerung angeben! Auf diese Weise erzeugen Sie Ereignisse, die erst nach einer gewissen Zeit ausgeführt werden – zum Beispiel den Aufruf von `zeitHerunterzaehlen()` nach einer Sekunde.

Und damit haben wir alle Bausteine zusammen:

- beim Start der Runde ein `zeitHerunterzaehlen()`-Ereignis in die Warteschlange stellen (mit einer Sekunde Verzögerung)
- am Ende von `zeitHerunterzaehlen()` ein weiteres `zeitHerunterzaehlen()`-Ereignis in die Warteschlange stellen (mit einer Sekunde Verzögerung), wenn nicht Runde oder Spiel zu Ende sind

Wie sieht nun das Ereignis genau aus?

Lassen Sie uns ein Ereignis bauen, das einen Verweis auf den auszuführenden Programmcode enthält (nämlich `zeitHerunterzaehlen()`). Das Stichwort für diese elegante Lösung heißt **Runnable**. Dahinter verbirgt sich zunächst einmal lediglich ein sehr einfaches Java-Interface:

```
public interface Runnable {
   public void run();
}
```

Klassen, die dieses Interface implementieren, müssen also eine Methode namens `run()` besitzen. Damit wird die Klasse **ausführbar** (**runnable**).

Wenn Sie Ihrer Activity dieses Interface (und die `run()`-Methode) verpassen, können Sie mit dem Handler ein Ereignis erzeugen, das nach einer Sekunde genau diese `run()`-Methode aufruft.

Erzeugen Sie aber zuerst ein Attribut mit dem Handler selbst, und zwar gleich bei der Deklaration, weil wir in der Activity immer nur genau eine Instanz benötigen:

```
private Handler handler = new Handler();
```

Lassen Sie die `GameActivity` das Interface `Runnable` implementieren:

```
public class GameActivity extends Activity implements OnClickListener,
Runnable;
```

Verwenden Sie den Handler, um am Ende von `starteRunde()` das Ereignis verzögert in die Warteschlange zu stellen:

```
private void starteRunde() {
   ...
   handler.postDelayed(this,1000);
}
```

Dabei verwenden Sie die Methode `postDelayed()` und übergeben ihr ein Runnable (nämlich die eigene Activity). Um was für eine Klasse es sich genau handelt, ist der Methode piepegal: Sie interessiert sich nur für das Runnable, und sie wird auch nichts anderes tun, als dafür zu sorgen, dass die Methode `run()` aufgerufen wird. Das ähnelt dem Bestellen beim Pizzataxi: Welcher Pizzabäcker sich darum kümmert und wie er aussieht, ist relativ egal – Hauptsache, er liefert.

Der zweite Parameter ist die gewünschte Verzögerung, in diesem Fall 1.000 Millisekunden, also eine Sekunde.

Schreiben Sie nun die simple Methode `run()`:

```
@Override
public void run() {
   zeitHerunterzaehlen();
}
```

Diese Methode muss `public` ein, denn sie wird von außen (durch die Ereignisverwaltung) aufgerufen. In der Methode rufen Sie natürlich `zeitHerunterzaehlen()` auf, und an deren Ende erzeugen Sie das nächste Ereignis:

```
private void zeitHerunterzaehlen() {
   ...
   handler.postDelayed(this, 1000);
}
```

Halten Sie sich vor Augen, was geschieht: Beim Start der Runde wird ein Ereignis in die Warteschlange gestellt, das nach einer Sekunde verarbeitet wird und zum Auruf Ihrer Methode `run()` führt. Dann wird die Game Engine aktiv und zählt die Zeit herunter – mit allem, was dazugehört. Schließlich erzeugt sie das nächste Ereignis, und eine Sekunde später passiert dasselbe noch mal.

Irgendwann aber muss die schönste Endlosschleife beendet werden, sonst würden immer mehr Mücken erscheinen, obwohl das Spiel längst beendet ist. Die betreffenden Bedingungen kennen wir bereits, und auch die Prüfmethoden sind schon darauf vorbereitet. Sorgen Sie also dafür, dass nur dann ein weiteres Ereignis erzeugt wird, wenn nicht Spiel oder Runde beendet sind, indem Sie den `postDelayed()`-Aufruf verschieben:

```
if(!pruefeSpielende()) {
   if(!pruefeRundenende()) {
      handler.postDelayed(this, 1000);
   }
}
```

Diese ineinander verschachtelten `if`-Bedingungen sorgen für das gewünschte Verhalten: Wenn das Spiel nicht zu Ende ist, wird geprüft, ob die Runde zu Ende ist, und nur wenn auch das nicht der Fall ist, geht das Spiel weiter mit dem nächsten Ereignis.

## 5.4 Der erste Mückenfang

Haben Sie den Code eingetippt und nachvollzogen? Alle eventuellen Tippfehler beseitigt? Wenn nicht, macht das auch nichts: Sie finden das fertige Projekt natürlich auf der Buch-DVD im dortigen *workspace*-Verzeichnis.

Es wird Zeit, das Spiel zu starten!


**Abbildung 5.11** Möge die Jagd beginnen.

## Retrospektive

Die ersten paar Spielrunden werden Ihnen eine ganze Reihe Erkenntnisse bringen.

Auf dem Emulator werden Sie ziemlich schnell feststellen, dass Sie mit der Maus relativ langsam sind. Hier zeigt sich der Unterschied zwischen den Welten: PC-Spiele müssen mit der Maus funktionieren, Android-Spiele mit Touchscreen. Berücksichtigen Sie das, wenn Sie sich ein Spielkonzept überlegen.

Im Gegensatz zu anderen Apps kommt es bei Spielen nicht nur darauf an, ob sie fehlerfrei funktionieren. Sie müssen außerdem weitere Kriterien erfüllen:

- Spiele dürfen nicht zu schwer sein: Sie müssen leicht beginnen und langsam schwieriger werden.
- Spiele müssen sofort verständlich sein. Niemand nimmt sich die Zeit, eine Anleitung zu lesen. Anspruchsvollere Spiele müssen daher zwingend ein Tutorial enthalten, das den Spieler an die Hand nimmt.
- Spiele müssen Spaß machen.

Das ist alles leicht gesagt, und die Schlussfolgerung lautet: Sie müssen Spiele noch mehr testen als andere Apps. Geben Sie ein Spiel mehreren Spielern in die Hand: Leuten mit schnellen Fingern, aber auch Personen, die noch nie ein Smartphone in der Hand hatten. Beobachten Sie Ihre Tester, und ziehen Sie Rückschlüsse.

Auch wenn Sie das Spiel nicht gleich komplett umschreiben können, an einigen Parametern können Sie leicht drehen.

Ist die Mückenjagd am Anfang zu einfach? Erscheinen so wenige Mücken, dass den Spielern langweilig wird?

Verdoppeln Sie die Anzahl der Mücken in der Methode `starteRunde()`:

```
muecken = runde * 20;
```

Aber Vorsicht, spätestens ab Runde 4 werden Sie von einem Schwarm überfallen, dessen Sie kaum mehr Herr werden.

Ist der Sekundentakt zu langsam?

Nun, im Vergleich mit der Zeit, die Sie beispielsweise mit dem Lesen dieses Kapitels verbringen, ist eine Sekunde ziemlich kurz. Aber sie ist lang in Relation zum Aufbau eines Fernsehbildes (alle 20 Millisekunden) oder zu der zeitlichen Auflösung, die unser Spiel maximal erreichen kann (1 Millisekunde).

Wir sind nicht auf den Sekundenrhytmus festgelegt. Um den Spielablauf zu verbessern, ändern Sie probeweise die Zeitscheibe (also das zeitliche Intervall unserer Game Engine) von einer ganzen auf eine Zehntelsekunde (100 ms).

Das ist gar nicht so kompliziert, denn Sie müssen dazu nicht viel tun. Die Berechnungen bleiben gleich, lediglich die Anzahl der Zeitscheiben ändert sich von 60 auf 600 und das Intervall für die Handler-Ereignisse von 1.000 auf 100. Um diese Werte leicht an allen Stellen, an denen sie vorkommen, ändern zu können, machen Sie sie zunächst zu Konstanten: Markieren Sie eine `1000` irgendwo in der `GameActivity`, und wählen Sie im Kontextmenü REFACTOR • EXTRACT CONSTANT (Abbildung 5.12).

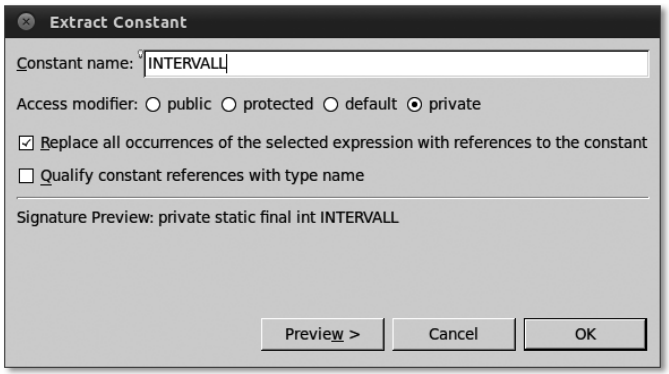



**Abbildung 5.12** Lassen Sie Eclipse alle vorkommenden Werte 1.000 durch eine Konstante ersetzen, indem Sie das erste Häkchen setzen.

Auf diese Weise ersetzt Eclipse alle Zahlen 1.000 durch eine einzige Konstante `INTERVALL`, deren Wert Sie dann leicht auf 100 ändern können.

Verfahren Sie genauso mit der Zahl 60, indem Sie eine Konstante `ZEITSCHEIBEN` einführen und sie anschließend auf 600 setzen.

Sie sehen jetzt übrigens, warum es sinnvoll war, bei der Berechnung der Länge des Zeitbalkens nicht durch 5 zu teilen, sondern zu schreiben: *300/60. Denn nur so konnte die Zahl 60 durch die Konstante `ZEITSCHEIBEN` ersetzt werden. Sich die einfache Kopfrechenaufgabe 300/60 vom Computer abnehmen zu lassen war im Nachhinein also eine gute Idee!

Immer denselben Hintergrund anzustarren wird Ihnen langweilig?

Gehen Sie an die frische Luft, machen Sie einen Spaziergang, und fotografieren Sie die Gegend. Skalieren Sie die Bilder auf 640 x 854 Pixel, und speichern Sie sie unter Dateinamen mit fortlaufender Nummer im Verzeichnis *drawable*: *hintergrund1.jpg*, *hintergrund2.jpg* etc.

Erweitern Sie die Methode `starteRunde()` um die Anzeige des jeweiligen Hintergrunds. Verpassen Sie dazu zunächst dem äußersten LinearLayout im Game-Layout die ID `hintergrund`.

Es gibt hier nur eine kleine Schwierigkeit: Sie können nicht einfach wie bisher die gewünschte Ressource mit `R.drawable.hintergrund` (plus Zahl) referenzieren, weil Sie sich nicht darauf verlassen können, dass der Android Resource Manager fortlaufende Zahlenwerte dafür vergibt.

Aber es gibt noch einen weiteren Weg, sich das gewünschte Bild zu holen:

```
int id = getResources().getIdentifier("hintergrund"+Integer.toString
(runde), "drawable", this.getPackageName());
```

Da wir den richtigen Resource-Identifier nicht kennen, holen wir ihn uns einfach anhand seines Namens, der aus `"hintergrund"` und der angehängten Rundenzahl besteht. Der zweite Parameter der Methode `getIdentifier()` ist der gewünschte Typ der Ressource, der dritte ist der Paketname Ihrer App.

Füttern Sie nun das Bild in den Hintergrund, aber nur, wenn es existiert (vielleicht erreicht ein Spieler überraschend eine Runde, für die Sie kein Foto gemacht haben):

```
if(id>0) {
   LinearLayout l = (LinearLayout) findViewById(R.id.hintergrund);
   l.setBackgroundResource(id);
}
```

Bei der Gelegenheit schalten Sie das ganze Spiel in den Vollbild-Modus um: Öffnen Sie das Android-Manifest, und geben Sie als THEME folgende magische Zeile ein:

```
@android:style/Theme.NoTitleBar.Fullscreen
```

Leider können Sie dies nicht mit dem BROWSE-Button auswählen, daher müssen Sie es eintippen.

Sie können ohne weitere Kenntnisse schon jetzt das Spiel um weitere Kniffe erweitern. Wie wäre es zum Beispiel mit einem gelegentlich erscheinenden Elefanten? Wer den antippt, bekommt gemeinerweise 1.000 Punkte abgezogen. Sie können dazu die Methode `eineMueckeAnzeigen()` ändern, indem Sie zufallsgesteuert manchmal anstelle der Mücke ein Bild von einem Elefanten reinmogeln:

```
if(rnd.nextFloat() < 0.05) {
   muecke.setImageResource(R.drawable.elefant);
   muecke.setTag(R.id.tier,ELEFANT);
} else {
   muecke.setImageResource(R.drawable.muecke);
}
```

Bei der gewählen Wahrscheinlichkeit von 0,05 erscheinen in 5 % aller Fälle Elefanten anstelle von Mücken. Um die Tiere voneinander zu unterscheiden, verwenden Sie einfach ein Tag. Dazu müssen Sie eine weitere ID in der Datei *ids.xml* anlegen, außerdem definieren Sie eine Konstante `ELEFANT`:

```
private static final String ELEFANT = "ELEFANT";
```

Der Inhalt des Strings ist egal, aber Sie sollten sich nicht selbst eine Falle stellen, indem Sie etwas anderes hinschreiben. Anstelle eines Strings können Sie auch ein beliebiges anderes Objekt verwenden, zum Beispiel ein `Integer` mit dem Wert 1. Entscheidend ist nicht der Inhalt des Objekts `ELEFANT`, sondern nur seine persönliche Anwesenheit.

In der Methode `onClick()` müssen Sie nun prüfen, ob der Spieler eine Mücke oder einen Elefanten erwischt hat:

```
if(muecke.getTag(R.id.tier) == ELEFANT) {
   punkte -= 1000;
} else {
   gefangeneMuecken++;
   punkte += 100;
}
```

Natürlich zählt der Elefant nicht als Mücke, daher dürfen Sie `gefangeneMuecken` nur im `else`-Fall erhöhen.

Übrigens können Sie sich die Navigation im Code erleichtern, indem Sie den View namens Outline reaktivieren (Sie erinnern Sich vielleicht, dass Sie ihn kurz nach der Eclipse-Installation geschlossen haben). Das hilfreiche Fensterchen zeigt Ihnen alle Methoden und Attribute an (Abbildung 5.13).

Experimentieren Sie mit den durchgestrichenen Icons: Mit dem ersten können Sie beispielsweise die Attribute ausblenden. Wie es sich gehört, verrät Eclipse Ihnen mit einem kleinen Fähnchen, was ein Icon bewirkt, wenn Sie mit der Maus einen Moment lang darauf verharren. An den farbigen Symbolen vor den Bezeichnern können Sie erkennen, ob diese `public` (grün) oder `private` (rot) deklariert sind. Doppelklicken Sie auf einen Methodennamen oder ein Attribut, um an die betreffende Stelle im Code zu springen.


**Abbildung 5.13** Der »Outline«-View von Eclipse stellt alle Elemente einer Klasse in einer Baumstruktur dar.

In weniger als 200 Zeilen Java-Code haben Sie ein einfaches Android-Spiel verwirklicht. Um die Fähigkeiten eines Smartphones richtig auszureizen, fehlt freilich noch einiges. Aber es kommen ja auch noch ein paar Kapitel ...

*»Ich glaube, ich habe irgendwas gehört.«*
*(Unbekannter Abenteurer, der in einer Höhle voller Dracheneier herumschleicht)*

# 6 Sound und Animation

Was wäre eine zünftige Raumschlacht ohne das Zischen von Laserstrahlen und das Wummern von Explosionen? Auch der berechtigte Hinweis auf die physikalische Tatsache, dass sich Schall im Vakuum des Weltraums nicht ausbreiten kann, ändert nichts an der unbestreitbaren Tatsache: Ein Spiel muss mit allen Sinnen genossen werden, und dazu gehört unzweifelhaft der Sound.


**Abbildung 6.1** Alles andere als »Lautlos im Weltraum«: Galaxy of Fire 2, ein Weltraumspiel für Android-Geräte mit leistungsstarkem Grafikchip

Glücklicherweise bietet Ihr Android-Smartphone nicht nur die nötigen Lautsprecher (oder Ohrhörer, wenn Sie Rücksicht auf Ihre Umgebung nehmen, wie es sich gehört), sondern auch einfach benutzbare Funktionen für Android-Entwickler.

Dann also an die Arbeit!

## 6.1 Sounds hinzufügen

Genau wie Grafiken sind Sounds aus technischer Sicht **Ressourcen**, die Sie in Form von Dateien Ihrem Projekt hinzufügen müssen.

Folglich benötigen Sie zunächst die fraglichen Sounddateien. Falls Sie gerade keine Mücke samt passendem, hochempfindlichen Mikrofon parat haben, machen Sie sich keine Sorgen: Sie können einen passenden Sound ohne Weiteres selbst erzeugen. Wenn Sie diesen kreativen Schritt überspringen möchten, können Sie aber auch einfach die Sounds verwenden, die auf der Buch-DVD im Projekt *Mueckenfang2* abgelegt sind.

### Sounds erzeugen

Um den Mückenfang zu untermalen, benötigen wir offensichtlich ein passsendes Summen. Um den Spieler aber nicht zu nerven, wie es echte Mücken tun, spielen wir nur beim Auftauchen einer neuen Mücke ein zwei Sekunden langes Summen ab. Das beantwortet auch die Frage, wie sich ein Summen abschalten lässt, wenn eine Mücke erwischt wird. Das Summen soll zunächst anschwellen und dann schnell ausgeblendet werden. Möglicherweise haben Sie schon ein Mikrofon zur Hand genommen und angefangen, das Geräusch mit den Lippen zu imitieren. Das ist ein legitimer Weg, Spielsounds zu generieren; meist verfälscht man sie anschließend noch leicht, indem man die Tonhöhe verändert.

Ich möchte Ihnen aber einen rein technischen Weg zeigen, der ohne Mikrofon und Lippen auskommt. Dazu benötigen Sie ein ziemlich mächtiges Soundprogramm namens Audacity. Das ist glücklicherweise Open Source und läuft auf allen gängigen Betriebssystemen (Sie finden es auf der Buch-DVD).

Installieren und starten Sie Audacity. Es begrüßt Sie mit einem leeren Fenster, einem umfangreichen Menü und einem Haufen Bedienelemente, die Sie im Augenblick nicht benötigen.

Audacity verfügt über mehrere leicht zu bedienende Tongeneratoren. Einen davon werden wir nun verwenden, Sie finden ihn im Menü unter ERZEUGEN • TONGENERATOR(2).

Hätten Sie gewusst, dass Mückenmännchen in einer anderen Frequenz summen als Weibchen? Da bekanntlich nur die Weibchen stechen, wäre es an dieser Stelle fatal, die falsche Frequenz zu wählen. Die richtige liegt bei rund 400 Hz, also 400 Flügelschlägen pro Sekunde. Natürlich ist der Summton kein reiner Sinus; der Einfachheit halber verwenden wir eine Sägezahn-Form. Nehmen Sie also im Tongenerator die nötigen Einstellungen vor (Abbildung 6.2), und lassen Sie das Geräusch erzeugen.

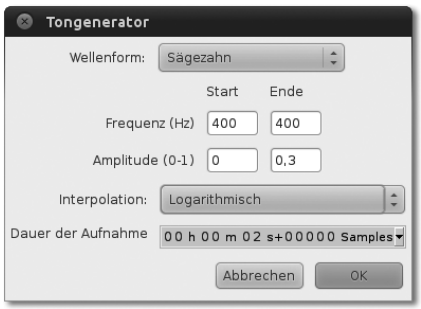



**Abbildung 6.2** Erzeugen Sie ein anschwellendes Mücken(weibchen)summen mit Audacitys Tongenerator.

Sie können sich das generierte Summen anhören, indem Sie auf den Abspielen-Button mit dem grünen Dreieck klicken.

Damit das Summen nicht zu abrupt abbricht, markieren Sie das letzte Viertel der Hüllkurve des Geräuschs mit der Maus und wählen im Menü EFFEKTE • AUSBLENDEN (Abbildung 6.3).

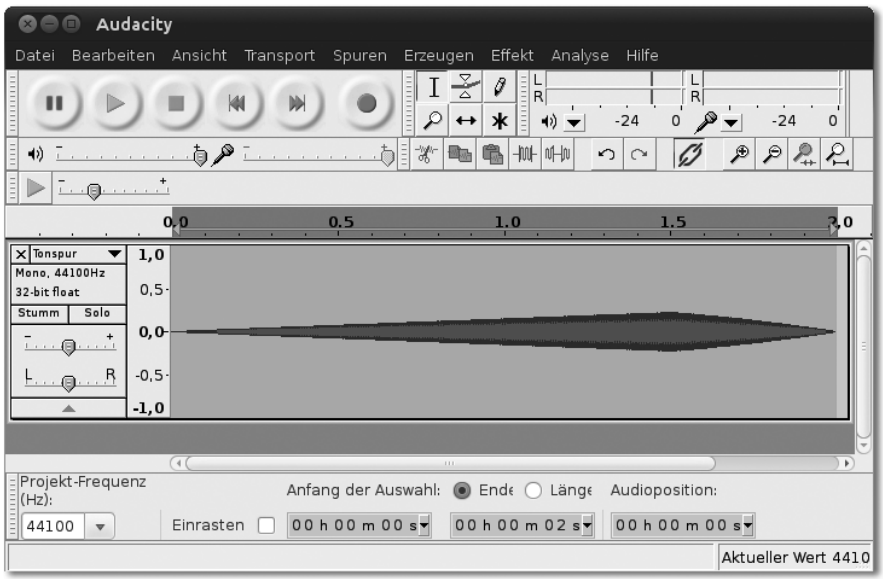



**Abbildung 6.3** Blenden Sie das Summen im hinteren Viertel aus.

Um das ganze Geräusch jetzt noch einmal zu hören, müssen Sie zunächst die gesamte Hüllkurve markieren (Strg + A). Exportieren Sie den Sound als MP3-Datei, indem Sie im Menü die Option DATEI • EXPORT wählen. Der korrekte Speicherort für die Sounddatei ist in Ihrem Projektverzeichnis der Pfad *res/raw*.

Legen Sie den Ordner *raw* an, wenn er noch nicht vorhanden ist. Achten Sie darauf, dass Sie wie bei allen Resource-Dateien keine Leerzeichen oder andere Sonderzeichen verwenden. Ich empfehle Ihnen als Dateinamen *summen.mp3*.

Wenn Sie Lust haben, experimentieren Sie ein wenig mit Audacity. Vielleicht gelingt Ihnen ein noch authentischerer, beängstigenderer Mückensound!

## Sounds als Ressource

Sie haben das Mückengeräusch als MP3-Datei gespeichert. Android unterstützt allerdings noch einen Haufen anderer Audioformate, und darin unterschiedliche Sampleraten, Auflösungen und Kanäle.

**Soundformate**

Worin unterscheiden sich eigentlich Sounds?

Im Gegensatz zur Schallplatte oder einer Musikkassette speichert ein Computer Geräusche nicht analog ab, sondern digital. Das bedeutet, dass der Verlauf des Tonsignals regelmäßig abgetastet und in diskreten Ganzzahlwerten gespeichert wird (Sampling). Dieser Vorgang geschieht beispielsweise auch, wenn Sie in ein Handy-Mikrofon sprechen.

Abhängig von der Anzahl verschiedener möglicher Lautstärkewerte und der Häufigkeit der Abtastung ergibt sich eine unterschiedliche Qualität – aber Sie erhalten logischerweise auch größere Dateien.

Eine Audio-CD hat beispielsweise eine Sampling-Rate von 44,1 kHz, also 44.100 Werten pro Sekunde. Die meisten Menschen stimmen der These zu, dass dies genügt, um keinen Unterschied zum analogen Original hören zu können (manche audiophile Hi-Fi-Puristen schreien an dieser Stelle empört auf).

Die Auflösung beträgt bei einer Audio-CD 16 Bit je Kanal, das entspricht 65.536 unterschiedlichen Lautstärkewerten. Das summiert sich zu um die 700 Megabyte Speicherbedarf für die 70 Minuten Musik, die maximal auf eine CD passen. Von Android unterstützte Dateiformate, die solche Audiodaten enthalten, sind PCM/WAV.

Waren CDs vor einigen Jahren noch das Nonplusultra, finden die meisten Menschen sie heutzutage unhandlich und ziehen MP3-Player vor. Der Trick besteht darin, die Daten nach dem Sampling zu komprimieren (im Fall des MP3-Formats mit je nach Kompressionsstärke hörbaren Verlusten). Es gibt noch weitere komprimierte Audioformate wie AAC und AAC+, die ebenfalls von Android unterstützt werden. An komprimierten Formaten unterstützt Android darüber hinaus AMR, FLAC, Ogg Vorbis.

Hinzu kommt das MIDI-Format, das nicht direkt Geräusche enthält, sondern Noten – es ist dann Aufgabe eines eingebauten Synthesizers, daraus Musik zu zaubern. Die Qualität hält sich allerdings in Grenzen.

Der Android Resource Manager legt gewohnt verlässlich einen Bezeichner namens `R.raw.summen` an, anhand dessen Sie den Sound identifizieren, wenn Sie ihn abspielen wollen (Abbildung 6.4).

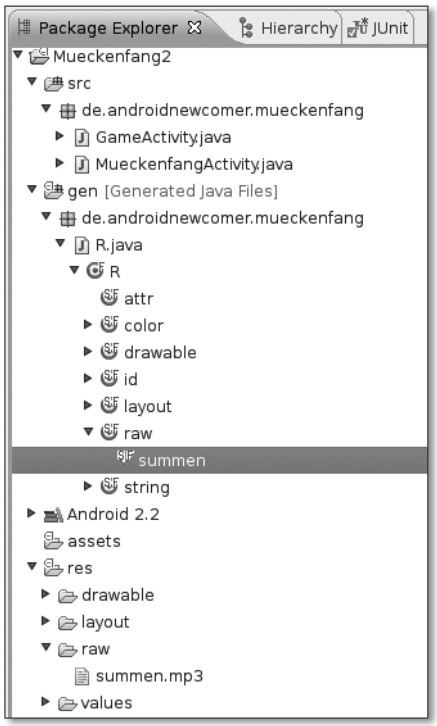



**Abbildung 6.4** Der Android Resource Manager erzeugt prompt einen Eintrag für Ihren Soundeffekt in der Klasse »R.java«.

Natürlich können Sie mehrere Dateien im *raw*-Verzeichnis ablegen, aber achten Sie darauf, dass sie trotz unterschiedlicher Endungen auch verschiedene Dateinamen aufweisen müssen. Denn die Endungen schneidet der Resource Manager ab, und aus *summen.mp3* und *summen.wav* würde zweimal `R.raw.summen` werden, was nicht funktionieren kann.

## 6.2 Sounds abspielen

Jetzt, da Sie Ihren ersten Sound ins Projekt eingefügt haben, müssen Sie ihn im richtigen Moment abspielen. Die richtige Stelle im Programmcode ist dafür zweifellos die Methode `eineMueckeAnzeigen()`.

Aber wie sorgen Sie dafür, dass das Summen im Lautsprecher ertönt?

## Der MediaPlayer

Das Android-System stellt für jede technische Finesse Ihres Smartphones Klassen zur Verfügung, die den Zugriff darauf erlauben. Das ist beim Sound nicht anders.

Die verantwortliche Klasse heißt `MediaPlayer`, und sie kann nicht nur Musik, sondern auch Videos abspielen. Es ist nicht ganz einfach, die Klasse `MediaPlayer` zu verwenden, denn sie verfügt über eine ganze Reihe Methoden und kann eine Menge unterschiedlicher Zustände annehmen – ähnlich übrigens wie eine typische GameEngine. Das hängt beispielsweise damit zusammen, dass eine Sounddatei zunächst geöffnet werden muss, um sie abspielen zu können. Sie erinnern sich sicher an die Sprachausgabe, die ebenfalls eine Initialisierungsphase benötigte. Darüber hinaus kann eine Datei abgespielt werden, es gibt eine Pause-Option, und schließlich kommt die Darbietung zu einem Ende, wenn das letzte Sample zum Lautsprecher geschickt wurde.

Ich habe Ihnen die Zustände der Klasse `MediaPlayer` in der am häufigsten vorkommenden Reihenfolge aufgeschrieben:

- Idle (untätig)
- Initialized
- Preparing
- Prepared
- Started
- Paused
- Stopped
- PlaybackCompleted

Um von einem Zustand in den nächsten zu gelangen, müssen Sie jeweils eine Methode aufrufen. Da einige Vorgänge asynchron im Hintergrund ablaufen, können Sie sich von der Klasse `MediaPlayer` benachrichtigen lassen, wenn sie den gewünschten Folgezustand erreicht hat.

**Wenn Sie's ganz genau wissen wollen ...**

Freunde unübersichtlicher Pfeildiagramme finden ein komplettes Zustandsdiagramm des MediaPlayers in der offiziellen Android-Dokumentation:

*http://developer.android.com/reference/android/media/MediaPlayer.html*

Glücklicherweise ist der Anwendungsfall, lediglich ein Mückensummen abzuspielen, kein besonders komplizierter. Trotzdem müssen Sie zwei Spezialfälle beachten:

- Ein Sound soll abgespielt werden, obwohl der vorherige noch nicht beendet ist.
- Das Spiel wird beendet, d. h., der Sound muss abgeschaltet werden.

Lassen Sie uns die Benutzung des MediaPlayers in mehreren Schritten durchgehen.

### MediaPlayer initialisieren

Da Sie das `MediaPlayer`-Objekt während der ganzen Lebensdauer der Activity benötigen, legen Sie es als Attribut an:

```
private MediaPlayer mp;
```

Erzeugen Sie dann ein `MediaPlayer`-Objekt. Man könnte vermuten, dass das mit dem Schlüsselwort `new` vonstatten geht, aber es gibt eine statische Methode in der `MediaPlayer`-Klasse, die Ihnen weitere Methodenaufrufe erspart. Das sieht dann so aus:

```
mp = MediaPlayer.create(this,R.raw.summen);
```

Hier können Sie direkt die gewünschte Resource-Referenz mitgeben. Die richtige Stelle, um das Objekt zu erzeugen, ist die `onCreate()`-Methode der `GameActivity`, schreiben Sie die obige Zeile also dort hinein.

Da Audiodaten relativ viel Speicher verbrauchen, ist es eine gute Idee, nicht das Auftauchen des Java-Entsorgungsunternehmens namens Garbage Collector abzuwarten, sondern selbst den Speicher freizugeben. Dies hat offensichtlich spätestens zu geschehen, wenn die Activity beendet wird. Um diesen Moment zu erwischen, können Sie die Methode `Activity.onDestroy()` überschreiben, die das Android-System aufruft, wenn die Activity beendet wird:

```
@Override
protected void onDestroy() {
   mp.release();
   super.onDestroy();
}
```

Wie Sie sehen, räumt `onDestroy()` nicht nur den MediaPlayer auf. Die Methode muss außerdem zwingend zum Schluss (nicht am Anfang!) die Elternmethode aufrufen, da dort weitere Aufräumarbeiten erledigt werden.

### Zurückspulen und Abspielen

Vielleicht hat Sie die Eclipse-Syntaxvervollständigung bereits darauf gebracht, wie die Methode heißt, um einen MediaPlayer zum Abspielen seines Sounds zu bringen. Ansonsten können Sie ja mal raten:

```
mp.start();
```

Schreiben Sie diese Zeile ans Ende der Methode `eineMueckeAnzeigen()`.

Allerdings ist es nicht damit getan, den Sound abzuspielen. Was passiert, wenn kurz vorher eine Mücke erschienen ist und deren Sound noch läuft?

Stellen Sie sich einen altmodischen Kassettenrekorder vor. Was müssten sie tun, wenn gerade ein Lied abgespielt wird und Sie es von vorn abspielen möchten?

- falls das Gerät gerade etwas abspielt, anhalten
- zurückspulen
- starten

Das sähe in Java wie folgt aus:

```
if(mp.isPlaying()) {
        mp.pause();
}
mp.seekTo(0);
mp.start();
```

Das können Sie so schreiben, allerdings klingt das manchmal etwas zerstückelt, weil der MediaPlayer mit der schnellen Abfolge der Befehle nicht hinterherkommt.

Die Methode zum Zurückspulen, `seekTo()`, funktioniert aber glücklicherweise auch während des Abspielens. Also können Sie die Abfrage des Abspiel-Zustands weglassen.

```
mp.seekTo(0);
mp.start();
```

Ergänzen Sie diese Zeilen am Ende von `eineMueckeAnzeigen()`, und probieren Sie das Spiel aus.

Sie werden feststellen, dass die Mücke weitersummt, selbst wenn Sie sie mit dem Finger zerquetschen. Da es aber gerade der Sinn des Spiels ist, die Nervensägen zum Schweigen zu bringen, pausieren Sie den MediaPlayer beim Treffen einer Mücke in der Methode `onClick()`:

```
mp.pause();
```

Verzieht sich eine Mücke durch die Methode `mueckenVerschwinden()`, müssen Sie den MediaPlayer ebenfalls anhalten. Ergänzen Sie also dieselbe Zeile wie in `onClick()` auch in der Methode `mueckenVerschwinden()`, und zwar im `if`-Block, der das Alter der Mücke abfragt:

```
if(alter > HOECHSTALTER_MS) {
   spielbereich.removeView(muecke);
   mp.pause();
}
```

Wenn Sie möchten, können Sie nun ein zweites Geräusch einbauen. Wie es klingt, wenn man eine Mücke zerquetscht, überlasse ich Ihrer Fantasie. Sie müssen lediglich für jeden Sound einen eigenen MediaPlayer erzeugen. Beachten Sie allerdings, dass das bei vielen Sounds eine Menge Speicher kostet, sodass Sie unter bestimmten Umständen besser einen einzelnen MediaPlayer recyceln und jeweils die nötige Audioressource so spät wie möglich laden. Leider müssen Sie dann aber mit einer gewissen Verzögerung rechnen. Für Sounds, die exakt zum richtigen Zeitpunkt ertönen müssen, eignet sich diese Methode nicht. Beispielsweise beim Abspielen von Songs geht es aber gar nicht anders: Sie können schlecht für jedes Musikstück einen MediaPlayer erzeugen und initialisieren. Das würde viel zu viel Speicher verbrauchen.

## 6.3 Einfache Animationen

Android verfügt über ziemlich umfangreiche Funktionen, mit denen Sie auf einfache Weise grafische Objekte in Bewegung versetzen können. Damit können Sie zwar weder die hübsche Zauberin in einem Fantasy-Rollenspiel zum Tanzen bringen noch Mücken zum Fliegen. Aber keine grafische Benutzeroberfläche kommt heute ohne diese kleinen Effekte aus, die Sie nur dann wahrnehmen, wenn Sie fehlen: Buttons werden scheinbar in den Bildschirm gedrückt, Pfeile blinken, Bitte-warten-Icons rotieren. Solche Animationen definieren Sie in speziellen XML-Dateien, und im Programmcode starten Sie sie je nach Bedarf.

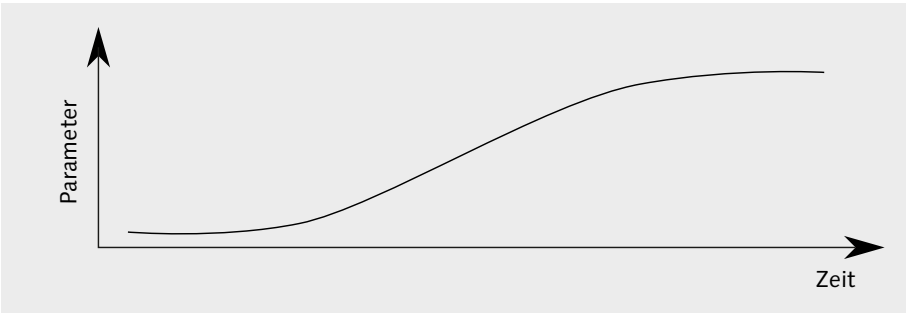



**Abbildung 6.5** Bei einer Animation wird ein Parameter allmählich verändert. Je weicher die Kurve, desto mehr freut sich das Auge.

Grundsätzlich funktioniert jede Animation auf dieselbe Art: Ein bestimmter Parameter eines Objekts wird mit der Zeit allmählich verändert (Abbildung 6.5). Dabei haben Sie eine Menge Freiheiten. Welche Parameter Sie auf diese Weise beeinflussen können, zeige ich Ihnen an einigen Beispielen.

## Views einblenden

Lassen Sie uns zunächst den Hauptbildschirm beim Starten des Spiels weich einblenden. Der Parameter, der dabei animiert wird, ist der Alpha-Wert. Ein Alpha-Wert von 1 bedeutet, dass ein Objekt sichtbar ist (das ist der Normalfall). Unsichtbar ist ein Objekt, wenn Alpha 0 ist. Bei Werten dazwischen ist das Objekt transparent: Je näher Alpha der 0 kommt, desto weniger ist davon zu sehen.

Wenn Sie also den Alpha-Wert langsam von 0 bis 1 ändern, beispielsweise im Laufe einer Sekunde, blenden Sie ein Objekt sanft ein. Wichtig ist, dass diese Änderung ruckelfrei geschieht, aber darum kümmert sich Android. Sie müssen nur die Animation definieren und beim Aufbau des Bildschirms starten.

Es wird Sie nicht wundern, dass eine Animationsdefinition eine Ressource ist und folglich in das Verzeichnis *res* gehört, genauer gesagt: in ein Unterverzeichnis namens *anim*. Legen Sie also ein solches an, anschließend benutzen Sie den Android XML Wizard, um eine neue Animation zu erstellen. Wählen Sie als Dateinamen *einblenden.xml*, und stellen Sie am unteren Rand des Wizards das Wurzelelement ALPHA ein (Abbildung 6.6).

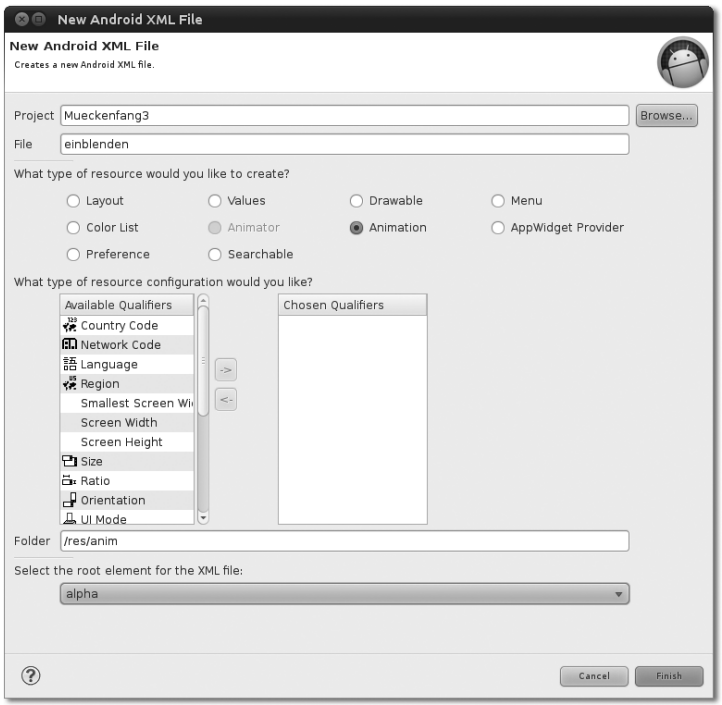



**Abbildung 6.6** Der XML Wizard erstellt Ihnen eine XML-Datei, die eine Animation beschreibt. Achten Sie darauf, das richtige Wurzelelement zu wählen.

Tja, und was Sie anschließend sehen, ist ein bisschen unerfreulich:

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

Sie werden erfolglos nach einer hübschen Benutzeroberfläche suchen, die die nackte XML-Datei vor Ihnen versteckt. Das Android-Eclipse-Plugin ermöglicht Ihnen lediglich die manuelle Bearbeitung – vielleicht ändert sich das in einer zukünftigen Version, aber im Moment kommen Sie nicht an der direkten Arbeit am XML-Code vorbei.

Ich werde Ihnen jetzt nicht die Feinheiten von XML erklären, denn die meisten Animationen lassen sich sehr übersichtlich definieren.

Schauen Sie sich das folgende Beispiel an:

```
<?xml version="1.0" encoding="utf-8"?>
<alpha xmlns:android="http://schemas.android.com/apk/res/android"
android:fromAlpha="0.0"
android:toAlpha="1.0"
android:duration="1000" />
```

Dieser XML-Code beschreibt eine Animation, die ein Objekt einblendet.

Drei Attribute innerhalb des `alpha`-Elements legen die gewünschten Parameter für die Animation fest. Da sind der Anfangswert (`android:fromAlpha`), der Endwert (`android:toAlpha`) und die Dauer (`android:duration`) in Millisekunden. Es gibt außerdem ein sogenanntes **Namespace-Attribut** (das mit dem `xmlns`), das immer gleich lautet. Es sorgt unter anderem dafür, dass Sie innerhalb dieses XML-Codes die Vorzüge der Syntaxvervollständigung von Eclipse genießen können. Tippen Sie zum Beispiel `android:` ein, und drücken Sie anschließend Strg + Leertaste, oder warten Sie einen Moment. Sofort können Sie unter den erlaubten Attributen wählen. Sogar auf akzeptable Werte zwischen den Anführungszeichen prüft Eclipse Ihre Eingaben.

Höchste Zeit, die Animation auszuprobieren!

Als ersten Schritt laden Sie die Animation in der `onCreate()`-Methode der `MueckenfangActivity`:

```
animationEinblenden = AnimationUtils.loadAnimation(this,
R.anim.einblenden);
```

Deklarieren Sie das nötige Attribut weiter oben in der Klasse, wo schon die restlichen versammelt sind. Am schnellsten geht das, indem Sie Eclipse selbst den

Fehler korrigieren lassen, der durch die fehlende Deklaration entsteht: Drücken Sie Strg + 1, und wählen Sie den zweiten Korrekturvorschlag (CREATE FIELD).

Die richtige Stelle, um die Animation zu starten, ist die `onResume()`-Methode. Holen Sie sich zunächst eine Referenz auf den Hintergrund-View. Falls Sie diesem LinearLayout noch keine ID verpasst haben, nennen Sie es `wurzel`. Anschließend starten Sie die Animation:

```
View v = findViewById(R.id.wurzel);
v.startAnimation(animationEinblenden);
```

Da die Methode `startAnimation()` in der Basisklasse `View` definiert ist, ist es an dieser Stelle nicht erforderlich, das Resultat von `findViewById()` in `LinearLayout` zu casten.

Probieren Sie's aus: Der Bildschirm wird jetzt beim Start weich eingeblendet.

### Wackelnde Buttons

Lassen Sie uns nun dem biederen Startbildschirm des Mückenfangs eine weitere sinnvolle Animation hinzufügen: Wenn der Benutzer auch nach zehn Sekunden den START-Button noch nicht gefunden hat, lassen wir ihn wackeln. Den Button, nicht den Benutzer.

Erzeugen Sie eine neue Animations-XML-Datei namens *wackeln.xml*, diesmal mit dem Wurzelelement `rotate`. Eine Rotation beschreiben Sie, indem Sie den Anfangswinkel und den Endwinkel (in Grad) angeben:

```
<?xml version="1.0" encoding="utf-8"?>
<rotate  xmlns:android="http://schemas.android.com/apk/res/android"
 android:fromDegrees="-20"
 android:toDegrees="20"
 android:duration="50"
 />
```

Laden Sie erneut die Animation in der `onCreate()`-Methode:

```
animationWackeln = AnimationUtils.loadAnimation(this, R.anim.wackeln);
```

Das zugehörige Attribut haben Sie bestimmt schon angelegt, ohne dass ich Ihnen erklären muss, wie's geht. Um die Animation zeitverzögert auszulösen, schreiben Sie eine neue private, innere Klasse, die das Interface `Runnable` implementiert:

```
private class WackleButton implements Runnable {
   @Override
   public void run() {
```

```
        startButton.startAnimation(animationWackeln);
    }
}
```

Legen Sie für den START-Button ein passendes Attribut an, das Sie in der `onCreate()`-Methode setzen:

```
startButton = (Button) findViewById(R.id.button1);
```

Um das Runnable zeitverzögert zu starten, ist ein Handler erforderlich. Das kennen Sie schon von der `GameActivity`, sodass Sie ohne große Erklärung den Handler deklarieren können:

```
private Handler handler = new Handler();
```

Fehlt nur noch der zeitverzögerte Aufruf des Runnables `WackleButton` am Ende von `onResume()`:

```
handler.postDelayed(new WackleButton(), 1000*10);
```

Wenn Sie das jetzt Spiel jetzt starten und zehn Sekunden warten, wird der START-Button einmal kurz hin- und herzucken. Das ist noch längst nicht der gewünschte Effekt! Erstens dreht Android Views standardmäßig um ihre linke obere Ecke, nicht um die Mitte. Das können Sie leicht ändern, indem Sie der Animationsdefinition zwei weitere Attribute hinzufügen:

```
android:pivotX="50%"
android:pivotY="50%"
```

Der Pivotpunkt ist in diesem Zusammenhang der Durchstoßpunkt der Achse, um die Android ein Objekt dreht. Er liegt jetzt genau in der Mitte des Buttons (Abbildung 6.7).

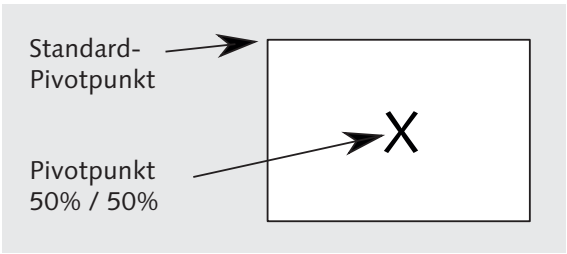



**Abbildung 6.7** Um diesen Punkt dreht sich die Welt. Mindestens aber der Button.

Jetzt müssen Sie die Animation nur noch mehrfach hintereinander wiederholen:

```
android:repeatCount="10"
```

Augenblick, das ist noch nicht ganz alles! In dieser Form würde die Rotation zwar zehnmal ablaufen, aber immer nur von –20° nach +20°, also ein Stückchen von schräg links nach schräg rechts. Es fehlt der Rückweg!

Zum Glück können Sie Android leicht mitteilen, dass jeder zweite Durchlauf rückwärts stattfinden soll. Schreiben Sie einfach:

```
android:repeatMode="reverse"
```

Wenn Sie die App jetzt ausprobieren, funktioniert alles wie geplant.

Es gibt allerdings einen kleinen Nebeneffekt. Drücken Sie mal den START-Button, bevor er wackelt. Fangen Sie dann ein paar Mücken. Am Ende der Jagd kommen Sie zum Startbildschirm zurück, und was passiert?

Der Button wackelt.

Warum tut er das? Weil der Handler immer noch existiert und bei nächster Gelegenheit das ihm anvertraute Runnable ausführt – in diesem Fall nach Rückkehr auf den Startbildschirm und völlig unerwünscht.

Sie müssen beim Verlassen des Startbildschirms das verzögerte Runnable aus der Ereigniswarteschlange des Handlers löschen. Dazu müssen Sie es aber zunächst identifizieren können, also deklarieren Sie es als privates Attribut der Activity:

```
private Runnable wackelnRunnable = new WackleButton();
```

In `onResume()` übergeben Sie jetzt dieses Objekt an den Handler:

```
handler.postDelayed(wackelnRunnable, 10000);
```

Und wenn die Activity vom Bildschirm verschwindet (also zum Beispiel beim Start des Spiels), entfernen Sie das `wackelnRunnable` aus der Ereigniswarteschlange. Dazu müssen Sie die Methode `onPause()` der Activity überschreiben:

```
@Override
protected void onPause() {
   super.onPause();
   handler.removeCallbacks(wackelnRunnable);
}
```

### Interpolation

In Abbildung 6.5 habe ich Ihnen eine »weiche« Animation untergejubelt, die in den bisherigen Beispielen allerdings noch nicht zum Einsatz gekommen ist. Wie auch im Straßenverkehr gibt es mehr als eine Möglichkeit, von A nach B zu gelangen. Die einfachste Variante ist der gerade Weg – für eine Animation bedeutet

dass, die Werte zwischen Anfangs- und Endzustand durch einfachen Dreisatz auszurechnen. Dies ist die **lineare Interpolation**, die als Voreinstellung bei den vorangegangenen Animationen zum Einsatz kam.

Aber es sieht einfach »cooler« aus, wenn eine Animation sanft beginnt, dann etwas beschleunigt, um am Schluss wieder abzubremsen. Das ähnelt dem Stop-and-go-Verkehr, wenn Sie mit einem Auto an einer Kreuzung losfahren und bei der nächsten Ampel wieder halten müssen. Die zugehörige Interpolation hört bei Android auf den Namen `accelerate_decelerate_interpolator`.

Es gibt eine ganze Reihe vordefinierter Interpolatoren (Abbildung 6.8), theoretisch können Sie sogar einen eigenen schreiben.

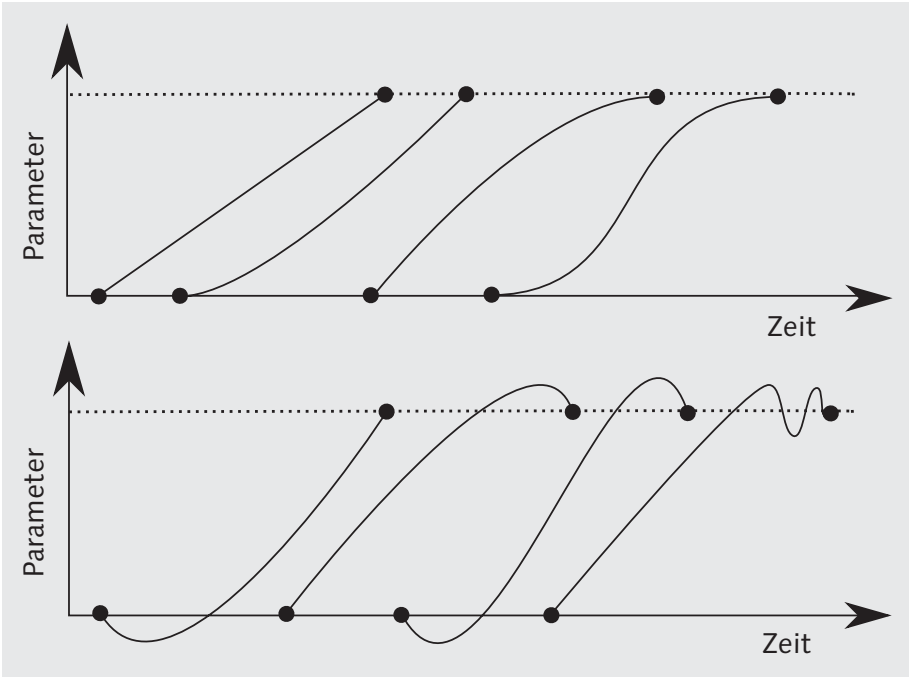



**Abbildung 6.8** Androids Interpolator-Aufgebot. Oben von links: linear_interpolator, accelerate_interpolator, decelerate_interpolator, accelerate_decelerate_interpolator. Unten von links: anticipate_interpolator, overshoot_interpolator, anticipate_overshoot_interpolator und last not least bounce_interpolator.

Lassen Sie uns einen hübschen Interpolator ausprobieren – und zwar an den Mücken. Was liegt näher, als beim Treffen einer Mücke diese mit einer kleinen Animation verschwinden zu lassen?

Schreiben Sie also zunächst das passende Animations-XML, und nennen Sie die Datei *treffer.xml*.

```
<?xml version="1.0" encoding="utf-8"?>
<scale  xmlns:android="http://schemas.android.com/apk/res/android"
android:fromXScale="1"
android:toXScale="0"
android:fromYScale="1"
android:toYScale="0"
android:pivotX="50%"
android:pivotY="50%"
android:duration="250"
android:interpolator="@android:anim/accelerate_decelerate_interpolator"
/>
```

Neben `alpha` und `rotate` ist der dritte animierbare Parameter `scale`, also die Größe. Sie sehen leicht am XML-Code, dass innerhalb von 250 Millisekunden sowohl Breite als auch Höhe vom Maßstab (**Scale**) 1 (entsprechend 100 %) auf 0 zusammengequetscht werden, und zwar unter Verwendung von `accelerate_decelerate_interpolator` (Sie können auch einfach die anderen ausprobieren).

Aber Vorsicht! Da jede Animation eine Viertelsekunde dauert, wird es vorkommen, dass mehrere gleichzeitig laufen – wenn ein Spieler vor Ende einer Animation schon die nächste Mücke erwischt. Daher können Sie diesmal nicht ein einziges Animationsobjekt erzeugen, sondern brauchen bei jedem Treffer ein frisches.

Ergänzen Sie daher in `GameActivity.onClick()` folgende Zeilen:

```
Animation animationTreffer = AnimationUtils.loadAnimation(this,
R.anim.treffer);
muecke.startAnimation(animationTreffer);
```

Wenn Sie sich die `onClick()`-Methode anschauen, fällt Ihnen ziemlich schnell auf, dass Sie nicht mehr, wie bisher üblich, an dieser Stelle die Mücke aus dem `spielbereich` entfernen können – dann würde man von der Animation rein gar nichts sehen. Entfernen Sie also die Zeile.

Aber irgendwie muss die Mücke ja verschwinden, und zwar genau dann, wenn die Animation vollständig abgespielt ist. Sie ahnen es sicher schon: Android teilt Ihnen auf Wunsch gerne mit, wann es so weit ist. Dazu müssen Sie lediglich einen `AnimationListener` anbieten. Sie brauchen also mal wieder eine private Klasse , und die sieht wie folgt aus:

```
private class MueckeAnimationListener implements AnimationListener {
   @Override
   public void onAnimationEnd(Animation animation) {
   }
```

```
    @Override
    public void onAnimationRepeat(Animation animation) {
    }
    @Override
    public void onAnimationStart(Animation animation) {
    }
}
```

Sie müssen zwar zwangsläufig alle drei Methoden der Schnittstelle `AnimationListener` implementieren, aber nur die erste benötigt Code. Es gibt allerdings einen Haken: Die Methode `onAnimationEnd()` erfährt leider nicht, bei **welcher** Mücke die Animation beendet ist. Auch die Animation selbst lüftet dieses Geheimnis nicht – Sie müssen also wohl oder übel der Klasse `MueckeAnimationListener` eine Referenz auf die Mücke mitgeben. Eine Instanz der inneren Klasse merkt sich dann die Referenz und kann die richtige Mücke entfernen. Fügen Sie also der inneren Klasse ein Attribut `muecke` hinzu, und schreiben Sie der Einfachheit halber einen Konstruktor, der dieses Attribut als Parameter mitbekommt:

```
private class MueckeAnimationListener implements AnimationListener {
    private View muecke;
    public MueckeAnimationListener(View m) {
        muecke = m;
    }
    ...
}
```

Leider war das noch nicht die ganze Wahrheit.

Wenn Sie jetzt einfach in `onAnimationEnd()` die `muecke` aus dem Spielbereich entfernen, knallt's früher oder später. Animation und Bildschirmaufbau sind nämlich nicht synchronisiert: Es kann passieren, dass Sie die Mücke entfernen, während Android gerade versucht, sie zu zeichnen. Das Resultat ist ein Crash der App. Deshalb müssen Sie den Umweg über den Handler gehen. Nur so ist gewährleistet, dass der View erst entfernt wird, wenn nichts mehr schiefgehen kann.

Um nicht noch eine innere Klasse mit Konstruktor zu bauen, zeige ich Ihnen jetzt, wie eine kurze und elegante, aber ein wenig unübersichtliche Variante aussieht:

```
@Override
public void onAnimationEnd(Animation animation) {
    handler.post(new Runnable() {
        @Override
        public void run() {
```

```
            spielbereich.removeView(muecke);
        }
    });
}
```

Wir übergeben der Methode `handler.post()` eine sogenannte anonyme innere Klasse (fett hervorgehoben). Sie heißt **anonym**, weil sie keinen Namen erhält, sondern gleich mit `new` ein (ebenfalls namenloses) Objekt erzeugt wird. Die anonyme Klasse implementiert das Interface `Runnable`, und folglich müssen Sie dessen Methode `run()` implementieren. Und in dieser darf endlich die Mücke unfallfrei vom Bildschirm verschwinden.

Der große Vorteil: Das fragliche Objekt `muecke` ist der anonymen Klasse bekannt, weil sie eine innere Klasse von `MueckeAnimationListener` ist, die über die richtige Referenz verfügt.

Rein theoretisch hätten wir übrigens auch den `AnimationListener` als anonyme innere Klasse in der Methode `onClick()` verwirklichen können. Aber mit dem Runnable darin wären das *zwei* ineinander verschachtelte anonyme Klassen, und so elegant und kompakt der Code dann auch wird – der Übersicht dient so etwas nicht. Ich persönlich programmiere zwar manchmal so etwas, aber nur, weil ich hoffe, dass niemand außer mir den Code je lesen muss – oder, falls doch, dass derjenige dann nicht bewaffnet ist.

Jetzt können Sie in `onClick()` einen Listener mit der angeklickten Mücke erzeugen und an die Animation hängen:

```
animationTreffer.setAnimationListener(new MueckeAnimationListener
(muecke));
```

Bleibt noch eine Kleinigkeit: Was passiert, wenn der Spieler eine »verschwindende« Mücke noch einmal antippt? Derzeit würde das als weiterer Treffer gewertet werden – doppelte Punktzahl! Das darf natürlich nicht passieren.

Glücklicherweise ist die Bedingung sehr simpel: Wir müssen `onClicks` ignorieren, wenn die fragliche Mücke gerade in eine Animation verwickelt ist. Also schreiben Sie einfach:

```
public void onClick(View muecke) {
    if(muecke.getAnimation()==null) {
        ...
    }
}
```

Falls Sie den Mücken gerne auch beim Auftauchen eine Animation verpassen möchten, schaffen Sie das jetzt sicher ohne meine Hilfe.

Denken Sie aber an drei Dinge:

- Es kann sein, dass mehrere Mücken gleichzeitig erscheinen – *ein* `Animation`-Attribut in der Klasse genügt also auch in diesem Fall nicht.
- Leider wird die Darstellung bei zu vielen gleichzeitig laufenden Animationen ruckeln. Übertreiben Sie's also nicht!
- Wenn Sie den Overshoot-Interpolator in Verbindung mit einer Vergrößerung von 0 auf 100 % verwenden, sieht das nicht wie eine Mücke aus, sondern wie eine Ente, die gerade von einem Tauchgang zurückkehrt.

## 6.4 Fliegende Mücken

Ist Ihnen das Spiel langweilig geworden? Es gibt nichts Schlimmeres, was einem Spiel passieren kann. Trotz Sound ist es einfach zu leicht, die Mücken zu treffen, es sei denn, es werden viele auf einmal.

Wenn Sie schon mal in der Realität auf Mückenjagd gegangen sind, werden Sie wissen, dass die Mistviecher selten an einem Ort sitzenbleiben. Was liegt also näher, als die Mücken auch in unserem Spiel in Bewegung zu versetzen?

### Grundgedanken zur Animation von Views

Bewegung ist, technisch betrachtet, nichts anderes als die ständige Änderung des Aufenthaltsortes. Sie müssen also lediglich in kurzen zeitlichen Abständen die Bildschirmkoordinaten einer Mücke verändern, um die Illusion einer Bewegung zu erzeugen.

Aber in welche Richtung und wie schnell soll sie sich bewegen?

### Geschwindigkeit festlegen

Offensichtlich müssen Sie einmal zu Beginn eine Geschwindigkeit festlegen, die aus einem horizontalen und einem vertikalen Bestandteil besteht. So etwas nennen Physiker gerne einen **zweidimensionalen Vektor**, aber für den Moment nennen wir die beiden Variablen einfach `vx` und `vy`.

Damit nicht alle Mücken gleich schnell in die gleiche Richtung driften, müssen Sie jeder Mücke ihre eigene Geschwindigkeit mitgeben. Sie haben bereits eine Methode kennengelernt, um so etwas zu bewerkstelligen: Jeder Mücke wurde ihr Geburtsdatum als **Tag** mitgegeben. Was liegt also näher, als die Geschwindigkeit auf die gleiche Weise einzubauen?

Fügen Sie dazu zunächst in die Datei *ids.xml* zwei neue Tag-IDs ein, die Sie `vx` und `vy` nennen.

Legen Sie die Werte für die Geschwindigkeit mit dem Zufallsgenerator fest:

```
int vx = zufallsgenerator.nextInt(3)-1;
int vy = zufallsgenerator.nextInt(3)-1;
```

Da die Methode `nextInt()` Zufallszahlen zwischen 0 (inklusive) und dem übergebenen Parameter (exklusive) ausspuckt, erhalten wir auf diese Weise Geschwindigkeiten für die x- bzw. y-Richtung zwischen –1 und +1. Auch die 0 ist mit von der Partie, sodass sich einige Mücken nicht bewegen werden – im Schnitt eine von neun, weil es 3 mal 3 = 9 Kombinationen gibt und nur eine den Stillstand bedeutet.

Diesen Fall sollten Sie ausschließen. Falls also sowohl `vx` als auch `vy` 0 sind, muss der Zufallsgenerator erneut ran – so lange, bis er etwas anderes als zwei Nullen generiert. In Java sieht das so aus:

```
int vx;
int vy;
do {
   vx = zufallsgenerator.nextInt(3)-1;
   vy = zufallsgenerator.nextInt(3)-1;
} while(vx==0 && vy==0);
```

Hierbei handelt es sich um eine Schleife, deren Wiederholungsbedingung am Ende in den Klammern hinter `while()` steht. Der Inhalt der geschweiften Klammern wird so oft wiederholt, wie die Bedingung erfüllt ist, mindestens aber einmal.

Da die Variablen `vx` und `vy` immer nur innerhalb des Codeblocks existieren, in dem sie deklariert sind, müssen wir sie vor die Schleife ziehen.

Die Geschwindigkeit muss noch mit dem Maßstab des Displays multipliziert werden:

```
vx = (int)Math.round(massstab*vx);
vy = (int)Math.round(massstab*vy);
```

Hängen Sie die Geschwindigkeit als Nächstes als Tags an die Mücke:

```
muecke.setTag(R.id.vx, new Integer(vx));
muecke.setTag(R.id.vy, new Integer(vy));
```

Anschließend kümmen wir uns um den eigentlichen Bewegungsvorgang.

## Mücken bewegen

Wir haben schon eine Methode, die in regelmäßigen zeitlichen Abständen aufgerufen wird: `zeitHerunterzaehlen()`. Es ist naheliegend, die Bewegung darin unterzubringen. Allerdings wäre es unschön, den gesamten Code dafür einfach in die Methode zu packen, weil sie dadurch zu lang und unübersichtlich wird. Schreiben Sie also eine neue Methode `mueckenBewegen()`, und rufen Sie sie in `zeitHerunterzaehlen()` auf, und zwar nach `mueckenVerschwinden()`, denn Mücken, die verschwunden sind, müssen wir nicht mehr bewegen. Umgekehrt ausgedrückt: Würden wir Mücken bewegen, die unmittelbar danach ohnehin verschwinden, hätten wir Rechenzeit verschwendet.

### Wie Fehler zu Code werden

Ein beliebter Trick zum Vermeiden von Tipparbeit ist anwendbar, wenn es um das Schreiben neuer Methoden geht.

Tippen Sie zuerst den Aufruf der neuen Methode ein. Eclipse wird Ihnen die Zeile rot unterstreichen, weil es die Methode natürlich nicht kennt.

Drücken Sie Strg + 1 (oder klicken Sie mit der Maus auf das kleine rote Kreuz am linken Rand des Editor-Fensters). Daraufhin verrät Ihnen Eclipse nicht nur den genauen Fehler, sondern schlägt auch sofort vor, wie er zu beheben ist: indem man die Methode hinzufügt. Eclipse zeigt sogar schon, wie die Methode ausschauen wird. Drücken Sie einfach ↵, um den Code automatisch zu erzeugen (Abbildung 6.9).


**Abbildung 6.9** Aus einem Fehler wird dank Strg + 1 die gewünschte Methode.

Es liegt auf der Hand, dass wir in der Bewegen-Methode alle Mücken auf dem Spielfeld einmal betrachten müssen. Das kommt Ihnen bekannt vor? Richtig, in der Methode `mueckenVerschwinden()` geschieht genau dasselbe. Also übernehmen wir einen Teil des dortigen Codes:

```
private void mueckenBewegen() {
   spielbereich = (FrameLayout) findViewById(R.id.spielbereich);
   int nummer=0;
```

```
    while(nummer<spielbereich.getChildCount()) {
        ImageView muecke = (ImageView) spielbereich.getChildAt(nummer);
        int vx = (Integer) muecke.getTag(R.id.vx);
        int vy = (Integer) muecke.getTag(R.id.vy);
        // und nun bewegen ...
        nummer++;
    }
}
```

In der `while`-Schleife wird jede Mücke im `spielbereich` einmal herangezogen. Aus den Tags lesen wir die Geschwindigkeit ab. Da Java an dieser Stelle nicht wissen kann, welche Objekte da an den Tags hängen, müssen Sie sie explizit in `Integer` umwandeln. Die Resultate können Sie mühelos Variablen vom nativen Typ `int` zuweisen.

Nun also zur eigentlichen Bewegung!

Werfen Sie noch einmal einen Blick auf die Stelle im Code, an der die Mücken positioniert werden:

```
FrameLayout.LayoutParams params = new FrameLayout.LayoutParams(
muecke_breite,muecke_hoehe);
params.leftMargin = links;
params.topMargin = oben;
params.gravity= Gravity.TOP + Gravity.LEFT;
spielbereich.addView(muecke,params);
```

Die Position der Mücke steckt also in Form von rechtem und linkem Rand in den `LayoutParams`. Also können Sie die Mücke bewegen, indem Sie sich die aktuellen `LayoutParams` holen, die Ränder ändern und der Mücke die neuen Parameter verpasssen.

```
FrameLayout.LayoutParams params = (android.widget.FrameLayout.
LayoutParams) muecke.getLayoutParams();
params.leftMargin += vx;
params.topMargin += vy;
muecke.setLayoutParams(params);
```

Eine kleine Komplikation gibt es in der ersten Zeile dieses Codeblocks: Sie müssen die `LayoutParams`, die Ihnen die `getLayoutParams()`-Methode zurückliefert, explizit umwandeln. Das ist vor allem deswegen höchst unübersichtlich, weil beide Klassen `LayoutParams` heißen, sich aber in unterschiedlichen Packages befinden und noch dazu lokale Klassen anderer Klassen sind:

```
android.view.View.LayoutParams
android.widget.FrameLayout.LayoutParams
```

Die Methode `View.getLayoutParams()` ist so deklariert, dass sie Ersteres zurückliefert. Sie aber wissen, dass es sich in Wirklichkeit um die zweite Version handelt (die von der ersten erbt), also können Sie sie umwandeln.

Probieren Sie das Spiel jetzt aus.

Sind Ihnen die Mücken zu langsam?

Nun, vielleicht sollten sie abhängig von der Runde immer schneller werden?

Versuchen Sie ruhig mal Folgendes:

```
params.leftMargin += vx*runde;
params.topMargin += vy*runde;
```

In der ersten Runde bewegen sich die Mücken also mit einer Geschwindigkeit von 1, dann doppelt so schnell, dreimal so schnell etc.

Viel Spaß bei der Jagd mit eingeschaltetem Turbo!

### Bilder programmatisch laden

Auch wenn vermutlich keiner von uns Experte für das avionische Verhalten von Stechmücken ist: Dass die Viecher seitwärts fliegen, ist doch eher unwahrscheinlich. Die richtige Richtung ist: Rüssel voraus.

Leider können Sie nicht ohne Weiteres Android dazu bringen, die ImageViews, die die Mücken zeigen, zu drehen. Wir brauchen daher einen alternativen Plan.

Wie wäre es, wenn für jede der acht Richtungen eine Bild-Ressource zur Verfügung stehen würde? Dann müsste nur beim Festlegen der Richtung das richtige Bild geladen werden.

Welches Bild zu einer Flugrichtung gehört, sollte am besten aus dem Dateinamen hervorgehen. Schnappen Sie sich also ein Grafikprogramm Ihrer Wahl, laden Sie die Mücke, und speichern Sie gedrehte Kopien. Dazu eignen sich Programme wie Photoshop oder GIMP. Wenn Sie die Mücke als Vektorgrafik angelegt haben, exportieren Sie einfach für jede der acht Positionen eine Bitmap. Benennen Sie die Bilder nach den Himmelsrichtungen, also *muecke_n.png* für die, die nach oben fliegt, *muecke_so.png* für die nach rechts unten etc. (Abbildung 6.10).

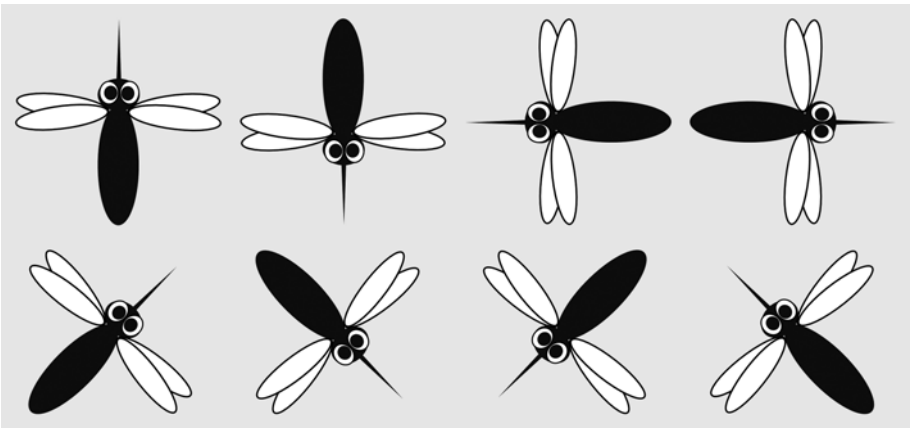


**Abbildung 6.10** Achten Sie beim Drehen der Mücken darauf, dass Höhe und Breite jeweils gleich sind.

Wenn Sie sich die Arbeit sparen möchten, finden Sie die gedrehten Mücken natürlich auch auf der beiliegenden DVD.

Diese Vorgehensweise bedeutet einen Kompromiss: Zwar ersparen Sie dem Smartphone die rechenintensive Drehung von Grafiken, dafür wird die App etwas mehr vom begrenzt verfügbaren Speicher verbrauchen. Es kommt in der App-Entwicklung oft darauf an, die verschiedenen Nachteile abzuwägen und den effizientesten Weg zu wählen. Die acht Mücken verbrauchen weniger als 20 KByte – damit kann man gut leben. Bei 20 Megabyte sähe die Sache sicher anders aus.

Sorgen Sie nun dafür, dass die jeweils richtige Mücke geladen wird. Es gibt mehrere Möglichkeiten, das zu tun – egal, welche man verwendet, es ist sicher eine gute Idee, das in einer eigenen Methode zu tun.

Schreiben Sie also eine Methode `setzeBild()`, die als Parameter erstens den ImageView für die Mücke, zweitens die gewünschte Richtung erhält:

```
private void setzeBild(ImageView muecke, int vx, int vy) {
}
```

Rufen Sie die Methode in `eineMueckeAnzeigen()` auf, nachdem die Richtung bekannt ist, aber unmittelbar vor der Skalierung von `vx` und `vy`:

```
setzeBild(muecke, vx, vy);
vx = (int)Math.round(massstab*vx);
vy = (int)Math.round(massstab*vy);
```

Anschließend sind die Werte von `vx` und `vy` nicht mehr im Bereich –1 bis +1 und für unsere Methode nicht mehr auszuwerten.

Etwas weiter oben wurde bisher der Mücke das Standardbild *muecke.png* zugewiesen:

```
muecke.setImageResource(R.drawable.muecke);
```

Diese Zeile können Sie jetzt löschen.

Sicher ist Ihnen nicht entgangen, dass die eigentliche Methode `setzeBild()` bislang leer ist. Das Beste verwahrt man sich ja bis zum Schluss ...

Ich zeige Ihnen jetzt drei verschiedene Möglichkeiten, diese Methode zu schreiben. Anhand der eingesetzten Techniken lernen Sie wieder eine Menge über Java, und am Ende können Sie sich aussuchen, welche Variante Ihnen am besten gefällt. Denn oft haben Sie tatsächlich diese Entscheidungsfreiheit: Wenn Ausführungsgeschwindigkeit und Speicherbedarf keine Rolle spielen, ist die konkrete Implementierung Geschmackssache.

### If-else-Abfragen

Es gibt eine überschaubare Anzahl verschiedener Richtungen: acht. Sie können daher einfach acht `if`-Abfragen implementieren:

```
private void setzeBild(ImageView muecke, int vx, int vy) {
   if(vx==-1 && vy==-1)
      muecke.setImageResource(R.drawable.muecke_nw);
   if(vx==-1 && vy== 0)
      muecke.setImageResource(R.drawable.muecke_w );
   if(vx==-1 && vy==+1)
      muecke.setImageResource(R.drawable.muecke_sw);
   if(vx== 0 && vy==-1)
      muecke.setImageResource(R.drawable.muecke_n);
   if(vx== 0 && vy==+1)
      muecke.setImageResource(R.drawable.muecke_s);
   if(vx==+1 && vy==-1)
      muecke.setImageResource(R.drawable.muecke_no);
   if(vx==+1 && vy== 0)
      muecke.setImageResource(R.drawable.muecke_o);
   if(vx==+1 && vy==+1)
      muecke.setImageResource(R.drawable.muecke_so);
}
```

Diese Variante erfordert nicht viel Nachdenken – Sie müssen lediglich aufpassen, dass bei der jeweiligen Kombination aus `vx` und `vy` auch die richtige Grafik verwendet wird.

Wie Sie sehen, brauchen Sie hier für eine relativ einfache Aufgabe 16 Zeilen, und dabei habe ich schon diverse geschweifte Klammern weggelassen.

Hinzu kommt, dass Java hier beim Programmablauf jede `if`-Bedingung einmal prüft, obwohl wir wissen, dass nur eine einzige zutreffen kann. Um das zu umgehen, könnten Sie jede Abfrage um einen vorzeitigen Abbruch der Methode ergänzen:

```
if(vx==-1 && vy==-1) {
   muecke.setImageResource(R.drawable.muecke_nw);
   return;
}
```

Kürzer wird der Code dadurch aber auch nicht ...

Etwas eleganter sind eingestreute `else`-Verzweigungen:

```
if(vx==-1 && vy==-1)
   muecke.setImageResource(R.drawable.muecke_nw);
else if(vx==-1 && vy== 0)
   muecke.setImageResource(R.drawable.muecke_w );
else if(vx==-1 && vy==+1)
   muecke.setImageResource(R.drawable.muecke_sw);
else ...
```

Diese Variante erfordert ebenfalls nur den Durchlauf der `if`-Anweisungen bis zum »Treffer«. Das sind im schlimmsten Fall immer noch sieben Abfragen.

Glücklicherweise gibt es elegantere Lösungen.

### Zweidimensionale Arrays

Sie haben bereits einfache Arrays primitiver Datentypen kennengelernt, und wenn ich Ihnen jetzt verrate, dass es auch zweidimensionale Arrays gibt, geht Ihnen möglicherweise ein Licht auf.

In einem zweidimensionalen Array lässt sich mühelos eine Tabelle wie die folgende ablegen:

| | | |
|---|---|---|
| `R.drawable.muecke_nw` | `R.drawable.muecke_n` | `R.drawable.muecke_no` |
| `R.drawable.muecke_w` | | `R.drawable.muecke_o` |
| `R.drawable.muecke_sw` | `R.drawable.muecke_s` | `R.drawable.muecke_so` |

In den neun Feldern dieser Tabelle stehen für die betreffende Richtung die IDs, die der Android Resource Manager für das jeweilige Bild angelegt hat.

Eine solche Tabelle kann in Form eines zweidimensionalen Arrays in Java leicht als konstantes, wenngleich etwas unübersichtliches Attribut in einer Klasse angelegt werden:

```
private static final int MUECKEN_BILDER[][] = {
   {R.drawable.muecke_nw, R.drawable.muecke_n, R.drawable.muecke_no},
   {R.drawable.muecke_w,  R.drawable.muecke,   R.drawable.muecke_o},
   {R.drawable.muecke_sw, R.drawable.muecke_s, R.drawable.muecke_so} };
```

Als primitiven Datentyp verwendet dieses Array `int`, denn nichts anderes sind die Resource-IDs. Sie sehen, dass Sie für die initiale Zuweisung der horizontalen und vertikalen Dimension lediglich die geschweiften Klammern ineinander verschachteln müssen. Jede innere geschweifte Klammer enthält eine Zeile der obigen Mückentabelle.

Jetzt müssen Sie nur noch passend zum Koordinatenpaar `vx` und `vy` das richtige Array-Element herausfinden. Sind `vx` und `vy` beide –1, zum Beispiel das Element links oben (es hat die Array-Indizes 0,0) etc. Da die Geschwindigkeitswerte von –1 bis +1 gehen, der Array-Index aber jeweils von 0 bis 2, genügt eine einfache Addition von +1:

```
private void setzeBild(ImageView muecke, int vx, int vy) {
   muecke.setImageResource(MUECKEN_BILDER[vy+1][vx+1]);
}
```

Sie müssen hier den äußeren Index zuerst angeben, um die richtige Zeile der Tabelle zu erwischen, und im zweiten Klammerpaar die Spalte.

Voilà – Sie haben dasselbe Ziel wie im vorangegangenen Abschnitt erreicht, und das mit einer einzigen Zeile (zugegeben: zuzüglich der Deklaration des Arrays `MUECKEN_BILDER`)! Man kann sehr oft durch geschicktes Ablegen von Daten in Arrays umfangreiche `if`-Konstruktionen vermeiden und damit Rechenzeit sparen – selbst wenn es sich wie im vorliegenden Fall vermutlich nur um Nanosekunden handelt.

Aber ist das Konstanten-Array eigentlich notwendig? Wieso haben wir eigentlich den acht Mückenbildern sprechende Namen gegeben? Kann man diesen Umstand nicht ausnutzen?

Man kann.

### Resource-IDs ermitteln

Bisher haben Sie die Ressource des Bildes anhand des vom Resource Manager erzeugten Bezeichners (z. B. `R.drawable.muecke`) identifiziert. Es gibt aber auch die Möglichkeit, die ID einer Ressource anhand des Namens (z. B. `"muecke_nw"`) zu ermitteln (vielleicht erinnern Sie sich an den Abschnitt »Retrospektive« in Kapitel 5):

```
int id = getResources().getIdentifier("muecke_nw",
   "drawable", this.getPackageName());
```

Die Methode `getResources()` ist in der Basisklasse `Activity` definiert. Wir nutzen also eine in allen `Activity`-Objekten zur Verfügung stehende Funktion, um auf das `Resources`-Objekt der App zuzugreifen. Das wiederum stellt mit `getIdentifier()` eine Möglichkeit zur Verfügung, eine ID zu ermitteln. Sie müssen als Parameter lediglich den Namen, den Typ und den Paketnamen der App übergeben.

Jetzt müssen Sie nur noch aus den Geschwindkeitswerten `vx` und `vy` den passenden String ermitteln. Lassen Sie uns dafür erneut ein zweidimensionales Array verwenden:

```
private static final String HIMMELSRICHTUNGEN[][] = {
   {"nw", "n", "no"},
   {"w",  "",  "o" },
   {"sw", "s", "so"} };
```

Der Vorteil dieses Arrays ist, dass es nicht auf Mücken festgelegt, ist. Verallgemeinerung ist immer sinnvoll, weil Sie so später den Code wiederverwenden können.

Jetzt müssen Sie nur noch den Parameter für den Aufruf von `getIdentifier()` aus dem immer gleichen Teil `"muecke_"` und der richtigen Himmelsrichtung zusammensetzen. Letztere erhalten Sie, wenn Sie die Koordinaten als Indizes für das Array `HIMMELSRICHTUNGEN[]` verwenden:

```
"muecke_"+HIMMELSRICHTUNGEN[vy+1][vx+1]
```

Bleibt das eigentliche Setzen der ID des richtigen Bildes mit der Methode `setImageResource()`, und die Methode ist fertig:

```
private void setzeBild(ImageView muecke, int vx, int vy) {
   muecke.setImageResource(
      getResources().getIdentifier(
         "muecke_"+HIMMELSRICHTUNGEN[vy+1][vx+1],
         "drawable", this.getPackageName()
      )
   );
}
```

Diese Implementierung ist möglicherweise die komplizierteste – und auf den ersten Blick für jemanden, der den Code nicht kennt, recht unverständlich. Sie ist dafür flexibel, weil sie sich prinzipiell auch auf Bilder anwenden lässt, die nicht `"muecke_"` heißen. Dafür ist diese Variante etwas langsamer als die vorangegangene, weil jedes Mal `getResources().getIdentifier()` aufgerufen wird und nicht nur das obligatorische `muecke.setImageResource()`.

Sie sehen an diesem Beispiel, dass es oft unterschiedliche Lösungen für Programmieraufgaben gibt, und jede hat ihre Vor- und Nachteile. Es ist Ihre Aufgabe als Entwickler, die jeweils beste zu wählen – wenn das nicht eindeutig ist, aber nach Möglichkeit nicht die schlechteste.

## Retrospektive

In diesem Kapitel haben Sie Mücken das Summen und das Fliegen beigebracht. Gratulation, damit haben Sie für Spiele-Apps mit nicht zu anspruchsvollen Animationsbedürfnissen eine Menge an Grundlagen gelernt. Nebenbei haben Sie zum ersten Mal mit zweidimensionalen Arrays hantiert.

Natürlich ist unser Mückenfang immer noch weit davon entfernt, ein atemberaubendes Spiel zu sein. Wie Sie sicher schon festgestellt haben, steckt der Teufel im Detail:

- Die diagonal fliegenden Mücken sind schneller ...
- ... und außerdem größer.

Während Sie das zweite Problem leicht in den Griff bekommen, indem Sie die Grafiken noch mal bearbeiten und die nicht diagonalen Mücken etwas verkleinern, ist die Sache mit der Geschwindigkeit schon kniffliger.

Was ist eigentlich die Ursache dafür?

Nun, die »horizontalen« Mücken legen pro Zeitscheibe einen Weg der Länge 1 zurück (skaliert mit der Bildschirmauflösung). Die »diagonalen« Mücken legen jedoch sowohl in horizontaler als auch in vertikaler Richtung die Länge 1 zurück.

Falls Sie sich an die Dreiecksformel von Pythagoras erinnern, können Sie leicht ausrechnen, dass die resultierende Entfernung nicht 1 ist, sondern Wurzel 2, also etwa 1,414. Damit sind die diagonalen Mücken glatte 41 % schneller als ihre Artgenossen (Abbildung 6.11).

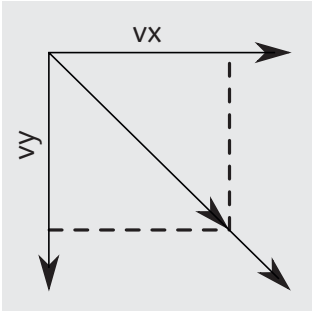



**Abbildung 6.11** Die diagonalen Mücken sind zu schnell (langer diagonaler Pfeil). Sie dürfen in jeder Zeitscheibe eigentlich nur so weit kommen, wie die gestrichelten Linien zeigen.

Um diese Angelegenheit in den Griff zu bekommen, bleibt Ihnen nichts anderes übrig, als mit Kommazahlen zu arbeiten. Setzt man im Satz von Pythagoras die längste Seite gleich 1, verrät der nächstbeste Taschenrechner schnell, dass die horizontale und vertikale Seite je 0,707 lang sein müssen (Abbildung 6.12). Die diagonalen Mücken dürfen folglich nur jeweils um 0,707 anstelle einer ganzen Einheit in jede Richtung verschoben werden.

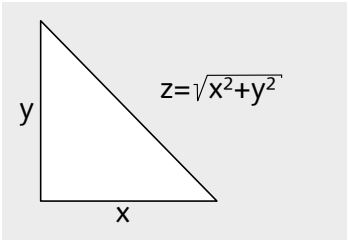



**Abbildung 6.12** Hier verrät Pythagoras, wie man aus zwei Seiten eines rechtwinkligen Dreiecks die Länge der dritten ausrechnen kann.

Im Fall der diagonalen Bewegung müssen wir also sowohl `vx` als auch `vy` mit 0,707 multiplizieren – und dann ordentlich runden, denn auf einem niedrig aufgelösten Display könnte die Geschwindigkeit 0 werden, wenn man einfach nur die Kommastellen abschneidet.

Als Bedingung für die diagonale Richtung schauen wir einfach nach, ob beide Geschwindigkeitskomponenten ungleich 0 sind, und setzen dann den nötigen Korrekturfaktor entsprechend:

```
double faktor = 1.0;
if(vx!=0 && vy!=0) {
   faktor = 0.70710678;
}
```

Die Rundung hatten wir ja schon in der Methode `eineMueckeAnzeigen()` stehen, dort müssen Sie, abgesehen von obigen Zeilen, also nur noch den `faktor` ergänzen:

```
vx = (int)Math.round(massstab*vx*faktor);
vy = (int)Math.round(massstab*vy*faktor);
```

Sie sehen mal wieder, dass man beim Programmieren von Spielen irgendwann nicht mehr um etwas Mathematik herumkommt. Vor allem Geometriekenntnisse können nicht schaden, denn meistens geht es um Bewegung, Entfernungen und Geschwindigkeiten. Sobald Kräfte wie Gravitation, Federn oder Raketenantriebe ins Spiel kommen, brauchen Sie nicht nur Mathematik, sondern auch noch Physik. Aber glücklicherweise kommen die Mücken auch ohne solche Kräfte klar.

Sie wiegen ja so gut wie nichts.

*»Bitte Sie in diesen Formular eingeben PIN und Nummer von Ihre Konto, bedankt vielmals!«*
*(anonym)*

# 7 Internet-Zugriff

Alleine spielen macht nur halb so viel Spaß. Sieht man sich den Erfolg von Spielen wie »World of Warcraft« an, ist es offensichtlich, dass das Internet ganz neue Spielerfahrungen ermöglicht. Bevor Sie jetzt versuchen, Mücken kreuz und quer durch das World Wide Web zu jagen, sollten wir uns eine überschaubare Anforderung ausdenken.

Wie wäre es, wenn sich unsere Mückenjäger miteinander messen könnten, indem sie ihre Rekorde miteinander vergleichen?

## 7.1 Highscores speichern

Bislang vergisst das Mückenspiel nach »Game Over« jede vorherige Runde. Grandiose Leistungen sollten jedoch nicht in Vergessenheit geraten – folglich brauchen Sie eine Highscore-Anzeige. Aber selbst nach Beenden des Spiels soll sich das Spiel den Rekord merken, daher wird es nicht genügen, ihn in einem Attribut abzulegen. Natürlich stellt Android dafür einfach zu verwendende Hilfsmittel zur Verfügung.

### Highscore anzeigen

Als Erstes benötigen Sie einen Ort, um den aktuellen Highscore anzuzeigen. Dazu eignet sich natürlich der Startbildschirm am besten. Fügen Sie dem also einfach zwei TextViews hinzu; dem einen verpassen Sie einen neuen String mit dem Inhalt »Highscore« (oder »Rekord«, wenn Ihnen das lieber ist), dem anderen die ID `highscore` (Abbildung 7.1).

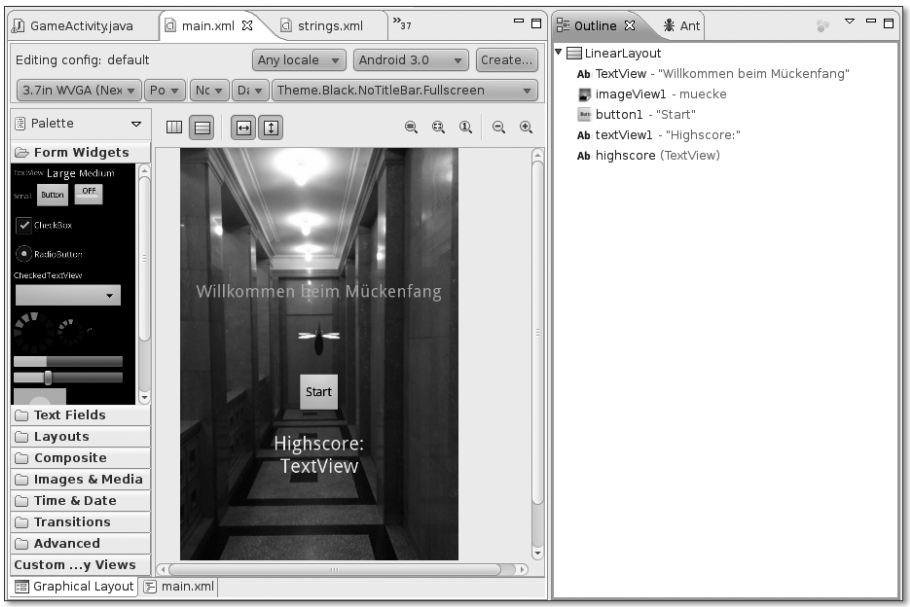



**Abbildung 7.1** Fügen Sie zwei TextViews für die Anzeige des Highscores ein.

Um den Highscore anzuzeigen, wenn die Activity angezeigt wird, schreiben Sie ihn einfach in den richtigen `TextView`:

```
@Override
protected void onResume() {
   super.onResume();
   TextView tv = (TextView) findViewById(R.id.highscore);
   tv.setText(Integer.toString(leseHighscore()));
}
```

`onResume()` wird von Android aufgerufen, wenn die Activity auf dem Bildschirm erscheint – also vor und auch nach dem Spiel.

Die Methode `leseHighscore()` legen Sie erst einmal als simplen Platzhalter an, damit Eclipse aufhört, über ihr Fehlen zu schimpfen. Drücken Sie einfach Strg + 1, um die Implementierung der Methode an Eclipse zu delegieren:

```
private int leseHighscore() {
   return 0;
}
```

Wenn die `MueckenfangActivity` den Highscore anzeigen soll, dann darf sie sich auch für dessen Verwaltung zuständig fühlen. Wie aber gelangt die erspielte Punktzahl am Ende des Spiels von der `GameActivity` in die `MueckenfangActivity`?

### Activities mit Rückgabewert

Dazu muss Erstere offensichtlich einen Wert zurückgeben, was sie bisher nicht tut. Man kann sie jedoch leicht dazu verdonnern, indem man anstelle von `startActivity()` eine andere Methode verwendet:

```
startActivityForResult(new Intent(this,GameActivity.class),1);
```

Der `int`-Parameter mit dem beliebigen Wert `1` wird durchgereicht, wenn die `GameActivity` ihr Ergebnis zurückgibt. Sie tut das, indem sie dafür sorgt, dass die Methode `onActivityResult()` der `MueckenfangActivity` aufgerufen wird. In dieser Methode können Sie also die erspielte Punktzahl verarbeiten. Falls sie höher ist als der bisherige Highscore, speichern Sie die Punktzahl als neuen Highscore:

```
@Override
protected void onActivityResult(int requestCode, int resultCode,
Intent data) {
   if(requestCode==1) {
      if(resultCode > leseHighscore()) {
         schreibeHighscore(resultCode);
      }
   }
}
```

Die Methode `schreibeHighscore()` müssen wir natürlich noch implementieren. Lassen Sie Eclipse zunächst eine leere Methode erzeugen (Strg + 1):

```
private void schreibeHighscore(int highscore) {
}
```

Nun muss die `GameActivity` die Punktzahl als Ergebnis zurückgeben. Dazu dient die Methode `setResult()`. Rufen Sie sie in der Methode `gameOver()` auf, und übergeben Sie die erreichte Punktzahl:

```
private void gameOver() {
   setResult(punkte);
   ...
}
```

Auf diese Weise ist dafür gesorgt, dass `onActivityResult()` mit der Punktzahl aufgerufen wird. Bleibt noch eine Aufgabe: die Punktzahl permanent zu speichern und wieder zu laden.

### Werte permanent speichern

Jede Android-App darf über einen Speicherbereich verfügen, der **Shared Preferences** heißt. Darin können Sie Zahlen, Strings oder andere primitive Datentypen

unter beliebigen Schlüsselbegriffen abspeichern. Für Bilder, Sounds oder andere umfangreichere Daten ist hier allerdings kein Platz.

Die Shared Preferences bleiben erhalten, wenn Ihre App beendet wird; sogar ein Update auf eine neue Version überleben sie. Lediglich bei einer Deinstallation werden normalerweise auch die Shared Preferences gelöscht.

Sie können von verschiedenen Activities Ihrer App auf dieselben Daten zugreifen (daher »shared«). Holen Sie sich eine Referenz auf den Datenhort, indem Sie die Methode `getSharedPreferences()` aufrufen:

```
SharedPreferences pref = getSharedPreferences("GAME", 0);
```

Indem Sie unterschiedliche `String`-Parameter übergeben, können Sie auf unabhängige Datenbereiche zugreifen, aber wir interessieren uns zunächst nur für einen und nennen ihn `"GAME"`.

Um einen Wert in den Datenbereich zu schreiben, benötigen Sie ein `Editor`-Objekt. Das holen Sie sich wie folgt:

```
SharedPreferences.Editor editor = pref.edit();
```

Der Editor schreibt schließlich den gewünschten Wert unter Angabe eines Schlüsselnamens in den zuvor gewählten Datenspeicher namens `"GAME"`:

```
editor.putInt("HIGHSCORE", highscore);
```

Als letzten Schritt müssen Sie dem Editor sagen, dass Sie nichts weiter für ihn zu tun haben. Dann erst schreibt er die Daten endgültig:

```
editor.commit();
```

Damit können Sie jetzt die Methode zum Speichern des Highscores aus den vier Zeilen zusammenschrauben:

```
private void schreibeHighscore(int highscore) {
   SharedPreferences pref = getSharedPreferences("GAME", 0);
   SharedPreferences.Editor editor = pref.edit();
   editor.putInt("HIGHSCORE", highscore);
   editor.commit();
}
```

Das Lesen eines Wertes aus den Shared Preferences ist deutlich einfacher, weil der Umweg über das `Editor`-Objekt entfällt:

```
private int leseHighscore() {
   SharedPreferences pref = getSharedPreferences("GAME", 0);
   return pref.getInt("HIGHSCORE", 0);
}
```

Ahnen Sie, wozu der zweite Parameter in der Methode `getInt()` gut ist?

Das ist der Vorgabewert, der zurückgegeben wird, falls unter dem Schlüssel `"HIGHSCORE"` noch nichts in den Shared Preferences abgelegt ist. Um diesen Sonderfall müssen Sie sich also überhaupt nicht weiter kümmern. Sehr praktisch, oder?

## Rekordhalter verewigen

Natürlich können Sie auf ähnliche Weise dem erfolgreichen Spieler erlauben, seinen Namen gemeinsam mit seiner Punktzahl zu verewigen. Es ist ein Leichtes, den Namen als String in den Shared Preferences zu speichern.

Zunächst muss ein passendes Texteingabefeld her. Das kennen Sie schon von der SagHallo-App, packen Sie also einfach eines auf den Startbildschirm (*main.xml*), und verzieren Sie es mit einem freundlichen Text und einem SPEICHERN-Button. Da diese Elemente nur sichtbar sein sollen, wenn dem Spieler erlaubt wird, seinen Namen einzugeben, ist es eine gute Idee, sie alle drei in ein neues LinearLayout zu verschachteln. Indem wir später dieses Layout-Element sichtbar und unsichtbar schalten, müssen wir das nicht für jedes einzelne Element darin tun (Abbildung 7.1).

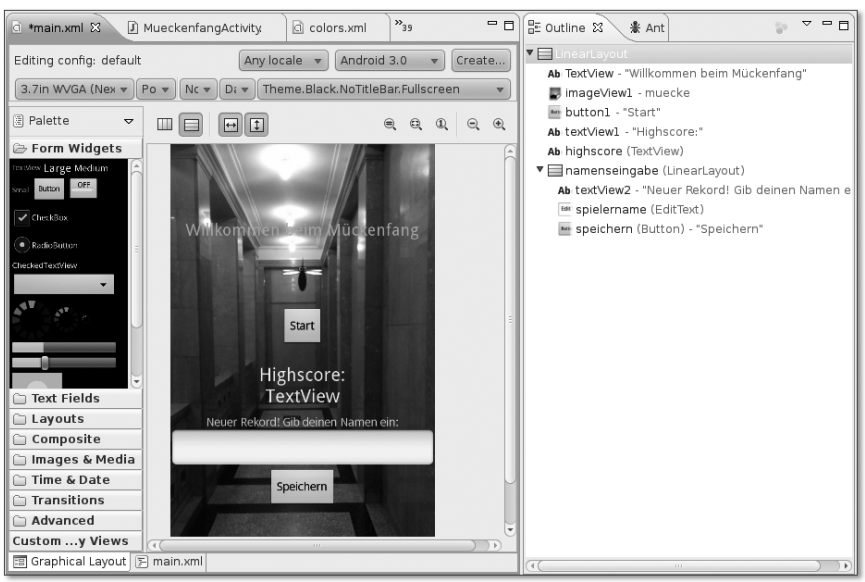



**Abbildung 7.2** Positioneren Sie alle Elemente für die Namenseingabe in einem LinearLayout-Objekt.

Verpassen Sie dem LinearLayout die ID `namenseingabe`, dem `EditText`-Element die ID `spielername` und dem SPEICHERN-Button die ID `speichern`. Für den TextView, der den Spieler bittet, seinen Namen einzugeben, benötigen Sie keine spe-

zielle ID, weil der Text statisch bleibt. Schreiben Sie ein paar freundliche Worte als neue String-Ressource in diesen TextView, und versehen Sie den neuen Button auf ähnliche Weise mit einer geeigneten Aufschrift.

**Die virtuelle Tastatur**

Beim Benutzen unterschiedlicher Android-Apps wird Ihnen schon aufgefallen sein, dass es verschiedene Varianten der Bildschirmtastatur gibt: nur Ziffern zum Beispiel oder eine eigene Taste für das @. Sie können die Tastatur bestimmen, indem Sie einem `EditText`-Element das Property `inputType` mitgeben. Mögliche Werte sind `phone`, `number` oder `textEmailAddress`.

Es gibt auch `textPersonName`, das für den vorliegenden Fall naheliegend ist. Aber Vorsicht: Sie verhindern dadurch nicht, dass der Benutzer die Tastatur umschaltet und Sonderzeichen eingibt. Wenn Sie das auf jeden Fall verhindern möchten, beispielsweise weil Sie die Eingabe an einen Programmteil weiterreichen, der mit Sonderzeichen nicht klarkommt, müssen Sie dem `EditText` einen `InputFilter` mitgeben. Das könnte beispielsweise so aussehen:

```
InputFilter filter = new InputFilter() {
   public CharSequence filter(CharSequence source, int start, int end,
Spanned dest, int dstart, int dend) {
      for(int i = start; i < end; i++) {
         if (!Character.isLetter(source.charAt(i)) &&
source.charAt(i)!=' ') {
            return "";
         }
      }
      return null;
   }
};
nickname.setFilters(new InputFilter[] { filter });
```

Ein solcher Filter prüft bei jeder einzelnen Eingabe, ob ein Zeichen zulässig ist (im Beispiel auf Buchstaben und Leerzeichen). Falls ja, gibt die `filter()`-Methode `null` zurück, sonst einen leeren String.

Da das `namenseingabe`-LinearLayout an verschiedenen Stellen sichtbar und unsichtbar geschaltet werden muss, gönnen Sie ihm ein privates Attribut als Referenz, das Sie in der Methode `onCreate()` initialisieren. Verfahren Sie genauso mit dem neuen Button, und verdrahten Sie seinen `OnClickListener`:

```
private LinearLayout namenseingabe;
private Button speichern;
public void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.main);
   namenseingabe = (LinearLayout) findViewById(R.id.namenseingabe);
   speichern = (Button) findViewById(R.id.speichern);
```

```
    speichern.setOnClickListener(this);
    ...
}
```

Schalten Sie außerdem die Namenseingabe gleich hier zunächst einmal unsichtbar, denn das ist ja der Ausgangszustand:

```
namenseingabe.setVisibility(View.INVISIBLE);
```

**Sichtbar oder gar nicht da**

Es gibt für Android-Layout-Elemente zwei unterschiedliche Arten der Unsichtbarkeit:

- `View.INVISIBLE`: Das Element ist unsichtbar, verbraucht aber genauso viel Platz, als wäre es sichtbar.
- `View.GONE`: Das Element ist nicht nur unsichtbar, sondern verbraucht auch keinen Platz – als würde es gar nicht existieren.

Wenn Sie `View.GONE` verwenden, müssen Sie bedenken, dass sich der Rest des Layouts verschieben kann. Probieren Sie daher im Layout-Editor genau aus, wie sich diese Art der Unsichtbarkeit auswirkt.

Sichtbar werden soll die Namenseingabe, wenn ein neuer Rekord erreicht wurde, also in `onActivityResult()`:

```
if(resultCode > leseHighscore()) {
    schreibeHighscore(resultCode);
    namenseingabe.setVisibility(View.VISIBLE);
}
```

Dass wir den Namen und den Rekord nicht gleichzeitig speichern, ist eine Ungenauigkeit, die wir in diesem Beispiel außer Acht lassen können.

Wenn der Benutzer seinen Namen eingegeben hat, wird er den SPEICHERN-Button drücken, was zum Aufruf Ihrer Methode `onClick()` führt. Die startet im Moment lediglich das Spiel, weil sie sich nicht dafür interessiert, welcher Button gedrückt wurde – Kunststück, bisher gab es ja auch nur einen.

Welcher Button gedrückt wurde, erfahren Sie anhand seiner ID:

```
@Override
public void onClick(View v) {
        if(v.getId() == R.id.button1) {
                startActivityForResult(new Intent(this,
                                        GameActivity.class),1);
        } else if(v.getId() == R.id.speichern) {
                schreibeHighscoreName();
        }
}
```

Ihnen fällt an dieser Stelle sicher auf, dass `button1` jetzt kein besonders aussagekräftiger Name mehr ist; wenn Sie wollen, können Sie ihn ändern.

Die Methode zum Speichern des eingegebenen Namens setzen Sie aus den bereits bekannten Aufrufen von `findViewById()` und `SharedPreferences` zusammen:

```
private void schreibeHighscoreName() {
   TextView tv = (TextView) findViewById(R.id.spielername);
   String name = tv.getText().toString().trim();
   SharedPreferences pref = getSharedPreferences("GAME", 0);
   SharedPreferences.Editor editor = pref.edit();
   editor.putString("HIGHSCORE_NAME", name);
   editor.commit();
}
```

Wie Sie sehen, müssen Sie sich zunächst eine Referenz auf das Eingabefeld beschaffen, dann den Inhalt mit `getText()` auslesen (und mit `trim()` vorn und hinten Leerzeichen entfernen) und das Resultat schließlich auf die vom Highscore bekannte Weise in die Shared Preferences schreiben. Als Schlüssel verwenden wir `"HIGHSCORE_NAME"`.

Um den Rekordhalter standesgemäß anzuzeigen, benötigen Sie eine Methode, die den Namen ausliest:

```
private String lies HighscoreName() {
   SharedPreferences pref = getSharedPreferences("GAME", 0);
   return pref.getString("HIGHSCORE_NAME", "");
}
```

Sorgen Sie nun noch dafür, dass der Besitzer des Highscores in `onResume()` angezeigt wird, aber nur, falls es überhaupt einen Highscore gibt. Schreiben Sie aber lieber eine eigene Methode dafür, die Sie in `onResume()` aufrufen – und auch in `onClick()` nach dem Speichern des Namens, damit gleich der richtige Rekordhalter erscheint:

```
private void HighscoreAnzeigen() {
   int highscore = leseHighscore();
   if(highscore>0) {
      tv.setText(Integer.toString(highscore) + " von " +
      leseHighscoreName());
   } else {
      tv.setText("-");
   }
}
@Override
```

```
protected void onResume() {
   super.onResume();
   highscoreAnzeigen();
}
public void onClick(View v) {
   ...
   if(v.getId() == R.id.speichern) {
      schreibeHighscoreName();
      highscoreAnzeigen();
      namenseingabe.setVisibility(View.INVISIBLE);
   }
}
```

Wie Sie sehen, lassen wir außerdem nach dem Speichern des neuen Rekordhalters die gesamte Namenseingabe wieder verschwinden.

Kommen Sie bitte nicht auf die Idee, den Code von `highscoreAnzeigen()` in `onResume()` stehenzulassen und von `onClick()` aus einfach `onResume()` aufzurufen. Mit etwas Glück funktioniert es, aber wahrscheinlicher ist ein Crash, der Sie für die unauthorisierte Einmischung in die Mechanismen des Activity-Lebenszyklus bestraft.

Wenn kein Highscore existiert, ist `highscore` übrigens gleich 0, und wir geben einfach einen Strich aus. Stattdessen können Sie natürlich auch einen Text wie »noch kein Highscore vorhanden« anzeigen lassen.

**Dynamische Texte übersetzbar machen**

Sie wissen bereits, dass Sie Ihre App sehr leicht übersetzen können: einfach für die gewünschte Sprache eine eigene *strings.xml*-Datei anlegen und in einem Ordner namens *values_cc* speichern, wobei *cc* das Sprachkürzel ist. Beispielsweise gehört die deutsche Sprachdatei nach *values_de*, eine französiche nach *values_fr*.

Was aber geschieht mit Strings, die fest im Programmcode stehen, weil ein Layout-Element dynamisch gefüllt wird? Im vorliegenden Fall ist das zum Beispiel der String `" von "`.

Die einfachste Lösung ist, für das eine Wörtchen eigene String-Ressourcen in die *strings.xml*-Dateien zu packen. Damit Ihnen Android den String automatisch aus der richtigen Sprachdatei holt, verwenden Sie einfach die `getResources()`-Methode:

```
getResources().getString(R.string.von)
```

Übergeben Sie an `getString()` die richtige Resource-ID, und Sie müssen sich nur noch um die nötigen Leerzeichen kümmern:

```
" "+getResources().getString(R.string.von)+" "
```

Und Leerzeichen muss man glücklicherweise nicht übersetzen.

## 7.2 Bestenliste im Internet

Was ist schöner, als den eigenen Highscore zu bewundern? Ganz einfach: ihn im Internet mit anderen zu teilen. Am schönsten ist eine Liste der besten 100 Spieler, damit sich mehr Mückenfänger im Internet wiederfinden.

Eine Internet-Bestenliste muss zwei wesentliche Funktionen bieten:

- Einsortieren neuer Rekorde
- Anzeige der besten Spieler

Um Speicher zu sparen, muss die Bestenliste nicht jeden Rekord entgegennehmen: Ist einer zu schlecht, um überhaupt in der Bestenliste zu erscheinen, muss er nicht gespeichert werden.

Offensichtlich benötigen Sie zwei relativ anspruchsvolle Komponenten für eine Internet-Bestenliste:

- den Server, der die Liste verwaltet
- einen Client, der auf den Server zugreift und Teil des Spiels ist

Während der Client in Android relativ leicht einzubauen ist, können Sie die Serverkomponente nicht ohne Weiteres in Java schreiben und in Ihren Webspace hochladen.

Aber zum Glück gibt es eine relativ einfache Lösung: die **Google App Engine**. Dort können Sie kostenlos Serveranwendungen hinterlegen, die Ihr Android-Spiel über das Internet erreichen kann – egal, auf welchem Handy es installiert ist. Das klingt auf den ersten Blick schwerer, als es ist – denn genau genommen müssen Sie wirklich nur ein wenig Java-Code produzieren, und um den Rest kümmert sich die Google App Engine!

Betrachten Sie den Bau einer Java-Serveranwendung als kleinen Exkurs, behalten Sie aber im Hinterkopf, dass viele besonders spannende Apps nicht ohne Serveranwendung auskommen. Daher wäre es fahrlässig, dieses wichtige Thema außen vor zu lassen.

**Ein fertiger Highscore-Server**

Sie müssen nicht unbedingt selbst einen Highscore-Server bauen und in der Google App Engine veröffentlichen. Verwenden Sie einfach meinen:

*http://myhighscoreserver.appspot.com/highscoreserver*

Leider kann ich Ihnen nicht garantieren, dass dieser Server permanent erreichbar ist oder auch noch ein paar Jahre nach dem Erscheinen dieses Buches noch funktionieren wird.

Übergehen Sie die nächsten Abschnitte, und machen Sie mit dem Client weiter (siehe Abschnitt »Die Internet-Erlaubnis«).

## Ein App Engine-Projekt

Um die Google App Engine verwenden zu können, müssen Sie zunächst einen kostenlosen Account anlegen, und zwar hier:

*http://code.google.com/intl/de-DE/appengine/*

Sie benötigen dazu einen Google-Account (den Sie als Besitzer eines Android-Handys ohnehin haben). Einmalig wird Ihnen ein Aktivierungscode per SMS geschickt, dazu müssen Sie Ihre Handynummer eingeben. Laden Sie außerdem die aktuelle Version des **Google App Engine SDK for Java** herunter, und vergessen Sie nicht das Google-Plugin für Eclipse (*http://code.google.com/intl/de-DE/eclipse/*).

Legen Sie auf der App-Engine-Webseite eine neue Applikation an, und merken Sie sich deren ID, denn die müssen Sie später in Eclipse eingeben. In Eclipse erzeugen Sie dann mit dem Google-Plugin eine neue Webanwendung (New • Web Application). Schalten Sie im zugehörigen Wizard das Google Web Toolkit aus, das brauchen wir hier nicht (Abbildung 7.3).

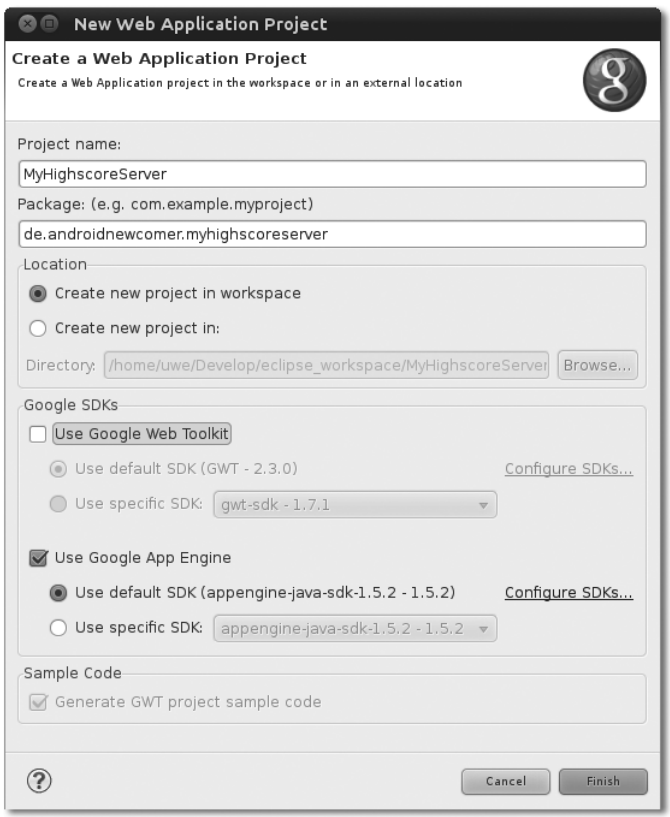



**Abbildung 7.3** Erstellen Sie ein neues Web Application Project.

Der Wizard erzeugt Ihnen daraufhin alle nötigen Dateien, und sage und schreibe eine davon ist Java-Code namens *HighscoreServerServlet.java* (der genaue Name der Datei hängt davon ab, wie Sie Ihr Projekt genannt haben).

Tatsächlich enthält die Klasse bereits Testcode, nämlich das gute alte »Hello World«. Möchten Sie die Anwendung mal ausprobieren, um eine Vorstellung davon zu haben, wie sie funktioniert?

Das ist leicht. Klicken Sie mit der rechten Maustaste auf das Projekt, und wählen Sie Run as • Web Application. Beobachten Sie die Ausgaben im Console-View von Eclipse. Nach einigen Sekunden werden Sie dort ungefähr Folgendes sehen:

```
INFO: The server is running at http://localhost:8888/
```

Öffnen Sie einen Browser, und geben Sie die genannte Adresse ein (die 8888 ist eine **Port-Nummer**, die abweichen kann). Sie sehen eine einfache Willkommensseite – das ist aber noch nicht Ihre Webapplikation. Die taucht als Link auf, Sie werden den Namen wiedererkennen, den Sie dem Projekt verpasst haben.

Klicken Sie den Link an, und schon erscheint das »Hello World«.

Was ist nun geschehen?

Der Browser hat eine Anfrage (Request) an die Adresse *localhost:8888/highscore-server* geschickt, und zwar über das Internet-Protokoll **HTTP** (Hypertext Transfer Protocol). Nichts anderes tut der Browser, wenn Sie im Web surfen, der Unterschied ist nur: Diesmal antwortet Ihr Rechner selbst (denn er heißt »localhost«). Genau genommen, antwortet die simulierte Version der App Engine, die Sie gerade gestartet haben. Um diese Antwort (Response) zu erzeugen, führt die App Engine den simplen Java-Code aus, der die »Hallo-Welt«-Ausgabe erzeugt (Abbildung 7.4). Diesen Java-Code nennt man **Servlet**.

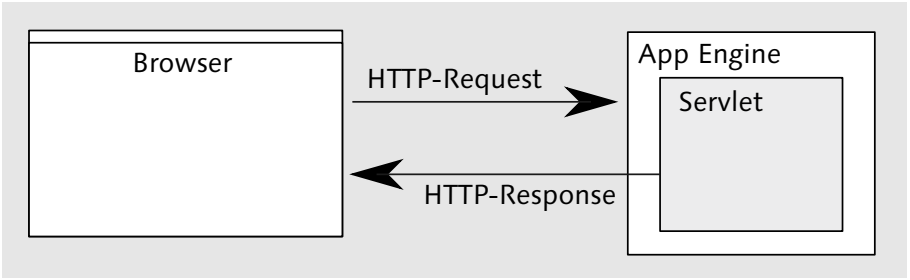



**Abbildung 7.4** Der Browser schickt eine Anfrage an die App Engine, und das Servlet antwortet.

Jetzt stellen Sie sich einfach vor, dass Ihr Servlet auf der »richtigen« App Engine läuft, von überall aus erreichbar ist und dass anstelle des Browsers Ihr Spiel die Requests ausführt und die Response entgegennimmt!

Sie müssen nur noch ein paar Parameter an die Requests hängen (z. B. einen neuen Highscore und den Namen des Spielers). Die Response ist dann die jeweils aktualisierte Bestenliste.

Um Ihre von Eclipse gestartete lokale App Engine zu stoppen, müssen Sie den roten Button im Console-View in Eclipse anklicken.

### URL-Parameter entgegennehmen

Sie haben sicher schon kompliziert aussehende URLs gesehen, in denen Parameter stehen. Wenn Sie etwa einen Suchbegriff bei Google eintippen und die Suche starten, tauchen in der Adresszeile eine ganze Menge undurchsichtiger Parameter auf. Diese Parameter werden genutzt, um einem Servlet mitzuteilen, was man von ihm will. Voneinander getrennt werden die Parameter mit `&`-Zeichen, und ihr Wert ist mit einem =-Zeichen angehängt. Zwischen der eigentlichen Adresse des Servlets und den Parametern steht ein `?`.

Schauen Sie sich folgende URL an:

*http://myhighscoreserver.appspot.com/highscoreserver?game=mueckenfang&points=4000&name=Paul+Panther*

Offenbar handelt es sich hier um einen (recht bescheidenen) neuen Highscore, den ein Spieler namens Paul Panther erreicht hat. Da in einer URL keine Leerzeichen erlaubt sind, müssen Sie sie durch + ersetzen. Es gibt übrigens noch andere Einschränkungen, über die später bei der Implementierung des Clients zu reden sein wird. Offensichtlich wäre es beispielsweise sehr ungünstig, wenn der Name des Spielers ein `&` oder ein = enthalten würde.

Sie sehen, dass wir gleich mit einplanen, dass der Highscore-Server für mehr als ein Spiel funktioniert (Parameter `game`).

Immer wenn ein HTTP-GET-Request bei Ihrem Servlet ankommt, wird die Methode `doGet()` aufgerufen (es gibt auch POST-Requests, aber die sollen uns im Moment nicht kümmern). Im ersten Schritt muss die Methode die Parameter aus der URL auslesen. Dazu dient das `HttpServletRequest`-Objekt, das der Methode von der App Engine übergeben wird:

```
String game = req.getParameter("game");
String name = req.getParameter("name");
```

Bei dem numerischen Parameter `points` müssen Sie etwas aufpassen, denn `getParameter()` liefert immer einen String zurück. Wenn der gewünschte Parameter in der URL fehlt, erhalten Sie allerdings den Wert `null`, der nicht in eine Zahl konvertiert werden kann. Beschaffen Sie sich also den numerischen Highscore-Wert wie folgt:

```
String pointsStr =  req.getParameter("points");
int points = 0;
if(pointsStr != null) {
   points = Integer.parseInt(pointsStr);
}
```

Falls das Servlet auf diese Weise Punkte und einen Namen erhalten hat, speichern Sie den Highscore:

```
if(points>0 && name!=null) {
   addHighscore(game, name, points);
}
```

Anschließend geben Sie die aktuelle Bestenliste zurück:

```
returnHighscores(resp, game, max);
```

Dass diese Methode den `HttpServletResponse`-Parameter `resp` benötigt, dürfte Sie nicht verwundern. Neben dem Parameter für das Spiel habe ich Ihnen hier noch ein `max` hinzugemogelt, das die Länge der Highscore-Liste begrenzt. Den Wert können Sie einfach aus einem weiteren URL-Parameter namens `max` extrahieren:

```
String maxStr =  req.getParameter("max");
int max = 10;
if(maxStr!=null) {
   max = Integer.parseInt(maxStr);
}
```

Sie sehen, dass hier die 10 als sinnvoller Standard verwendet wird, falls kein Parameter `max` übergeben wurde.

## Daten im High Replication Datastore speichern

Nun ist es an der Zeit, die beiden fehlenden Methoden zum dauerhaften Speichern und Ausgeben von Highscores zu füllen. Dazu müssen wir Daten speichern und wieder auslesen. Die App Engine bietet dazu eine sehr übersichtliche Möglichkeit in Form des High Replication Datastores.

Sie müssen nicht viel über diesen Datastore wissen. Er ist sehr einfach aufgebaut: Unter sogenannten **Keys** können Sie beliebig viele **Entity**-Objekte speichern. Die Entity-Objekte wiederum können Name-Wert-Paare enthalten (Abbildung 7.5).

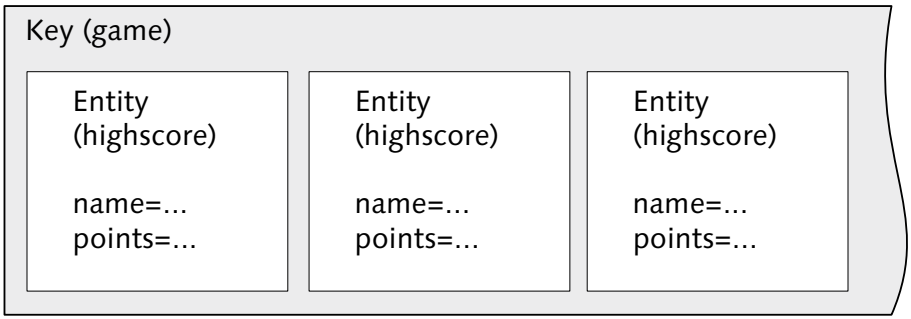



**Abbildung 7.5** Wir speichern je einen Key für jedes Spiel (game), und darin für jeden eingereichten Highscore ein Entity. Jedes Entity enthält den Namen (name) und die Punktzahl (points).

Die Methode zum Speichern eines neuen Highscores sieht wie folgt aus:

```
private void addHighscore(String game, String name, int points) {
   DatastoreService datastore = DatastoreServiceFactory.
                                getDatastoreService();
   Key gameKey = KeyFactory.createKey("game", game);
   Entity highscore = new Entity("highscore", gameKey);
   highscore.setProperty("name", name);
   highscore.setProperty("points", points);
   datastore.put(highscore);
}
```

Zunächst beschaffen wir uns eine Referenz auf den `datastore`, dann lassen wir uns von der `KeyFactory` einen Key mit dem Namen des Spiels erzeugen.

Anschließend entsteht ein neues Entity-Objekt, das den Namen `"highscore"` erhält und unter dem `gameKey` aufgehängt wird. Auf diese Weise werden alle Highscores ihrem jeweiligen Spiel zugeordnet, und selbst wenn ein neues Spiel unterstützt werden soll, müssen Sie nichts weiter tun: Es wird einfach implizit der nötige neue Key angelegt.

Das Entity `highscore` bekommt dann die beiden nötigen Werte verpasst, nämlich den Namen und die Punktzahl.

Die letzte Zeile speichert das Entity – also den Rekord – im High Replication Datastore.

### Highscores aus dem Datastore auslesen

Das Servlet kann jetzt also schon Highscores abspeichern – und zwar beliebig viele, bis Google Ihnen den Strom abdreht. Allerdings reicht vermutlich ein Leben nicht, um so viele Highscores an das Servlet zu schicken, dass das passiert. Sie können sich in Ihrer App-Engine-Oberfläche die **Quota Details** anschauen und darüber staunen, was Sie alles bei Google gratis kriegen:

- Requests: unlimited
- Datastore Entity Put Operations: unlimited
- High Replication Data: 0,5 Gigabyte (in 24 Stunden)

Beim Auslesen der Highscores werden wir uns selbstverständlich auf das jeweilige Spiel – also den passenden Key – beschränken. Deshalb sieht der Anfang der Methode `returnHighscores()` nicht sonderlich überraschend aus:

```
DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
Key gameKey = KeyFactory.createKey("game", game);
```

Die nächste Aufgabe ist es, die Entity-Objekte in der richtigen Sortierreihenfolge auszulesen, und zwar bis zum gewünschten Maximum.

Zum Auslesen verwenden wir eine Query (Abfrage):

```
Query query = new Query("highscore", gameKey);
```

Das `Query`-Objekt erzeugen wir mit Parametern für den Namen der gewünschten Entity-Objekte (`"highscore"`) und dem `gameKey`. Dann hängen wir ein Sortierkriterium an:

```
query.addSort("points", Query.SortDirection.DESCENDING);
```

Damit ist die Abfrage erstellt und kann nun auf den Datastore losgelassen werden. Das geschieht in der folgenden Zeile:

```
List<Entity> highscores = datastore.prepare(query).asList(
                          FetchOptions.Builder.withLimit(max));
```

Mit den spitzen Klammern verraten wir dem Compiler, dass wir eine Liste mit Entity-Objekten erwarten, außerdem erkennen Sie die Begrenzung auf `max` Rückgabewerte.

Bleibt nur noch eines: die Highscores auszugeben.

```
for(Entity e : highscores) {
   resp.getWriter().println(e.getProperty("name") + "," +
        e.getProperty("points"));
}
```

Die Schleife über alle Elemente der Liste `highscores` kommt Ihnen sicher bekannt vor. Für jedes Element wird der Code in den geschweiften Klammern einmal ausgeführt.

Das `HttpServletResponse`-Objekt `resp` verfügt über einen `PrintWriter`, den wir uns mit `getWriter()` holen können. Der Writer wiederum besitzt eine Methode `println()` (abgekürzt für »print line«), der wir übergeben, was als Antwort geschickt werden soll. Wir wählen hier das einfachste Format, das möglich ist, indem einfach Name und Punktzahl durch ein Komma getrennt werden.

Jede der bis zu zehn Zeilen der Antwort wird also einen Highscore enthalten.

Wenn Sie die Methode komplett geschrieben haben, können Sie das Servlet (zunächst lokal) ausführen.

Dann wechseln Sie zum Browser und geben in die Adressleiste ein:

*http://localhost:8888/highscoreserver?game=mueckenfang&name=Peter+Panther&points=1200&max=10*

Beim ersten Aufruf erhalten Sie daraufhin nur eine Zeile:

```
Peter Panther,1200
```

Ändern Sie den Namen und die Punktzahl in der Adressleiste, und drücken Sie erneut ↵:

*http://localhost:8888/highscoreserver?game=mueckenfang&name=Hans+Hase&points=900&max=10*

Das Resultat wird sein:

```
Peter Panther,1200
Hans Hase,900
```

Je nach Browser kann es sein, dass beides in einer Zeile erscheint. Das liegt daran, dass wir kein ordentliches HTML erzeugen. Schauen Sie sich den Quellcode des Browser-Fensters an, dann sehen Sie die Ausgaben zeilenweise.

Spielen Sie etwas mit Ihrem Highscore-Server, bis Sie sicher sind, dass er wie gewünscht funktioniert. Dann deployen Sie ihn auf Ihre App Engine. Kllicken Sie mit rechts auf das Projekt in Eclipse, und wählen Sie im Popup-Menü Google • Deploy to App Engine. Beim ersten Mal müssen Sie sich einmal authentifizieren, danach verbindet sich Ihr Eclipse automatisch mit der richtigen Applikation in Ihrem App-Engine-Account.

Anschließend können Sie das Servlet auch über die öffentliche Adresse aufrufen und ausprobieren und in Ihrer App-Engine-Konsole überwachen.

Jetzt, da Ihr Highscore-Server läuft, ist der Exkurs in die abenteuerliche Welt der Google App Engine auch schon beendet. Wir können uns als Nächstes dem Client zuwenden, der anstelle des bisher verwendeten Browsers künftig mit Ihrem Servlet sprechen wird.

## Die Internet-Erlaubnis

Um an die Highscores im Netz zu kommen, benötigt die App die Erlaubnis, aufs Internet zuzugreifen. Der Wunsch nach einer solchen Permission wird dem Benutzer vor der Installation einer App angezeigt – das haben Sie sicher schon selbst gesehen. Der Besitzer des Geräts muss einer App also explizit die Erlaubnis erteilen, kritische Funktionen zu verwenden. Im Fall des Internet-Zugriffs ist das besonders wichtig: Stellen Sie sich vor, eine an sich harmlose App, die augenscheinlich überhaupt keinen Internet-Zugriff benötigt, lädt heimlich im Hintergrund große Datenmengen herunter oder verschickt Spam im Auftrag zwielichtiger Gesellen!

Permissions werden im Android-Manifest Ihrer App verwaltet. Öffnen Sie also in Eclipse die Datei *AndroidManifest.xml*, und aktivieren Sie die Registerkarte Permissions.

Klicken Sie auf Add, und fügen Sie ein Element der Sorte Uses Permission hinzu. Auf der rechten Seite wählen Sie unter den unzähligen Permissions, die vordefiniert sind, android.permission.INTERNET aus (Abbildung 7.6).

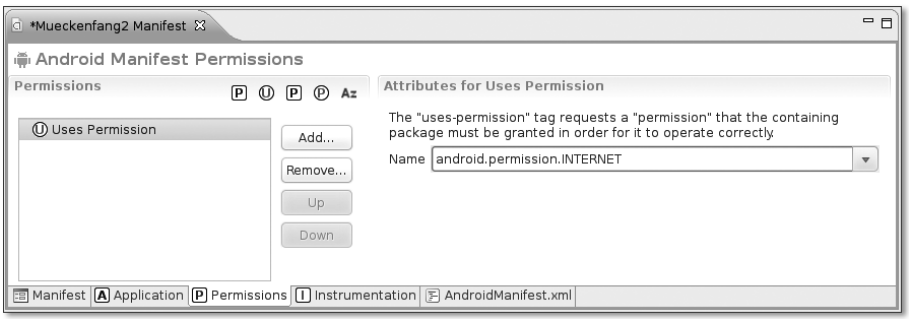



**Abbildung 7.6** Nur wenn Ihre App im Manifest die Internet-Permission anfordert, darf sie mit der weiten Welt dort draußen Kontakt aufnehmen.

Speichern Sie die Datei, damit Sie als Nächstes den eigentlichen Internet-Zugriff verwirklichen können.

### Der Android-HTTP-Client

Jetzt, wo der Server zur Verwaltung von Highscores zur Verfügung steht und die App mit ihm sprechen darf, können Sie sich daranmachen, die Android-Seite zu programmieren: den **Client**.

Fügen Sie dazu der Klasse `MueckenfangActivity` eine neue Methode hinzu:

```
private void internetHighscoresAnzeigen() {
   highscores = "";
}
```

Wie Sie sehen, leert diese Methode zunächst einmal alle möglicherweise vorhandenen Highscores. Das zugehörige `String`-Attribut `highscores` müssen Sie natürlich ebenfalls in der Klasse `MueckenfangActivity` deklarieren.

Um die Details der Kommunikation müssen Sie sich glücklicherweise nicht kümmern, denn Android stellt bereits eine Klasse zur Verfügung, die das Zwiegespräch mit dem Server für Sie erledigt: `AndroidHttpClient`.

Ein Objekt dieser Klasse erzeugen Sie ausnahmsweise nicht mit `new`, sondern mit einer statischen Methode:

```
AndroidHttpClient  client = AndroidHttpClient.newInstance("Mueckenfang");
```

Die Methode `newInstance()` erzeugt nicht nur das `AndroidHttpClient`-Objekt, es initialisiert es auch mit sinnvollen Standardwerten. Bloß einen Namen für den User Agent müssen Sie noch als Parameter übergeben. Der User Agent, in diesem Fall `"Mueckenfang"`, ist eine Zeichenfolge, die mit zum Server geschickt wird. Browser verwenden dies, um sich zu identifizieren – so kann der Server beispielsweise einem Handy-Browser eine optimierte Seite ausliefern, dem Desktop-Browser die umfangreichere Version (natürlich von oben bis unten mit Werbung gepflastert). Dem Highscore-Server ist der User Agent herzlich egal, Sie können also einen beliebigen String verwenden.

Als Nächstes müssen Sie eine HTTP-Anfrage basteln. Der Server unterstützt GET-Anfragen, also:

```
HttpGet request = new HttpGet(HIGHSCORE_SERVER_BASE_URL + "?game="
                     + HIGHSCORESERVER_GAME_ID);
```

Wie Sie sehen, wird der GET-Request aus ein paar String-Konstanten zusammengebastelt. Da die nämlich mit hoher Wahrscheinlichkeit noch an anderer Stelle gebraucht werden (beim Speichern neuer Highscores zum Beispiel), vermeiden wir auf diese Weise redundante Parameterwerte. Ansonsten besteht die Gefahr, dass Sie irgendwann einmal die URL nur an einer Stelle ändern und die andere vergessen – die Folgen wären unvorhersehbar.

Definieren Sie die beiden Konstanten in der Klasse `MueckenfangActivity`:

```
private static final String HIGHSCORE_SERVER_BASE_URL=
        "http://myhighscoreserver.appspot.com/highscoreserver";
private static final String HIGHSCORESERVER_GAME_ID =
        "mueckenfang";
```

Ändern Sie die URL, wenn Sie einen eigenen Server haben. Versuchen Sie nicht, die Adresse des Entwicklungsservers zu nehmen, den Eclipse hochfährt, wenn Sie das Servlet lokal testen: Jener Server läuft unter der Adresse *http://localhost*, und die kann weder Ihr Emulator noch Ihr Handy erreichen. Sie müssten Ihren Rechner schon mit dem Internet verbinden, und ihm einen von überall erreichbaren Namen verpassen, damit das funktioniert. (Wenn Sie das unbedingt probieren möchten, finden Sie bei *http://dyndns.com* einen passenden Service.)

Als Nächstes ist es an der Zeit, die Anfrage an den Server zu schicken:

```
HttpResponse response = client.execute(request);
```

Bis die Methode `execute()` des `AndroidHttpClient` zurückkehrt, können einige Sekunden vergehen. Ich werde Ihnen später erklären, warum das problematisch ist und wie man mit solchen Wartezeiten umgeht.

Die Methode `execute()` liefert eine `HttpResponse` zurück, aus der Sie jetzt die Ausgabe des Highscore-Servers extrahieren müssen. Die ist im `HttpEntity` der Antwort versteckt:

```
HttpEntity entity = response.getEntity();
```

Leider stellt `HttpEntity` den von uns begehrten Inhalt nicht einfach als String zur Verfügung, sondern nur als Stream. Streams – Datenströme – sind sehr häufig eingesetzte Klassen in Java. Für die Details müssen wir uns im Augenblick nicht interessieren; stellen Sie sich den Stream einfach als geheimnisvollen Textspender vor, der nur einzelne Buchstaben nacheinander ausspuckt (glücklicherweise in der richtigen Reihenfolge).

Um aus einem Stream zu lesen, benötigen Sie einen `InputStreamReader`.

```
InputStreamReader reader = new InputStreamReader(entity.getContent(),
"utf-8" );
```

Dem neuen `InputStreamReader` wird neben `entity.getContent()` (dem geheimnisvollen `InputStream`) der internationale Zeichensatz `"utf-8"` zugewiesen.

Jetzt kann der `reader` ein erstes Zeichen lesen:

```
int c = reader.read();
```

Der Rückgabewert der Methode `read()` ist ein `int`-Wert, der dem Unicode des zugehörigen Buchstabens entspricht – es sei denn, der geheimnisvollen Schachtel sind die Buchstaben ausgegangen, dann ist das Resultat –1. Hängen wir also das gelesene Zeichen an den String `highscores` an:

```
highscores += (char)c;
```

Hier ist eine explizite Konvertierung von `int` nach `char` erforderlich. Sie wollen aber nicht nur das erste Zeichen aus dem Stream, sondern alle. Wiederholen Sie also den Vorgang, solange die geheimnisvolle Schachtel Buchstaben liefert:

```
while(c != -1) {
   highscores += (char)c;
   c = reader.read();
}
```

Anschließend steht in `highscores` genau jener Text zur Verfügung, den der Highscore-Server ausgespuckt hat.

**Streams**

Datenströme sind wichtige Elemente in der modernen Programmierung. Verwechseln Sie Streams nicht mit Dateien: Ein Stream hat beispielsweise keine definierte Länge. Denken Sie ans Live-Streaming der Fußball-WM im Internet: Wer weiß schon, ob es Verlängerung oder Elfmeterschießen gibt?

Aber auch jede Webseite ist ein Beispiel für einen Stream: Ein Browser kann bereits mit der Verarbeitung des HTML-Codes beginnen, wenn die Seite noch gar nicht vollständig aus dem Netz geladen wurde. Auf diese Weise kann er Seiten viel schneller darstellen, als wenn er warten müsste, bis alle Daten angekommen sind.

Ein Eingabe-Datenstrom (Klasse `InputStream`) verrät Ihnen also nur, *ob* (und wie viele) Daten-Bytes im Moment gelesen werden können (Methode `available()`), und erlaubt Ihnen, ein oder mehrere Bytes auszulesen (verschiedene `read()`-Methoden). Erst wenn der Stream definitiv beendet ist und keine Daten mehr eintrudeln werden, gibt `available()` eine 0 zurück und `read()` –1.

Wenn Sie wissen, dass der `InputStream` Textdaten enthält, verwenden Sie einen `InputStreamReader`, der aus den zunächst nutzlosen Bytes des Streams lesbare Zeichen macht. Der Reader muss dazu wissen, in welcher Zeichensatz-Codierung die Textdaten vorliegen. Meist ist das heutzutage UTF-8, der Unicode-Zeichensatz, aber es kann auch sein, dass Ihnen ISO-8859-1 begegnet (der Windows-Zeichensatz). Wenn Sie selbst die Datenquelle geschrieben haben (wie den Highscore-Server), wissen Sie natürlich genau, welche Codierung er liefert. Ansonsten müssen Sie es herausfinden.

Sobald die Highscore-Liste etwas größer wird, wird es etwas langsam, sie Zeichen für Zeichen einzulesen. Ein wenig verwirrend, aber viel schneller ist ein »Leser im Leser«: Ein `BufferedReader`, der aus dem `InputStreamReader` zeilenweise liest:

```
InputStreamReader input = new InputStreamReader(entity.getContent(),
"UTF8" );
BufferedReader reader = new BufferedReader(input,2000);
List<String> highscoreList = new ArrayList<String>();
String line = reader.readLine();
while (line != null) {
   highscoreList.add(line);
   line = reader.readLine();
}
```

`OutputStreams` funktionieren übrigens analog zu `InputStream`s, bloß dass Sie zum Ausgeben von Byte-Daten eine `write()`-Methode verwenden und meist einen `PrintWriter`, um Texte auszugeben.

Sie erinnern sich bestimmt, dass der Server die Highscores in der Form

```
Nickname,Highscore
```

zeilenweise zurückgibt. Da das auf dem Bildschirm keine nachvollziehbare Darstellung wäre, gönnen wir uns eine kleine Konvertierung:

```
highscores = highscores.replace(",", ": ");
```

Die Methode `replace()` gibt eine Kopie des Strings zurück, auf dem sie angewendet wird, und ersetzt dabei jedes Auftreten des ersten Parameters durch den zweiten. Beachten Sie, dass die Methode *nicht* den String selbst verändert. Deshalb müssen Sie den Rückgabewert wieder dem String `highscores` zuweisen, um den gewünschten Effekt zu erzielen.

Eclipse markiert Ihnen schon die ganze Zeit einige Codestellen rot, ohne dass Sie sich einer Schuld bewusst sind. Das liegt daran, dass bei einem Internet-Zugriff eine Menge schiefgehen kann: Der Server könnte offline sein oder umgezogen; oder Sie befinden sich schlicht in einem Funkloch. Solche Ausnahmefälle erzeugen Exceptions tief in den Eingeweiden des `AndroidHttpClients`. Er kann dann nicht mehr sinnvoll weitermachen, daher bricht er seine Arbeit ab und wirft Ihnen eine bestimmte Sorte von Exceptions vor die Füße: eine IOException (**IO** steht für **I**nput und **O**utput).

Die Methoden `client.execute()` und `reader.read()` verraten in ihrer Deklaration, dass sie eventuell IOExceptions werfen, daher verlangt Eclipse, dass Sie sich darum kümmern.

Grundsätzlich haben Sie zwei Möglichkeiten, solche Exceptions zu behandeln:

- Sie machen es genau wie die Methode, die die Exception wirft: Sie brechen die Arbeit ab und werfen die Exception zur nächsthöheren Methode. Dazu müssen Sie Ihre eigene Methode um dieselbe Exception-Deklaration erweitern wie die Methode, die die Exception wirft: `throws IOException`. Die Pro-

grammierhilfe von Eclipse bietet Ihnen diese Lösungsmöglichkeit an, wenn Sie Strg + 1 drükken. Allerdings hilft Ihnen das im aktuellen Fall nicht, denn es würde das Problem nur verschieben: Irgendwo müssen Sie die Ausnahmesituation behandeln.

- Sie fangen die Exception. Dazu müssen Sie den »riskanten« Teil des Programmcodes mit einer `try-catch`-Konstruktion versehen. Innerhalb des `catch`-Blocks behandeln Sie dann die Ausnahmesituation.

Schauen Sie sich zunächst die gesamte Methode mitsamt der Ausnahmebehandlung an:

```
private void internetHighscoresAnzeigen() {
   try {
      highscores="";
      AndroidHttpClient  client = AndroidHttpClient.newInstance(
             "Mueckenfang");
      HttpGet request = new HttpGet(HIGHSCORE_SERVER_BASE_URL +
          "?game=" + HIGHSCORESERVER_GAME_ID);
      HttpResponse response = client.execute(request);
      HttpEntity entity = response.getEntity();
      InputStreamReader reader = new InputStreamReader(entity.
           getContent(), "UTF8" );
      int c = reader.read();
      while(c>0) {
         highscores += (char)c;
         c= reader.read();
      }
      highscores = highscores.replace(",", ": ");
   } catch (IOException e) {
      highscores = "Fehler";
   }
}
```

Wenn alles klappt, wird der Codeblock hinter `try` durchlaufen und der hinter `catch` nicht. Falls irgendwo im `try`-Block eine aufgerufene Methode eine IOException wirft, bricht Java die Verarbeitung ab und macht im `catch`-Block weiter. In den runden Klammern hinter dem Schlüsselwort `catch` steht die Exception-Klasse, die behandelt wird. Sie müssen für jede möglicherweise erzeugte Exception-Klasse einen eigenen `catch`-Block einrichten, aber glücklicherweise genügt im Moment einer. Wir zeigen dem Spieler einfach einen Fehler an, indem wir anstelle der Highscore-Liste das Wort »Fehler« in das Attribut `highscores` schreiben.

**Checked und unchecked Exceptions**

Grundsätzlich sind Exceptions in Java nichts anderes als Objekte von Klassen, die von der Klasse `Exception` erben. Allerdings gibt es eine wichtige Gruppe von Ausnahmen, die von `RuntimeException` erben. Beispiele dafür sind `NullPointerException` oder `IllegalArgumentException`. Solche Exceptions heißen *unchecked*, und sie werden weder mit dem Schlüsselwort `throws` deklariert noch mit `catch` gefangen – weil sie schlicht und einfach nie auftreten sollten. Natürlich tun sie das trotzdem: Aber dann müssen Sie als Programmierer ran und die Ursache beheben, die fast immer in einem Programmierfehler besteht. Es ist völlig sinnlos, eine solche Exception zu fangen und zu versuchen, weiterzumachen, als wäre nichts geschehen. Wenn eine unchecked Exception auftritt, wird Ihre App mit der bekannten Dialogbox »... wurde unerwartet beendet« geschlossen, und der Benutzer erhält die Möglichkeit, einen Fehlerbericht zu senden. Den finden Sie mitsamt der kompletten Stacktrace-Ausgabe der Exception in Ihrem App-Verwaltungsbereich des Android Markets.

*Checked* Exceptions dagegen – also zum Beispiel IOExceptions oder ConnectionTimeoutExceptions – werden mit `throws` deklariert, und Java verlangt, dass Sie sie mit `try-catch`-Blöcken behandeln. Zeigen Sie dem Benutzer eine Fehlermeldung, oder versuchen Sie, die fehlgeschlagene Operation zu wiederholen (bloß bitte nicht endlos). Gerade bei Smartphones kann die Internet-Verbindung kurzzeitig abreißen, aber Sekunden später schon wieder zur Verfügung stehen.

Wenn Sie sich entschließen, in einer Methode eine Exception mit `try-catch` zu behandeln, dann tun Sie das auch. Ich habe schon unzählige leere `catch`-Blöcke gesehen – fügen Sie denen keine weiteren hinzu. Zeigen Sie dem Benutzer einen Hinweis, was schiefgegangen ist, vielleicht kann er etwas unternehmen: Vielleicht den Schreibschutz von der SD-Karte entfernen oder Platz schaffen, wenn sie voll ist. Erzeugen Sie mindestens eine Ausgabe ins Systemprotokoll. Dann haben Sie im Fall des Falles immerhin einen Anhaltspunkt.

Wie ich bereits angedeutet habe, kann die Unterredung mit dem Highscore-Server die eine oder andere Sekunde in Anspruch nehmen. Deshalb dürfen Sie keinesfalls die Methode `internetHighscoresAnzeigen()` beispielsweise von `onResume()` aus aufrufen! Das würde die Benutzeroberfläche des Smartphones blockieren, bis der Server geantwortet hat – in einem Handynetz ist das ein inakzeptables Verhalten. Daher wird es von Android unterbunden – Ihnen bleibt nichts anderes übrig, als das Laden der Highscores von der Benutzeroberfläche zu entkoppeln, indem Sie den Vorgang in den Hintergrund verlagern.

## Background-Threads

Glücklicherweise kann ein Smartphone mehrere Dinge quasi gleichzeitig tun, genau wie Ihr Desktop-Rechner oder Notebook. Sie müssen also dafür sorgen, dass zum einen die Benutzeroberfläche funktioniert und nebenbei etwas anderes passiert: in diesem Fall das Laden der Bestenliste aus dem Internet.

Das Zauberwort dafür lautet: **Threads**.

Denken Sie jetzt nicht an Hälse, sondern an Internet-Foren: Dort gibt es in der Regel eine ganze Menge Threads, und in jedem diskutieren verschiedene Leute über unterschiedliche Dinge. Dabei sind diese Threads unabhängig voneinander, d. h., wenn sie den einen lesen, ist es völlig egal, was gerade im anderen passiert.

Etwa so funktionieren auch Threads in Java. Normalerweise läuft der Code Ihrer App in nur einem Thread, dem **Vordergrund-Thread** oder **UI-Thread** (UI = User Interface). Befehle in jedem Thread werden nacheinander verarbeitet, und wenn einer etwas länger dauert, dann ist die Benutzeroberfläche solange nicht benutzbar, weil sie keine Ereignisse entgegennehmen kann.

Sie können aber jederzeit einen zweiten, parallelen Thread starten und ihm die Aufgabe erteilen, bestimmten Code abzuarbeiten, der nicht mit der Benutzeroberfläche interagiert. Das nennt man **Background-Thread**. Der betreffende Code läuft also unsichtbar im Hintergrund (Abbildung 7.7).

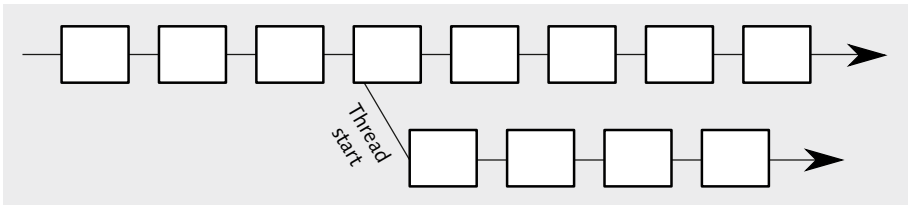



**Abbildung 7.7** Wenn der Hauptthread (oben) einen zweiten Thread startet (unten), läuft dieser quasi parallel.

Wenn Sie einen Thread erzeugen, können Sie ihm ein Objekt übergeben, das das Interface `Runnable` implementiert. Sobald Sie den Thread starten, wird er die Methode `run()` im angegebenen Runnable aufrufen. Wenn die `run()`-Methode zurückkehrt, beendet sich der Thread selbst.

Sie erinnern sich sicher, dass Sie schon in der `GameActivity` eine `run()`-Methode implementiert haben, die zum Herunterzählen der Zeit diente. Dazu haben Sie einfach die Activity selbst mit dem Interface `Runnable` versehen. Wir werden jetzt ein klein wenig anders vorgehen, weil wir, wie Sie später sehen werden, zwei unterschiedliche Runnables in der `MueckenfangActivity` benötigen.

Dazu schreiben Sie jetzt eine private, innere Klasse innerhalb der Klasse `MueckenfangActivity`:

```
private class HoleHighscores implements Runnable {
   @Override
   public void run() {
```

```
        internetHighscoresAnzeigen();
    }
}
```

Die innere Klasse `HoleHighscores` stellt lediglich die nötige `run()`-Methode bereit, um der Deklaration `implements Runnable` zu genügen. Da eine solche innere Klasse Zugriff auf alle Elemente der Klasse hat, in der sie definiert ist, kann `run()` einfach die Methode `internetHighscoresAnzeigen()` aufrufen.

Jetzt müssen Sie nur noch in `onResume()` einen neuen Thread erzeugen, ihm ein neues Objekt der Klasse `Runnable` übergeben und den Thread starten:

```
@Override
protected void onResume() {
    super.onResume();
    highscoreAnzeigen();
    Thread t = new Thread(new HoleHighscores());
    t.start();
}
```

Entscheidend an diesem Code ist, dass der Aufruf `t.start()` *sofort* zurückkehrt (manche Entwickler nennen das »einen Thread abspalten«). Damit kann der Hauptthread weiterlaufen und ist nicht blockiert. Die Methode `run()` in `HoleHighscores()` wird wenig später im neuen Thread aufgerufen – im Hintergrund, ohne die Benutzeroberfläche zu stören. Dadurch findet die gesamte Kommunikation mit dem Highscore-Server im Hintergrund statt.

Auf diese Weise umgeht die App die Einschränkung, dass der Hauptthread den `AndroidHttpClient` nicht benutzen darf. Allerdings haben wir jetzt ein neues Problem: Hintergrund-Threads ihrerseits dürfen nämlich nicht auf die Benutzeroberfläche zugreifen ...

### Die Oberfläche aktualisieren

Sicher ist Ihnen aufgefallen, dass die Methode `internetHighscoresAnzeigen()` zwar die Highscores in das Attribut `highscores` schreibt – aber ausgegeben wird das bislang noch nicht. Wohin auch?

Öffnen Sie daher zunächst das Layout *main.xml*, und fügen Sie im unteren Bereich einen ScrollView ein, der die Rekordliste anzeigen soll. Der ScrollView bekommt ein LinearLayout eingepflanzt – ScrollViews dürfen nur ein Kindelement enthalten, wir brauchen aber zwei. In das LinearLayout (mit vertikaler Ausrichtung) packen Sie zwei TextViews: einen mit dem festen Inhalt »Highscores« und einen mit der ID `highscores` (Abbildung 7.8).

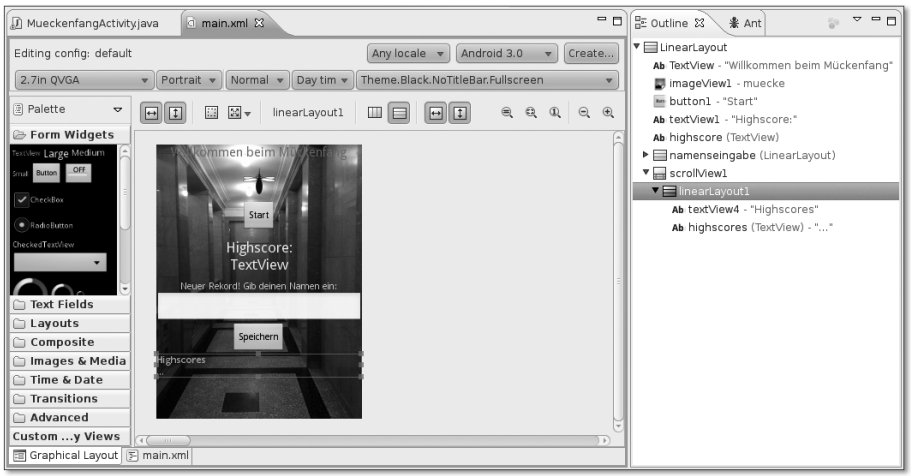



**Abbildung 7.8** Fügen Sie Layout-Elemente ein, um die Highscores anzuzeigen.

Der ScrollView erhält die volle Bildschirmbreite (`match_parent`) und eine feste Höhe von 100dp, das LinearLayout in Breite und Höhe `match_parent`. Die beiden TextViews versehen Sie mit `wrap_content` als Größe.

Jetzt ist ein passender Bildschirmbereich vorhanden, also brauchen Sie eine Methode, die den Inhalt des Attributs `highscores` ausgibt. Aber wo wird diese Methode aufgerufen? Nicht von `internetHighscoresAnzeigen()` aus, denn die läuft im falschen Thread.

Aber zweifellos ist die Stelle, an der diese Methode damit fertig ist, die Highscores zu beschaffen, der richtige Ort, um die eigentliche Ausgabe anzustoßen. Sie müssen nur dafür sorgen, dass nicht der Background-Thread die Aufgabe übernimmt (er darf das nicht), sondern der UI-Thread (der Hauptthread der App).

Dazu gibt es eine eigene Methode in der Klasse `Activity`: `runOnUiThread()`. Dieser Methode müssen Sie ein Runnable übergeben, dessen `run()`-Methode bei nächster Gelegenheit im UI-Thread ausgeführt wird.

Schreiben Sie also zunächst eine weitere innere Klasse, die ein Runnable implementiert und in der `run()`-Methode nichts weiter tut, als den Inhalt von `highscores` in den richtigen TextView zu schreiben:

```
private class ZeigeHighscores implements Runnable {
   @Override
   public void run() {
      TextView tv = (TextView) findViewById(R.id.highscores);
      tv.setText(highscores);
   }
}
```

Jetzt müssen Sie nur noch in `internetHighscoresAnzeigen()` an der richtigen Stelle (nach dem Ersetzen der Kommas durch Doppelpunkte) das neue Runnable im UI-Thread ausführen lassen:

```
runOnUiThread(new ZeigeHighscores());
```

Probieren Sie das Spiel jetzt aus. Sie werden sehen, dass die Startseite sofort auftaucht, und mit kurzer Verzögerung erscheinen die Highscores. Natürlich nur, wenn Sie schon welche auf dem Server gespeichert haben, etwa durch das Testen des Servers über die Adresszeile des Browsers. Ansonsten ist es an der Zeit, dafür zu sorgen, dass eigene Rekorde auch an den Highscore-Server geschickt werden.

**Zu Risiken und Nebenwirkungen von Threads**

Die Nebenläufigkeit von Threads ist nicht nur eine praktische Sache – sie ist aus der modernen Softwareentwicklung nicht mehr wegzudenken.

Allerdings hinken die Programmiertechniken dieser Anforderung ein wenig hinterher. Sie haben bereits gesehen, dass es durchaus ein wenig umständlich ist, mit verschiedenen Threads zu arbeiten – und in Ihrer App sind es nur *zwei*, mit denen Sie hantieren müssen. Bei Webservern und Webapplikationen (wie dem Highscore-Servlet) ist aber jeder eingehende Request ein eigener Thread!

Wenn davon eine Menge gleichzeitig eintreffen, kann es leicht passieren, dass zwei Threads gleichzeitig den gleichen Programmcode abarbeiten. Sie müssen sich sehr genau überlegen, welche Methoden oder Codeblöcke Sie vor solchem Mehrfachzugriff schützen müssen. Um das zu tun, müssen Sie Methoden mit dem Modifizierer `synchronized` versehen oder Code in `synchronized {}`-Blöcke packen.

So etwas kommt glücklicherweise in Andrdoid-Apps nur sehr selten vor, sodass Sie sich vorläufig nicht weiter den Kopf darüber zerbrechen müssen, warum ein Monster noch Schaden anrichtet, obwohl es in einem zweiten Thread schon tot ist.

## Highscores zum Server schicken

Ist Ihnen aufgefallen, dass es keinen großen Unterschied gibt zwischen dem Abrufen der Highscores und dem Einreichen eines neuen Rekords?

In beiden Fällen liefert der Highscore-Server die aktuelle Liste zurück. Der Rekord, bestehend aus Nickname und Punktzahl, wird in URL-Parametern an den Server übergeben. Ist die Punktzahl 0 oder fehlt der Name, ignoriert der Server die Parameter allerdings.

Das Abrufen einer Rekordliste ist also identisch mit dem Einreichen eines Rekords, Sie müssen bloß eine Punktzahl von 0 übergeben. Warum sollten Sie also eine fast identische Kopie der Methode `internetHighscoresAnzeigen()` schreiben, um einen Rekord zum Server zu schicken? Viel effizienter ist es, nur

eine Methode zu haben – die dafür beides kann. Ändern Sie einfach die vorhandene Methode ab, indem Sie ihr zwei Parameter übergeben und den Namen abkürzen:

```
private void internetHighscores(String nickname, int score)
```

Der bisherige Aufruf im Runnable `HoleHighscores` funktioniert natürlich nicht mehr, ändern Sie ihn wie folgt:

```
internetHighscores("",0);
```

Jetzt müssen Sie die beiden neuen Parameter noch an die URL hängen:

```
HttpGet request = new HttpGet(HIGHSCORE_SERVER_BASE_URL
   + "?game=" + HIGHSCORESERVER_GAME_ID
   + "&name=" + URLEncoder.encode(nickname)
   + "&points=" + Integer.toString(score));
```

Wichtig ist hier, dass Sie den `nickname` nicht ohne Weiteres an die URL hängen dürfen. Wie schon zuvor erwähnt, sind Leerzeichen in URLs verboten und müssen durch +-Zeichen ersetzt werden. Auch eine ganze Reihe anderer Zeichen ist nicht erlaubt. Glücklicherweise müssen Sie sich nicht mit den Details herumschlagen – die statische Methode `URLEncoder.encode()` gewährleistet, dass jedes Zeichen nötigenfalls konvertiert wird. Aufseiten des Servers wiederum ist es Aufgabe der Google App Engine, die Encodierung rückgängig zu machen – dort sind keine Maßnahmen im Code nötig.

Als Nächstes müssen Sie die Methode `internetHighscores()` beim Speichern eines neuen Rekords aufrufen. Natürlich muss das wiederum im Hintergrund geschehen, also bauen Sie sich eine neue `Runnable`-Klasse:

```
private class SendeHighscore implements Runnable {
   @Override
   public void run() {
      internetHighscores(leseHighscoreName(),leseHighscore());
   }
}
```

Der Einfachheit halber holt sich die `run()`-Methode hier den Namen und den Rekord aus den Shared Preferences, wo sie zuvor gespeichert wurden. Jetzt müssen Sie nur noch dieses Runnable nebenläufig starten, und zwar wenn der Spieler seinen Namen eingegeben hat und auf den Speichern-Button klickt:

```
@Override
public void onClick(View v) {
   if(v.getId() == R.id.button1) {
      startActivityForResult(new Intent(this,GameActivity.class),1);
```

```
    } else if(v.getId() == R.id.speichern) {
        schreibeHighscoreName();
        highscoreAnzeigen();
        namenseingabe.setVisibility(View.INVISIBLE);
        Thread t = new Thread(new SendeHighscore());
        t.start();
    }
}
```

Die beiden neuen Zeilen in der Methode `onClick()` habe ich im Listing fett hervorgehoben.

Mehr müssen Sie nicht tun! Denn für die ordnungsgemäße Aktualisierung der List sorgt die Methode `internetHighscores()` schon lange.

So, dann versuchen Sie mal, in der Highscore-Liste möglichst weit nach oben zu kommen ... wenn Sie Ihren eigenen Server verwenden, ist das keine Herausforderung. Aber nehmen Sie mal *meinen* Highscore-Server ...

## HTML darstellen

Man kann nicht behaupten, dass die Bestenliste in der jetzigen Form besonders attraktiv aussieht. Lassen Sie uns ein bisschen Farbe ins Spiel bringen!

Die Android-TextViews verstehen HTML-Code. Nicht alles, aber ein wenig. Genug jedenfalls, um Farben, Schriftarten und Bilder einzubauen. HTML, die »Sprache« des World Wide Web, fügt normalem Text Formatierungen hinzu, die ein Browser darstellen kann – oder eben ein TextView. Solche Formatierungsanweisungen bestehen aus sogenannten **Tags**, die den zu formatierenden Text umgeben:

```
Das ist ein <b>fetter</b> Rekord!
```

Die Tags in spitzen Klammern werden bei der Ausgabe interpretiert und verändern die Darstellung des Wortes dazwischen. Jedes Anfangs-Tag endet mit einem passenden Ende-Tag, das mit einem Schrägstrich beginnt.

Dies ist nicht der richtige Ort, Ihnen HTML beizubringen; Sie finden haufenweise Hilfen im Netz oder gute Bücher beim Verlag Ihres Vertrauens. Ohnehin unterstützt der TextView nur wenige HTML-Befehle, und die erklären sich fast von selbst (Tabelle 7.1).

| Unterstützte HTML-Tags |
| --- |
| fett |
| kursiv |
| <u>unterstrichen</u> |
| <font color="#RRGGBB">farbig</font> |
| neue<br>Zeile |
| <img src="bildname"> |

**Tabelle 7.1** Der TextView beherrscht nicht viel HTML – aber genug für ein paar ansehnliche Hervorhebungen.

Lassen Sie uns nun die Antwort des Highscore-Servers mit einigen HTML-Tags verschönern.

Um die Bestenliste einfacher handhaben zu können, konvertieren wir sie zunächst in ein `List`-Objekt, in dem jedes Element einer Zeile und damit einem Highscore-Eintrag entspricht:

```
List<String> highscoreList = new ArrayList<String>();
```

Diese Zeile legt eine leere Liste an. Die zwischen den spitzen Klammern angegebene Klasse `String` verrät Java, dass wir ausschließlich Strings in der Liste zu speichern gedenken. Was hier nach überflüssiger Schreibarbeit aussieht, erspart später explizite Typenumwandlungen und reduziert die Gefahr, dass die zur Laufzeit fehlschlagen.

Wie lässt sich die Antwort des Highscore-Servers nun aber in einzelne Zeilen zerlegen?

Freilich könnten Sie das mit einer Schleife über jedes einzelne Zeichen des Strings `highscores` erledigen. Allerdings wäre es reine Zeitverschwendung, das auf diese Weise zu tun – denn wie für viele oft anfallende Aufgaben gibt es auch für diese eine einfache Lösung, die Android von Haus aus mitbringt: die Hilfsklasse `SimpleStringSplitter`.

Erzeugen Sie eine Instanz dieser Klasse, und geben Sie dem Konstruktor das Zeichen, an dem er den Original-String auftrennen soll:

```
SimpleStringSplitter sss = new SimpleStringSplitter('\n');
```

Die Umschreibung `'\n'` steht dabei für **Newline**, ein spezielles Zeichen, das die einzelnen Zeilen voneinander trennt.

Füttern Sie als Nächstes den Originalstring in den `SimpleStringSplitter`:

```
sss.setString(highscores);
```

Schließlich durchlaufen Sie die gesplitteten Strings mit einer einfachen `while`-Schleife:

```
while(sss.hasNext()) {
   highscoreList.add(sss.next());
}
```

Solange der Splitter Ihnen etwas zu liefern imstande ist, wird seine Methode `hasNext()` `true` zurückgeben und `next()` den betreffenden Teilstring liefern.

Am Ende dieser Operation enthält die `highscoreList` für jeden Highscore eine Zeile.

Bevor Sie jetzt den HTML-String zusammensetzen, sollten Sie das Attribut `highscores` in `highscoresHtml` umbenennen – allerdings nur bei der Deklaration. Das bisherige Attribut `highscores` wird zu einer lokalen Variablen in der Methode `internetHighscores()`, denn außerhalb wird sie nicht mehr benötigt. Den HTML-Code schreiben wir daher jetzt in das Attribut `highscoresHtml`. Zunächst allerdings leeren wir es:

```
highscoresHtml = "";
```

Um das zu tun, muss der Code jedes Element der Liste `highscoresList` einmal anfassen und, versehen mit dem gewünschten HTML-Code, an `highscoresHtml` anhängen.

```
for(String s : highscoreList) {
   highscoresHtml += "<b>"
      + s.replace(",", "</b> <font color='red'>")
      + "</font><br>";
}
```

Der rechte Teil der +=-Operation sieht relativ unübersichtlich aus. Gewöhnen Sie sich daran: Wenn Sie zwei Sprachen mischen (hier Java und HTML), geht fast immer der Überblick verloren. Es gibt zwar Lösungsansätze, die wären für uns im vorliegenden Anwendungsfall aber alle fürchterlich übertrieben.

Sehen Sie sich also an, aus welchen Einzelteilen jede Highscore-Zeile zusammengesetzt wird.

Da ist zunächst das Tag `"<b>"`, das dafür sorgt, dass der nachfolgende Text fett dargestellt wird. Es folgt ein trickreicher `String.replace()`-Aufruf. Sie kennen diese Methode schon: In einer früheren Version hatte sie für uns das Komma durch einen Doppelpunkt ersetzt. Jetzt nimmt allerdings kompletter HTML-Code den Platz des Kommas ein:

```
</b> <font color='red'>
```

Denken Sie daran, dass vor dem Komma der Name des Spielers steht, und dahinter seine Punktzahl. Wenn obiger HTML-Code das Komma ersetzt, folgt auf den Namen das `</b>`, das die Fettschrift beendet, die wir am Anfang der Zeile begonnen haben. Es folgt ein Leerzeichen und anschließend ein `font`-Tag, das die Farbe der Schrift umschaltet, in diesem Fall auf Rot. Zwischen die einfachen Anführungszeichen könnten Sie auch ein paar andere Farbwörter schreiben, die in HTML definiert sind (`green`, `blue`, `black`, `white`, `yellow` ...). Alternativ können Sie hier einen RGB-Farbwert wie `#EECCCC` (Hellrot) hinterlegen, allerdings leider keine der in der *colors.xml*-Ressource definierten Farben.

Zum Schluss wird "`</font><br>`" angefügt, sodass die Punktzahl rot gefärbt wird und danach die Zeile endet.

Der HTML-Code eines jeden Highscore-Eintrags sieht also nach diesem kniffligen Umbau wie folgt aus:

```
<b>Name</b> <font color='red'>Punktzahl</font><br>
```

Bevor Sie jetzt ausprobieren, ob der Code das Gewünschte tut, müssen Sie dem TextView noch HTML anstelle des normalen Textes zuweisen. Die betreffende Zeile in der Methode `ZeigeHighscores()` muss die Hilfsklasse `Html` verwenden, um Ihren Code zu interpretieren:

```
tv.setText( Html.fromHtml( highscoresHtml ) );
```

Die statische Methode `Html.fromHtml()` liefert ein Objekt zurück, das das Interface `Spanned` implementiert, das wiederum von `CharSequence` erbt – dem von `setText()` erwarteten Parametertyp. Glücklicherweise müssen Sie sich um dieses Detail nicht weiter kümmern; Hauptsache, Sie können sich jetzt an der farbigen Highscore-Liste erfreuen.

### HTML mit Bildern

Sie können in HTML-Code Bilder einbinden. Aber woher kommen diese Bilder? Wenn Webseiten auf einem Server oder in einem Verzeichnis auf der Festplatte liegen, dann wird vom `<img>`-Tag aus auf eine separate Grafikdatei verwiesen, etwa so:

```
<img src="muecke.png">
```

Das `src`-Attribut gibt dabei die Quelle (**Source**) an. Wo aber soll Ihr TextView das Bild suchen, wenn Sie ihm ein `<img>`-Tag im HTML-Code unterjubeln? Auf der SD-Karte? In den Ressourcen Ihrer App? Im Internet?

Bedenken Sie, dass das Darstellen eines Bildes in einer HTML-Seite immer eine separate Anfrage an den Webserver zur Folge hat. Ganz ähnlich funktioniert das im HTML-fähigen TextView unter Android: Sie müssen eine Methode zur Verfügung stellen, um die Grafik ausliefern zu können.

Wie üblich dient dazu ein Interface: `ImageGetter`. Dieses Interface ist lokal in der Hilfsklasse `Html` definiert und erfordert die Implementierung einer Methode namens `getDrawable()`. Diese wird immer dann aufgerufen, wenn die grafikfähige Variante der Methode `Html.fromHtml()` auf ein `<img>`-Tag stößt.

Ergänzen Sie einfach mal ein Mückenbild vor dem `<br>`-Tag beim Zusammenbau der Highscore-Liste:

```
highscoresHtml += "<b>"
   + s.replace(",", "</b> <font color='red'>")
   + "</font><img src='muecke'><br>";
```

Verwenden Sie hier einfache Anführungszeichen, weil keine doppelten innerhalb eines Strings auftauchen dürfen (genau genommen können Sie es schon, indem Sie einen Backslash `\` vor das Anführungszeichen setzen, aber der macht den Code furchtbar unübersichtlich).

Lassen Sie nun die `MueckenfangActivity` das Interface `Html.ImageGetter` implementieren:

```
public class MueckenfangActivity extends Activity implements
       OnClickListener, Html.ImageGetter
```

Implementieren Sie die vom Interface geforderte Methode `getDrawable()`:

```
@Override
public Drawable getDrawable(String name) {
   int id = getResources().getIdentifier(name, "drawable", this.
            getPackageName());
   Drawable d = getResources().getDrawable(id);
   d.setBounds(0, 0, d.getIntrinsicWidth(), d.getIntrinsicHeight());
   return d;
}
```

Was tut diese Methode?

Sie kennen bereits die Methoden `getResources()` und `getIdentifier()`, die Ihnen zu einem bestimmten Namen die passende ID heraussuchen. In diesem Fall ist eine Ressource des Typs `drawable` gefragt, die anschließend anhand ihrer ID geladen wird.

Damit die Grafik in HTML eingebettet werden kann, müssen Sie noch einen kleinen Trick anwenden, indem Sie das Begrenzungsrechteck der Grafik auf seine eigene Breite und Höhe setzen – ansonsten funktioniert's nicht. Schließlich wird `drawable` mit `return d` zurückgegeben.

Als Letztes müssen Sie die passende Variante der `fromHtml()`-Methode in `ZeigeHighscores` aufrufen:

```
tv.setText(Html.fromHtml( highscoresHtml, MueckenfangActivity.this,
null ) );
```

Der zweite Parameter ist dabei die Referenz auf die Implementierung von `Html.ImageGetter`, in diesem Fall unsere `MueckenfangActivity` selbst. Da sich der Aufruf in einer anderen Klasse (namentlich `ZeigeHighscores`) befindet, müssen Sie das Schlüsselwort `this` mit dem Namen der äußeren Klasse verzieren. Sonst würde `this` nämlich `ZeigeHighscores` meinen, und die implementiert natürlich nicht den `ImageGetter`.

Wenn Sie jetzt Ihre App starten, werden Sie feststellen, dass die Mücke viel zu groß dargestellt wird. Natürlich: Ihre Maße sind nun einmal 50 x 50 Pixel, und Sie haben nichts anderes angegeben.

Leider erlaubt es das `<img>`-Tag unter Android nicht wie im »echten« HTML, Größen direkt in Form von Attributen anzugeben. Es gibt mehrere Wege aus diesem Schlamassel. Sie können einfach eine auf 30 x 30 Pixel verkleinerte Version der Mücke unter einem anderen Namen im *drawable*-Verzeichnis speichern und diese im `<img>`-Tag referenzieren. Oder Sie ändern in der `setBounds()`-Methode die Breite und Höhe direkt, indem Sie sie fest auf 30 Pixel setzen:

```
d.setBounds(0, 0, 30, 30);
```

Wenn Sie's richtig flexibel haben möchten, hängen Sie an den Namen der Grafik im `<img>`-Tag eine Zahl (getrennt etwa durch einen Strich an) und bauen die Methode `getDrawable()` so um, dass sie diese Zahl erkennt und die Grafik entsprechend skaliert.

Oder Sie überlegen sich, dass HTML vielleicht doch nicht so geeignet ist, Ihre mit Bildern gespickte Bestenliste anzuzeigen. In dem Fall schauen Sie sich den nächsten Abschnitt an, in dem ich Ihnen eine weitere, oft verwendete Methode der Listendarstellung erkläre.

## 7.3 Listen mit Adaptern

Für lange Listen mit komplexem Inhalt ist ein einziger, scrollfähiger TextView mit HTML darin nicht besonders gut geeignet. Vor allem ältere Smartphones werden vom erhöhten Speicherbedarf gern mal überfordert. Denken Sie an eine Liste mit 1.000 statt zehn Rekordhaltern, und Sie ahnen vielleicht, was ich meine.

### ListViews

Es gibt eine recht mächtige Alternative, nämlich **ListViews**. Das ist nichts anderes als ein scrollbarer Bereich, der eine gewisse Anzahl gleichartiger anderer Views enthält (die Listeneinträge). Der Clou an der Sache ist, dass diese Views wiederverwendet werden, wenn der Benutzer durch die Liste scrollt, und lediglich mit den jeweils passenden Inhalten versehen werden. Um Letzteres kümmert sich ein sogenannter **Adapter.** Das ist ein Objekt, das jedem logischen Eintrag einer Liste (etwa einem Highscore) ein Element des Listviews zuordnet.

Auch Android selbst nutzt ListViews exzessiv, sei es in den Einstellungen, im Market oder in anderen Apps. Es kann also nicht allzu verkehrt sein, zu verstehen, wie diese praktische Einrichtung funktioniert.

Um den Überblick zu behalten, lassen Sie uns einen ListView für eine »große« Highscore-Liste verwenden. Dazu basteln wir ein neues Layout, das wir aktivieren, wenn der Spieler die Top 10 antippt, die wir weiterhin auf der Startseite des Spiels anzeigen.

Natürlich wäre es möglich, für die Bestenliste eine komplett neue Activity zu erschaffen. Aber der gesamte Code zum Holen der Highscores befindet sich in der `MueckenfangActivity`. Dies ist eine der klassischen Situationen, in denen der Entwickler abwägen muss, welcher Lösungsweg bei gleichem Ergebnis leichter zu implementieren ist. Eine separate Activity hätte zwar einige Vorteile (so könnte man beispielsweise dem Android-Startmenü ein Icon hinzufügen, das direkt dorthin führt) – aber nichts, was wir unbedingt bräuchten. Die Auslagerung der ganzen Internet-Funktionalität in eine andere Klasse (beispielsweise eine gemeinsame Basisklasse für beide Activities) wäre eine Menge Arbeit. Also wählen wir den schnellsten Weg, von dem wir wissen, dass und wie er funktioniert!

Erzeugen Sie mit dem Wizard eine neue Android-XML-Datei namens *toplist.xml*. Verwenden Sie ein LinearLayout (mit Hintergrundgrafik) als Grundgerüst, und packen Sie an dessen oberes Ende einen TextView mit großer Schrift, dem Sie den schon vorhandenen String »Highscores« zuordnen.

Füllen Sie den Rest mit einem ListView, den Sie in der Palette in der Rubrik Composite finden. Setzen Sie Breite und Höhe des ListViews auf `match_parent`, und geben Sie ihm die ID `list` (Abbildung 7.9).

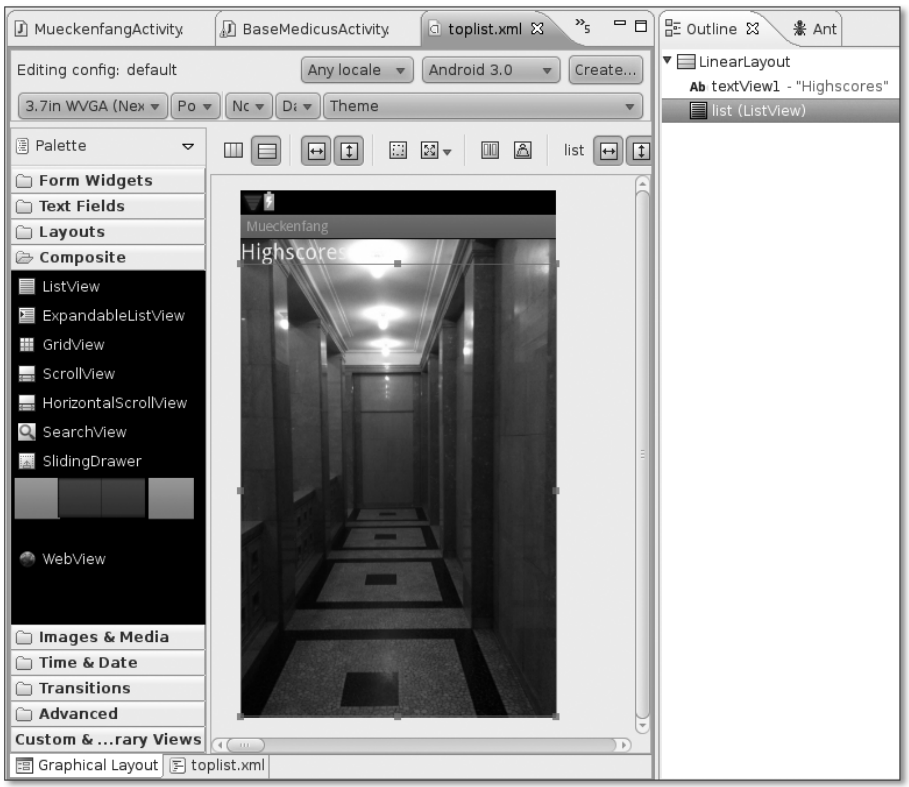



**Abbildung 7.9** Die Highscore-Liste besteht aus einer Überschrift, einem ListView, der den Rest füllt, und der obligatorischen Hintergrundgrafik.

Sie benötigen ein weiteres Layout, das einen einzelnen Listeneintrag repräsentiert. Stellen Sie sich vor, dass der ListView später dieses Layout für jeden Eintrag klont und der Reihe nach in seinem Bildschirmbereich platziert.

Nennen Sie dieses zweite Layout *toplist_element.xml*. Es basiert auf einem FrameLayout und erhält drei TextViews als Kindelemente. Der erste TextView erhält eine Breite von `50dp` und die ID `platz`. Dort werden wir die Platznummer anzeigen, beginnend bei 1. Der zweite TextView erhält die ID `name`. Verpassen Sie ihm einen linken Rand von `60dp` (`padding_Left`), damit er nicht mit dem Platz überlappt. Schließlich geben Sie dem dritten TextView die ID `punkte` und eine `layout_gravity right`. Damit erscheint die Punktzahl rechtsbündig. Gestalten Sie nun Schriftgrößen, -farben und Stil nach Geschmack, und speichern Sie die Datei (Abbildung 7.10).

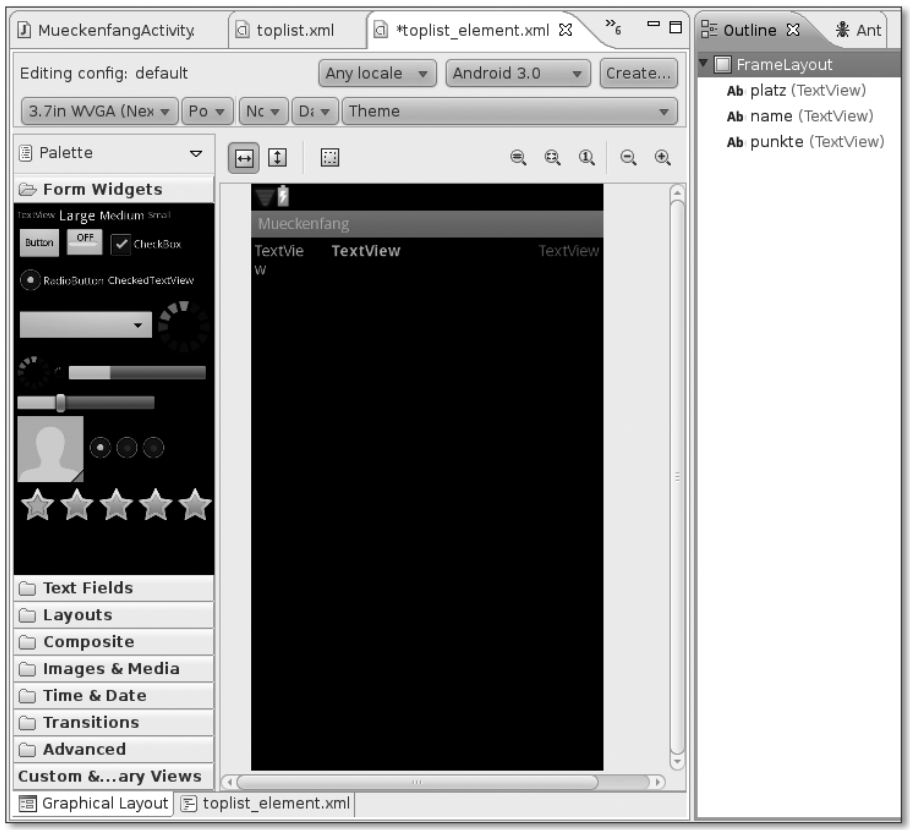



**Abbildung 7.10** Das Layout für die Listeneinträge erhält keine Hintergrundgrafik.

### Tipp: ListViews auf Hintergrundfarbe hinweisen

Wenn Sie durch Listen scrollen, werden Sie bemerken, dass einige hübsche visuelle Effekte ohne Ihr Zutun zur Anwendung kommen. Einträge werden am oberen und unteren Rand sanft ausgeblendet, und das Ganze funktioniert auch ganz wunderbar, solange der Bildschirmhintergrund schwarz ist.

Denn ListViews gehen stur davon aus, dass Sie keinen anderen als einen schwarzen Hintergrund verwenden, und setzen daher schwarze Farbe zum Ausblenden von Einträgen ein.

Das sieht zwangsläufig hässlich aus, wenn der Hintergrund mal farbig und mal weiß ist. Sie können den ListView aber darauf hinweisen, welche Farbe sich hinter ihm verbirgt – selbst herausfinden kann er es leider nicht.

Setzen Sie im ListView das Attribut `cacheColorHint` auf `#00000000`, also komplett transparent, um unabhängig vom Hintergrund eine akzeptable Darstellung zu erhalten.

Zunächst versehen Sie den bisherigen Highscore-TextView in der Methode `onCreate()` mit einem `onClickListener`. Bei der Gelegenheit führen wir gleich eine Referenz auf den View als Attribut ein:

```
highscores = (TextView) findViewById(R.id.highscores);
highscores.setOnClickListener(this);
```

Verwenden Sie dieses Attribut jetzt auch in `ZeigeHighscores`, um redundanten Code zu vermeiden.

Fügen Sie die nötige Abfrage in die `onClick()`-Methode ein:

```
} else if(v.getId() == R.id.highscores) {
   setContentView(R.layout.toplist);
   Thread t = new Thread(new HoleHighscores(100));
   t.start();
}
```

Mit `setContentView()` schaltet dieser Code direkt zu dem neuen Layout *toplist.xml* um und startet den Hintergrund-Thread mit dem Runnable `HoleHighscores`. Wir recyceln also dieses Runnable, was grundsätzlich eine gute Idee ist, und machen es gleichzeitig flexibel, indem wir die gewünschte Maximalanzahl der Scores als Parameter an den Konstruktor von `HoleHighscores` übergeben.

Implementieren Sie diesen Konstruktor wie folgt:

```
private class HoleHighscores implements Runnable {
   private int max;
   public HoleHighscores(int m) {
      max = m;
   }
   @Override
   public void run() {
      internetHighscores("",0,max);
   }
}
```

Der Konstruktor überträgt den ihm übergebenen Parameter an das private Attribut `max`, das wiederum in der Methode `run()` an `internetHighscores()` übergeben wird. Diese Methode müssen Sie jetzt erweitern, damit der Wert auch dort ankommt. Der zugehörige URL-Parameter heißt ebenfalls `max`, und glücklicherweise waren wir vorausschauend genug, ihn bereits in den Highscore-Server einzubauen.

```
private void internetHighscores(String nickname, int score, int max)
 {
```

```
...
HttpGet request = new HttpGet(HIGHSCORE_SERVER_BASE_URL
+ "?game=" + HIGHSCORESERVER_GAME_ID
+ "&nickname=" + URLEncoder.encode(nickname)
+ "&score=" + Integer.toString(score)
+ "&max=" + max);
```

Die Methode `internetHighscores()` ist bisher nur dazu in der Lage, die Antwort des Highscore-Servers in das Attribut `highscoresHtml` zu schreiben. Dort gehört sie aber nur hin, wenn der Hauptbildschirm im Standardmodus ist. Zeigt er gerade das Bestenliste-Layout an, muss die Liste der Rekorde an den ListView `list` überstellt werden. Und dazu dient ein Adapter.

### ArrayAdapter

Stellen Sie sich den ArrayAdapter vor wie den Vermittler zwischen dem sichtbaren ListView und den Inhalten, die er anzeigt. Im einfachsten Fall ist das eine Liste von Strings: Der ArrayAdapter erhält eine List<String> und stellt dem ListView immer die Strings zur Verfügung, die gerade im scrollbaren Bereich zu sehen sind (Abbildung 7.11).

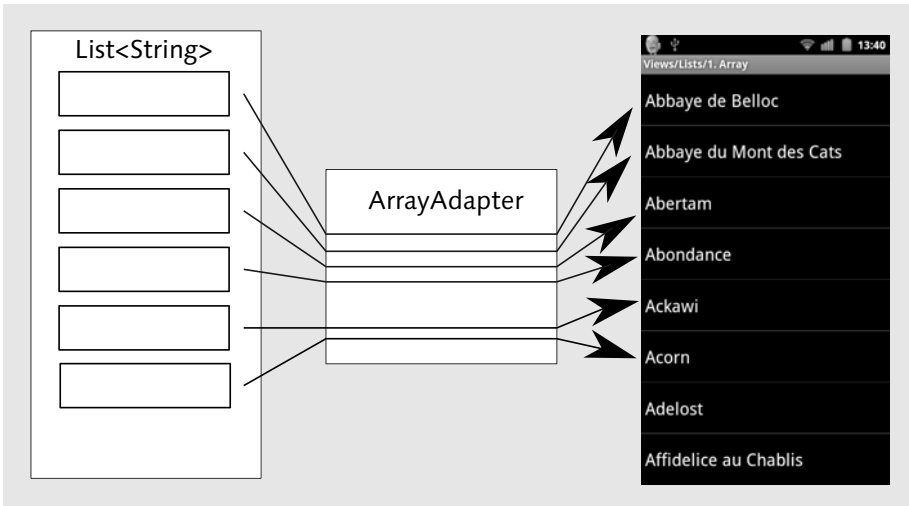



**Abbildung 7.11** Der ArrayAdapter vermittelt die Elemente einer String-Liste an einen ListView.

Wenn sich der Inhalt des Arrays ändert, müssen Sie übrigens nur den Adapter benachrichtigen, und der sorgt dafür, dass der ListView aktualisiert wird.

Einen solchen simplen ArrayAdapter können Sie nicht nur in ListViews verwenden, sondern beispielsweise auch in Auswahllisten. Um das zu zeigen, lassen Sie

uns dem Spieler ermöglichen, einen Schwierigkeitsgrad zu wählen: leicht, mittel, schwer.

Fügen Sie einen View vom Typ Spinner im Layout *main.xml* zwischen Mücke und START-Button ein, und geben Sie ihm die ID `schwierigkeitsgrad`. Setzen Sie eine Breite von `150dp`, damit der View nicht zu viel Platz einnimmt.

Verpassen Sie der Klasse `MueckenfangActivity` ein neues Attribut, um die Referenz auf den neuen Spinner zu speichern:

```
private Spinner schwierigkeitsgrad;
```

An gleicher Stelle fügen Sie den ArrayAdapter hinzu:

```
private ArrayAdapter<String> schwierigkeitsgradAdapter;
```

Die Klasse `String` in den spitzen Klammern legt fest, dass der Adapter mit einem Array aus Strings arbeiten wird.

Füllen Sie die View-Referenz in `onCreate()`:

```
schwierigkeitsgrad = (Spinner) findViewById(R.id.schwierigkeitsgrad);
```

Erzeugen Sie als Nächstes den Adapter. Es gibt mehrere Konstruktoren, davon verwenden wir den einfachsten:

```
schwierigkeitsgradAdapter = new ArrayAdapter<String>(this, android.
        R.layout.simple_spinner_
item, new String[] {"leicht","mittel","schwer"} );
```

Der erste Parameter ist ein `Context`-Objekt, mit dem der Adapter arbeiten soll. Wird ein `Context` erwartet, übergeben Sie einfach immer die aktuelle Activity, in diesem Fall also `this`. Parameter Nummer 2 ist die ID des Layouts, das für jeden Eintrag verwendet wird. Dazu liefert Android bereits eine Version mit, die Sie mit `android.R.layout.simple_spinner_item` referenzieren, anstatt eine eigene zu bauen. Der letzte Parameter ist schließlich das String-Array mit den gewünschten Auswahloptionen.

Wenn Sie später den Spinner antippen, öffnet sich eine Liste mit den wählbaren Einträgen. Dafür gibt es ein eigenes Layout, das Sie dem Adapter mitteilen:

```
schwierigkeitsgradAdapter.setDropDownViewResource(android.R.layout.
        simple_spinner_dropdown_item);
```

Schließlich müssen Sie nur noch den Spinner mit seinem Adapter verheiraten:

```
schwierigkeitsgrad.setAdapter(schwierigkeitsgradAdapter);
```

Natürlich soll sich die Auswahl des Spielers auch irgendwie auswirken.

Dazu müssen Sie zum Zeitpunkt des Spielstarts prüfen, welcher Eintrag im View ausgewählt ist. Dann können Sie der `GameActivity` diesen Wert als Schwierigkeitsgrad mitgeben.

Bearbeiten Sie also den ersten `if`-Zweig in der Methode `onClick()`. Ermitteln Sie zunächst, welcher Schwierigkeitsgrad ausgewählt ist:

```
int s = schwierigkeitsgrad.getSelectedItemPosition();
```

Die aktuell ausgewählte Position beginnt immer bei 0. Im Fall von »leicht« wird die Variable `s` also den Wert 0 erhalten, bei »schwer« 2.

Wie aber können Sie diese Zahl jetzt der `GameActivity` übergeben?

Bisher wird die an dieser Stelle wie folgt gestartet:

```
startActivityForResult(new Intent(this,GameActivity.class),1);
```

Ich werde Ihnen jetzt zeigen, wie Sie einem Intent zusätzliche Informationen anhängen. Der Übersicht halber müssen Sie dazu dem Intent eine eigene lokale Variable gönnen. Am einfachsten kriegen Sie das hin, wenn Sie in Eclipse `new Intent(this,GameActivity.class)` markieren und per Refactoring eine Variable einführen. Drücken Sie dazu Alt + ⇧ + L (oder im Kontextmenü: Refactor • Extract Local Variable). Eclipse fragt Sie nach dem Namen der neuen Variablen, nehmen Sie dafür einfach `i` oder `intent`.

Das Resultat des Refactorings sieht dann wie folgt aus:

```
Intent i = new Intent(this,GameActivity.class);
startActivityForResult(i,1);
```

Jetzt können Sie zwischen den beiden Zeilen eine weitere einfügen und ein sogenanntes **Extra** an den Intent hängen:

```
i.putExtra("schwierigkeitsgrad", s);
```

Extras bestehen immer aus einen Namen und einem Wert, in diesem Fall einem `int`. Sie können auch Strings oder beinahe beliebige andere Objekte anhängen. Auf diese Weise können Sie alle relevanten Daten an eine andere Activity übergeben.

Nun muss die `GameActivity` diesen Wert wieder aus dem Intent extrahieren, am besten in `onCreate()`:

```
schwierigkeitsgrad = getIntent().getIntExtra("schwierigkeitsgrad", 0);
```

Sie sehen, dass Sie abhängig vom Typ des mitgegebenen Extras die richtige Methode aufrufen und darüber hinaus den verwendeten Namen als ersten Parameter übergeben müssen.

Die 0 als zweiter Parameter dient als Standardwert, falls aus irgendeinem Grund das Extra fehlen sollte.

Wie der Schwierigkeitsgrad sich aufs Spiel auswirkt, bleibt Ihnen überlassen. Beachten Sie jedoch, dass es für die schwerere Variante mehr Punkte geben muss, damit ein Obermückenfänger nicht auf der schweren Stufe viel weniger Punkte erspielt als ein Anfänger.

Das ist eine Frage des Balancings, auf das wir hier nicht ausführlich eingehen können. Versuchen Sie's für den Anfang mit zwei simplen Eingriffen:

```
muecken = runde * (20 + schwierigkeitsgrad*10);
```

Dies ändert die Anzahl der Mücken in `starteRunde()` so, dass ein mutiger Jäger, der die schwierige Version wählt, praktisch in Runde 2 startet und dann auch noch jede zweite überspringt.

Erhöhen Sie auf angemessene Weise die Punktzahl pro getroffener Mücke in `onClick()`:

```
punkte += 100 + schwierigkeitsgrad*100;
```

Der Experte erhält also gleich dreimal so viele Punkte für jede getroffene Mücke, nämlich 300. Im mittleren Schwierigkeitsgrad ist jede Mücke immerhin 200 Punkte wert. Nur wenn `schwierigkeitsgrad` gleich 0 ist, bleibt's bei den 100 Punkten.

Sie sehen, dass die sinnvolle Verarbeitung eines Spinner-Views fast mehr Aufwand ist als dessen Darstellung selbst. Sobald allerdings etwas Komplizierteres erscheinen soll als simple Strings, ändert sich die Lage.

### Eigene Adapter

Die Highscore-Liste soll nicht nur je einen String pro Zeile anzeigen, sondern eine Platznummer, einen Namen und eine Punktzahl. Dafür haben Sie bereits das Layout *toplist_element.xml* gebastelt. Lassen Sie uns nun eine Adapter-Klasse bauen, die für jeden Eintrag in der Highscore-Liste das Einzelelement-Layout untereinander darstellt.

Erzeugen Sie mit dem Wizard (Strg + N) eine neue Klasse namens `ToplistAdapter`, die von `ArrayAdapter<String>` erbt:

```
package de.androidnewcomer.mueckenfang;
import android.widget.ArrayAdapter;
public class ToplistAdapter extends ArrayAdapter<String> {
}
```

Der Klassenname `String` in den spitzen Klammern gibt Java bekannt, dass unser Adapter als Datenbasis eine `List<String>` verwenden wird. Eine solche erhalten wir ja vom Highscore-Server zurück, und es wird Aufgabe des Adapters sein, aus einem einzelnen String jeweils Name und Highscore zu extrahieren.

Wie aber kommt der Adapter an die Liste der Strings?

Am einfachsten übergeben Sie die Liste dem Adapter als Konstruktor-Argument. Ohnehin erwartet Eclipse von Ihnen, dass Sie mindestens einen der Konstruktoren aus der Basisklasse `ArrayAdapter` überschreiben, denn der Standard-Konstruktor ist beim Adapter sinnlos und daher nicht verfügbar. Implementieren Sie also den folgenden Konstruktor:

```
public ToplistAdapter(Context ctx, int textViewResourceId,
      List<String> list) {
   super(ctx, textViewResourceId, list);
   context = ctx;
   toplist = list;
}
```

Abgesehen vom obligatorischen Aufruf des geerbten Konstruktors, erkennen Sie, dass sich der Adapter die Parameter für den aktuellen Context (das wird die `MueckenfangActivity` sein) und natürlich die eigentliche Topliste merkt. Führen Sie dazu die nötigen privaten Attribute ein:

```
private Context context;
private List<String> toplist;
```

In seiner jetzigen Form tut der Adapter noch nicht viel, insbesondere liefert er keine Views zurück, die der ListView anzeigen könnte. Ändern Sie das, indem Sie die Methode `getView()` überschreiben:

```
@Override
public View getView(int position, View convertView, ViewGroup parent)
 {
}
```

Diese Methode wird später ohne Ihr Zutun vom ListView verwendet, wenn er sich die Views holen möchte, die an der Stelle `position` erscheinen sollen. Dabei startet `position` immer bei 0 für den obersten Listeneintrag.

Jetzt müssen Sie die nötigen Views natürlich erzeugen. Dazu beschaffen Sie sich als Erstes eine Referenz auf den `LayoutInflater`. Das ist ein Dienst, der aus Layout-Dateien fertige View-Hierarchien erzeugt:

```
LayoutInflater inflater = (LayoutInflater) context.getSystemService(
Context.LAYOUT_INFLATER_SERVICE);
```

Als Nächstes beauftragen Sie den `LayoutInflater`, Ihre XML-Datei in Views zu verwandeln:

```
View element = inflater.inflate(R.layout.toplist_element, null);
```

Das Objekt `element` entspricht jetzt dem FrameLayout an der Wurzel Ihres Layouts *toplist_element.xml*.

Die einzelnen TextViews innerhalb des FrameLayouts beschaffen Sie sich wie üblich mit der Methode `findViewById()`, in diesem Fall aufgerufen auf `element`.

```
TextView platz = (TextView) element.findViewById(R.id.platz);
TextView name  = (TextView) element.findViewById(R.id.name);
TextView punkte= (TextView) element.findViewById(R.id.punkte);
```

Was gilt es nun dort einzutragen? Der Reihe nach: die Platznummer, dann den Namen und schließlich die Punktzahl.

Die Platznummer ist leicht – das ist einfach `position` plus 1:

```
platz.setText(Integer.toString(position+1)+".");
```

Wir hängen noch `"."` an, um die Platzierung als solche zu kennzeichnen.

Name und Punktzahl müssen Sie jetzt aus dem richtigen `toplist`-Eintrag an der Stelle `position` extrahieren. Verwenden Sie einen `SimpleStringSplitter`, um die durch Komma getrennten Bereiche Name und Punktzahl einzeln zu erhalten und in ihre TextViews zu schreiben:

```
SimpleStringSplitter sss = new SimpleStringSplitter(',');
sss.setString(toplist.get(position));
name.setText(sss.next());
punkte.setText(sss.next());
```

Als Letztes geben Sie den fertig gefüllten View an die aufrufende Methode zurück:

```
return element;
```

Sie müssen noch eine weitere Methode des Adapters überschreiben, die der ListView später verwenden wird. Verständlicherweise möchte der ListView wissen, wie viele Einträge Ihre Liste besitzt:

```
@Override
public int getCount() {
   return toplist.size();
}
```

Jetzt müssen Sie nur noch alles zusammenbauen.

Ändern Sie die Methode `internetHighscores()` so, dass sie mit den beiden Anzeigemodi klarkommt:

```
// standard layout
if(findViewById(R.id.highscores) != null) {
   highscoresHtml = "";
   for(String s : highscoreList) {
      highscoresHtml += "<b>" + s.replace(",",
      "</b> <font color='red'>") + "</font><img src='muecke'><br>";
   }
   runOnUiThread(new ZeigeHighscores());
}
```

Das war die Version für die HTML-Liste. Sie wird aktiv, wenn der HTML-fähige ListView `R.id.highscores` im Layout vorhanden ist, also der betreffende `findViewById()`-Aufruf ein Ergebnis zurückgibt.

Im Fall der großen Highscore-Liste erzeugen Sie den Adapter und starten ein anderes Runnable:

```
// toplist layout
if(findViewById(R.id.list)!=null) {
   list = (ListView) findViewById(R.id.list);
   adapter = new ToplistAdapter(this, 0, highscoreList);
   runOnUiThread(new ZeigeTopliste());
}
```

Deklarieren Sie die nötigen Attribute:

```
private ToplistAdapter adapter;
private ListView list;
```

Das noch fehlende Runnable weist dem ListView schließlich seinen Adapter zu und teilt selbigem mit, dass neue Daten zur Verfügung stehen:

```
private class ZeigeTopliste implements Runnable {
   @Override
   public void run() {
      list.setAdapter(adapter);
      adapter.notifyDataSetChanged();
   }
}
```

Wenn Sie die App jetzt ausprobieren, werden Sie zwar erfreut feststellen, dass die komplette Highscore-Liste funktioniert, allerdings arbeitet nun der ZURÜCK-Button nicht mehr wie erwartet: Er führt nicht zum Startbildschirm, sondern beendet die App. Das ist logisch, denn »Zurück« meint immer Activities, und wir haben ja bloß die eine.

Glücklicherweise lässt sich die Funktion des ZURÜCK-Buttons sehr leicht nachbauen. Überschreiben Sie einfach die Methode `onKeyDown()`, und starten Sie das ganze Spiel neu, wenn der Benutzer den ZURÜCK-Button gedrückt hat:

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
   if(event.getKeyCode() == KeyEvent.KEYCODE_BACK) {
      if(findViewById(R.id.list)!=null) {
         startActivity(new Intent(this,MueckenfangActivity.class));
         finish();
         return true;
      }
   }
   return super.onKeyDown(keyCode, event);
}
```

### Recyceln von Views

Leider ist das Auspacken eines Layouts eine »teure« Angelegenheit, das heißt: Es dauert eine Weile. Selbst wenn wir hier von Millisekunden reden: Halten Sie sich vor Augen, dass der Adapter für jeden Eintrag in der Liste immer denselben stupiden Vorgang ausführen muss, während der Benutzer durch den ListView scrollt. Spätestens für komplexere Listeinträge wird das schnell unwirtschaftlich.

Deshalb übergibt der ListView der Methode `getView()` des Adapters den »alten« View als Parameter `convertView`. Falls dieser gesetzt ist, können Sie also ihn verwenden, statt einen neuen vom `LayoutInflater` anzufordern. Umgekehrt ausgedrückt: Sie müssen nur dann den Inflater bemühen, wenn der Parameter `convertView` leer ist:

```
public View getView(int position, View convertView, ViewGroup parent
) {
   View element=convertView;
   if(element==null) {
      LayoutInflater inflater = (LayoutInflater) context.
            getSystemService(Context.LAYOUT_INFLATER_SERVICE);
      element = inflater.inflate(R.layout.toplist_element, null);
   }
   ...
```

Verwenden Sie immer dieses Entwurfsmuster, um Speicher und Rechenzeit zu sparen. Es ist weder kompliziert noch fehleranfällig.

Ein versöhnlicher Abschluss für einen Abschnitt, der nicht ganz unkompliziert war – aber ohne Listen kommen die wenigsten Apps aus, und wenn Sie wissen, wie die Adapter funktionieren, sind sie schnell gebaut.

In diesem Kapitel haben Sie eine Menge Technologien kennengelernt und vielleicht sogar Ihre erste Serveranwendung für die Google App Engine geschrieben. Sie haben gesehen, dass die App-Entwicklung ein weites Feld ist, das sich oftmals auf Nebenschauplätzen abspielt. Allerdings können Sie ja mal die bisher produzierten Programmzeilen zählen (es sind etwas über 500). Wenn Sie jetzt subtrahieren, was Ihnen Eclipse dank automatischer Codeerzeugung abgenommen hat, werden Sie feststellen, dass Sie wirklich nicht viel programmieren müssen, um zu interessanten Ergebnissen zu kommen – das nötige *Gewusst-wie* genügt.

*»Ich sehe was, was du nicht siehst, und das ist … infrarot.«*
*(unfair spielende Digicam)*

# 8 Kamera und Augmented Reality

Digicams sind bloß für Schnappschüsse gut, und die klitzekleinen Linsen, die in Smartphones üblicherweise verbaut sind, beschränken Aufnahmebedingungen und Bildqualität? Entscheidend ist, was man mit dem aufgenommenen Bild anstellt. Moderne Apps erkennen beispielsweise Barcodes auf Produktverpackungen, andere können Texte lesen und sogar übersetzen – alles mithilfe mächtiger Software im Hintergrund, versteht sich.

Der Fantasie sind keine Grenzen gesetzt, was den Einsatz der in jedem Android-Gerät eingebauten Kamera betrifft. Freilich kostet aufwendige Bildmanipulation mehr Rechenpower, als ein Telefon liefern kann, aber selbst mit einfachen Mitteln lässt sich Erstaunliches bewirken.

Lassen Sie uns also die Digicam des Smartphones verwenden, um den Mückenfang noch spannender zu gestalten: Zunächst ersetzen wir den Bildschirmhintergrund durch einen Blick durch die Kameralinse. Dadurch wird der Bildschirm des Handys ein Fenster in die Wirklichkeit, in die nicht-reale Elemente (die Mücken) eingeblendet werden – der erste Schritt zur **Augmented Reality**.

Der Übersicht halber habe ich für dieses Kapitel eine separate Kopie des Projektverzeichnisses auf die Buch-DVD gepackt. Sie finden es unter dem Namen *Mueckenfang3*.

## 8.1 Die Kamera verwenden

Vielleicht ist Ihnen aufgefallen, dass es in der Palette des Layout-Editors keinen View gibt, der so ausschaut, als könne er mit der Kamera zu tun haben. Es gibt lediglich den VideoView, der eignet sich aber nur zum Abspielen von Filmen, nicht für die eingebaute Kamera.

Uns bleibt daher nichts anderes übrig, als einen eigenen View zu bauen, dessen Inhalt nichts anderes ist als das Vorschaubild der Kamera. Der beste Ansatz ist es, eine eigene Klasse von einer anderen abzuleiten.

### Der CameraView

Die Basisklasse für die eigene Kameravorschau ist der `SurfaceView` – das ist ein View, dessen sichtbarer Inhalt komplett der Kontrolle des Programmierers unterliegt. Praktisch jeder Inhalt ist möglich – von simplen Zeichnungen über 3-D-Action bis hin zum Vorschaubild der Kamera.

Erstellen Sie also als Erstes eine neue Klasse namens `CameraView`, die von `SurfaceView` erbt:

```
public class CameraView extends SurfaceView {
}
```

Um auf die Zeichenfläche zuzugreifen, benötigen wir ein `SurfaceHolder`-Objekt, das mit unserem View über ein Interface kommuniziert. Dieses Interface heißt `SurfaceHolder.Callback`, und die Klasse `CameraView` muss es implementieren. Erzeugen Sie außerdem ein Attribut für das eigentliche Kamera-Objekt:

```
public class CameraView extends SurfaceView implements SurfaceHolder
.Callback {
   SurfaceHolder surfaceHolder;
   Camera camera;
   ...
}
```

Lassen Sie sich von Eclipse die Methodenrümpfe hinzufügen, die nötig sind, um dem `Callback`-Interface zu genügen, indem Sie die automatische Fehlerkorrektur die Arbeit erledigen lassen.

Bevor Sie die die neuen Methoden mit Code füllen, müssen Sie verstehen, wie die drei Komponenten miteinander interagieren (Abbildung 8.1).

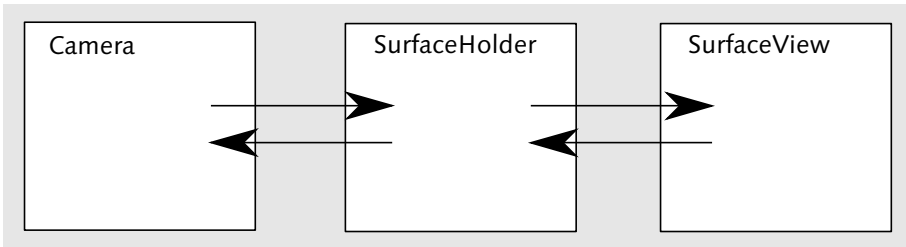



**Abbildung 8.1** Die Kamera projiziert ihr Vorschaubild über den »SurfaceHolder« in den »SurfaceView«.

Der `SurfaceView` ist dafür verantwortlich, die Kamera und den `SurfaceHolder` miteinander zu verknüpfen, und zwar abhängig von den Ereignissen, die das `Callback`-Interface meldet.

Zunächst aber müssen Sie dafür sorgen, dass der `SurfaceHolder` überhaupt existiert – schreiben Sie also erst einmal einen Konstruktor:

```
public CameraView(Context context, AttributeSet attrs) {
   super(context, attrs);
   surfaceHolder = getHolder();
   surfaceHolder.addCallback(this);
   surfaceHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
}
```

Nach dem Aufruf des Elternkonstruktors wird das Attribut für den `SurfaceHolder` gefüllt, dann das Objekt mit dem Callback verbunden. Die letzte Zeile setzt den Typ des `SurfaceHolders` auf denjenigen, den die Kamera für ihre Vorschau benötigt.

Leider müssen Sie noch einen zweiten, identischen Konstruktor schreiben, der allerdings nur den `Context`-Parameter erhält:

```
public CameraView(Context context) {
   super(context);
   surfaceHolder = getHolder();
   surfaceHolder.addCallback(this);
   surfaceHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
}
```

Android verwendet je nach Bedarf den einen oder den anderen Konstruktor, daher müssen beide vorhanden sein. Achten Sie darauf, dass die zweite Version den passenden Elternkonstruktor ohne `attrs` verwendet.

Als Nächstes kümmern wir uns um die drei Methoden, die zum `Callback`-Interface gehören. Da wäre zunächst einmal die wichtigste Methode, nämlich `surfaceCreated()`. Es wird Sie nicht überraschen, dass diese Methode von Android aufgerufen wird, sobald die sichtbare Surface erzeugt wurde und bereit ist, irgendetwas anzuzeigen. Der perfekte Zeitpunkt, um das `Camera`-Objekt zu erzeugen. Allerdings dient dazu diesmal kein Konstruktor, sondern eine statische Methode der Klasse `Camera` namens `open()`:

```
public void surfaceCreated(SurfaceHolder holder) {
   camera = Camera.open();
```

Verknüpfen Sie als Nächstes die Kameravorschau mit dem `SurfaceHolder`:

```
   camera.setPreviewDisplay(holder);
```

Einer kleinen Zusatzmaßnahme bedarf es, um die Kamera auf den Hochkant-Modus umzustellen, den unser Mückenfang verwendet:

```
   camera.setDisplayOrientation(90);
```

Eclipse wird an dieser Stelle versuchen, Ihnen verständlich zu machen, dass die `setPreviewDisplay()`-Methode eine IOException werfen könnte. Also müssen Sie den Aufruf in eine ordentliche `try-catch`-Struktur einbetten, da es nicht infrage kommt, die Exception weiter nach außen zu werfen. Da eine IOException anzeigt, dass etwas mit der Kamera nicht stimmt, geben Sie die sicherheitshalber im Fehlerfall mit der Methode `release()` frei. Die ganze Methode sieht dann wie folgt aus:

```
public void surfaceCreated(SurfaceHolder holder) {
   camera = Camera.open();
   camera.setDisplayOrientation(90);
   try {
      camera.setPreviewDisplay(holder);
   } catch (IOException exception) {
      camera.release();
      camera = null;
   }
}
```

Als ordentliche Programmierer räumen wir hinter uns auf. Implementieren Sie daher die zweite von drei Methoden des `Callback`-Interfaces wie folgt:

```
public void surfaceDestroyed(SurfaceHolder holder) {
   camera.stopPreview();
   camera.release();
   camera = null;
}
```

Abgesehen vom expliziten Beenden des Vorschaumodus, sehen Sie hier dieselben Zeilen wie im `catch`-Block der vorangegangenen Methode – also nichts Neues.

Schließlich müssen Sie noch `surfaceChanged()` implementieren. Diese Methode wird immer dann von Android aufgerufen, wenn sich die Größe des sichtbaren Bereiches des `SurfaceViews` ändert – mindestens aber einmal.

Verpassen Sie in dieser Methode der Kameravorschau die richtige Größe (die der Methode freundlicherweise übergeben wird). Dazu müssen Sie die Parameter der Kamera auslesen, ändern und wieder speichern:

```
Camera.Parameters parameters = camera.getParameters();
parameters.setPreviewSize(w, h);
camera.setParameters(parameters);
```

Leider hat meine Erfahrung gezeigt, dass manche Smartphone-Kameras mit diesem Aufruf nicht klarkommen. Sie lassen die Änderung der Größe einfach nicht zu und quittieren den letzten Aufruf mit einer RuntimeException. Glücklicherweise funktioniert in solchen Fällen fast immer die Standard-Einstellung. Deshalb dürfen wir eine Exception an dieser Stelle ausnahmsweise einfach fangen und ignorieren:

```
try {
   Camera.Parameters parameters = camera.getParameters();
   parameters.setPreviewSize(w, h);
   camera.setParameters(parameters);
} catch (Exception e) {
   Log.w("CameraView", "Exception:" , e);
}
```

An dieser Stelle eine Warnung: Fangen Sie nie eine unchecked Exception, ohne in irgendeiner Form darauf zu reagieren! Die Folge ist im besten Fall unvorhersehbares Verhalten Ihrer App, im schlimmsten Fall Exceptions an völlig anderen Stellen (Folgefehler).

Fangen Sie unchecked Exceptions nur, wenn Sie genau wissen, warum Sie das tun, und erzeugen Sie mindestens eine Log-Ausgabe.

**Logging in Android**

Sie können jederzeit in einer App Anmerkungen ins systemweite Protokoll schreiben. Mit Bordmitteln am Handy können Sie das Protokoll zwar nicht auslesen, aber mit Apps wie *Android System Info* können Sie einen Blick hineinwerfen. Wenn Sie genug hilfreiche Log-Ausgaben erzeugen, können Sie Glück haben: Manchmal ist ein Benutzer Ihrer App dazu bereit, extra Android System Info zu installieren und Ihnen ein Logfile zu mailen, in dem Sie tatsächlich ablesen können, was schiefgeht.

Solange Ihr Handy per USB am Rechner hängt (oder ein Emulator läuft), können Sie in Eclipse im View namens Logcat live verfolgen, was ins Logfile geschrieben wird. Sie werden diese Ansicht oft sehr hilfreich finden, das kann ich Ihnen versprechen.

Einträge ins Logfile haben verschiedene Stufen, zum Beispiel *debug*, *info* oder *warning*, die Sie mit der zugehören Methode der Klasse `Log` erzeugen können. In Android System Info erscheinen die Zeilen abhängig von der Stufe in unterschiedlichen Farben.

Geben Sie als ersten Parameter immer einen String an, der darüber Aufschluss gibt, in welchem Teil einer App der Eintrag geschrieben wird. Darüber hinaus können Sie einen beliebigen Text ausgeben lassen, eine Exception oder auch Zahlen:

```
Log.d("MeineApp","Dies ist ein Debug-Log.");
```

Mehr zum Thema Logging erfahren Sie im Abschnitt »Logging einbauen«.

Als letzten Schritt müssen Sie in `surfaceChanged()` natürlich noch die eigentliche Preview starten:

```
camera.startPreview();
```

Wenn Sie die Klasse fehlerfrei implementiert und gespeichert haben, können Sie sie als Ihren ersten selbst gebauten View ins Layout integrieren.

### CameraView ins Layout integrieren

Im Moment besteht das Layout des Spielbildschirms aus einem LinearLayout mit einem Foto als Hintergrund. In diesem LinearLayout sind der obere Informationsbereich, der Spielbereich und die unteren Anzeigebalken integriert.

Leider können Sie Ihren CameraView nicht als Hintergrund des LinearLayouts verwenden. Wir benötigen einen neuen Hintergrund, und der jetzige wird zum Vordergrund. Ändern Sie als erstes die ID des LinearLayouts in `vordergrund`.

Anschließend verwenden Sie die praktische Funktion Wrap in Container des Layout-Editors, um das Vordergrund-LinearLayout zum Kind eines neuen Views zu machen, der dann der neue Hintergrund wird. Als Layout wählen Sie ein FrameLayout, als ID `hintergrund` (Abbildung 8.2).

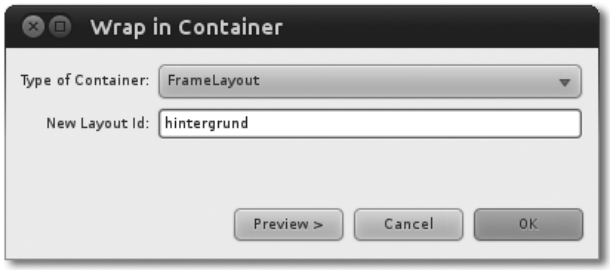



**Abbildung 8.2** Der alte Hintergrund wird in ein neues FrameLayout eingebettet.

Entfernen Sie nun das Hintergrundbild des neuen Vordergrunds, indem Sie das `background`-Attribut des LinearLayouts leeren.

**Direktes Editieren des Layout-XML**

Bisher habe ich Ihnen jeden Blick auf den XML-Code erspart, der sich hinter den Layouts verbirgt. Als der Layout-Editor noch nicht so mächtig war wie heute, führte kein Weg am unübersichtlichen XML vorbei. Heute können Sie Android-Apps schreiben, ohne je XML-Code zu Gesicht zu bekommen.

Bis auf eine Kleinigkeit. Falls Sie genau wie ich vergeblich nach einer Möglichkeit suchen, dem Attribut `background` »nichts« zuzuweisen, bleibt Ihnen nichts anderes übrig, als direkt am XML herumzufummeln.

Klicken Sie also auf den unteren Anfasser, auf dem `game.xml` steht.

Suchen Sie am oberen Ende den folgenden XML-Code, der das LinearLayout für den Vordergrund definiert:

```
<LinearLayout android:orientation="vertical"
   android:layout_width="fill_parent"
   android:background="@drawable/hintergrund"
   android:layout_height="fill_parent"
   android:id="@+id/vordergrund">
```

Die Reihenfolge der Attribute in diesem XML-Element kann sich unterscheiden, und auch die Formatierung. Löschen Sie das Attribut `android:background` komplett (also inklusive =-Zeichen und dem, was dahintersteht), dann schalten Sie wieder zurück zum grafischen Layout-Editor.

Vielleicht haben Sie aber Glück, und den Entwicklern des Layout-Editors fällt ihr Versäumnis in nächster Zeit auf (manchmal sind die Jungs ziemlich fix). Dann schaffen Sie es vielleicht, weiterhin ohne XML-Code zu leben.

Falls Sie Ihren bisherigen Code verwenden und in der `GameActivity` eine Änderung des Hintergrundbildes eingebaut haben, müssen Sie diesen Code jetzt löschen.

Jetzt fehlt Ihrem Layout nur noch der CameraView!

Dabei handelt es sich um eine selbst gebaute Komponente, die Sie in der Palette des Layout-Editors unter der Rubrik Custom & Library Views finden. Eventuell müssen Sie den Refresh-Button anklicken, damit der CameraView auftaucht.

Ziehen Sie ihn einfach in das Hintergrund-FrameLayout, achten Sie aber darauf, dass der CameraView hinter dem Vordergrund-LinearLayout zu liegen kommt. Maßgeblich ist die Reihenfolge, die Sie im Outline-Fenster begutachten und ändern können: Stellen Sie sich einfach vor, dass Android die Komponentenliste von oben nach unten abarbeitet und zeichnet. Weiter oben stehende Kinder eines FrameViews erscheinen also *hinter* weiter unten stehenden (Abbildung 8.3).

Wenn Sie das Layout richtig zusammengebaut haben, können Sie schon erkennen, dass der grau dargestellte CameraView hinter den Einblendungen im Vordergrund liegt. Eine Kameravorschau bietet Ihnen Eclipse natürlich nicht – dazu müssen Sie später Ihre App starten.

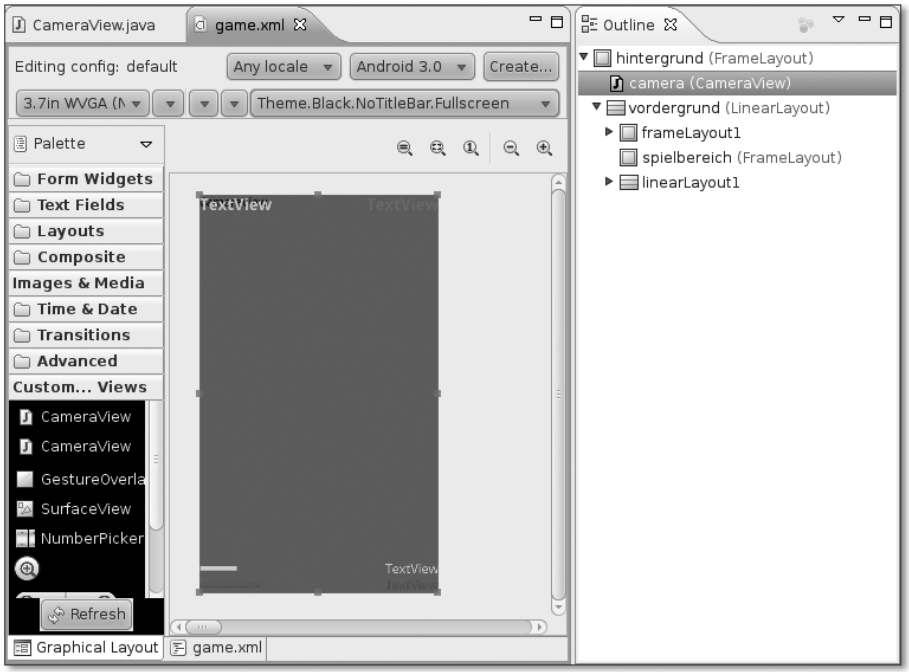



**Abbildung 8.3** Achten Sie darauf, dass der CameraView in der Hierarchie das erste Kind des FrameLayouts im Hintergrund ist.

## Die Camera-Permission

Wie fast alle Hardwarekomponenten dürfen Sie auch die Kamera nicht ohne explizite Erlaubnis verwenden. Man stelle sich vor, eine App, die vorgeblich einem ganz anderen Zweck dient, macht gelegentlich heimlich Fotos und lädt sie ins Internet hoch – was für ein Spaß! Bloß nicht für alle Betroffenen.

Öffnen Sie also das Manifest Ihrer App, und fügen Sie einen passenden Eintrag unter den Permissions hinzu (Abbildung 8.4).

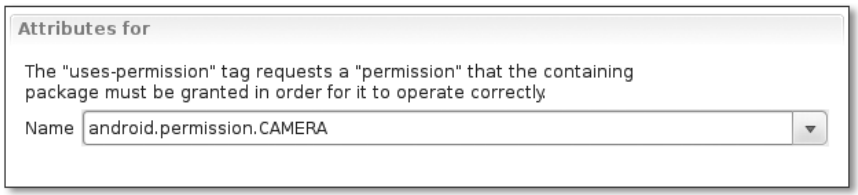



**Abbildung 8.4** Ohne Erlaubnis keine Fotos und keine Vorschau

Wenn Sie Ihre App jetzt starten, sehen Sie, was andere Menschen nicht sehen: Mücken, die in der Landschaft herumfliegen und die man mit der Fingerspitze loswerden kann (Abbildung 8.5). Ein Traum, oder?


**Abbildung 8.5** Am Ende des Kapitels steht der Mückenfang in Nachbars Garten.

## 8.2 Bilddaten verwenden

Wussten Sie eigentlich, dass Mücken keine Tomaten mögen? Nein? Nun, möglicherweise gilt das nur für die Mücken in unserem Spiel. Wir werden nämlich als Nächstes eine zweite Möglichkeit einbauen, um die Viecher loszuwerden: Richten Sie die Kamera so auf eine Tomate, dass Tomate und Mücke einander berühren, um die Mücke zu terminieren!

Zugegeben, das Spielchen wird auch mit einer Kirsche oder einer Clownnase funktionieren. Denn wir werden als Nächstes das Kamerabild nach roten Gegenständen absuchen, und jede Mücke, die einem roten Gegenstand zu nahe kommt, vernichten. Und zwar mit doppelter Punktzahl!

### Bilddaten anfordern

Bislang wird die Kameravorschau lediglich auf den Bildschirm gebracht – um die eigentlichen Bilddaten kümmert sich die App dabei überhaupt nicht. Allerdings benötigen Sie die Bilddaten, wenn Sie darin nach roten Objekten suchen wollen. Also müssen Sie eine Möglichkeit implementieren, die Daten von der Kamera anzufordern. Damit die Game Engine komfortabel auf die Bilddaten zugreifen kann, übergeben Sie die nötige Schnittstelle zunächst im CameraView an die Kamera:

```
public void setOneShotPreviewCallback(PreviewCallback callback) {
   if(camera!=null) {
```

```
        camera.setOneShotPreviewCallback(callback);
    }
}
```

Das Interface `PreviewCallback` enthält nur eine Methode: `onPreviewFrame()`. Obiger Code reicht ein Objekt, das `PreviewCallback` implementiert, an die Methode `setOneShotPreviewCallback()` weiter. Der Name der Methode verrät es: Der Callback wird nur einmal verwendet (`OneShot`), nämlich wenn die Kamera das nächste Vorschaubild auf dem Bildschirm darstellt. Sofort danach löscht sie den Callback.

Auf diese Weise müssen Sie zwar den Callback immer wieder neu setzen, wenn Sie ein neues Bild erhalten möchten. Andererseits aber kann es nicht passieren, dass die Kamera Ihren Code mit Daten überhäuft und Sie mit der Verarbeitung nicht hinterherkommen.

Die Strategie wird also wie folgt aussehen:

- Die Game Engine setzt den Callback.
- Die Kamera schickt bei nächster Gelegenheit Bilddaten zurück.
- Die Game Engine erhält die Bilddaten und wertet sie aus.
- Alles beginnt wieder von vorn, bis das Spiel vorbei ist.

Speichern Sie als Erstes in der `GameActivity` eine Referenz auf den CameraView:

```
private CameraView cameraView;
...
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.game);
    cameraView = (CameraView) findViewById(R.id.camera);
    ...
```

Lassen Sie die `GameActivity` das Interface `PreviewCallback` implementieren:

```
public class GameActivity extends Activity implements Runnable,
OnClickListener, PreviewCallback
```

Die benötigte Methode, die dann später von der Kamera aufgerufen wird, heißt `onPreviewFrame()`:

```
@Override
public void onPreviewFrame(byte[] bild, Camera camera) {
}
```

Die Bilddaten verbergen sich später in dem `byte[]`-Array namens `data`.

Fordern Sie nun in der Methode `spielStarten` Bilddaten an:

```
cameraView.setOneShotPreviewCallback(this);
```

Das war der einfache Teil. Jetzt müssen Sie die Daten, die `onPreviewFrame()` erhält, nach roten Gegenständen durchsuchen.

### Bilddaten auswerten

Bevor Sie damit beginnen können, die Bilddaten auszuwerten, müssen Sie sich darüber im Klaren sein, wie sie organisiert sind. Ein `byte[]`-Array ist so ziemlich die unsortierteste Datenquelle, die Sie sich vorstellen können: Sie enthält eine bestimmte Anzahl Bytes, und in welcher Weise diese die Bilddaten repräsentieren, ist eine Frage der Kameraparameter. Das Standardformat für Vorschaubilder hört auf die Namen NV21 und YCbCr_420_SP.

Das klingt nach einem Mittelding aus Zugnummer und Weltformel, und die gute Nachricht an dieser hoch komplizierten Sache ist: Die Methode, die aus diesen wüsten Bilddaten die Farbe eines Punktes ermittelt, müssen Sie weder schreiben noch verstehen – ich habe das für Sie erledigt.

Wenn Sie möchten, können Sie sich im Netz Informationen über das interne Bildformat ansehen und anschließend darüber fluchen, dass Android kein einfacheres Format für die Vorschaubilder verwendet.

Anschließend werfen Sie einen Blick auf die Methode `holePixel()`, die geradezu magische Operationen ausführt, um Ihnen die Farbe eines Bildpunkts zu verraten:

```
private int holePixel(byte[] bild, int breite, int hoehe, int x, int
 y) {
   if(x>=breite || x<0 || y>=hoehe || y<0)  return 0;
   int frameSize = breite * hoehe;
   int uvp = frameSize + (y >> 1) * breite;
   int Y = (0xff & ((int) bild[breite*y + x])) - 16;
   if (Y < 0) Y = 0;
   int v = (0xff & bild[uvp+2*(x/2)]) - 128;
   int u = (0xff & bild[uvp+2*(x/2)+1]) - 128;
   int y1192 = 1192 * Y;
   int r = (y1192 + 1634 * v);
   int g = (y1192 - 833 * v - 400 * u);
   int b = (y1192 + 2066 * u);
   if (r < 0) r = 0; else if (r > 262143) r = 262143;
   if (g < 0) g = 0; else if (g > 262143) g = 262143;
   if (b < 0) b = 0; else if (b > 262143) b = 262143;
```

```
    return 0xff000000 | ((r << 6) & 0xff0000) | ((g >> 2) & 0xff00) |
 ((b >> 10) & 0xff);
}
```

Ein YUV-Format ist eines, in dem die Bildinformation aus Luminanz (Lichtstärke, Y) und Chrominanz (Farbanteil, UV) zusammengesetzt ist. Sie müssen sich nicht für die Details interessieren, aber wenn ich Ihnen verrate, dass beispielsweise das analoge Fernsehformat PAL auf diese Weise arbeitet, können Sie vielleicht beim nächsten Small Talk punkten. Schwarz-Weiß-Fernseher (ja, so was gab's mal!) verwendeten lediglich den Luminanz-Anteil. Vielleicht erkennen Sie im Code der Methode `holePixel()`, dass die Helligkeit aus dem vorderen Teil des `bild[]`-Arrays entnommen wird und die Farbwerte aus dem hinteren Teil. Wenn nicht, macht das nichts: Freuen Sie sich einfach darüber, dass Ihnen jemand die Arbeit abgenommen hat, diese ziemlich anstrengende Methode zu schreiben, und kümmern Sie sich um den Rest.

Beginnen Sie also mit dem Schreiben der Methode `onPreviewFrame()`, die immer dann aufgerufen wird, wenn ein neues Vorschaubild bereitsteht:

```
public void onPreviewFrame(byte[] bild, Camera camera) {
}
```

Ermitteln Sie nun zunächst Breite und Höhe des Bildes:

```
int breite = camera.getParameters().getPreviewSize().width;
int hoehe = camera.getParameters().getPreviewSize().height;
```

Fahren Sie fort mit einer Überprüfung, ob überhaupt das erwartete Format vorliegt. Nur dann müssen Sie alle vorhandenen Mücken überprüfen.

```
if (camera.getParameters().getPreviewFormat() == ImageFormat.NV21) {
   mueckenAufTomatenPruefen(bild, breite, hoehe);
}
```

Die eigentliche Analyse der Bilddaten ist keine Sache von zwei oder drei Zeilen. Sie müssen sich einige Gedanken machen, damit die nötigen Methoden korrekt und vor allem schnell arbeiten.

**Bildinhalte analysieren**

Grundsätzlich ist Bildverarbeitung eine aufwendige Angelegenheit. Das liegt nicht so sehr an den Rechenoperationen, sondern an deren schierer Menge: Ein simples Vorschaubild kommt schon auf über 200.000 Bildpunkte – möchten Sie irgendeine Operation mit jedem Pixel durchführen, muss der zuständige Code genauso oft durchlaufen werden.

Daher gilt es, im jeweiligen Fall zu optimieren. Beschränken Sie sich auf die Rechenoperationen, die wirklich notwendig sind. Wenn Sie beispielsweise das Vorschaubild nach roten Objekten durchsuchen möchten, prüfen Sie nicht die Farbe jedes Bildpunktes, sondern suchen Sie nur an der kleinstmöglichen Anzahl von Stellen.

Beispielsweise können Sie ein grobmaschiges Raster von Messpunkten über den Bildschirm legen. Prüfen Sie auf diese Weise nur die Knotenpunkte eines Netzes mit einem Linienabstand von zehn Pixeln, so sparen Sie glatt 99 % des Rechenaufwands, weil nur einer von 10 x 10, also 100 Pixeln, betrachtet werden muss! Objekte, die kleiner sind als 10 x 10 Pixel, gehen Ihnen zwar durch die Lappen, aber in den meisten Fällen können Sie damit leben. Aber nicht mit einer unglaublich trägen, nahezu unbenutzbaren App.

### Tomaten gegen Mücken

Im letzten Schritt müssen Sie für jede Mücke prüfen, ob sie einen roten Bildbereich berührt. Dazu kommen Sie um etwas Rechnerei nicht herum. Erstens müssen Sie davon ausgehen, dass das Vorschaubild eine andere Auflösung hat als der Bildschirm. Beispielsweise liefert eine Kamera im Hochformat eine Vorschau von 480 x 480 Pixeln, aber der Bildschirm ist 854 x 480 Pixel groß (was noch dazu zu einer unschönen Verzerrung führt).

Stellen Sie sich ein Quadrat vor, das eine Mücke völlig umschließt. Wenn Sie die Koordinaten dieses Quadrats in die Koordinaten des Vorschaubildes umrechenen, wissen Sie, welchen Bereich Sie nach roter Farbe durchsuchen müssen.

Wie aber ermittelt die Game Engine, ob ein Bereich rote Farbe enthält? Zunächst müssen Sie definieren, was »Rot« überhaupt bedeutet. Natürlich können Sie nicht einfach nach dem RGB-Farbcode `#FF0000` suchen, denn das ist reines Rot, das in der Natur kaum vorkommt. Es genügt ebenfalls nicht, zu prüfen, ob die Rotkomponente den größten Wert aller drei Farbanteile hat – das mag auch für Braun oder helles Rosa gelten. Nein, als Rot dürfen nur solche Bildpunkte gelten, bei denen die Differenz zwischen Rotanteil und Durchschnittshelligkeit ein Minimum von 100 überschreitet.

Beispiele für diese Defintion des Rotanteils:

- Weiß hat einen Rotanteil von 0, weil der Rotanteil und die Durchschnittshelligkeit genau gleich groß sind (nämlich 255 oder hexadezimal `FF`). Für Schwarz gilt dasselbe.
- Braun (zum Beispiel `#502828`, also RGB 80, 40, 40) hat einen Rotanteil von etwa 27 (80 minus 160 durch 3).

- Flieder (#C896C8 oder 200, 150, 200) hat einen Rotanteil von knapp 17 (200 minus 150 durch 3).
- Eine reife Cocktailtomate auf einem Foto (z. B. *http://goo.gl/WaChO*) hat die Farbe #F42400 (244, 36, 0) und einen Rotanteil von satten 150.

Die Aufgabe besteht jetzt also darin, im zu einer Mücke gehörenden Ausschnitt des Vorschaubildes die Pixel zu zählen, deren Rotanteil über 50 liegt. Falls wir mehr als zehn solcher Pixel finden (weniger könnten Bildrauschen sein), soll die Mücke den fürchterlichen Tomatentod sterben.

Lassen Sie uns zunächst eine Methode schreiben, die alle roten Pixel in einem Ausschnitt des Vorschaubildes zählt:

```
private int zaehleRotePixel(byte[] bild, int breite, int hoehe, Rect
 ausschnitt) {
   int anzahl = 0;
   for(int x=ausschnitt.left; x<ausschnitt.right; x++) {
      for(int y=ausschnitt.top; y<ausschnitt.bottom; y++) {
         if(istPixelRot(bild, breite, hoehe,x,y)) {
            anzahl++;
         }
      }
   }
   return anzahl;
}
```

Schauen Sie sich vor allem die beiden ineinander verschachtelten Schleifen an. Die äußere Schleife durchläuft alle Spaltennummern vom linken Rand des Rechtecks `ausschnitt` bis zum rechten. Dabei wird die letzte Spalte weggelassen, weil die Zählung der Spalten bei 0 beginnt und folglich bei `ausschnitt.right` keine Spalte mehr existiert. Die innere Schleife wird für jede Spalte des Bildes ausgeführt, und zwar vom oberen bis zum unteren Rand des Ausschnitts. Das Ganze funktioniert so ähnlich wie ein Rasenmäher, bloß dass Sie mit dem normalerweise abwechselnd den Rasen rauf- und runterfahren.

Somit wird die Zeile im Inneren der beiden Schleifen für jeden Bildpunkt im Auswahlbereich einmal ausgeführt. Dort rufen wir eine andere Methode auf, die die Pixelfarbe überprüft, und zählen im positiven Fall die Variable `anzahl` hoch. Die Methode `istPixelRot()` müssen Sie natürlich noch schreiben:

```
private boolean istPixelRot(byte[] bild, int breite, int hoehe, int
x, int y) {
   int farbe = holePixel(bild, breite, hoehe, x, y);
   return Color.red(farbe)-
(Color.red(farbe)+Color.green(farbe)+Color.blue(farbe))/3 > 60 ;
}
```

Diese Methode holt sich zunächst mittels der magischen Methode `holePixel()` den kombinierten Farbwert des Pixels an der gewünschten Stelle. Danach folgt die Abfrage des Rotwertes, die wir weiter oben diskutiert haben. Dabei unterstützt die Klasse `Color` mit ihren drei statischen Methoden `red()`, `blue()` und `green()`. Diese Methoden extrahieren aus dem RGB-Farbwert `farbe` den Anteil der jeweiligen Farbe. Falls die Differenz aus Rotanteil und dem Mittelwert aller Farbanteile größer als 50 ist, ist der Pixel für uns rot.

Schreiben Sie als Letztes die Methode `mueckenAufTomatenPruefen()`:

```
private void mueckenAufTomatenPruefen(byte[] bild, int breite,
int hoehe) {
   int nummer=0;
   while(nummer<spielbereich.getChildCount()) {
      ImageView muecke = (ImageView) spielbereich.getChildAt(nummer);
      if(mueckeBeruehrtTomate(muecke, bild, breite, hoehe)) {
         mp.pause();
         gefangeneMuecken++;
         punkte += 100 + schwierigkeitsgrad*100;
         bildschirmAktualisieren();
         spielbereich.removeView(muecke);
      } else {
         nummer++;
      }
   }
}
```

Auch diese Methode benötigt neben den Bilddaten die Breite und Höhe des Bildes als Parameter, die weiter durchgereicht werden.

Wie Sie alle Mücken auf dem Spielfeld einmal in einer `while()`-Schleife betrachten können, wissen Sie schon aus der Methode `mueckenBewegen()`. Prüfen Sie nun für jede Mücke, ob sie eine Tomate berührt. Falls ja, ergreifen Sie dieselben Maßnahmen wie beim erfolgreichen Antippen einer Mücke, bloß gibt's für die Tötung per Tomaten doppelt so viele Punkte (wenn Sie möchten, können Sie das natürlich anders handhaben).

Als Letztes schreiben Sie die Methode `mueckeBeruehrtTomate()`. Noch eine Methode!? Ja, denn Sie könnten zwar theoretisch den ganzen Code in eine einzige Methode quetschen (`onPreviewFrame()`), aber damit wäre jeglicher Überblick verloren. Erinnern Sie sich daran, dass Methoden möglichst nicht länger als eine Bildschirmseite in Eclipse werden sollten. Außerdem sollte sich jede Methode nur um eine Aufgabe kümmern, nicht um mehrere.

Die Methode `mueckeBeruehrtTomate()` muss natürlich als Resultat einen `boolean`-Wert zurückgeben. Außerdem muss sie die Koordinaten zwischen Bildschirmpixeln (Mücken) und Vorschaubild umrechnen. Da das Vorschaubild um 90° verdreht ist (es ist immer horizontal ausgerichtet, das Spielfeld aber vertikal), müssen Sie ein wenig mit Breite und Höhe experimentieren:

```
private boolean mueckeBeruehrtTomate(ImageView muecke, byte[] bild,
int breite, int hoehe) {
   float faktorHorizontal = hoehe*1.0f /
 getResources().getDisplayMetrics().widthPixels;
   float faktorVertikal = breite*1.0f /
 getResources().getDisplayMetrics().heightPixels;
   Rect ausschnitt = new Rect();
   ausschnitt.bottom= Math.round(hoehe -
 faktorHorizontal * muecke.getLeft());
   ausschnitt.top   = Math.round(hoehe -
 faktorHorizontal * muecke.getRight());
   ausschnitt.right = Math.round(faktorVertikal * muecke.getBottom()
);
   ausschnitt.left  = Math.round(faktorVertikal * muecke.getTop());
   int rotePixel = zaehleRotePixel(bild, breite, hoehe, ausschnitt);
   if(rotePixel > 10) {
      return true;
   }
   return false;
}
```

Wenn sich also mehr zehn rote Pixel im Bereich der Mücke befinden, interpretieren wir das als Berührung mit einer Tomate.

Probieren Sie das Spiel in dieser Form ruhig einmal aus. Sie werden sehen, dass es nicht träger wirkt als bisher. Trotzdem besteht noch Potenzial zur Optimierung:

- `zaehleRotePixel()` könnte sich auf jeden zweiten Pixel in jeder zweiten Zeile beschränken und so 75 % Rechenzeit sparen.
- Sie könnten anstelle von `zaehleRotePixel()` eine Methode `enthaeltBereichRotePixel()` bauen, der der Grenzwert 10 übergeben wird. Diese Methode könnte dann bei Erreichen des Grenzwerts aufhören zu zählen.
- Richtet man die Kamera auf eine komplett rote Fläche, stirbt jede Mücke sofort nach ihrem Erscheinen. Um diese Schummelei zu verhindern, könnte eine zusätzliche Methode den Rotanteil des gesamten Bildschirms berechnen und bei einem bestimmten Grenzwert die Tötung durch Tomaten unterbinden. Natürlich dürfte diese Methode nicht jeden einzelnen Pixel des Vorschaubildes testen, sondern müsste mit einem Raster von z. B. einem von 100

Pixeln arbeiten. Beispielsweise könnte die gerade konzipierte Methode `enthaeltBereichRotePixel()`auch das leisten: Sie müssten ihr nur als zusätzlichen Parameter eine Rasterweite übergeben.

- Die Funktionen zum Auslesen der Pixel aus einem Vorschaubild sind nicht ohne Weiteres in anderen Apps wiederverwendbar. Schade, oder?

**Klassen extrahieren**

Spätestens wenn Sie eine weitere App schreiben, die Vorschaubilder der Kamera analysieren möchte, werden Sie sich darüber ärgern, dass der zugehörige Code tief in der `GameActivity` versteckt ist. Sie können nicht einfach eine fertige Klasse rüber in Ihr neues Projekt kopieren und direkt verwenden. Wiederverwendbarkeit ist ein wichtiges Konzept in der Softwareentwicklung, und Eclipse und Java machen es Ihnen leicht.

Erzeugen Sie zunächst eine neue Klasse namens `NV21Image.java`. Verschieben Sie dann einfach die drei Methoden `holePixel()`, `zaehleRotePixel()` und `istPixelRot()` dort hinein. Ändern Sie die Zugriffs-Modifier von `private` in `public`.

Entfernen Sie die Parameter `bild`, `breite` und `hoehe` aus allen drei Parameterlisten (sowie aus den Aufrufen innerhalb der Klasse), und fügen Sie der Klasse stattdessen `private`-Attribute gleichen Namens hinzu:

```
private byte[] bild;
private int breite;
private int hoehe;
```

Als Letztes schreiben Sie einen Konstruktor, der diese drei Attribute als Parameter erhält. Das geht per Rechtsklick: Wählen Sie im Kontextmenü Source • Generate Constructor using fields. Erzeugen Sie anschließend noch Getter für `breite` und `hoehe`, indem Sie erneut das Kontextmenü bemühen (Abbildung 8.6).

Jetzt können Sie in der `GameActivity` in `onPreviewFrame()` ein `NV21Image`-Objekt erstellen:

```
NV21Image nv21 = new NV21Image(bild, breite, hoehe);
```

Ändern Sie zuletzt die Signatur der Methode `mueckenAufTomatenPruefen()`, denn die kommt jetzt prima mit dem `NV21Image` aus:

```
private void mueckenAufTomatenPruefen(NV21Image nv21)
```

Analog müssen Sie `mueckeBeruehrtTomate()` ändern. Schließlich können Sie dort die alles entscheidende Methode der Klasse `NV21Image` aufrufen:

```
int rotePixel = nv21.zaehleRotePixel(ausschnitt);
```

Der Umbau belohnt Sie mit kürzerem, übersichtlicherem Code und einer Klasse `NV21Image`, die Sie ohne Weiteres in anderen Projekten verwenden können – wenn Sie zusätzliche Methoden brauchen, die nach andersfarbigen Pixeln suchen, können Sie sie leicht hinzufügen.


**Abbildung 8.6** Eclipse nimmt Ihnen beim Erzeugen von Gettern und Settern Tipparbeit ab.

Sie sehen, dass Bildverarbeitung eine komplizierte Angelegenheit ist, die jedoch zu erstaunlichen Ergebnissen führt. Man könnte ein eigenes Buch zu dem Thema schreiben, und um desssen Autor nicht die Arbeit abzunehmen, soll der Ausflug in die Bildverarbeitung hiermit beendet sein.

*»... kein Schießpulver, ein Kompass, der nicht nach Norden zeigt ...*
*Sie sind ohne Zweifel der schlechteste Pirat, von dem ich je gehört habe.«*
*»Aber Ihr **habt** von mir gehört.«*
*(aus »Fluch der Karibik«)*

# 9 Sensoren und der Rest der Welt

Sicher haben Sie als Kind auch einmal eine magnetische Nadel auf ein Stück Korken geklebt und diesen auf dem Wasser schwimmen lassen. Faszinierend, welche Kraft das allgegenwärtige Erdmagnetfeld hat! Und wie nützlich sie ist! Man kann damit sogar Amerika entdecken (oder zumindest Inseln in der Karibik). Aber Sie kennen das ja: Immer dann, wenn man gerade einen Kompass braucht, um den Heimweg (oder das Kreuz auf einer Schatzkarte) zu finden, hat man keinen zur Hand. Aber das Smartphone hat man natürlich immer dabei!

Das Dumme an einer Magnetnadel ist, dass sie sich nicht dafür interessiert, auf *welches* Magnetfeld sie da gerade reagiert: auf das der Erde, auf das eines Elektromotors oder sogar auf ein induziertes Feld in einem Stück Alteisen. Ein Kompass funktioniert nur vernünftig, wenn wenig Störungen in der Nähe sind. Erstaunlich genug, dass er bei einem Smartphone, das immerhin vollgestopft ist mit Elektronik, überhaupt noch einigermaßen funktioniert.

Stellen Sie sich vor, das Smartphone zeigt auf dem Bildschirm einen Pfeil an, der immer nach Norden zeigt. Das bedeutet, dass der Pfeil sich relativ zum Bildschirm dreht, wenn Sie das Gerät drehen. Abgesehen von der grafischen Darstellung, benötigen Sie vor allem eines, um eine solche App zu verwirklichen: den Magnetfeld-Sensor.

Schauen wir mal, wohin uns der Android-Kompass führt ...

## 9.1 Himmels- und sonstige Richtungen

Sensoren für das Erdmagnetfeld gibt es schon seit dem 11. Jahrhundert in China, aber in Handys erst seit dem 21. Jahrhundert. Während chinesische Kompasse stets nach Süden zeigen, ist die Sache im Android-Handy etwas komplizierter: Verbaut ist nämlich ein *dreidimensionaler* Magnetfeldsensor.

Diese Tatsache hat eine ganze Reihe Vor- und Nachteile. Wir kommen darauf zu sprechen, während wir mithilfe einer einfachen App in den folgenden Abschnit-

ten das Smartphone in einen Kompass verwandeln, mit dem Kolumbus Indien bestimmt nicht verfehlt hätte.

### Der SensorManager

Als ersten Schritt zu Ihrem eigenen Kompass legen Sie ein neues Android-Projekt in Eclipse an. Nennen Sie es **Kompass**, legen Sie einen Package-Namen fest, und wählen Sie Android 1.6 oder neuer als TARGET. Geben Sie `KompassActivity` als Namen der zu erstellenden Activity ein, und los geht's.

Alle Sensoren werden von einem zentralen Android-Dienst verwaltet, dem `SensorManager`. Holen Sie sich eine Referenz auf diesen Dienst, indem Sie die Methode `getSystemService()` verwenden:

```
@Override
protected void onCreate(Bundle b) {
   super.onCreate(b);
   sensorManager = (SensorManager)getSystemService(Context.
   SENSOR_SERVICE);
}
```

Je nachdem, welche `Context.SENSOR`-Konstante Sie übergeben, erhalten Sie eine Referenz auf einen anderen Dienst. Im Moment interessieren wir uns aber nur für den `SensorManager`.

Natürlich müssen Sie ein Attribut in Ihrer Activity einführen:

```
private SensorManager sensorManager;
```

Bitten Sie als Nächstes den `SensorManager` um eine Referenz auf den richtigen Sensor:

```
magnetfeldSensor = sensorManager.getDefaultSensor(Sensor.TYPE_
ORIENTATION);
```

Auch dieses Attribut müssen Sie natürlich in Ihrer Klasse deklarieren:

```
private Sensor magnetfeldSensor;
```

Jetzt müssen Sie nur noch die Daten des Sensors auslesen. Wenn Sie nun mit dem Content-Assist eine Methode namens `magnetfeldSensor.getData()` suchen, werden Sie keinen Erfolg haben. Die Sache ist ein klein wenig komplizierter.

### Rufen Sie nicht an, wir rufen Sie an

Wie die meisten Dienste arbeitet auch der `SensorManager` asynchron. Sie fragen ihn also nicht in regelmäßigen Abständen nach der aktuellen Nordrichtung, son-

dern Sie melden einen **Listener** an, der benachrichtigt wird, wenn es etwas Neues gibt. Das ist effizient: Nur, wenn sich Sensorwerte ändern, muss Code ausgeführt werden, ansonsten kann das Handy sinnvollere Dinge tun, zum Beispiel Strom sparen.

Mehr noch: Wenn Ihre `KompassActivity` gar nicht im Vordergrund ist, muss sie auch keine Sensordaten empfangen. Folglich ist der richtige Weg der folgende:

- Sobald die Activity aktiviert wird, abonnieren wir Sensor-Ereignisse.
- Sobald ein Sensor-Ereignis eintrifft, zeichnen wir die Kompassnadel.
- Sobald die Activity gestoppt wird, kündigen wir unser Abo.

Sie kennen ja schon die Ereignisbehandlungsmethoden der `Activity`-Klasse. Wird eine Activity in den Vordergrund geholt, wird, `onResume()` aufgerufen. Überschreiben Sie also diese Methode, um das Abo beim `SensorManager` abzuschließen:

```
@Override
protected void onResume() {
   super.onResume();
   sensorManager.registerListener(this, magnetfeldSensor,
         SensorManager.SENSOR_DELAY_GAME);
}
```

Sie sehen, dass der `SensorManager` für die Verwaltung der Abonnements zuständig ist, nicht der Sensor. Den müssen wir lediglich als zweiten Parameter übergeben.

Als ersten Parameter müssen Sie einen `SensorEventListener` angeben, genauer gesagt: ein Objekt, das dieses Interface anbietet. Um nicht extra eine eigene Klasse dafür zu bauen, muss wie schon so oft die Activity selbst herhalten. Ergänzen Sie also die `implements`-Anweisung:

```
public class KompassActivity extends Activity implements
SensorEventListener {
   ...
}
```

Der dritte Parameter des `registerListener()`-Aufrufs definiert die gewünschte zeitliche Auflösung. Es gibt vier verschiedene, die ich Ihnen in aufsteigender Genauigkeit aufliste:

- `SENSOR_DELAY_NORMAL`
- `SENSOR_DELAY_UI`
- `SENSOR_DELAY_GAME`
- `SENSOR_DELAY_FASTEST`

Sie erfahren am schnellsten über eine Änderung am Magnetfeld – also an der Ausrichtung des Handys –, wenn Sie `SENSOR_DELAY_FASTEST` wählen. Leider genehmigt sich dieser Modus die größte Portion Batterieladung. Für die grafische Darstellung einer Kompassnadel ist `SENSOR_DELAY_GAME` ein brauchbarer Kompromiss.

Vergessen Sie nicht, den Listener wieder abzumelden, wenn die Activity vom Bildschirm verschwindet – in dem Fall wird `onPause()` aufgerufen:

```
@Override
protected void onPause() {
   sensorManager.unregisterListener(this);
   super.onPause();
}
```

Fällt Ihnen etwas auf? Im Gegensatz zu Klingelton-Abos ist die Abmeldung viel einfacher als die Anmeldung.

Werfen Sie nun einen Blick auf die entscheidende Methode, die immer dann aufgerufen wird, wenn neue Messwerte vorliegen: `onSensorChanged()`.

Die eigentlichen Messwerte des Magnetfeldes stecken in einem Attribut des `SensorEvent`-Objekts, das der `SensorManager` dieser Methode übergibt. Die Methode implementieren wir später, vorläufig muss ein leerer Rumpf reichen, um der `implements`-Deklaration zu genügen:

```
public void onSensorChanged(SensorEvent event) {
}
```

Das Interface `SensorEventListener` verlangt noch eine zweite Methode, die wir allerdings nur schreiben, weil wir müssen. Verwenden werden wir sie nicht.

```
public void onAccuracyChanged(Sensor sensor, int accuracy) {
}
```

Jetzt fehlt nur noch die grafische Darstellung.

### Die Kompassnadel und das Canvas-Element

Sie werden sich schon gefragt haben, wie Sie eine Kompassnadel zeichnen können. Mit den bisher besprochenen Mitteln der Views und Layouts funktioniert das nicht: Es fehlt die Möglichkeit der stufenlosen Drehung.

TextViews können Texte darstellen, Buttons Knöpfe, ImageViews Bilder – aber keiner unterstützt eine Drehung um einen beliebigen Winkel.

Deshalb werden wir jetzt einen eigenen View implementieren. Solche Custom Views kümmern sich komplett selbst um ihr Aussehen. Unser `KompassnadelView` wird eine weiße Nadel im gewünschten Winkel auf eine schwarze Fläche zeichnen.

Ein Custom View ist zunächst einmal nichts anderes als eine Klasse, die von der Klasse `View` erbt:

```
public class KompassnadelView extends View {
}
```

Die Klasse erhält einen Konstruktor, der später einige Vorbereitungen vornehmen wird:

```
public KompassnadelView(Context context) {
   super(context);
}
```

Der `KompassnadelView` muss natürlich den Winkel kennen, in dem er die Nadel zeichnen soll. Erzeugen Sie also ein privates Attribut und eine `set`-Methode:

```
private float winkel=0;
public void setWinkel(float winkel) {
   this.winkel = winkel;
   invalidate();
}
```

Die Methode `invalidate()` teilt Android mit, dass der zuletzt gezeichnete Inhalt des Views nicht mehr aktuell ist. Der Aufruf sorgt bewirkt, dass Android bei nächster Gelegenheit dafür sorgt, dass der View neu gezeichnet wird.

Vielleicht fragen Sie sich, warum Sie das Neuzeichnen nicht selbst an dieser Stelle erledigen: Es könnte sein, dass Sie das dann zu oft tun oder vom falschen Thread aus. Der Aufruf von `invalidate()` ist der sichere Weg.

Nun zum eigentlichen Zeichnen des Views. Immer, wenn der View neu gezeichnet werden muss, ruft Android die Methode `onDraw()` auf, die Sie folglich überschreiben müssen:

```
@Override
protected void onDraw(Canvas canvas) {
}
```

Was brauchen Sie, um zu zeichnen? Zunächst einmal eine Leinwand (engl. **Canvas**). Die übergibt Ihnen Android freundlicherweise, jetzt fehlen nur noch Pinsel und Farbe. Zunächst aber sorgen Sie dafür, dass die Leinwand einen schwarzen Hintergrund erhält:

```
canvas.drawColor(Color.BLACK);
```

Bereiten Sie als Nächstes den Pinsel vor. Wir werden bei jedem Malvorgang denselben verwenden, also deklarieren Sie das nötige `Paint`-Objekt als privates Attribut:

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

Setzen Sie im Konstruktor die gewünschten Eigenschaften des Pinsels:

```
zeichenfarbe.setAntiAlias(true);
```

Antialiasing sorgt für glatte Kanten, indem »halbe Pixel« durchscheinend gezeichnet werden. Es vermeidet unschöne Treppchen-Effekte.

```
zeichenfarbe.setColor(Color.WHITE);
zeichenfarbe.setStyle(Paint.Style.FILL);
```

Alles, was Sie zeichnen, kann einen **Rand** und eine **Fläche** haben. Sie können wählen, ob der Pinsel nur den Rand (`STROKE`), nur die Fläche (`FILL`) oder beides (`FILL_AND_STROKE`) zeichnet.

Nun zurück zur `onDraw()`-Methode. Die Kompassnadel soll die größtmögliche Länge haben: Das ist je nach Orientierung des Bildschirms entweder die Höhe oder die Breite. Speichern Sie zunächst diese Werte in zwei Variablen `hoehe` und `breite`, damit der Rest übersichtlicher wird:

```
int breite = canvas.getWidth();
int hoehe = canvas.getHeight();
```

Bestimmen Sie nun den kleineren der beiden Werte als Länge der Nadel:

```
int laenge = Math.min(breite, hoehe);
```

Als Nächstes definieren wir einen Pfad. Der Pfad beschreibt den Weg, den der virtuelle Pinsel auf der Leinwand zurücklegt, um die Fläche zu umschließen, die im Auge des Betrachters wie eine Kompassnadel aussieht.

Erzeugen Sie also zunächst ein `Path`-Objekt:

```
Path pfad = new Path();
```

Jetzt wird der Pinsel geschwungen! Dazu müssen Sie dem Pfad Koordinaten nennen, zu denen der Pinsel sich bewegen soll. Der Mittelpunkt der Leinwand soll der Punkt `0,0` sein (das ist nötig, um später die Drehung einfacher zu machen). Dementsprechend hat der Punkt oben in der Mitte – also wo die Spitze der Nadel nach Norden zeigt – die Koordinaten `0,-laenge/2`.

Von dort führen wir den Pinsel nach unten, aber nicht in die Mitte, sondern etwas nach rechts, dann die gleiche Strecke nach links und zurück. So entsteht ein länglicher Keil (Abbildung 9.1).

```
Path pfad = new Path();
pfad.moveTo(0, -laenge/2);
pfad.lineTo(laenge/20, laenge/2);
pfad.lineTo(-laenge/20, laenge/2);
pfad.close();
```

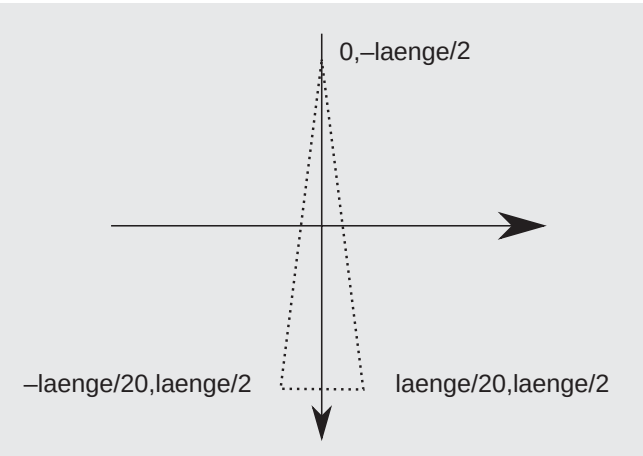



**Abbildung 9.1** Der Pinsel zeichnet die keilförmige Kompassnadel, beginnend an der Spitze, im Uhrzeigersinn.

Der erste Pinselpunkt wird mit der Methode `moveTo()` bestimmt, die beiden folgenden mit `lineTo()`. Die Methode `close()` schließt die gezeichnete Figur, indem sie den Pinsel zum Ausgangspunkt fährt.

Bevor Sie jetzt den Pfad auf die Leinwand malen, müssen Sie das Bezugssystem drehen, und zwar um den Wert im Attribut `winkel`.

```
canvas.translate(breite/2, hoehe/2);
canvas.rotate(winkel);
canvas.drawPath(pfad, zeichenfarbe);
```

Die `translate()`-Methode verschiebt den Bezugspunkt in die Mitte der Leinwand. Ansonsten würde die anschließende Drehung mit `rotate()` um die untere linke Ecke drehen, was nicht im Sinne des Erfinders ist. Die letzte Zeile schließlich zeichnet endlich die Nadel.

### View und Activity verbinden

Als letzten Schritt müssen Sie `KompassnadelView` und `KompassActivity` verknüpfen.

Erzeugen Sie in der `onCreate()`-Methode eine Instanz des Views:

```
view = new KompassnadelView(this);
```

Definieren Sie diesen View nun als (einzigen) Inhalt der Activity:

```
setContentView(view);
```

Sie haben `setContentView()` bisher immer für in XML definierte Layouts verwendet. Stattdessen können Sie, wie Sie sehen, jeden View als Parameter übergeben. Dieser wird dann als einziger Inhalt der Activity auf dem Bildschirm angezeigt.

Natürlich müssen Sie das Objekt `view` als Attribut deklarieren:

```
private KompassnadelView view;
```

Zuletzt verpassen Sie dem View den richtigen Winkel, sobald Ihnen Android einen neuen Messwert übermittelt:

```
public void onSensorChanged(SensorEvent event) {
   if (view != null) {
      view.setWinkel(-event.values[0]);
   }
}
```

Der gewünschte Winkel steckt im ersten Element des Messwerte-Arrays `event.values[]`. Das Minuszeichen ist nötig, weil der Kompasswinkel im Uhrzeigersinn definiert ist, das Canvas-Element aber andersherum dreht: gegen den Uhrzeigersinn. Das ist für Mathematiker die »natürliche« Drehrichtung. Das Minus könnten sie statt an dieser Stelle natürlich auch in der `onDraw()`-Methode des Views unterbringen.

Wenn Sie die App jetzt starten, haben Sie Ihren ersten Kompass selbst gebastelt – ganz ohne Korken, Magnetnadel und Wasserbad.

## 9.2 Wo fliegen sie denn?

Ich habe Sie bereits darauf hingewiesen, dass der Android-Kompass ein dreidimensionaler ist. Sie können daher mit seiner Hilfe die genaue Orientierung des Geräts im Raum feststellen.

Wozu das?

Sie kennen vielleicht einige Apps, deren Clou es ist, dass sie grafische Objekte ins Kamerabild einblenden, die sich mit bewegen, wenn Sie das Gerät schwenken. Genau diesen Effekt verdanken Sie dem 3-D-Kompass: Wenn das Gerät jederzeit genau weiß, in welche Richtung es schaut, können Sie ausrechnen, wo genau auf dem Bildschirm ein grafisches Objekt erscheinen muss. **Augmented Reality** heißt das zugehörige Buzzword.

Natürlich ruckelt das Ganze ein wenig, weil Sensor und Berechnungen träger sind als die Kamera oder gar der Benutzer. Wir werden trotzdem diese Chance ergreifen und die Technologie für die Darstellung eines Mückenschwarms verwenden, gegen den jedes sommerliche schwedische Seeufer einpacken kann.

Den Programmcode finden Sie auf der Buch-DVD unter dem Projektnamen *Mueckenfang4.* Sie können auch eine Kopie Ihres aktuellen Mueckenfang-Projekts verwenden; entfernen Sie jedoch alle Funktionen zur Tomaten-Erkennung und die Bewegung der Mücken. Augmented Reality *und* Bildverarbeitung – das ist zu viel der Rechenarbeit, kostet zu viel Prozessorleistung, Speicher, Strom und Hirnschmalz.

## Sphärische Koordinaten

Der Raum, in dem wir uns als Menschen bewegen, ist dreidimensional. Folglich gilt das auch für den **virtuellen Raum**, den wir mit Mücken bevölkern und dem realen überlagern werden.

**Dreidimensional** heißt zunächst nur, dass drei Zahlen erforderlich sind, um einen Punkt in diesem Raum zu definieren. Allerdings ist es möglich, verschiedene Koordinatensysteme dafür zu verwenden. Ein Beispiel ist das **kartesische Koordinatensystem**. Es verwendet drei senkrecht aufeinanderstehende Achsen x, y und z, um einen Punkt anhand seiner Abstände von einem Anfangspunkt zu definieren.

Wir werden allerdings das Kugelkoordinatensystem bevorzugen. Es hantiert nicht mit Achsen, sondern mit zwei Winkeln und einer Entfernung. Stellen Sie sich Kugelkoordinaten wie drei aufeinanderfolgende Anweisungen vor: »Drehen Sie sich 30 Grad nach links, dann schauen Sie 25 Grad schräg nach oben, und gehen Sie 8,6 Lichtjahre geradeaus, um den Stern Sirius zu erreichen.«

Der Trick an der Sache: Wir tun einfach so, als befänden sich alle Mücken im gleichen Abstand vom Spieler (bzw. vom Handy). Dadurch ist die Entfernungskoordinate immer gleich, was uns eine Menge Arbeit abnimmt. Es verbleiben zwei sphärische Koordinaten (Abbildung 9.2).

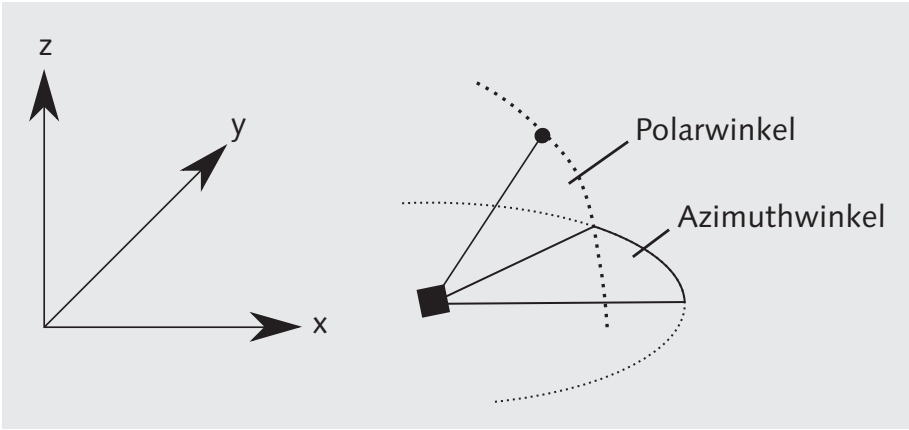



**Abbildung 9.2** Kartesische Koordinaten (links) und sphärische Koordinaten (rechts) – Azimuth- und Polarwinkel definieren den Aufenthaltsort der Mücke (Punkte), wenn die Entfernung vom Betrachter (Raute) immer gleich ist.

Wenn wir später Mücken schlüpfen lassen, benötigen wir also wie bisher zwei Zahlen, um ihren Aufenthaltsort zu definieren: den Azimuth- und den Polarwinkel:

```
int azimuth = zufallsgenerator.nextInt(360);
int polar   = zufallsgenerator.nextInt(61)-30;
```

Die beiden Werte muss sich die Mücke bis zu ihrem Tod merken, also speichern wir sie in zwei neuen Tags:

```
muecke.setTag(R.id.azimuth, new Integer(azimuth));
muecke.setTag(R.id.polar, new Integer(polar));
```

Während der Azimuth ein beliebiger Winkel zwischen 0 und 360° sein kann, sollten wir den Polarwinkel beschränken auf Werte zwischen –30° (schräg unten) und +30° (schräg oben). Sonst verrenken sich die Spieler den Hals, und das können wir nicht verantworten.

Damit die Mücke erst dann auf dem Bildschirm erscheint, wenn sie im Blickfeld ist, schalten Sie die Sichtbarkeit zunächst auf »unsichtbar«:

```
muecke.setVisibility(View.INVISIBLE);
```

Der Einfachheit halber erhalten alle Mücken in dieser Variante des Spiels dasselbe Aussehen:

```
muecke.setImageResource(R.drawable.muecke_n);
```

Wie aber können Sie jetzt die Mücken vor der virtuellen Kamera sichtbar machen?

### Die virtuelle Kamera

Wenn wir den Himmel mit virtuellen Mücken bevölkern, brauchen wir eine virtuelle Kamera, um sie zu sehen. Diese virtuelle Kamera schaut immer in dieselbe Richtung wie die eingebaute Kamera im Gerät. Wenn der Spieler das Handy schwenkt, schwenkt er gleichzeitig unsere virtuelle Kamera. Das bedeutet zwei Dinge (Abbildung 9.3):

- Wenn der Spieler sich um seine eigene Achse dreht, ändert sich der **Azimuthwinkel** der Kamera (die Himmelsrichtung).
- Wenn der Spieler das Handy auf- oder abwärts schwenkt, ändert sich der **Polarwinkel** der Kamera (die Neigung).

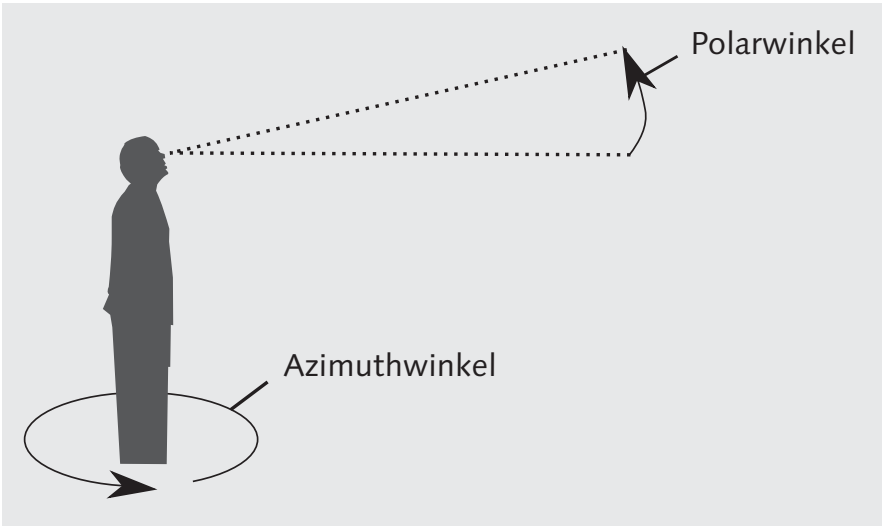



**Abbildung 9.3** Der Azimuthwinkel ist der Winkel zwischen der Nordrichtung und der Himmelsrichtung, in die der Spieler durch seine Kamera schaut. Der Polarwinkel ist der Neigungswinkel des Handys im Vergleich zur aufrechten Position.

Die beiden Winkel können wir mit den Positionswinkeln der Mücken vergleichen. Auf diese Weise kommen immer genau jene Mücken ins Bild, die in der betreffenden Richtung am virtuellen Himmel stehen. Die Kamera ist also nichts anderes als ein Ausschnitt des Himmels, begrenzt vom Bildschirmrand (Abbildung 9.4).

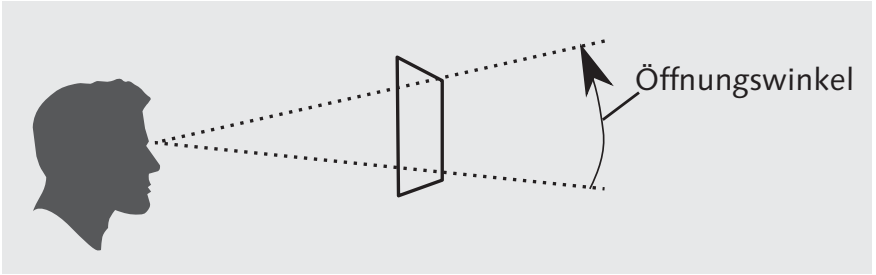



**Abbildung 9.4** Aus der Perspektive des Betrachters (links) begrenzt der Kameraausschnitt (Mitte) den sichtbaren Bereich der Welt auf einen kleinen Winkelbereich. Es gibt einen vertikalen und einen horizontalen Öffnungswinkel.

Wie groß ist dieser Ausschnitt? Nun, nicht allzu groß. Sie können ja mal schätzen oder mit einem Geodreieck nachmessen (bitte nicht ins Auge stechen). Wenn Sie's ganz genau haben wollen, teilen Sie Höhe oder Breite des Bildschirms durch den Abstand zwischen Auge und Handy (alles in Zentimetern) und berechnen mit dem nächstbesten Taschenrechner den **Arcussinus.**

Oder glauben Sie mir einfach, dass ein horizontaler Öffnungswinkel von 10° und ein vertikaler von 15° (bei hochkant gehaltenem Handy) ganz gut hinkommt.

Im Kamerabild sichtbar sind also alle Mücken, deren Azimuthwinkel zwischen –5° und +5° liegen und deren Polarwinkel zwischen –7,5° und +7,5° liegt, wenn der Spieler genau nach Norden schaut (Azimuthwinkel gleich 0) und geradeaus, nicht nach unten oder oben (Polarwinkel gleich 0).

Dreht sich der Spieler um seine Achse, sagen wir um 5° nach links, sieht er nicht mehr den virtuellen Himmel zwischen den Azimuthwinkeln –5° und +5°, sondern zwischen –10° und 0°.

Schwenkt der Spieler die Handykamera um 5° nach oben, sieht er den Himmel zwischen den Polarwinkeln –7,5°+5° = –2,5° und +7,5°+5° = 12,5°.

An welcher Stelle auf dem Bildschirm erscheinen aber nun die Mücken?

### Mücken vor der virtuellen Kamera

Immer wenn der Spieler sich mit dem Handy so bewegt, dass sich einer seiner beiden Positionswinkel ändert, müssen wir erneut prüfen, welche Mücken zu sehen sind und wo genau.

Also registrieren Sie in der Methode `onResume()` den `SensorListener`, den Sie ja schon vom Kompass kennen:

```
@Override
protected void onResume() {
   super.onResume();
   sensorManager.registerListener(this, sensor, SensorManager.
         SENSOR_DELAY_FASTEST);
}
```

`SensorManager` und `Sensor` holen Sie sich schon in `onCreate()`:

```
sensorManager = (SensorManager)getSystemService(SENSOR_SERVICE);
sensor = sensorManager.getDefaultSensor(Sensor.TYPE_ORIENTATION);
```

Die nötigen Attribute lauten wenig überraschend:

```
private SensorManager sensorManager;
private Sensor sensor;
```

Vergessen Sie nicht, den `SensorListener` wieder abzumelden, wenn die Activity inaktiv wird:

```
@Override
protected void onPause() {
   sensorManager.unregisterListener(this);
   super.onPause();
}
```

Schließlich lassen Sie die `GameActivity` das Interface `SensorEventListener` implementieren und ergänzen die nötigen Methoden:

```
@Override
public void onAccuracyChanged(Sensor arg0, int arg1) {
}
@Override
public void onSensorChanged(SensorEvent event) {
   float azimuthKamera = event.values[0];
   float polarKamera   = -90 - event.values[1];
   mueckenPositionieren(azimuthKamera, polarKamera);
}
```

Wie schon beim Kompass ist die Methode `onAccuracyChanged()` eine lästige Pflicht ohne weitere Funktion.

In `onSensorChanged()` aber spielt die Musik. Genau hier entnehmen Sie dem `SensorEvent`-Parameter die beiden Winkel, allerdings müssen Sie den Polarwinkel umrechnen, weil der Spieler das Handy aufrecht hält.

Schreiben Sie nun die Methode `mueckenPositionieren()`. Darin gehen Sie alle Mücken im Spielbereich durch – egal, ob gerade sichtbar oder nicht:

```
private void mueckenPositionieren(float azimuthKamera,
float polarKamera) {
   spielbereich = (FrameLayout) findViewById(R.id.spielbereich);
   int nummer=0;
   while(nummer<spielbereich.getChildCount()) {
      ...
      nummer++;
   }
}
```

Diese Schleifenkonstruktion kennen Sie ja bereits von allen anderen Fällen, in denen alle Mücken durchlaufen werden müssen.

Holen Sie sich jetzt innerhalb der Schleife für jede Mücke zunächst deren virtuelle Position aus den Tags:

```
ImageView muecke = (ImageView) spielbereich.getChildAt(nummer);
int azimuth = (Integer) muecke.getTag(R.id.azimuth);
int polar  = (Integer) muecke.getTag(R.id.polar);
```

Berechnen Sie dann die Position relativ zu den Winkeln der virtuellen Kamera:

```
float azimuthRelativ = azimuth - azimuthKamera;
float polarRelativ = polar - polarKamera;
```

Es ist ganz wichtig, dass Sie ab hier mit `float`-Werten arbeiten. Bei einer Bildschirmbreite, die 10° entspricht, gäbe es sonst nur zehn Spalten, in denen Mücken erscheinen könnten – ein furchtbares Hin- und Herspringen wäre die Folge.

Es folgt eine Fallunterscheidung: Mücken, die gerade nicht im Bildfeld sind, können Sie unsichtbar machen, den Rest müssen Sie anzeigen:

```
if(istMueckeInKamera(azimuthRelativ, polarRelativ)) {
   ...
   muecke.setVisibility(View.VISIBLE);
} else {
   muecke.setVisibility(View.GONE);
}
```

Schreiben Sie zunächst die Methode `istMueckeInKamera()`:

```
private boolean istMueckeInKamera(float azimuthDifferenz,
float polarDifferenz) {
   return (Math.abs(azimuthDifferenz) <= KAMERABREITE_AZIMUTH/2)
     && (Math.abs(polarDifferenz) <= KAMERABREITE_POLAR/2);
}
```

Die Methode bestimmt die Absolutwerte der relativen Winkel und vergleicht sie mit der Breite der virtuellen Kamera. Mücken sind sichtbar, wenn sowohl azimuthaler als auch polarer Winkel klein genug sind.

Sobald Sie wissen, welche Mücken im Bildausschnitt sichtbar sein müssen, können Sie ausrechnen, wo sie im Layout zu positionieren sind. Das ist mathematisch nichts anderes als Dreisatz: Wir wissen, dass zwischen linkem und rechtem Bildschirmrand 10° Azimuth liegen. 10° entsprechen also in der Horizontalen genau der Bildschirmbreite in Pixeln. Vertikal entspricht der Öffnungswinkel von 15° dementsprechend der Bildschirmhöhe in Pixeln.

Spätestens jetzt haben Sie zumindest im Kopf schon Konstanten für die beiden Öffnungswinkel angelegt, denn die Werte werden offenbar an mehreren Stellen benötigt und müssen später vielleicht justiert werden:

```
public static final int KAMERABREITE_AZIMUTH = 10;
public static final int KAMERABREITE_POLAR   = 15;
```

Sie müssen also nur noch die Differenz von Mücken-Azimuthwinkel und Kamera-Azimuthwinkel in Pixel umrechnen. Dann dasselbe für den Polarwinkel (Abbildung 9.5), alles bezogen auf den Mittelpunkt des Spielbereichs. Eine kleine Korrektur ist noch nötig, weil wir zum Zeichnen die linke obere Ecke des Mückenbildchens benötigen und nicht den Mittelpunkt.

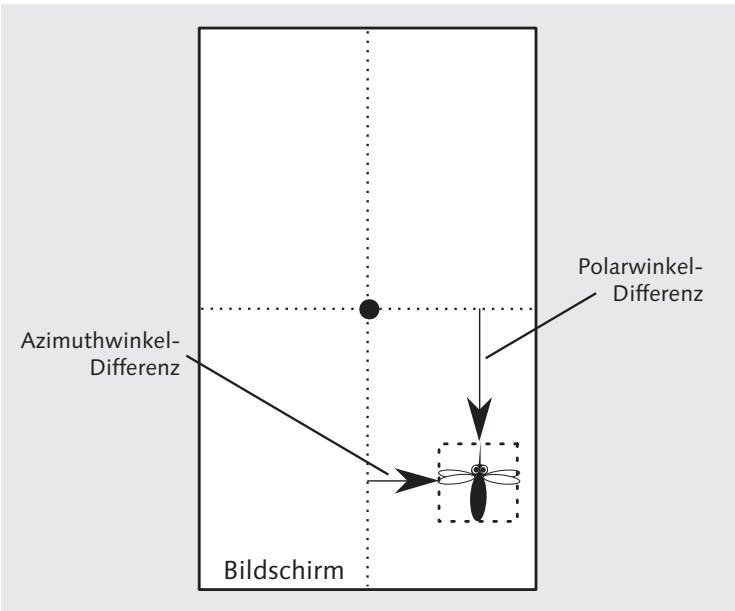



**Abbildung 9.5** Die Winkel-Differenzen zwischen Kamera- und Mückenkoordinaten lassen sich leicht in Bildschirmkoordinaten umrechnen.

Das Resultat packen Sie in die `LayoutParams` des ImageViews der Mücke:

```
FrameLayout.LayoutParams params = (android.widget.FrameLayout.
 LayoutParams) muecke.getLayoutParams();
params.leftMargin =  spielbereich.getWidth()/
 2 + Math.round(spielbereich.getWidth()*azimuthRelativ/
 KAMERABREITE_AZIMUTH) -muecke.getWidth()/2;
params.topMargin = spielbereich.getHeight()/2 -
 Math.round(spielbereich.getHeight()*polarRelativ/
KAMERABREITE_POLAR) - muecke.getHeight()/2;
muecke.setLayoutParams(params);
```

Wenn Sie das Spiel jetzt ausprobieren, werden Sie sehen, dass es ganz schön schwierig ist, die Mücken zu finden. Sie hören überall das Summen, aber Sie finden die Mücken eher per Zufall. Als erste Maßnahme stellen Sie die Lebensdauer der Mücken auf 60 Sekunden, denn wenn man mal so ein Vieh gefunden hat und es verschwindet, bevor man es zerquetschen kann, ist man schnell frustriert.

Trotzdem bleibt das Spiel sehr anstrengend. Das liegt vor allem daran, dass der Kompass ziemlich träge ist: Wenn der Spieler sich in eine bestimmte Richtung dreht und dort nichts sieht, dreht er sich schnell weiter – dabei war der Kompass noch gar nicht damit fertig, den richtigen Winkel zu bestimmen.

Was da hilft, ist ein Radar. Stellen Sie sich vor, Sie wären eine Fledermaus und würden Ultraschall-Rufe aussenden, die von den Mücken zurückgeworfen werden. Mit Ihrem inneren Auge könnten Sie dann in jede Richtung schauen – und sich in die richtige Richtung drehen.

Daher werden wir als Nächstes auf Basis des `KompassViews` aus dem vorangegangenen Abschnitt einen Radarschirm konstruieren, der die Mücken in der Umgebung anzeigt.

### Der Radarschirm

Im Wesentlichen basiert der Radarschirm auf den grafischen Hilfmitteln, die Sie bereits beim Zeichnen des Kompasses kennengelernt haben. Ausgangspunkt ist wiederum eine Klasse, die von `View` erbt. Allerdings wird der Radarschirm weder den ganzen Bildschirm ausfüllen noch der einzige View sein, daher müssen wir auf einige Vereinfachungen verzichten.

Der neue `RadarView` wird mithilfe des Layout-Editors in *game.xml* eingebaut, und damit das funktioniert, benötigt er einen zusätzlichen Konstruktor:

```
public RadarView(Context context, AttributeSet attrs) {
   super(context,attrs);
}
```

Der `AttributeSet`-Parameter enthält die Einstellungen, die Sie im Layout-Editor vornehmen. Wir müssen uns zwar nicht um sie kümmern, aber Android muss sie kennen – daher funktioniert der View nicht ohne diesen Konstruktor, der den zugehörigen Elternkonstruktor aufruft.

Der Radarschirm benötigt zwei Zeichensstifte: einen, um einen Kreis zu zeichnen, an dem sich der Spieler orientieren kann, und einen zweiten, der die Mücken in einer anderen Farbe malt.

Um die beiden Zeichenstifte nicht in beiden Konstruktoren zusammenzubauen, schreiben wir eine gemeinsame `init()`-Methode, die jeweils aufgerufen wird:

```
public RadarView(Context context) {
   super(context);
   init();
}

public RadarView(Context context, AttributeSet attrs) {
   super(context,attrs);
   init();
}
```

Was muss die `init()`-Methode nun alles leisten?

Damit die Linien, die der Radar zeichnen wird, auf allen Geräten gleich dick aussehen, beschaffen wir uns zunächst den Bildschirm-Maßstab:

```
massstab = getResources().getDisplayMetrics().density;
```

Initialisieren Sie nun die beiden Zeichenstifte, einen für die Hilfslinien und einen für die Mücken:

```
mueckenfarbe.setAntiAlias(true);
mueckenfarbe.setColor(Color.RED);
mueckenfarbe.setStyle(Paint.Style.STROKE);
mueckenfarbe.setStrokeWidth(5*massstab);
linienfarbe.setAntiAlias(true);
linienfarbe.setColor(Color.WHITE);
linienfarbe.setStrokeWidth(1*massstab);
linienfarbe.setStyle(Paint.Style.STROKE);
```

Sie sehen, dass der `massstab` bei der Bestimmung der Linienbreite zum Einsatz kommt. Da Handy-Bildschirme unterschiedlich hohe Pixelauflösungen besitzen, sorgt dies für annähernd gleich breite Linien.

Nutzen Sie die Programmierhilfe von Eclipse (Strg + 1), um die fehlenden Attribute in die Klasse einzufügen:

```
private Paint mueckenfarbe = new Paint();
private Paint linienfarbe = new Paint();
private float massstab;
```

Wenn Sie gerade dabei sind, fügen Sie zwei weitere Attribute hinzu, die der Radarschirm braucht, um zu funktionieren: den Winkel zur Nordrichtung und das FrameLayout, das die Mücken enthält. Aus diesen Objekten kann der `RadarView` dann alle nötigen Informationen auslesen, um den Schirm zu zeichnen.

```
private float winkel=0;
private FrameLayout container;
```

Die `GameActivity` wird später diese Attribute setzen müssen: den Winkel ständig und den `container` einmal zu Beginn. Schreiben Sie also die nötigen Setter-Methoden.

```
public void setContainer(FrameLayout container) {
   this.container = container;
}
public void setWinkel(float winkel) {
   this.winkel = winkel;
   invalidate();
}
```

Die Methode `setWinkel()` unterrichtet die Elternklasse außerdem darüber, dass bei nächster Gelegenheit die Grafik neu zu zeichnen ist. Apropos zeichnen:

```
@Override
protected void onDraw(Canvas canvas) {
   canvas.drawColor(getResources().getColor(R.color.transgrau));
   if(container==null) return;
   ...
}
```

Wie Sie sehen, zieht die Methode die Reißleine, wenn der `container` noch nicht gesetzt wurde. In dem Fall wird lediglich der durchscheinende Hintergrund gemalt. Dessen Farbe entnimmt obige Implementierung übrigens aus der `color.xml`-Resource-Datei. Sie sehen, dass es ungemein praktisch ist, überall auf dieselben Ressourcen zugreifen zu können. Die Farbe `R.color.transgrau` ist definiert als `#88888888`, also halb durchsichtiges Grau. Sie können natürlich jede Farbe verwenden, die Ihnen besser gefällt.

Für die folgenden Zeichenoperationen ist es wie schon beim Kompass sinnvoll, sich Breite, Höhe und Radius in lokalen Variablen zu merken:

```
int breite = getWidth();
int hoehe = getHeight();
int radius = Math.min(breite, hoehe)/2;
```

Zeichnen Sie nun einen Kreis, um den Radarbereich zu begrenzen:

```
canvas.drawCircle(breite/2, hoehe/2, radius, linienfarbe);
```

Die ersten beiden Parameter bestimmen den Mittelpunkt des Kreises, der dritte den Radius. Dieser Kreis wird also den Zeichenbereich des `RadarViews` bestmöglich ausfüllen.

Als Nächstes zeigen wir dem Spieler, welchen Ausschnitt des Radars er auf dem Bildschirm sieht. Wir hatten ja in der `GameActivity` 20° als sichtbare Breite festgelegt, also befehlen Sie dem Canvas-Element nun, ein Kuchenstück mit diesem Öffnungswinkel zu zeichnen:

```
canvas.drawArc(new RectF(0,0,breite,hoehe),-
100, 20, true, linienfarbe);
```

Leider sind die Parameter für die `drawArc()`-Methode nicht so übersichtlich wie beim Kreis. Anstelle eines Mittelpunkts müssen Sie ein Rechteck angeben, das die vollständige Ellipse umschließt, von der Sie einen Ausschnitt zeichnen möchten. Solange Breite und Höhe gleich sind, reden wir hier natürlich von einem Quadrat und einem Kreis.

Knifflig ist Parameter Nummer 2: Er definiert den Anfangswinkel des zu zeichnenden Kreisbogens, allerdings gerechnet von einer Stelle ganz rechts außen – also im Osten. Intuitiv richtig ist es aber, den Ausschnitt nach oben zu zeichnen, also –90° vom Ausgangspunkt entfernt.

Damit der Ausschnitt an der richtigen Stelle erscheint, muss er also bei –90° – 10° = –100° beginnen. Die Länge von 20° übergeben Sie als dritten Parameter (Abbildung 9.6).

Der vorletzte Parameter befiehlt der Methode, nicht nur den Kreisbogen zu zeichnen, sondern auch den Umriss eines Kuchenstücks.

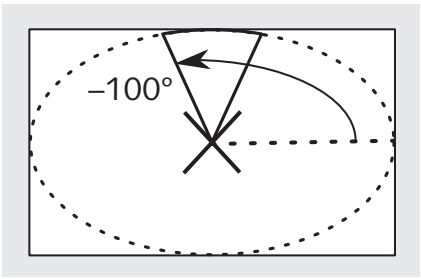



**Abbildung 9.6** Die Methode »drawArc()« verwendet als Bezugspunkte das äußere, begrenzende Rechteck und die Ostrichtung als Nullpunkt der Winkelangabe. Deshalb beginnt der Kreisbogen bei einem Winkel von –100° (Pfeil).

Um anstrengende trigonometrische Übungen zu vermeiden, zeichnen wir auch die Mücken als Kreisbögen – das ist die einfachste Möglichkeit, den uns bekannten Azimuthwinkel in eine grafische Darstellung umzumünzen. Die Kreisbögen werden dabei so kurz, dass sie wie Punkte aussehen. Die etwas größere Dicke des Zeichenstiftes `mueckenfarbe` sorgt für eine gute Sichtbarkeit.

Um die Mücken zu zeichnen, holen Sie sich wie schon an diversen Stellen in der `GameActivity` die Kinder des Spielbereichs:

```
int nummer = 0;
while(nummer < container.getChildCount()) {
   ImageView muecke = (ImageView) container.getChildAt(nummer);
   nummer++;
}
```

Sobald Sie den ImageView haben, ermitteln Sie zunächst aus dem Tag den Azimuthwinkel der Mücke :

```
float azimuth = (Integer)muecke.getTag(R.id.azimuth);
```

Bleibt als Letztes der Aufruf der Zeichenmethode, die gleichzeitig eine der kompliziertesten Programmzeilen in diesem Buch ist:

```
canvas.drawArc(new RectF(breite*0.1f, hoehe*0.1f, breite*0.9f, hoehe
*0.9f), azimuth + winkel-90, 5, false, mueckenfarbe);
```

Das begrenzende Rechteck ist diesmal nicht ganz so groß wie der View – sonst würden die Mücken auf dem weißen Außenkreis landen. Also schneiden wir außen 10 % Breite und Höhe ab. Wie Sie sehen, tragen die Faktoren `0.1f` und `0.9f` ihren nativen Datentyp – `float` – als Postfix `f` mit sich herum. Würden Sie die Buchstaben `f` weglassen, würde Java die Zahlen als `double` auffassen. Der Konstruktor `RectF` erwartet aber `float`, nicht `double`.

Der Startwinkel des Kreisbogens ist der Azimuth der Mücke, korrigiert um den Drehwinkel gegenüber der Nordrichtung (`winkel`) und die obligatorischen –90°, weil `drawArc()` gerne im Osten mit dem Zeichnen beginnen würde, wir aber im Norden. Die Zahl `5` ist die Winkellänge des Kreisbogens, 5° also. Das `false` verhindert, dass `drawArc()` ein ganzes Tortenstück zeichnet, und der letzte Parameter, der Zeichenstift, ist der einzige, der keine Erklärung erfordert.

Als Nächstes müssen Sie den `RadarView` ins Layout *game.xml* einfügen (Abbildung 9.7).

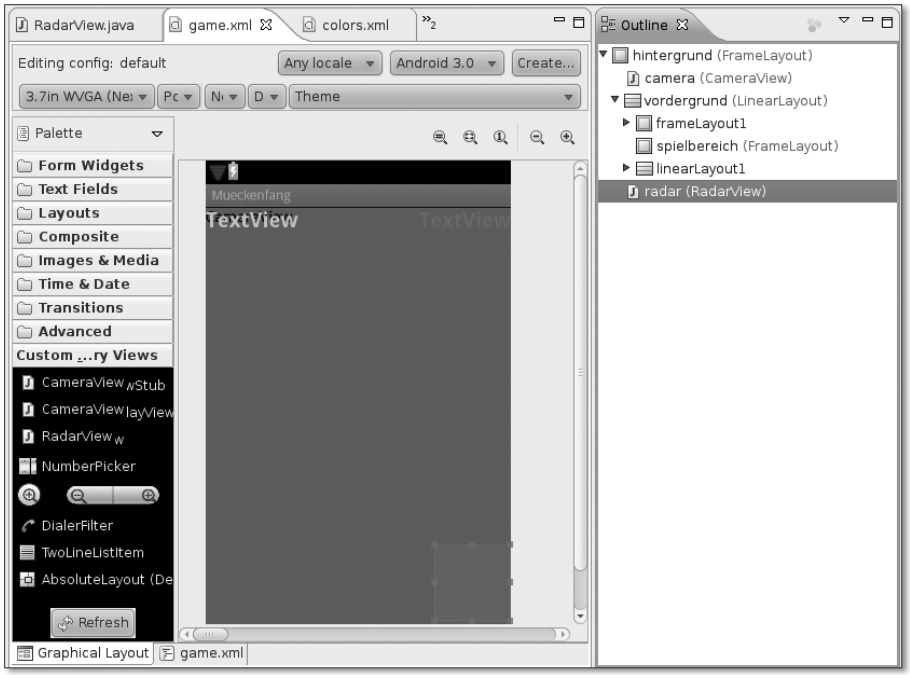



**Abbildung 9.7** Fügen Sie den »RadarView« mit dem Layout-Editor hinzu.

Sie finden selbst gebaute Views im Layout-Editor auf der Palette in der Rubrik Custom Views. Achten Sie darauf, dass Sie den `RadarView` als direktes Kind des Hintergrund-FrameLayouts anlegen. Setzen Sie Breite und Höhe auf `80dp`, die Layout-Gravitation auf `bottom` und `left` (oder `right`, wenn Sie Rechtshänder sind, Sie werden beim Spielen merken, warum) und den unteren Rand (`margin bottom`) auf `50dp`, damit das Radar nicht die Punkteleisten überdeckt. Setzen Sie die ID des Views auf `radar`.

Jetzt können Sie in der `GameActivity` in der `onCreate()`-Methode eine Referenz auf den `RadarView` in einem neuen Attribut namens `radar` speichern:

```
radar = (RadarView) findViewById(R.id.radar);
```

Geben Sie dem Radar bekannt, wo es die Mücken finden kann:

```
radar.setContainer(spielbereich);
```

Und zu guter Letzt teilen Sie dem Radar in `onSensorChanged()` die aktuelle Blickrichtung des Spielers mit:

```
radar.setWinkel(-event.values[0]);
```

Das Minuszeichen dient wie schon beim Positionieren der Mücken dazu, die Drehrichtungen von Kompass und Geometrie anzugleichen.

Wenn Sie das Spiel jetzt ausprobieren, werden Sie sehen, dass es deutlich einfacher (und interessanter) geworden ist, auf Mückenjagd zu gehen. Es kostet etwas Übung, das Handy beim Antippen einer Mücke nicht zu verdrehen, sodass das Ziel aus dem Bildschirm verschwindet. Das ist sehr irritierend: Die Mücke ist weg, aber nicht, weil man sie getroffen hat – und im Radar ist sie noch da.

Wenn Sie mögen, können Sie das ganze Spiel auf Querformat umbauen – das erleichtert die Suche deutlich, weil in horizontaler Richtung mehr Platz auf dem Bildschirm ist.

Man kann sich außerdem ziemlich komplizierte Methoden überlegen, um die scheinbare Bewegung der Mücken auf dem Bildschirm vom Ruckeln zu befreien – aber glauben Sie mir, am Ende steht und fällt die Spielbarkeit mit der Geschwindigkeit des Sensors.

Haben Sie jetzt genug Mücken gejagt? Okay, dann lassen Sie uns den kleinen Vampiren (oder ihren zerdrückten Überbleibseln) den Rücken kehren und einen weiteren Sensor des Smartphones für eine ganz andere Art von App heranziehen.

## 9.3 Beschleunigung und Erschütterungen

Nehmen Sie sich einen dünnen Faden, und binden Sie ein nicht zu leichtes Objekt daran fest (Handy, Hantel, Hund). Wenn Sie den Faden dann am oberen Ende festhalten, hängt das Objekt am anderen Ende senkrecht nach unten. Ich verrate Ihnen sicher nichts Neues, wenn ich Ihnen sage, dass dies eine Auswirkung der Erdanziehungskraft ist. Wenn Sie gerade nichts anderes als Ihren Hund für das Experiment finden, verzichten Sie vielleicht lieber drauf, weil das Ergebnis evident ist.

Aber setzen Sie sich mal mit Ihrem Fadenpendel in einen Zug, und beobachten Sie, was passiert, wenn der losfährt oder bremst. Auch dann wird das Pendel ausgelenkt, und zwar entgegen der Beschleunigungsrichtung. Während der Fahrt geradeaus mit gleichbleibender Geschwindigkeit bleibt das Pendel dagegen unbeeinflusst.

Fahren Sie mit einem Karussel, wird das Pendel nach außen gelenkt, und zwar von der Zentrifugalkraft, die umso größer ist, je schneller sich das Karussel dreht. Jede Kraft entspricht einer Beschleunigung, und das Accelerometer kann sie messen – in jede der drei Richtungen des Raums.

Jetzt stellen Sie sich ein solches Pendel vor, das ein bisschen kleiner ist – sagen wir, ein paar Mikrometer groß – und in Ihrem Handy eingebaut ist. Dann wissen Sie,

wie der Beschleunigungssensor prinzipiell funktioniert. Außerdem liegen sofort einige Anwendungsfälle auf der Hand: Der Sensor weiß immer genau, wo unten ist. Deshalb kann er schon kleinste Neigungen registrieren. So funktionieren Spiele, bei denen das Handy als Lenkrad dient, und auch die beliebten Kugel-Labyrinthe. Gar nicht zu reden von dem Bierglas, das sich beim Neigen leert (»iBeer«).

Der Beschleunigungssensor ist empfindlich genug, um auch leichte Erschütterungen zu registrieren. Wenn die App die Sensorwerte geschickt ausliest, kann sie damit Schritte zählen oder die Geräusche eines Percussion-Instruments simulieren.

Bevor wir versuchen, das Handy in eine Rumba-Rassel zu verwandeln, lassen Sie uns mit einem einfachen Beispiel beginnen und einen Schrittzähler bauen.

## Ein Schrittzähler

Die einfachste Form des Schrittzählers zählt Erschütterungen einer gewissen Intensität und zeigt deren Anzahl auf dem Bildschirm an. Legen Sie zunächst ein neues Android-Projekt an, und nennen Sie es **Schrittzaehler** (bitte vermeiden Sie den Umlaut ä). Ändern Sie im vom Wizard angelegten Layout *main.xml* die Schriftgröße und ID des TextViews, und fügen Sie einen ZURÜCKSETZEN-Button hinzu (Abbildung 9.8).

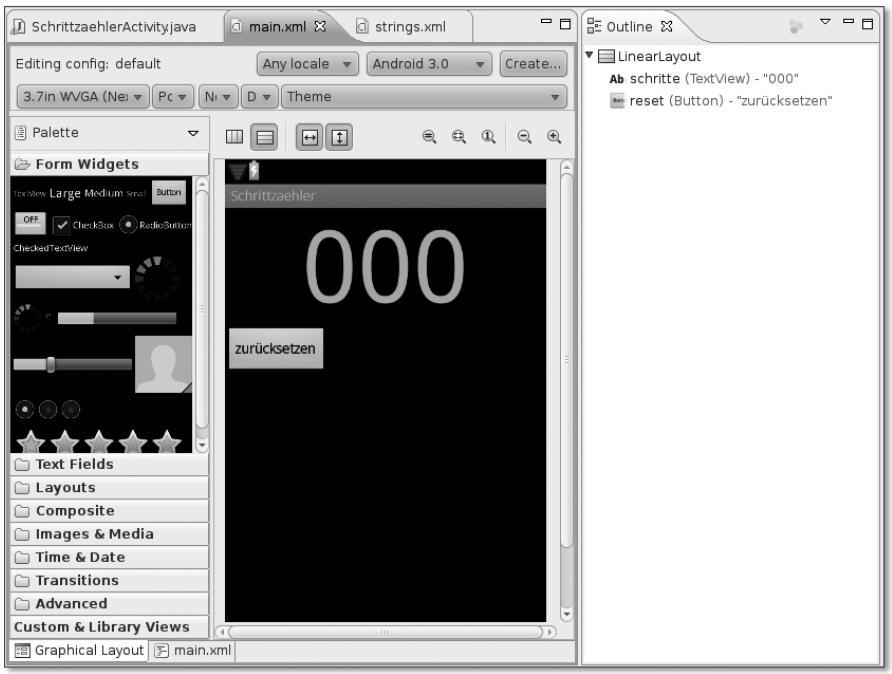



**Abbildung 9.8** Das Layout für den Schrittzähler ist denkbar einfach.

Im Android-Manifest tragen Sie für die SchrittzaehlerActivity als Screen Orientation Portrait ein. Würden Sie das nicht tun, würde die Orientierung des Bildschirms automatisch wechseln. Dabei würde Android jedes Mal die Activity neu starten, und im folgenden einfachen Beispiel wäre das überaus ungünstig, weil die Schrittzählung wieder bei 0 beginnen würde.

Treffen Sie zunächst in der Activity die nötigen Vorbereitungen, um den Sensor abzufragen und die Schrittzahl darstellen zu können.

Holen Sie sich in der `onCreate()`-Methode Referenzen auf den `SensorManager` und den richtigen Sensor:

```
sensorManager = (SensorManager) getSystemService(Context.SENSOR_
SERVICE);
sensor = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
```

Die zugehörigen Attribute erzeugt Ihnen Eclipse, indem Sie die praktische Tastenkombination Strg + 1 drücken.

```
private SensorManager sensorManager;
private Sensor sensor;
```

Um Schritte zu zählen, brauchen wir ein simples `int`-Attribut:

```
privat int schritte=0;
```

Führen Sie ein Attribut für den TextView ein, der die Schritte anzeigt:

```
private TextView textView;
```

Setzen Sie in der `onCreate()`-Methode dieses Objekt auf die Referenz zum richtigen View im Layout:

```
textView = (TextView) findViewById(R.id.schritte);
```

Schreiben Sie eine kleine Methode namens `aktualisiereAnzeige()`, die den aktuellen Wert von `schritte` auf den Bildschirm bringt:

```
private void aktualisiereAnzeige() {
   textView.setText(Integer.toString(schritte));
}
```

Rufen Sie diese Methode am Ende von `onCreate()` einmal auf, um sicherzustellen, dass beim Start der App 0 Schritte angezeigt werden.

Verbinden Sie den Zurücksetzen-Button mit einer `onClick()`-Methode, indem Sie die Activity das Interface `OnClickListener` implementieren lassen:

```
public class SchrittzaehlerActivity extends Activity implements
OnClickListener
...
Button b = (Button) findViewById(R.id.reset);
b.setOnClickListener(this);
...
@Override
public void onClick(View v) {
   if(v.getId() == R.id.reset) {
      schritte=0;
      aktualisiereAnzeige();
   }
}
```

Die `onClick()`-Methode prüft der Vollständigkeit halber, ob tatsächlich der ZURÜCKSETZEN-Button gedrückt wurde (man kann ja nie wissen), und setzt die Schritte dann zurück.

Damit ist die Activity im Großen und Ganzen implementiert, allerdings fehlt noch die gesamte Funktionalität, um die Erschütterungen festzustellen. Darum kümmern wir uns als Nächstes.

### Mit dem SensorEventListener kommunizieren

Um die Erschütterungen zu messen, schreiben wir der Übersicht halber eine separate Klasse, die gleichzeitig die Rolle des `SensorEventListeners` übernimmt, also das zugehörige Interface implementiert:

```
package de.androidnewcomer.schrittzaehler;
public class ErschuetterungListener implements SensorEventListener {
   @Override
   public void onAccuracyChanged(Sensor s, int a) {
   }
   @Override
   public void onSensorChanged(SensorEvent event) {
   }
}
```

Wie üblich kümmern wir uns nicht um `onAccuracyChanged()`-Ereignisse. Die Methode `onSensorChanged()` wird später versuchen, an den Sensordaten zu erkennen, ob ein Fuß des Handy-Trägers auf den Boden aufgetroffen ist. In dem Fall muss der `ErschuetterungListener` die `SchrittzaehlerActivity` darüber informieren, damit die die angezeigte Schrittzahl um 1 erhöht. Es gibt unterschiedliche Möglichkeiten, diese Kommunikation zu realisieren, wir wählen diesmal einen Handler.

Sie kennen ja bereits diese magischen Objekte, mit denen man sogar die Kommunikation über die gefährlichen Grenzen von Threads hinweg hinbekommt. In diesem Fall wird die Activity den Handler verwalten und eine Referenz an den Listener übergeben. Die Activity wiederum wird auf die via Handler eingehende Erschütterungsnachricht (**Message**) reagieren, indem sie die Anzeige aktualisiert (Abbildung 9.9).

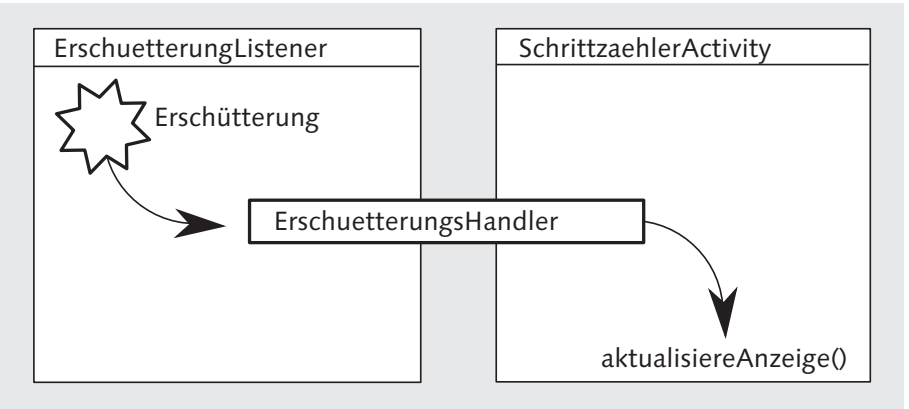



**Abbildung 9.9** Der »ErschuetterungListener« benutzt den »ErschuetterungsHandler«, um die »SchrittzaehlerActivity« über einen zurückgelegten Schritt zu unterrichten.

Das geht nicht mit der Standard-Version der Klasse `Handler`. Daher benötigen wir eine von `Handler` abgeleitete Klasse, die die Methode `handleMessage()` überschreibt und die Schrittzahl erhöht:

```
private class ErschuetterungsHandler extends Handler {
   @Override
   public void handleMessage(Message msg) {
      schritte++;
      aktualisiereAnzeige();
   }
}
```

Der eigentliche Inhalt der Nachricht `msg` interessiert uns nicht – es genügt, *dass* eine Nachricht kommt. In anderen Anwendungsfällen könnte es sein, dass das `Message`-Objekt zusätzliche Daten enthält.

`ErschuetterungsHandler` ist eine lokale Klasse innerhalb der `Schrittzaehler-Activity`. So kann sie leicht auf die Attribute und Methoden der umschließenden Klasse zugreifen. Erzeugen Sie in der Activity ein Attribut, das Sie gleich mit dem eigenen Handler füllen:

```
private ErschuetterungsHandler handler = new ErschuetterungsHandler();
```

Bisher kennt der `ErschuetterungListener` den Handler noch nicht. Da der genau einmal gesetzt werden muss, bietet es sich an, ihn dem Listener als Konstruktor-Parameter zu übergeben:

```
public class ErschuetterungListener implements SensorEventListener {
private Handler handler;
   public ErschuetterungListener(Handler h) {
      handler = h;
   }
   ...
}
```

Jetzt können Sie in der Activity ein Attribut für den Listener erzeugen und ihn in der Methode `onResume()` mit dem Beschleunigungssensor verdrahten:

```
private ErschuetterungListener listener = new ErschuetterungListener
(handler);
...
@Override
protected void onResume() {
super.onResume();
   sensorManager.registerListener(listener, sensor,
         SensorManager.SENSOR_DELAY_GAME);
}
```

Vergessen Sie nicht, den Listener in `onPause()` wieder abzumelden:

```
@Override
protected void onPause() {
   sensorManager.unregisterListener(listener);
   super.onPause();
}
```

Jetzt fehlt nur noch das eigentliche Erkennen eines Schritts.

### Schritt für Schritt

Es gibt eine ganze Reihe möglicher Herangehensweisen, an den Sensordaten einen Schritt zu erkennen. Wenn Sie die eingangs dieses Buches vorgestellte App Tricorder installiert haben, können Sie ja mal den Beschleunigungssensor anschalten und beobachten, wie eine Kurve mit verschieden starken Ausschlägen entsteht, wenn Sie durch die Gegend gehen oder laufen (Abbildung 9.10).

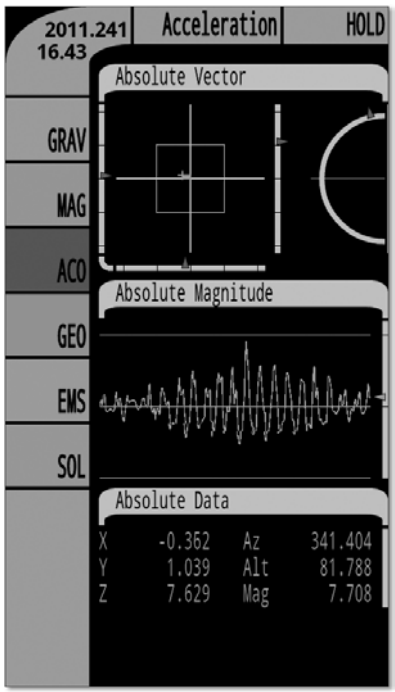



**Abbildung 9.10** Der Tricorder zeigt die Messkurve des Beschleunigungssensors während des Gehens, Joggens und des erneuten Gehens.

Sie sehen, dass beim Joggen eine hübsche Sinuskurve entsteht. Das liegt daran, dass das Handy wie der gesamte Körper relativ gleichmäßig auf und ab bewegt wird. Beim Gehen sind die Ausschläge deutlich geringer – und dementsprechend schwerer zu identifizieren. Vermutlich könnte man zu diesem Thema ganze Bücher füllen, aber ich möchte deren Autoren ungern die Arbeit abnehmen. Deshalb werde ich Ihnen eine relativ einfache Methode vorstellen, die sich nach der Auf-und-ab-Bewegung richtet.

Die Herausforderung an der Erkennung eines Schrittes ist es, nicht zu viele und nicht zu wenige Ereignisse zu produzieren. Das ist schwierig, weil der Beschleunigungssensor sehr empfindlich ist. Wir können nicht einfach jeden zweiten Schnittpunkt mit der 0-Achse als Schritt interpretieren – das Rauschen würde uns pro Sekunde mehr Phantom-Schritte bescheren, als ein realer Mensch in einer Stunde laufen kann.

Der Trick besteht nun darin, auf die zu einem Schritt gehörenden Ereignisse *der Reihe nach* zu warten. Zunächst muss die Beschleunigung in Richtung $x$ einen gewissen Schwellwert *über*schreiten (Körper und Handy beschleunigen aufwärts). Dann warten wir darauf, dass der Messwert einen gewissen, negativen

Schwellwert *unter*schreitet (abwärts). Sobald das geschieht, ist der Schritt vollständig. Erst dann warten wir wieder auf die nächste Überschreitung des Schwellwerts (Abbildung 9.11). Zur Unterscheidung, ob der Schritt begonnen hat oder nicht, dient ein `boolean`-Attribut namens `schrittBegonnen`.

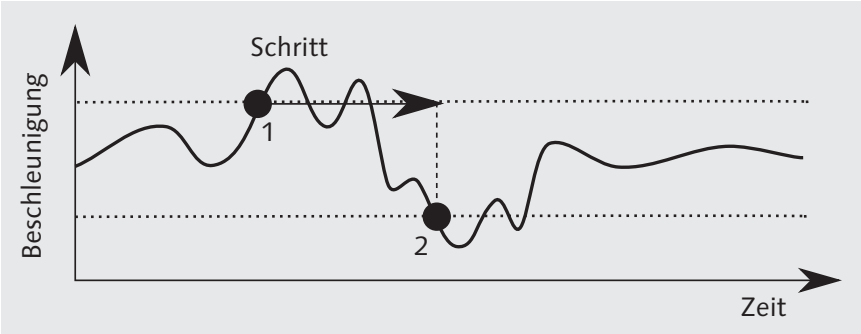



**Abbildung 9.11** Der Schritt gilt als begonnen, sobald der obere Schwellwert überschritten wird (erster Punkt), und als beendet, sobald der untere Schwellwert unterschritten wird (zweiter Punkt). Beachten Sie, dass unsere Logik die anderen Schnittpunkte mit den Schwellwerten ignoriert.

In Java-Code sieht die Schritterkennung so aus:

```
@Override
public void onSensorChanged(SensorEvent event) {
   float x = event.values[0];
   if(schrittBegonnen) {
      if(x > schwellwert) {
         schrittBegonnen=false;
         handler.sendEmptyMessage(1);
      }
   } else {
      if(x < -schwellwert) {
         schrittBegonnen = true;
      }
   }
}
```

Als Standardwert für die Schwellwert-Konstante funktioniert der Wert 1,0 ganz gut:

```
private static final float SCHWELLWERT = 1.0f;
private float schwellwert = SCHWELLWERT;
```

Es kann gut sein, dass wir dem Benutzer erlauben wollen, diesen Wert zu justieren. Deshalb fügen Sie der Klasse vorsichtshalber schon mal einen Setter hinzu:

```
public void setSchwellwert(float wert) {
   schwellwert = wert;
}
```

Probieren Sie die App jetzt aus. Vergessen Sie nicht, das USB-Kabel zu entfernen, bevor Sie mit dem Handy durch die Wohnung laufen. Alternativ können Sie das Handy schütteln oder mit der freien Hand anstoßen, um das Zählen der Schritte zu testen.

Bevor Sie jetzt mit Ihrer App eine Runde joggen gehen, muss ich Sie auf einen fatalen Nachteil hinweisen: Sobald die Activity nicht mehr aktiv ist, hört sie auf zu zählen. Das liegt daran, dass wir den Listener in der `onPause()`-Methode abmelden. Natürlich könnten Sie das einfach bleiben lassen – elegant ist diese Lösung aber nicht. Deshalb werde ich Ihnen als Nächstes erklären, wie Sie das Zählen der Schritte im Hintergrund ablaufen lassen können.

## 9.4 Hintergrund-Services

Bisher haben Sie jegliche Programmlogik in Activities verpackt. Android stellt allerdings noch andere Möglichkeiten bereit. In diesem Abschnitt geht es um Services: Dienste, die im Hintergrund laufen können, mehr oder weniger unabhängig von der Benutzeroberfläche.

Normalerweise laufen auf einem Android-Handy diverse Services, die Sie gar nicht bemerken. Öffnen Sie die Einstellungen-App, und wählen Sie Applikationen verwalten, um auf der Registerkarte Ausgeführte die laufenden Dienste zu sehen. Sie sehen dort vermutlich die magischen Google Services, außerdem den Market und vielleicht ein E-Mail-Programm.

Services können durchaus mit der Benutzeroberfläche interagieren. Der Market-Service blendet beispielsweise eine Benachrichtigung ein, wenn er Updates für installierte Apps gefunden hat.

Für viele Anwendungen ist es von Vorteil oder sogar unbedingt nötig, die ganze Zeit aktiv zu sein, aber nur bei Bedarf sichtbar zu werden. Der Schrittzähler ist ein gutes Beispiel, daher werden wir ihn jetzt in einen Service verwandeln.

### Eine Service-Klasse

Wenn Sie eine eigene Activity erzeugen, schreiben Sie eine Klasse, die von Androids Basisklasse `Activity` erbt. Eigene Service-Klassen erben dagegen von der Basisklasse `Service`.

Erzeugen Sie eine neue Klasse `SchrittzaehlerService`, und verpassen Sie ihr alle Attribute, die bisher in der Activity verwendet wurden. Verschieben Sie auch den `ErschuetterungsHandler` hinüber in den Service:

```
public class SchrittzaehlerService extends Service {
   private SensorManager sensorManager;
   private Sensor sensor;
   private ErschuetterungsHandler handler =
           new ErschuetterungsHandler();
   private ErschuetterungListener listener =
           new ErschuetterungListener(handler);
   private int schritte=0;
   private class ErschuetterungsHandler extends Handler {
      @Override
      public void handleMessage(Message msg) {
         schritte++;
      }
   }
   @Override
   public IBinder onBind(Intent intent) {
      return null;
   }
}
```

Die Methode `onBind()` müssen Sie schreiben, weil sie in der Basismethode als `abstract` definiert ist. Das ähnelt dem Fall, wenn eine Klasse das Interface `SensorEventListener` implementiert und zwangsweise `onAccuracyChanged()` enthalten muss. Wir benötigen `onBind()` nicht und werden zunächst dafür sorgen, dass der Service Ihre Schritte ordentlich zählt.

Ähnlich wie eine Activity hat auch ein Service einen Lebenszyklus (**Lifecycle**), dessen wichtigste Ereignisse mit passenden Behandlungsmethoden versehen sind. Wird ein Service zum ersten Mal gestartet, wird die `onCreate()`-Methode aufgerufen, und wenn er beendet wird, `onDestroy()`. Implementieren Sie also diese beiden Methoden, und füllen Sie sie mit dem entsprechenden Code aus der Activity, der den `SensorListener` abonniert und abmeldet:

```
@Override
public void onCreate() {
   super.onCreate();
   sensorManager = (SensorManager) getSystemService(Context.
                   SENSOR_SERVICE);
   sensor = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
            sensorManager.registerListener(listener, sensor,
            SensorManager.SENSOR_DELAY_GAME);
```

```
}
@Override
public void onDestroy() {
   sensorManager.unregisterListener(listener);
   super.onDestroy();
}
```

Genau wie eine Activity funktioniert auch ein Service nicht ohne Bürokratie: Sie müssen ihn im Android-Manifest anmelden (Abbildung 9.12).

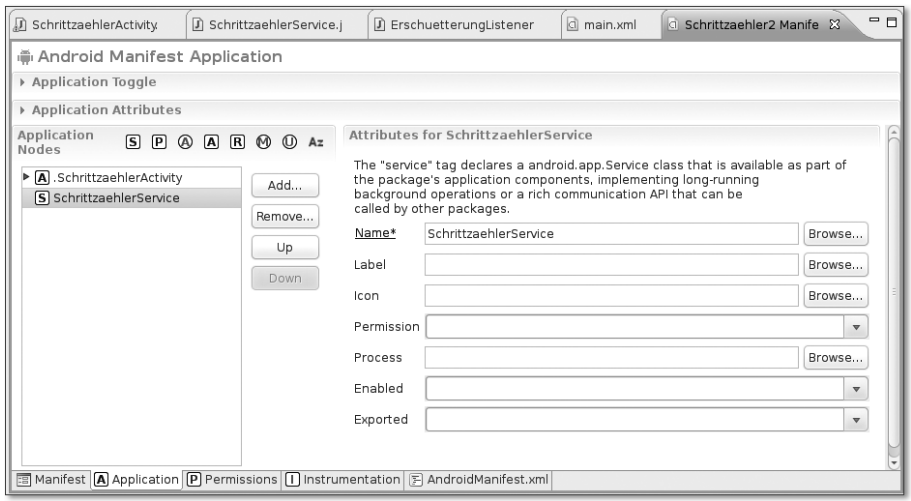



**Abbildung 9.12** Tragen Sie den »SchrittzaehlerService« im Android-Manifest ein. Achten Sie darauf, nach dem Klick auf den »Add«-Button »Service« auszuwählen, nicht »Activity«.

Sobald Ihr neuer Service startet, wird er beginnen, Schritte zu zählen, bis er beendet wird. Letzteres kann übrigens auch passieren, wenn der Speicher Ihres Handys bis oben hin voll ist. Das System beendet dann den Service und gibt den Speicher frei. Die `onDestroy()`-Methode wird dabei zwar nicht aufgerufen, aber das fleißige Aufräumkommando wird auch den `SensorListener` nicht verschonen. Sobald Android meint, dass genug freier Speicher verfügbar ist, wird der Service automatisch wieder gestartet, sodass er wieder anfängt, Schritte zu zählen – natürlich bei 0. Diesen Spezialfall lassen wir im Rahmen dieses Buches außer Acht.

Ob der Service nun wirklich tut, was er soll, können Sie im Moment nicht mit Sicherheit sagen, weil Sie keinerlei Rückmeldung auf dem Bildschirm haben. Um diese fehlende Verbindung kümmern wir uns als Nächstes.

### Service steuern

Einen Service, der im Hintergrund läuft, kann man starten und stoppen. Um dem Benutzer die Kontrolle zu erlauben, fügen Sie dem Layout *main.xml* zwei Buttons hinzu (Abbildung 9.13). Nennen Sie sie `start` und `stop`, und verpassen Sie ihnen passende Beschriftungen.

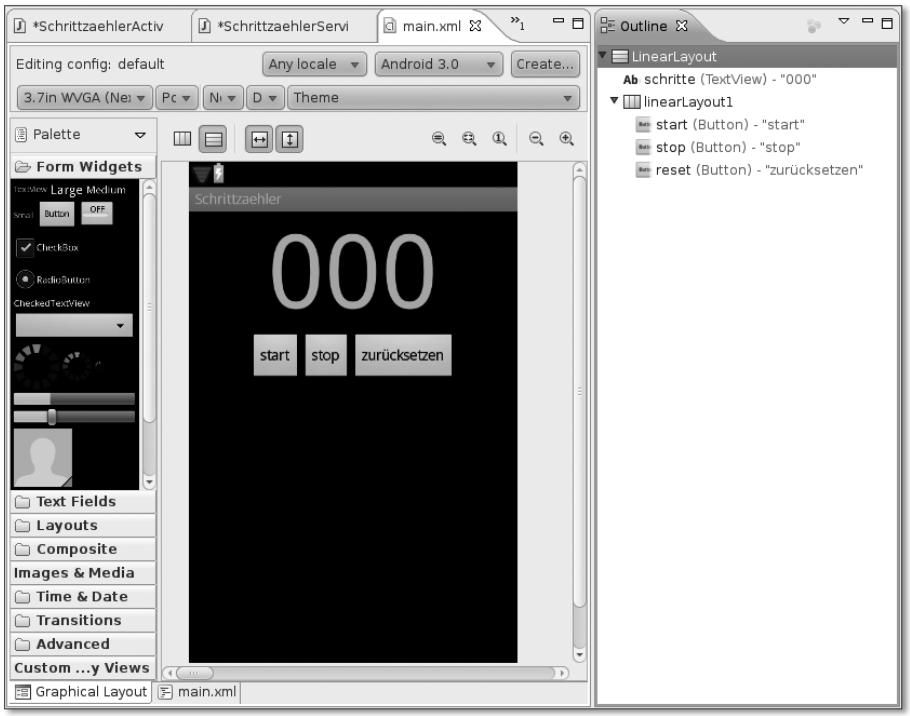



**Abbildung 9.13** Fügen Sie dem Schrittzähler-Layout Buttons zum Starten und Stoppen des Services hinzu.

Verdrahten Sie die `OnClickListener` der Buttons mit der Activity in deren `onCreate()`-Methode:

```
Button b = (Button) findViewById(R.id.reset);
b.setOnClickListener(this);
b = (Button) findViewById(R.id.start);
b.setOnClickListener(this);
b = (Button) findViewById(R.id.stop);
b.setOnClickListener(this);
```

Behandeln Sie die Click-Ereignisse in der `onClick()`-Methode:

```
@Override
public void onClick(View v) {
   if(v.getId()== R.id.start) {
      startService(new Intent(this,SchrittzaehlerService.class));
   }
   if(v.getId()== R.id.stop) {
      stopService(new Intent(this,SchrittzaehlerService.class));
   }
   if(v.getId()== R.id.reset) {
   }
}
```

Das Starten eines Services ähnelt dem Starten einer Activity. Sie erzeugen ein `Intent`-Objekt, dem Sie die Klasse des Services nennen, und übergeben es an die Methode `startService()` anstelle von `startActivity()`.

Im Gegensatz zu Activities können Sie Services explizit stoppen, indem Sie `stopService()` aufrufen.

Zwar können Sie jetzt den Service starten und stoppen, aber die Schrittzahl ist immer noch nicht auf dem Bildschirm. Es fehlt noch der Datenaustausch zwischen Activity und Service.

### Einfache Service-Kommunikation

Die Kommunikation zwischen Activity und Service muss zwei Dinge leisten:

- Service  Activity: Anzahl Schritte
- Activity  Service: Zurücksetzen der Schritte auf 0

Es gibt eine ganze Reihe von Möglichkeiten, diese Kommunikation zu implementieren. Da sie im gleichen Prozess und Thread stattfindet, können prinzipiell alle Klassen aufeinander zugreifen. Gefährlich ist lediglich die Tatsache, dass möglicherweise Service oder Activity gerade aus dem Speicher geräumt wurden. In dem Fall darf die App nicht abstürzen: Der Service sollte dann einfach mit der Schulter zucken und beim nächsten Schritt erneut versuchen, der Activity die Schrittanzahl zu übermitteln. Und die Activity sollte, wenn der Service gerade nicht läuft, darauf verzichten, die Schritte zurückzusetzen – das geschieht beim nächsten Service-Start ja ohnehin.

Ich werde Ihnen an dieser Stelle nicht die offizielle Kommunikationsmethode vorstellen, die relativ kompliziert und alles andere als leicht verständlich ist. Stattdessen greifen wir zu einem recht einfachen Trick: statische Attribute.

Ändern Sie mal in der Service-Klasse das Attribut `schritte` entsprechend:

```
public static int schritte=0;
```

**Zur Erinnerung: Der Modifier »static«**

Statische Attribute oder Methoden existieren unabhängig von Objekten. Sie sind vielmehr an die Klasse gebunden. Daher stehen sie in allen Objekten zur Verfügung – und sogar, wenn gar keine existieren. Dafür gilt die Highlander-Regel: Es gibt das statische Attribut nur einmal. Jedes Objekt der Klasse und die Klasse selbst greifen auf ein und dasselbe zu.

Während statische Methoden häufig für Hilfsfunktionen eingesetzt werden, dienen statische Attribute meist (zusammen mit dem `final`-Modifier) als Konstanten. Oder eben zum einfachen Datenaustausch über Klassengrenzen hinweg, unabhängig von Objekten.

Dieses statische Attribut wird bereits erzeugt, wenn die Klasse selbst geladen wird – und es existiert unabhängig davon, ob der Service läuft oder nicht! Dank der Sichtbarkeit `public` darf die `SchrittzaehlerActivity` ohne Weiteres auf dieses Attribut zugreifen, zum Beispiel, um es auf 0 zu setzen, wenn der Benutzer den ZURÜCKSETZEN-Button drückt:

```
public void onClick(View v) {
   if(v.getId() == R.id.reset) {
      SchrittzaehlerService.schritte = 0;
      textView.setText("0");
   }
   ...
}
```

Leider eignet sich das Attribut nicht für den umgekehrten Weg. Woher soll die Activity wissen, wann sich der Wert geändert hat? Theoretisch könnte die Activity in regelmäßigen Abständen nachschauen, aber das wäre wenig elegant.

Wir ebnen dem Service einen anderen Weg, um sich mitzuteilen: einen **statischen** Handler. Der Service wird über diesen Handler Nachrichten mit dem aktuellen Schritte-Wert schicken, ganz so, wie es auch der `ErschuetterungListener` tut. Über den neuen Handler gebietet die Activity, denn er repräsentiert einen Empfänger: Er existiert genauso lange wie die Activity.

```
private EreignisHandler ereignisHandler = new EreignisHandler();
private class EreignisHandler extends Handler {
   @Override
   public void handleMessage(Message msg) {
      textView.setText(Integer.toString(msg.what));
   }
}
```

Sie sehen, dass dieser Handler nichts anderes tut, als den Zahlenwert der Nachricht in den TextView zu schreiben.

Jetzt müssen Sie nur noch den Service über diesen Handler informieren. Das muss an zwei Stellen geschehen:

- gleich nach dem Start des Services
- wenn die Activity neu dargestellt wird, also in `onResume()`

Letzteres ist nötig für den Fall, dass die Activity eine Weile inaktiv war und von Android neu erzeugt wird, der Service aber noch läuft. In dem Fall muss der Service nicht neu gestartet werden, der Handler muss aber trotzdem installiert werden. Ergänzen Sie also dieselbe Zeile in `onResume()` und bei der Behandlung des START-Buttons:

```
SchrittzaehlerService.ereignisHandler = ereignisHandler;
```

Natürlich muss der Service über das passende statische Attribut verfügen:

```
public static Handler ereignisHandler;
```

Hier genügt die Klasse `Handler`, von der unser `EreignisHandler` erbt. Dieses Detail geht den Service aber nichts an, denn er muss uns nur eine simple Nachricht schicken, wenn er die Schrittanzahl erhöht. Genau genommen, geschieht das im `ErschuetterungsHandler`:

```
private class ErschuetterungsHandler extends Handler {
   @Override
   public void handleMessage(Message msg) {
      schritte++;
      if(ereignisHandler != null) {
         ereignisHandler.sendEmptyMessage(schritte);
      }
   }
}
```

Ganz wichtig ist es hier, zu prüfen, ob der `ereignisHandler` überhaupt gesetzt ist. Wurde die Activity aus dem Speicher geräumt, ist das nicht der Fall, und der folgende Methodenaufruf würde eine `NullPointerException` verursachen, weil `ereignisHandler` gleich `null` ist.

Wird die Activity später wieder gestartet, empfängt sie dank des neu installierten `ereignisHandlers` sofort wieder Schritt-Benachrichtungen. Bevor die erste eintrifft, sollte die Activity aber schon den letzten bekannten Wert anzeigen:

```
textView.setText(Integer.toString(SchrittzaehlerService.schritte));
```

Probieren Sie den Schrittzähler nun aus. Sie können jetzt die Activity in den Hintergrund schicken, indem Sie zum Beispiel die Home-Taste antippen oder eine andere App starten. Solange ein wenig Speicher frei ist, läuft der Service weiter

und zählt Ihre Schritte. Sie können jederzeit Ihre App wieder starten, egal, ob über das Icon im Menü oder das Festhalten der Home-Taste – und Sie sehen stets die aktuelle Anzahl Schritte.

Services sind unvermeidliche Werkzeuge, um im Hintergrund dauerhaft laufende Programmlogik zu ermöglichen. Sie dürfen lediglich nie vergessen, dass Service und Activity auch unabhängig voneinander existieren können. Keiner von beiden darf sich darauf verlassen, dass der andere gerade anwesend ist.

## 9.5 Arbeiten mit Geokoordinaten

Viele beliebte Smartphone-Apps beziehen ihre Magie aus Geokoordinaten. Google Maps weiß (wenn Sie möchten) immer, wo Sie gerade sind und wo die nächste Pizzeria, öffentliche Toilette oder Telefonzelle steht. Dank Satellit bis auf ein paar Meter genau.

Was viele Smartphone-Besitzer nicht wissen: Es gibt eine zweite, gröbere Ortsbestimmung, die ohne Satellit auskommt. Diese netzwerkbasierte Ortsbestimmung (**Network based Location**) kann aus der Identifikation der Mobilfunkzelle sowie der WLAN-Netzwerke in der Nähe mithilfe eines Online-Dienstes von Google erstaunlich genau abschätzen, wo Sie sich aufhalten. Der Vorteil: Diese Ortung kostet bei Weitem nicht so viel Batterieladung wie GPS. Und sie funktioniert in geschlossenen Räumen.

Als Beispielanwendung werden wir als Nächstes eine App schreiben, die (zum Beispiel) Ihren Weg ins Büro aufzeichnet und auf einer Karte darstellen kann.

Für den unwahrscheinlichen Fall, dass Sie die App nicht komplett selbst schreiben wollen, finden Sie das Eclipse-Projekt auf der beigelegten DVD unter dem Verzeichnis *WegInsBuero*.

### Der Weg ins Büro

Den Weg ins Büro (oder wohin auch immer) aufzuzeichnen ähnelt in gewisser Hinsicht der Schrittzähler-App: Die Aufzeichnung des Weges hat im Hintergrund zu geschehen – also als Service. Die Benutzeroberfläche besteht nur aus Buttons zum Starten und Stoppen sowie zum Betrachten des Wegs. Der Service muss in diesem Fall nicht einmal von sich aus Informationen zurück an die Oberfläche übermitteln: Er wird sich die Wegpunkte einfach speichern. Erst auf Wunsch des Benutzers wird die Activity die Punkte abrufen und auf einer Karte darstellen.

Beginnen Sie mit dem Layout *main.xml*. Wichtig ist dabei die Benennung der Buttons `start`, `stop` und `ansehen`.

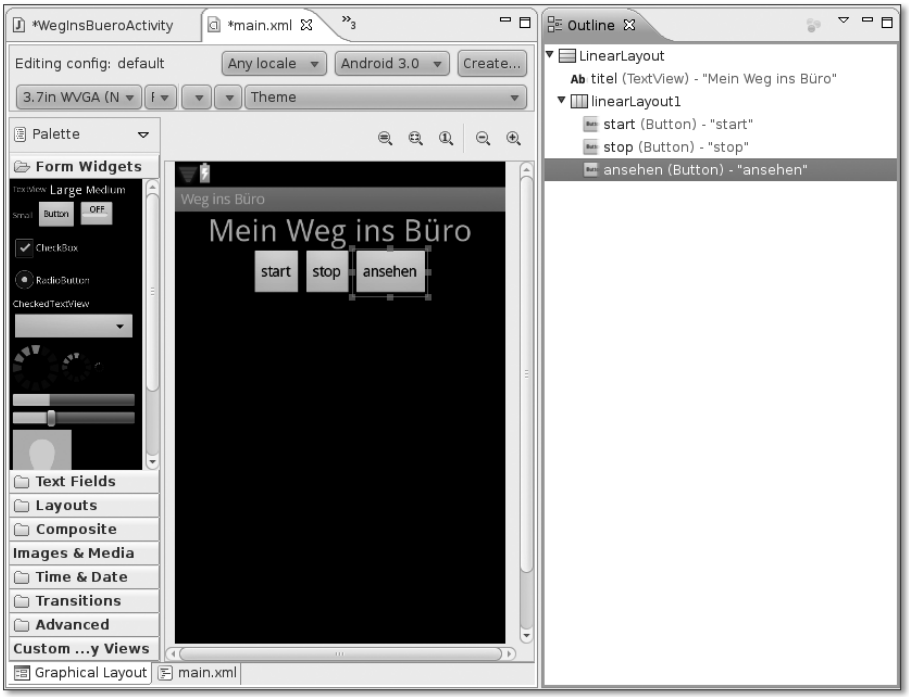



**Abbildung 9.14** Das Layout für den Weg ins Büro ist sehr überschaubar.

Die Activity der App ist demzufolge ähnlich übersichtlich. Sie implementiert den hinlänglich bekannten `OnClickListener` und verdrahtet die drei Buttons mit der eigenen `onClick()`-Methode:

```
@Override
public void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.main);
   Button b = (Button) findViewById(R.id.start);
   b.setOnClickListener(this);
   b = (Button) findViewById(R.id.stop);
   b.setOnClickListener(this);
   b = (Button) findViewById(R.id.ansehen);
   b.setOnClickListener(this);
}
```

Zwei der Buttons starten und stoppen den zugehörigen Service:

```
@Override
public void onClick(View v) {
   if(v.getId()== R.id.start) {
      startService(new Intent(this,WegAufzeichnungsService.class));
   }
   if(v.getId()== R.id.stop) {
      stopService(new Intent(this,WegAufzeichnungsService.class));
   }
}
```

Um den Ansehen-Button kümmern wir uns zuletzt. Vorher ist der Service an der Reihe, der die Wegpunkte einsammeln wird.

### Koordinaten ermitteln

Aufgabe des `WegAufzeichnungsService`, den Sie jetzt implementieren werden, ist es, die vom Smartphone zur Verfügung gestellten Wegpunkte zu speichern. Beginnen Sie mit dem Grundgerüst eines jeden Service:

```
public class WegAufzeichnungsService extends Service {
   @Override
   public IBinder onBind(Intent intent) {
      return null;
   }
   @Override
   public void onCreate() {
      super.onCreate();
   }
   @Override
   public void onDestroy() {
      super.onDestroy();
   }
}
```

Wenn Sie den Service ins Android-Manifest eintragen, fügen Sie gleich noch die nötige Erlaubnisanfrage hinzu, um die Position bestimmen zu dürfen. Im Gegensatz zu den Bewegungssensoren betrachtet das System die Geokoordinaten nämlich zu Recht als sensible Daten, für deren Verwendung jede App die Genehmigung des Benutzers benötigt.

Fügen Sie drei Uses-Permission-Einträge hinzu:

```
android.permission.ACCESS_COARSE_LOCATION
android.permission.ACCESS_FINE_LOCATION
android.permission.INTERNET
```

Die erste Permission schaltet die netzwerkbasierte Positionsbestimmung frei, die zweite die Satellitennavigation. Die Internet-Permission ist erforderlich, um Kartendaten aus dem Netz zu laden.

Die Geokoordinaten stellt Ihnen ein Systemservice namens `LocationManager` zur Verfügung, den Sie in einem Attribut speichern.

```
private LocationManager locationManager;
```

Die richtige Stelle, um das Attribut zu füllen, ist die `onCreate()`-Methode Ihres Services:

```
locationManager = (LocationManager) this.getSystemService(
                                    Context.LOCATION_SERVICE);
```

Ähnlich wie bei den Sensoren müssen Sie einen Listener als Abonnenten anmelden, um Daten zu erhalten, diesmal ist es ein `LocationListener`.

```
locationManager.requestLocationUpdates(LocationManager.NETWORK_
                PROVIDER, 0, 0, this);
```

Der erste Parameter legt fest, dass sich der Listener nur für die netzwerkbasierte Ortsbestimmung interessiert (wir werden die GPS-Variante später ausprobieren).

In bekannter Manier möge der Service selbst das geforderte Interface implementieren:

```
public class WegAufzeichnungsService extends Service implements
LocationListener
```

Lassen Sie Eclipse die nötigen Methoden implementieren, indem Sie Strg + 1 drücken, um den vermeintlichen Syntaxfehler zu beheben.

Von den vielen Methoden, die jetzt in Ihrem Service landen, interessiert nur eine einzige:

```
public void onLocationChanged(Location location) {
}
```

Der `LocationManager` wird diese Methode immer dann mit einer aktuellen Ortsangabe versorgen, wenn eine neue zur Verfügung steht. Die Aufgabe der Methode ist simpel: Sie muss lediglich das Objekt speichern. Dazu legen wir eine statische `ArrayList` im Service an:

```
public static List<Location> weg = new ArrayList<Location>();
```

Der Vorteil der `static`-Deklaration ist, dass die Activity später sehr leicht auf die Liste zugreifen kann.

Die Methode `onLocationChanged()` muss jetzt nur noch den neuen Wegpunkt an die Liste hängen:

```
weg.add(location);
```

Übrigens enthält das `Location`-Objekt nicht nur Orts-, sondern auch Zeitangaben, sodass Sie später sogar ein Geschwindigkeitsprofil Ihres zurückgelegten Weges erstellen können.

Lassen Sie uns als Letztes beim Starten des Services die statische Liste leeren. Ergänzen Sie in `onCreate()`:

```
weg.clear();
```

Eine vorher gespeicherte Wegstrecke wird so gelöscht. Wenn Sie möchten, können Sie später einen Button zum Zurücksetzen einbauen; wie das funktioniert, wissen Sie noch von der Schrittzähler-App.

Vergessen Sie nicht, in der `onDestroy()`-Methode das Abo zu kündigen:

```
locationManager.removeUpdates(this);
```

Im letzten Schritt ergänzen wir die App um die grafische Darstellung des zurückgelegten Weges.

### Karten und Overlay

So gut wie jedes Smartphone hat Zugriff auf Google Maps, und es ist technisch möglich, Karten dieses Dienstes in eigene Apps einzubinden. Allerdings gibt es zwei Nachteile:

- Um in einer App Google Maps nutzen zu können, benötigen Sie einen API Key. Der kostet zwar nichts, aber es ist etwas umständlich, ihn zu beschaffen.
- Die Lizenzbedingungen von Google Maps sind kompliziert und verbieten alles Mögliche, zum »Ausgleich« erlauben Sie jedoch Google, die Bedingungen jederzeit einseitig zu verändern.

Anstelle von Google Maps nutzen wir zum Darstellen der Karte und des Wegs ins Büro daher eine andere Methode: mapsforge. Das ist ein Open-Source-Projekt, das auf dem Kartenmaterial von Openstreetmaps.org (OSM) basiert. Es wird ähnlich Wikipedia von der Community gepflegt, d. h., auch Sie können ein fehlendes Toilettenhäuschen einzeichnen oder eine verrutschte Bushaltestelle an die richtige Stelle schieben.

Als Bonus obendrauf ist die Verwendung von mapsforge in einer eigenen App äußerst einfach. Also los!

**mapsforge**

Entwickelt und zur Verfügung gestellt wird mapsforge von der Freien Universität Berlin. Wenn Sie möchten, können Sie das Kartenmaterial sogar auf der SD-Karte Ihres Handys speichern. Das spart Datenverkehr, und die Darstellung ist viel schneller, weil die Karten nicht aus dem Internet nachgeladen werden müssen. Dafür erwartet Sie ein einmaliger Download von nicht unwesentlichen Ausmaßen: 1 GByte umfasst beispielsweise die Karte Deutschlands. Sie können aber auch kleinere Karten herunterladen, etwa von den Bundesländern. Sie finden den Download-Bereich für Deutschland hier:

*ftp://ftp.mapsforge.org/maps/europe/germany/*

Die Webseite des mapsforge-Projekts finden Sie unter:

*http://mapsforge.org*

Dort finden Sie auch weitere Dokumentationen.

Die Funktionen von mapsforge sind in einer externen Programmbibliothek enthalten, die Sie Ihrem Eclipse-Projekt hinzufügen müssen. Kopieren Sie die Bibliothek von der Buch-DVD, oder laden Sie sie von folgender Webseite herunter:

*http://code.google.com/p/mapsforge/downloads/detail?name=mapsforge-map-0.2.4.jar*

Java-Bibliotheken enden immer auf *.jar*. Sie sind nichts anders als ZIP-Archive, die Sie mit jedem halbwegs potenten Archivprogramm näher in Augenschein nehmen können. Es befinden sich diverse *.class*-Dateien, also kompilierte Java-Klassen, darin, fein säuberlich einsortiert in ihre Package-Ordner.

Um die Details müssen Sie sich hier nicht weiter kümmern. Wichtig ist, dass Sie die Datei *mapsforge-map-0.2.4.jar* in ein neues Verzeichnis *lib* innerhalb Ihres Projekts kopieren. Wenn Sie das getan haben, müssen Sie in Eclipse Ihren Projektbaum aktualisieren (F5), damit die Bibliothek dort sichtbar wird.

Klicken Sie dann mit der rechten Maustaste auf die Bibliothek, und wählen Sie Build Path • Add to build path. Daraufhin verändert sich das Icon vor der Bibliothek, zusätzlich erscheint sie unter einem neuen Knoten namens References Libraries in Ihrem Projekt (Abbildung 9.15). Dort können Sie die Bibliothek sogar aufklappen und alle enthaltenen Packages und Klassen auflisten.

Um die Karte anzuzeigen, benötigen Sie eine spezielle Activity, die diesmal nicht von der Standard-Klasse `Activity` abgeleitet wird, sondern von `MapActivity`, einer Klasse, die mapsforge zur Verfügung stellt.

```
public class WegAnsehenActivity extends MapActivity {
}
```


**Abbildung 9.15** Fügen Sie die mapsforge-Programmbibliothek Ihrem Projekt hinzu.

Wie schon beim Kompass-Beispiel kommt auch diese Activity ohne Layout-XML aus, weil sie nur einen einzigen View enthält: einen `MapView`, ebenfalls eine von mapsforge zur Verfügung gestellte Klasse.

Erzeugen Sie den `MapView` in der `onCreate()`-Methode der Activity, und weisen Sie ihn mit `setContentView()` als einzigen View der Activity zu.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   MapView mapView = new MapView(this, MapViewMode.
                                 OSMARENDER_TILE_DOWNLOAD);
   mapView.setClickable(true);
   mapView.setBuiltInZoomControls(true);
   setContentView(mapView);
}
```

Der View wird mit einem bestimmten Modus erzeugt, der auf den komplizierten Namen `MapViewMode.OSMARENDER_TILE_DOWNLOAD` hört. Dahinter verbirgt sich nichts anderes als die Kartenansicht mit automatischem Download der Kartenstücke (**Tiles**) aus dem Internet.

Zwei weitere Zeilen machen die Karte klickbar und blenden die Zoom-Buttons ein. Das gewährleistet, dass Sie den Kartenausschnitt intuitiv benutzen können, wie Sie es von Google Maps gewohnt sind. Das alles leistet der `MapView`.

Fehlt nur noch der zurückgelegte Weg. Auch dafür bringt mapsforge Unterstützung mit, und zwar in Form von Overlays. Stellen Sie sich ein Overlay wie eine Klarsichtfolie vor, die über den Kartenausschnitt gelegt wird. Verschieben Sie den Kartenausschnitt, bewegt sich die Folie mit. Ähnliches gilt fürs Zoomen.

Jetzt müssen Sie nur noch den zurückgelegten Weg, den der Service gespeichert hat, auf die Folie zeichnen. Dazu muss die `WegAnsehenActivity` die vom Service gefundenen Wegpunkte in einen OverlayWay umwandeln. Das ist natürlich nur sinnvoll, wenn mehr als ein Wegpunkt existiert:

```
List<Location> weg = WegAufzeichnungsService.weg;
if(weg.size()>1) {
   ...
}
```

Zum Zeichnen brauchen Sie einen Zeichenstift, wie Sie ihn schon vom Kompass kennen:

```
Paint paint = new Paint();
paint.setAntiAlias(true);
paint.setStyle(Paint.Style.STROKE);
paint.setStrokeWidth(7);
paint.setColor(Color.BLUE);
```

Der `MapView` erlaubt es, gleich mehrere Overlays anzuzeigen, die alle mit demselben `paint`-Objekt gezeichnet werden:

```
ArrayWayOverlay overlayArray = new ArrayWayOverlay(paint, paint);
```

Jetzt erzeugen Sie ein `OverlayWay`-Objekt und ein Array mit `GeoPoint`-Objekten, die der `MapView` anstelle von `Location`-Objekten verwendet:

```
OverlayWay way = new OverlayWay();
GeoPoint[] wegpunkte = new GeoPoint[weg.size()];
```

Das Array `wegpunkte` wird mit der richtigen Größe angelegt, nämlich der Anzahl der Wegpunkte aus dem Service. Als Nächstes übertragen Sie die `Location`-Wegpunkte in das `GeoPoint`-Array:

```
for(int i=0; i<weg.size(); i++) {
   Location ort = weg.get(i);
   wegpunkte[i] = new GeoPoint(ort.getLatitude(),ort.getLongitude());
}
```

Jetzt müssen Sie den erstellten Weg nur noch an den `MapView` übergeben:

```
way.setWayData(new GeoPoint[][] { wegpunkte });
overlayArray.addWay(way);
mapView.getOverlays().add(overlayArray);
```

Das sieht etwas kompliziert aus, weil `setWayData()` ein zweidimensionales Array erwartet – wir könnten auch mehr als einen Weg übergeben.

Fast fertig! Jetzt zentrieren Sie die Kartenansicht noch auf den Ausgangspunkt des Weges. Dazu dient der MapController des `MapViews`:

```
mapView.getController().setCenter(wegpunkte[0]);
```

Schließlich sorgen Sie dafür, dass ein Klick auf den Ansehen-Button in der Haupt-Activity die `WegAnsehenActivity` startet:

```
@Override
public void onClick(View v) {
   ...
   if(v.getId() == R.id.ansehen) {
      startActivity(new Intent(this,WegAnsehenActivity.class));
   }
}
```

Fertig ist Ihre erste App mit Geokoordinaten und Karte (Abbildung 9.16)! Natürlich können Sie auch diese App noch beliebig umbauen oder erweitern. Denken Sie zum Beispiel an Wegpunkte für Start und Ziel, an Live-Updates der Kartenansicht oder die Möglichkeit, Wege zu speichern und zu laden.

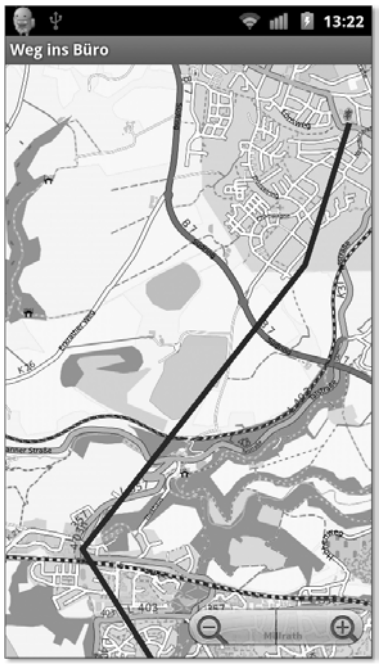



**Abbildung 9.16** Die App zeichnet auch ohne GPS den zurückgelegten Weg in die Karte – die Genauigkeit hängt von der Netzabdeckung ab.

*»Tricks kenn ich nicht. Bloß Flüche und Verwünschungen.«*
*(Magister Magicus Krassus der Fürchterliche)*

# 10 Tipps und Tricks

Bisher habe ich Ihnen einige Möglichkeiten, die Android bietet, anhand von Beispiel-Apps vorgeführt. Eine ganze Reihe Tricks habe ich dabei bereits eingeflochten, aber an einigen Stellen hätte ich Sie zu weit vom Pfad des Erfolgs fortgelockt, wenn ich weiter vom Thema abgewichen wäre. Deshalb reiche ich Ihnen einen Stapel Tipps an dieser Stelle nach, ohne dafür jeweils eine Beispiel-App zu basteln. Inzwischen verfügen Sie über genug Grundlagen, um meine Tipps an der richtigen Stelle nutzen zu können.

## 10.1 Fehlersuche

Es ist genauso erfreulich wie selten, dass eine App auf Anhieb funktioniert. Was tun, wenn etwas schiefgeht?

Kommt drauf an, *was* schiefgeht. Jeder Fehler bedeutet zunächst einmal einen Forschungsauftrag an den Entwickler. Das ähnelt der Frage, warum Vögel Federn haben. Um der Antwort auf die Spur zu kommen, kann man Fossilien ausgraben und untersuchen oder einen Schöpfer postulieren. Ich versichere Ihnen, dass die erstgenannte Variante bei der Suche nach Fehlern in Android-Apps zielführender ist als die zweite.

Sie erleben zunächst einmal nur einen **Effekt** des Fehlers – bei Android nicht selten in Form eines »Tut-uns-leid!«-Dialogs (Abbildung 10.1).


**Abbildung 10.1** Ein Fehler ist aufgetreten, und eines ist ziemlich sicher: Es einfach »erneut zu versuchen« wird nicht helfen.

Schlüpfen Sie in die Rolle eines Sherlock Holmes, um der Ursache des Fehlers auf die Spur zu kommen ...

### Einen Stacktrace lesen

Wenn Sie Ihr Smartphone mit Eclipse verbunden haben oder mit dem Emulator arbeiten, können Sie sehr genau verfolgen, was die ganze Zeit auf dem Handy passiert.

Dazu dient der View namens LogCat. Aktivieren Sie diese Ansicht, indem Sie im Menü Window • Show View • Other... • Android • LogCat auswählen. LogCat ist nichts anderes das Protokoll aller Geschehnisse auf dem Gerät, und selbstverständlich gehören auch Fehlermeldungen dazu. Grundsätzlich besteht jede Meldung aus einer oder mehreren Zeilen, die alle mit Datum und Uhrzeit versehen sind (Abbildung 10.2).

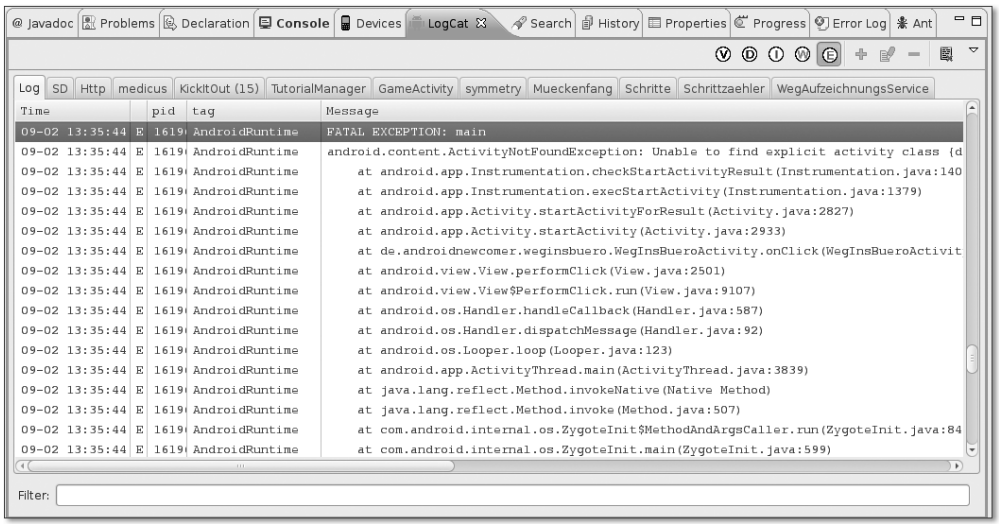



**Abbildung 10.2** Der »LogCat«-View zeigt Ihnen, was auf dem Handy alles geschieht.

Außerdem verfügt jeder Eintrag über eine Protokoll-Stufe (Log Level). Es gibt fünf Stufen mit absteigender Wichtigkeit, die in unterschiedlichen Farben dargestellt werden:

- E = Error
- W = Warnung
- I = Info
- D = Debug
- V = Verbose (ausführlich)

Die Einträge verraten Ihnen außerdem ihren Ursprung über die **Prozess-ID** (**pid**) und ein **Tag**. Der Rest jeder Zeile ist für die eigentliche Meldung reserviert.

Leider ballert Ihnen Android das Protokoll hauptsächlich mit Meldungen voll, die Sie gerade nicht besonders interessant finden. Aber das LogCat-Fenster bietet eine Reihe von Möglichkeiten, sich in dem ganzen Durcheinander zurechtzufinden.

Klicken Sie beispielsweise das rote Icon mit dem E an. Dies beschränkt die Ausgaben im Fenster auf Zeilen mit der Protokollstufe **Error**. Falls Ihre App ein »Tut uns leid« erzeugt, ist der Klick auf das E eine gute Idee. Zum Vorschein kommt dann eine ganze Anzahl Zeilen, die zu dem Fehlerereignis gehören. Diese Zeilen, die Sie beispielhaft in Abbildung 10.2 sehen, nennt man **Stacktrace**.

Sie lesen den Stacktrace normalerweise von oben nach unten. Als Erstes steht da die eigentliche Fehlermeldung, die das »Tut uns leid!« verursacht hat. Sie lautet meistens `FATAL EXCEPTION: main`. Ursache ist immer eine Exception, die entweder im Android-System selbst oder innerhalb Ihrer App aufgetreten ist, aber mit keinem `catch`-Block gefangen wurde.

Die zweite Zeile des Stacktrace enthält die genaue Bezeichnung der aufgetretenen Exception, zum Beispiel `ActivityNotFoundException`. Wenn Sie großes Glück haben, ist die Fehlersuche an dieser Stelle schon beendet: Manchmal können Sie aus dieser Meldung sofort auf die Ursache schließen. Im vorliegenden Beispiel wurde versucht, eine Activity zu starten, die nicht im Manifest angemeldet ist. Dies ist ein häufiger Fehler – so häufig, dass die Android-Macher sogar die Lösung mit in die ausführliche Meldung geschrieben haben: »Have you declared this activity in your AndroidManifest.xml?«

Natürlich ist die Fehlersuche nicht immer so einfach.

Schauen Sie sich ein weiteres Beispiel an:

```
FATAL EXCEPTION: main
java.lang.RuntimeException: Unable to create service de.androidnewcom
er.weginsbuero.WegAufzeichnungsService: java.lang.NullPointerException
        at android.app.ActivityThread.handleCreateService(
           ActivityThread.java:2076)
        at android.app.ActivityThread.access$2500(
           ActivityThread.java:123)
        at android.app.ActivityThread$H.handleMessage(
           ActivityThread.java:993)
        at android.os.Handler.dispatchMessage(Handler.java:99)
        at android.os.Looper.loop(Looper.java:123)
        at android.app.ActivityThread.main(ActivityThread.java:3839)
        at java.lang.reflect.Method.invokeNative(Native Method)
        at java.lang.reflect.Method.invoke(Method.java:507)
```

```
        at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.
           run(ZygoteInit.java:841)
        at com.android.internal.os.ZygoteInit.main(
           ZygoteInit.java:599)
        at dalvik.system.NativeStart.main(Native Method)
Caused by: java.lang.NullPointerException
        at de.androidnewcomer.weginsbuero.WegAufzeichnungsService.
           onCreate(WegAufzeichnungsService.java:33)
        at android.app.ActivityThread.handleCreateService(
           ActivityThread.java:2066)
        ... 10 more
W/ActivityManager( 1695): Force finishing activity de.androidnewcomer.
weginsbuero/.WegInsBueroActivity
```

In diesem Fall ist offenbar das Starten des `WegAufzeichnungsService` fehlgeschlagen, und zwar mit einer `NullPointerException`. Der Code versucht also, ein Objekt zu verwenden, das `null` ist, also uninitialisiert. Jetzt müssen Sie den Rest des Stacktrace lesen, um der Ursache auf die Spur zu kommen.

Jede Zeile bezeichnet eine Stelle im Code, an der eine andere Methode aufgerufen wird, und zwar rückwärts. Sie sehen also zuerst die unterste Methode, dann die Stelle, wo diese Methode aufgerufen wurde etc. Sie sehen, dass sich das alles in Klassen abspielt, die Sie nicht geschrieben haben – bis es an einer Stelle heißt: `Caused by:`.

Android hat hier die `NullPointerException` gefangen und eine neue daraus gemacht. Hier, tief unten, finden Sie die ursprüngliche Exception und auch den Verweis auf Ihren eigenen Code. In diesem Beispiel war die fehlerhafte Zeile in *WegAufzeichnungsService.java*, Zeile 33.

Dort steht:

```
weg.clear();
```

Ursache für eine `NullPointerException` ist fast immer, dass Sie auf eine Methode oder ein Attribut einer nicht initialisierten Variablen zugreifen. Das ist hier offensichtlich `weg`.

Suchen Sie die Stelle, an der `weg` deklariert wird. In diesem Beispiel:

```
public static List<Location> weg;
```

Prüfen Sie anschließend, ob die Variable je mit irgendeinem Objekt initialisiert wird, bevor der Zugriff in Zeile 33 erfolgt. Das ist nicht der Fall, also liegt der Fehler auf der Hand: Die Initialisierung fehlt, und richtig lautet die Deklaration:

```
public static List<Location> weg = new ArrayList<Location>();
```

Ich kann Ihnen hier natürlich nicht jede mögliche Exception erläutern, dazu gibt es zu viele. Nur so viel: Wenn Sie minutenlang ratlos auf Ihren Code starren, zeigen Sie die Stelle einem Bekannten, oder machen Sie Kaffeepause, und schauen Sie danach noch mal drauf. Irgendwann finden Sie den Fehler. Und verlassen können Sie sich auf eines: Mit den Java-Stacktraces kommen Sie Fehlern viel schneller auf die Spur als Generationen von Entwicklern vor Ihnen, die sich mit weniger gesprächigen Programmiersprachen herumschlagen mussten.

**Logs von anderen Geräten**

Es ist ziemlich unpraktisch, wenn Ihnen ein Benutzer einen Fehler meldet, den Sie auf Ihrem Gerät aber partout nicht nachvollziehen können.

Tatsächlich ist nicht jedes Android-Gerät gleich; auf manchen mag eine App laufen, die auf anderen nur »Tut uns leid« sagt.

Wenn Sie des Besitzers nicht habhaft werden können, um sein Handy an Ihr Eclipse zu stöpseln, gibt es glücklicherweise zwei andere Methoden:

1. Die kostenlose App Android System Info ermöglicht es, das LogCat-Protokoll direkt auf dem Gerät einzusehen. Mehr noch: Das Protokoll lässt sich in einer Datei auf SD-Karte speichern und von dort aus per Mail verschicken. Vielleicht haben Sie Glück, und der Benutzer, der den Fehler gemeldet hat, unterstützt Sie, indem er die App installiert und Ihnen das Protokoll schickt.
2. Wenn Sie eine App im Android Market veröffentlicht haben, können Benutzer Fehlerberichte einsenden. Das ist eine Möglichkeit, die natürlich nicht jeder Benutzer wahrnimmt, denn er kann die Frage des Handys, ob er einen Bericht senden möchte, natürlich ablehnen. Sie finden diese Berichte in Ihrer Entwicklerkonsole auf der Market-Webseite, die ich Ihnen in einem späteren Kapitel vorstellen werde. Die Berichte enthalten meist aussagekräftige Stacktraces und selbstverständlich auch die Versionsnummer der App, in der sie aufgetreten sind.

## Logging einbauen

Sie müssen nicht unbedingt warten, bis ein fataler Fehler auftritt und Android einen Stacktrace ins Protokoll schreibt – Sie können auch ohne Weiteres selbst von Ihrer App aus Informationen ausgeben. Man nennt das **Logging**. Für kompliziertere Anwendungen ist es fast unerlässlich – vor allem, wenn Berechnungen im Hintergrund ablaufen, sodass weder ein ander Benutzer noch Sie selbst auf Anhieb sehen können, was genau geschieht.

Um Zeilen ins Protokoll zu schreiben, benutzen Sie statische Methoden der Klasse `Log`. Für jede Protokollstufe gibt es eine Methode. Fehlermeldungen können Sie beispielsweise wie folgt ausgeben:

```
Log.e("WegAufzeichnungsService", "Fehlerbeschreibung");
```

Die Logging-Methoden erwarten zwei oder drei Parameter. Das genannte Beispiel ist eine einfache Variante mit zwei Parametern. Der erste ist das Log-Tag, das Sie bereits aus der LogCat-Ansicht kennen. Es dient dazu, zu erkennen, dass die Meldung von Ihrer App kommt. Wählen Sie einen eindeutigen Bezeichner, und definieren Sie ihn am besten als Konstante:

```
public static final String LOGTAG = "MeineApp";
```

Das vermeidet Tippfehler, die einzelne Einträge vermeintlich vor Ihnen verstecken.

Parameter Nummer 2 ist eine beliebige Ausgabe, die Ihnen bei der Untersuchung des Verhaltens Ihrer App hilft. Beispielsweise können Sie mit passenden Log-Kommandos sichtbar machen, wann ein Service startet oder beendet wird:

```
public void onCreate() {
   super.onCreate();
   Log.d(LOGTAG, "service startet");
   ...
}
public void onDestroy() {
   Log.d(LOGTAG, "service beendet");
   super.onDestroy();
}
```

Natürlich können Sie auch aktuelle Zahlenwerte ausgeben, die Aufschluss über den Programmablauf geben:

```
public void onLocationChanged(Location location) {
   weg.add(location);
   Log.d(LOGTAG, "Aktuelle Weglänge: " + weg.size() );
}
```

Sie können mit dem LogCat-View in Eclipse die Ausgabe auf Ihre eigene App begrenzen, indem Sie einen Log-Filter einrichten. Klicken Sie auf das grüne Plus-Icon, und geben Sie einen Namen für den Filter ein. Tragen Sie außerdem Ihr verwendetes Tag ein (Abbildung 10.3). Wenn Sie möchten, können Sie den Filter auch auf eine Protokollstufe beschränken, aber dafür gibt es ja die farbigen Buttons.

Achten Sie darauf, dass Sie innerhalb der Log-Ausgaben keine komplizierten Berechnungen durchführen. Ich habe schon einmal von Applikationen gehört, die zehnmal schneller wurden, wenn man das Logging ausschaltete ...

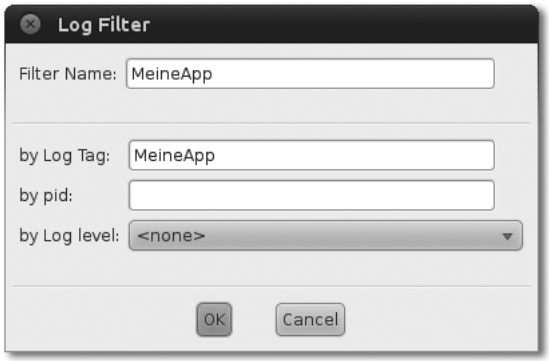



**Abbildung 10.3** Eigene Log-Filter helfen ungemein, im Wust der Android-Meldungen die eigenen aufzufinden.

Die Faustregel lautet: Protokollieren Sie während der Entwicklungsphase alles, was Sie brauchen. Wenn Sie sicher sind, dass ein Programmteil korrekt arbeitet, werfen Sie alle Log-Befehle raus, oder machen Sie sie mithilfe einer `boolean`-Konstanten global abschaltbar:

```
if(LOGGING) Log.d(LOGTAG, "Nebensächliche Information");
```

Lassen Sie nur das Logging in der App, das Sie wirklich brauchen. Denken Sie außerdem daran, keine sicherheitskritischen Informationen zu loggen, beispielsweise Ihren Google Maps API Key. Denn jeder Benutzer kann diese Informationen prinzipiell sehen. Verraten Sie im Protokoll nichts über Ihre App, das gegen Sie verwendet werden kann.

## Schritt für Schritt debuggen

Manchmal hilft auch kein Log weiter. Sie wissen überhaupt nicht, was Ihre App gerade tut? Sie würden gerne genau sehen, welcher Code gerade ausgeführt wird?

Auch das geht mit Eclipse.

Überlegen Sie sich zunächst, an welcher Stelle der Ablauf der App unterbrochen werden soll. Doppelklicken Sie dann im Editor links von der gewählten Zeile auf den Rand des Fensters. Es entsteht ein grüner Breakpunkt. An dieser Stelle wird die App stehenbleiben und Ihnen tiefe Einblicke gewähren. Allerdings klappt das nicht, wenn Sie die App wie bisher mit Run starten. Wählen Sie stattdessen Debug oder das Icon mit dem grünen Käfer.

Sobald Ihre App die betreffende Stelle erreicht, pausiert sie, und Eclipse schaltet auf die Debug-Perspektive um (Abbildung 10.4).

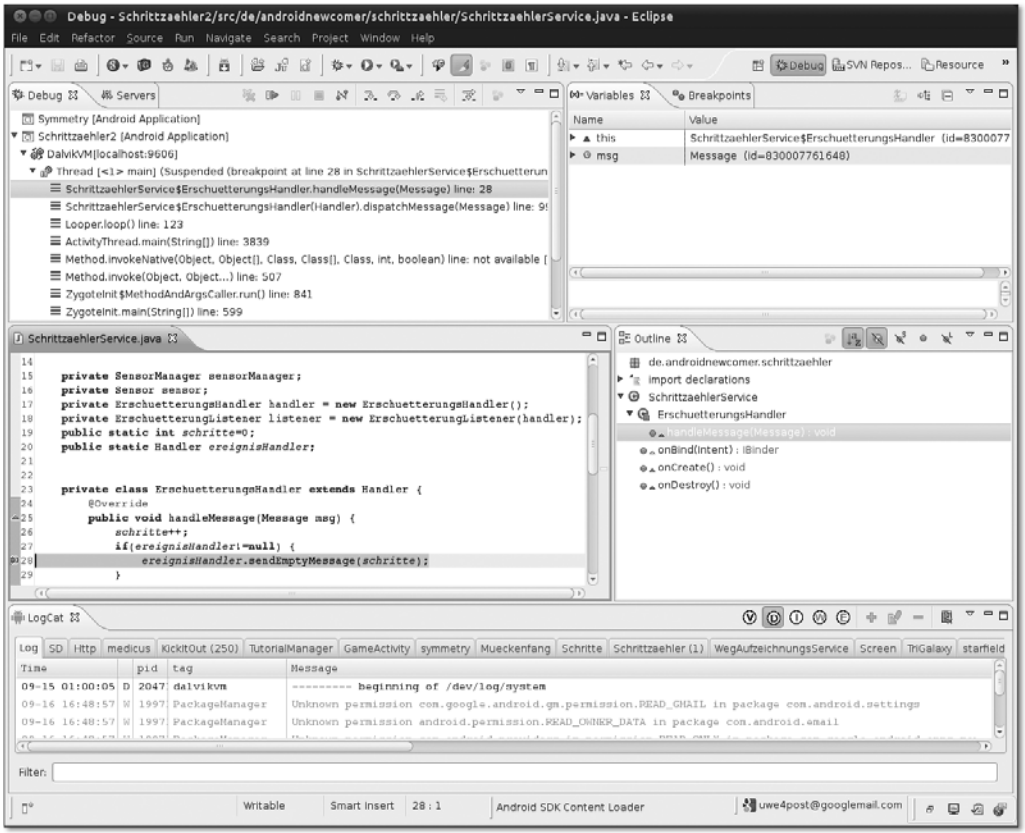



**Abbildung 10.4** Wenn die App am Breakpunkt stoppt, schaltet Eclipse in die »Debug«-Perspektive um.

Die DEBUG-Perspektive ist mit zusätzlichen Views gepflastert, die Ihnen genaue Einblicke in die App erlauben. Links oben sehen Sie im DEBUG-View den aktuellen Stacktrace. So können Sie herausfinden, von wo aus Ihre App aufgerufen wurde.

Rechts oben finden Sie einen sehr spannenden View namens VARIABLES. Darin aufgelistet sind alle Variablen, die die App an der momentanten Stelle kennt. Sie können sich in Ruhe die Werte ansehen – vielleicht ist einer anders, als Sie es erwartet haben. An dieser Stelle können Sie sogar Werte ändern, um zu untersuchen, wie Ihre App darauf reagiert!

Am oberen Rand des DEBUG-Views finden Sie mehrere Icons, die es erlauben, die App direkt zu steuern: Mit dem grünen Pfeil können Sie die App weiterlaufen lassen, bis sie auf den nächsten Breakpunkt trifft. Oft sehr hilfreich sind die Icons mit den kleinen Pfeilen: Sie erlauben es, jeweils nur die nächste Zeile auszuführen.

Auf diese Weise können Sie Schritt für Schritt beobachten, was Ihre App gerade tut, wie sich Variableninhalte ändern und welche Methoden Ihre App aufruft.

Breakpunkte können Sie entweder löschen, indem Sie erneut darauf doppelklicken; alternativ finden Sie eine Liste im BREAKPOINTS-View. Um zurück zur Java-Perspektive zu schalten, verwenden Sie die Buttons rechts oben im Eclipse-Fenster.

Es kostet eine Menge Zeit und außerdem eine gehörige Portion Geschick, Breakpunkte an der richtigen Stelle zu setzen, um mit dem schrittweisen Debugging Fehler zu identifizieren – aber wenn es hart auf hart kommt, haben Sie keine andere Möglichkeit.

## 10.2 Views mit Stil

Die Android-Oberfläche – bestehend aus Textfeldern, Buttons etc. – sieht an sich ganz hübsch aus, hat aber zwei Nachteile:

- Fast jeder Smartphone-Hersteller passt das Aussehen seinen Vorstellungen an, sodass Apps auf verschiedenen Phones unterschiedlich aussehen.
- In Spielen sehen graue Buttons mit Hervorhebungen in Orange (oder Blau oder Grün, je nach Hersteller) meist billig aus.

Wenn Sie Ihrer App einen einheitlichen Stil verpassen wollen, müssen Sie das Standard-Erscheinungsbild überschreiben. Wie das funktioniert, zeige ich Ihnen in den folgenden Abschnitten auf Basis des Projekts **Schrittzaehler2**.

### Hintergrundgrafiken

Wenn Ihnen die grauen Standard-Buttons nicht gefallen, können Sie einfach deren Hintergrundgrafik austauschen. Sie können ein Grafikprogramm wie Inkscape verwenden, um einen alternativen Hintergrund zu basteln. Zeichnen Sie beispielsweise ein Rechteck mit runden Ecken, farbiger Fläche, Schatten oder einem Flammen-Effekt am Rand (Abbildung 10.5).

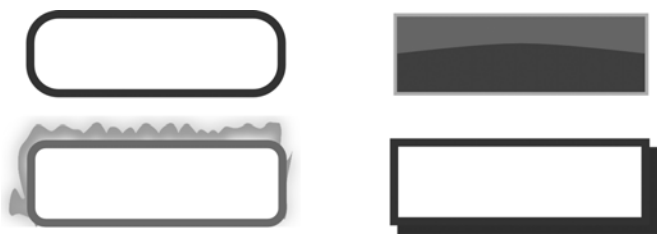


**Abbildung 10.5** Sie können mit jedem Grafikprogramm leicht Button-Hintergründe nach eigenen Vorstellungen gestalten.

Speichern Sie Ihren Hintergrund als PNG-Grafik im Verzeichnis *drawable*. Das PNG-Format unterstützt im Gegensatz zu JPG Alpha-Transparenz, sodass Sie wirklich runde Ecken und andere Effekte erzeugen können, die ziemlich flott ausschauen.

Stellen Sie dann einfach im Layout-Editor für Ihre Buttons den richtigen Hintergrund ein, indem Sie Ihre Grafik als Property BACKGROUND auswählen (Abbildung 10.6). Prinzipiell funktioniert das für alle Views, zum Beispiel auch für `EditText`.


**Abbildung 10.6** Eigene Button-Hintergründe verschönern die Schrittzähler-App.

Ihnen werden sofort ein paar Nachteile dieser Methode auffallen:

- Sie müssen das BACKGROUND-Property für jeden Button einzeln ändern.
- Die Buttons nehmen die Breite der Grafik an.
- Wenn die Schrift länger ist als die Grafik, wird diese horizontal gestreckt. Das sieht nicht schön aus.

Der nächste Abschnitt erklärt Ihnen, wie Sie alle Buttons auf einen Streich mit einer Hintergrundgrafik versehen können.

## Styles

Ein Android-**Style** ist eine Zusammenfassung mehrerer View-Eigenschaften unter einem Namen. Anstatt allen Views, die diese Eigenschaften erhalten sollen, jede einzelne zu verpassen, genügt es, ihnen den Style zuzuweisen.

Sie können mehrere Styles in einer Resource-Datei speichern. Erzeugen Sie eine neue Resource-Datei namens *styles.xml* im Verzeichnis *res/values*.

Fügen Sie anschließend mit dem Resource-Editor ein neues Style-Element ein, das `rundeButtons` heißt. Dieser Style erhält vorläufig ein `Item`, das so heißen muss wie das entsprechende View-Property, in diesem Fall also `android:background`. Als Wert tragen Sie eine Referenz auf die gewünschte Hintergrundgrafik ein. Leider gibt es hier keinen komfortablen Auswahldialog, Sie müssen also darauf achten, dass Sie sich nicht vertippen. Achten Sie auch darauf,

dem Namen der Grafik das magische `@drawable/` voranzustellen und auf die Dateiendung (*.png*) zu verzichten (Abbildung 10.7).

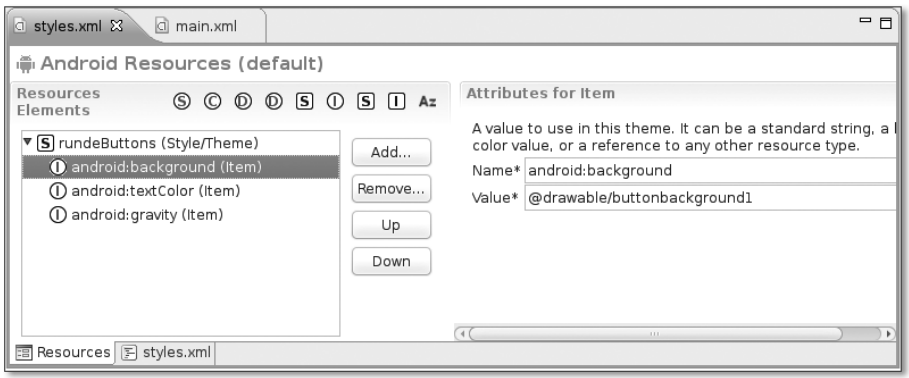



**Abbildung 10.7** Wenn Sie Style-Items erstellen, die auf Grafiken verweisen, müssen Sie dem Namen des Grafik ein »@drawable/« voranstellen.

Sie können dem Style beliebig viele weitere Property-Items hinzufügen, beispielsweise TEXTCOLOR. Im Fall von Buttons sollten Sie auf jeden Fall das GRAVITY-Property auf den Wert `center` setzen. Denn Ihr Style *ersetzt* den Standard-Style, und ohne `center` würden die Beschriftungen in der linken oberen Ecke der Buttons landen.

Weisen Sie schließlich den Buttons Ihren Style mit dem Layout-Editor zu.

## Themes

Wenn Sie eine App mit mehr als einem Button gebastelt haben oder gar mehrere Styles für unterschiedliche View-Typen, dann können Sie auf einen Schlag alle Styles allen Views zuordnen, indem Sie die Styles zu einem **Theme** zusammenfassen.

Themes sind Sammlungen von Styles. Sie können ein Theme einer Activity oder Ihrer ganzen App zuweisen. Allerdings wendet Android dann jeden Style, den Sie definiert haben, auf jeden infrage kommenden View an. Im Fall des BACKGROUND-Propertys ist das natürlich nicht das, was Sie wollen – es sei denn, Sie mögen Ihre Hintergrundgrafik so sehr, dass es Sie nicht stört, wenn der Bildschirm voll davon ist.

Sie müssen also dafür sorgen, dass die Button-Styles nur auf Buttons angewendet werden.

Das ist aber nicht alles: Ein Theme muss sich um das Aussehen *aller* Views kümmern. Es kann freilich nicht in Ihrem Sinne sein, diese unzähligen Styles alle

selbst festzulegen. Deshalb gibt es die Möglichkeit, ein Theme zu erstellen, das alle Werte eines anderen Themes *erbt* und nur ausgewählte überschreibt.

Technisch ist ein Theme ein Style, wenn Sie jetzt also Ihr neues Theme anlegen, sorgen Sie durch einen passenden Namen für eine leichte Unterscheidbarkeit. Nennen Sie den neuen Style beispielsweise `meinTheme`. Tragen Sie als Parent `android:style/Theme.Black` ein. Das ist eines der im System verfügbaren Themes und hat sinnvolle Voreinstellungen. Welche Themes Android Ihnen zur Verfügung stellt, können Sie sehr leicht im Layout-Editor sehen: Dort gibt es ein Dropdown-Menü, mit dem Sie auswählen können, in welchem Theme Ihr Layout dargestellt wird. Merken Sie sich einfach den Namen Ihres Lieblingsthemes, und schreiben Sie ihn in das Feld PARENT. In diesem Dropdown finden Sie später übrigens auch Ihr selbst definiertes Theme, sodass Sie Ihre App in ihrem neuen Kleid beurteilen können, ohne Sie auf dem Handy starten zu müssen.

Fügen Sie dem neuen Theme ein Item mit dem Namen `android:buttonStyle` hinzu. Wie Sie leicht erraten können, ist das der von Android verwendete Style für Buttons. Dessen Wert ändern Sie einfach auf Ihren eigenen Style, indem Sie dessen Namen eintragen. Vergessen Sie nicht das `@style/`-Präfix (Abbildung 10.8).

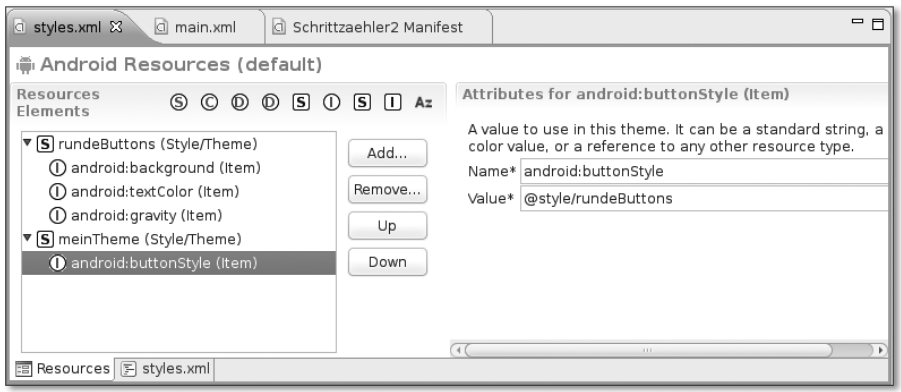



**Abbildung 10.8** Um den buttonStyle des Parent-Themes zu überschreiben, legen Sie einfach das entsprechende Item in Ihrem Theme an.

Als Letztes müssen Sie Ihr Theme Ihrer App zuordnen. Das geschieht im Android-Manifest. Es gibt unter den APPLICATION ATTRIBUTES ein Eingabefeld für das Theme, das Sie dort zudem bequem mit einem Auswahldialog einstellen können (Abbildung 10.9).

Alternativ können Sie Themes separat für Activities eintragen. Das geschieht ebenfalls im Manifest, bloß unterhalb der jeweiligen Activity-Elemente.

Fertig! Jetzt können Sie das Resultat bewundern.

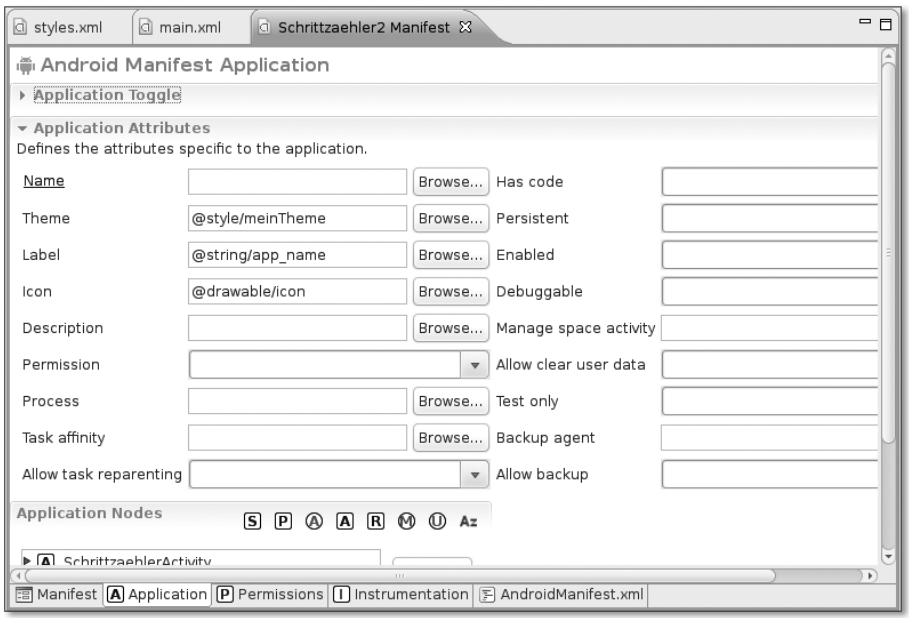



**Abbildung 10.9** Tragen Sie Ihr Theme ins Android-Manifest ein.

## Button-Zustände

Ihr Button besitzt jetzt genau eine Hintergrundgrafik. Das bedeutet, dass es keinen sichtbaren Unterschied gibt zwischen einem normalen, einem gedrückten oder einem unbenutzbaren Button.

Anstelle *einer* Hintergrundgrafik müssten Sie also mehrere hinterlegen. Aber wie soll das funktionieren, wenn es nur ein `background`-Attribut gibt?

Der Trick besteht darin, als `background`-Attribut nicht direkt auf die Hintergrundgrafik zu verweisen, sondern auf eine *Liste* von Grafiken. Eine solche Liste ist eine besondere Resource-Datei, die im *drawable*-Verzeichnis liegt. Anstelle einer Grafik lädt Android dann diese Liste und verarbeitet sie. Erstellen Sie also eine neue Resource-Datei namens *button.xml* mit dem Android-Wizard. Wählen Sie als RESOURCE-TYP DRAWABLE und als ROOT ELEMENT SELECTOR.

Leider gibt es für Resource-Dateien dieser Art noch keinen komfortablen Editor. Sie müssen sich also jetzt einmal mehr mit XML-Code herumschlagen. Glücklicherweise bietet Ihnen Eclipse immerhin eine Edit-Hilfe, wenn Sie Strg + Leertaste drücken.

Die vom Wizard erzeugte leere Datei sieht so aus:

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

An diesen Zeilen müssen Sie nichts ändern, Sie müssen bloß welche ergänzen.

Für jeden Button-Status benötigen Sie ein `item`-Element, das zum einen den betreffenden Status benennt und zum anderen die zugehörige Grafik. Sehen Sie sich das folgende Beispiel an:

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
<item android:drawable="@drawable/buttonbackground4"
      android:state_pressed="true"/>
<item android:drawable="@drawable/buttonbackground2"
      android:state_pressed="false"/>
</selector>
```

Der Hintergrund `buttonbackground4` (hellgrüner Rand) wird dem Status `pressed` (also »gedrückt«) zugewiesen, während `buttonbackground2` (blauer Rand) »nicht gedrückt« bedeutet.

Es gibt eine Reihe von Zuständen, denen Sie Grafiken zuordnen können:

- `focused`
- `selected`
- `enabled`
- `pressed`

Auch Kombinationen sind möglich.

Auf ganz ähnliche Weise können Sie für andere Views multiple Hintergrundgrafiken festlegen, beispielsweise für Checkboxen. Da gibt es dann den Zustand `checked`, dem Sie ein Bild mit einem Haken zuordnen sollten.

### 9-Patches

Kommen wir zurück zu einem weiteren Ausgangsproblem der eigenen Button-Hintergründe: Je nach Größe des Buttons erscheint die Hintergrundgrafik verzerrt.

Glücklicherweise verfügt Android über ein erstaunlich einfaches Hilfsmittel, um dieses Problem zu lösen, ohne dass Sie haufenweise Grafiken für unterschiedliche Seitenverhältnisse bereitstellen müssen. Das Zauberwort lautet: **9-Patch**.

Ein 9-Patch ist eine normale PNG-Grafik, die von einem 1 Pixel breiten Rahmen umgeben ist. Der obere und der linke Rand dieses Rahmens bestimmen durch gesetzte Pixel, welcher Bereich der Grafik gestreckt werden darf und welcher nicht.

Bevor Sie jetzt anfangen, in Ihrem Lieblings-Bildbearbeitungsprogramm einzelne Pixel mit der Lupe zu verschieben, starten Sie besser ein kleines Tool, das Teil des Android SDK ist. Sie finden es im Verzeichnis *tools*, und sein Name lautet draw9patch. Wenn Sie das Tool auf Ihrem PC oder Mac gestartet haben, ziehen Sie einfach Ihre bisherige Button-Hintergrundgrafik auf das Programmfenster und lassen sie dort fallen. Dann können Sie mit der Maus bequem links und oben Pixel einzeichnen und sehen gleichzeitig im rechten Fensterbereich, wie unterschiedlich gestreckte Views Ihren Hintergrund darstellen werden (Abbildung 10.10). Wo Sie schwarzen Rand einzeichnen, streckt Android die Grafik, der Rest bleibt unskaliert.

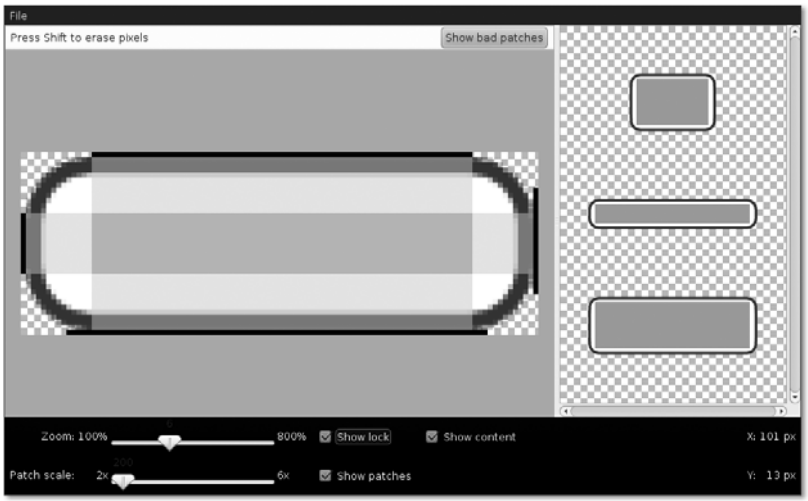



**Abbildung 10.10** Markieren Sie im draw9patch-Tool jenen Bereich Ihrer Grafik, der gestreckt werden darf.

Am rechten und unteren Rand können Sie mit zwei weiteren Pixelleisten definieren, in welchem Bereich Android Inhalte wie Button-Beschriftungen anzeigen darf. Damit können Sie vermeiden, dass Aufschriften den Rand berühren oder überschreiten. Schalten Sie die Checkboxen SHOW PATCHES und SHOW CONTENT ein, um während der Arbeit noch besser die Auswirkungen Ihrer gesetzten Pixel überwachen zu können.

Mit der ⇧-Taste können Sie Pixel wieder entfernen. Speichern Sie die neue Grafik unter einen anderen Namen, zum Beispiel *buttonbackground2*. Das Tool hängt automatisch die Endung *.9.png* an.

Android unterscheidet Grafiken nur anhand ihres Dateinamens ohne Endung. Deshalb muss Ihr 9-Patch einen anderen Namen erhalten, sonst wüsste Ihre App nicht, ob sie das 9-Patch oder dessen Ursprungsgrafik verwenden soll.

Ändern Sie einfach in Ihrem Style den Namen des Drawables, damit das 9-Patch verwendet wird.

Übrigens genügt es völlig, wenn Sie anstelle eines länglichen einen quadratischen 9-Patch basteln – der innere Bereich wird ohnehin skaliert. Eine quadratische Grafik ist kleiner und verbraucht weniger Speicher.

Natürlich funktionieren 9-Patches nicht nur als Button-Hintergrund, sondern beispielsweise auch bei `EditText`-Views.

Mit Styles, Themes und 9-Patches verfügen Sie über mächtige Hilfsmittel, um auch größeren Apps schnell und effizient das gewünschte Aussehen zu verpassen.

## 10.3 Dialoge

Zwischen Activities und/oder Layouts zu wechseln ist die bevorzugte Methode, Bildschirminhalte auszutauschen. Aber manchmal möchten Sie dem Benutzer vielleicht nur eine einfache Information anzeigen, etwa eine Fehlermeldung oder eine Warnung. Dazu möchten Sie nicht die aktuelle Activity verlassen, weil Sie bei der Rückkehr den letzten Zustand aufwendig restaurieren müssten. Die Lösung: Dialoge.

Dialoge sind Layouts, die über die aktuelle Bildschirmdarstellung gelegt werden. Wenn ein Dialog wieder geschlossen wird, kommt der vorherige Bildschirminhalt wieder zum Vorschein. Die laufende Activity bleibt bestehen.

Besitzt ein Dialog einen transparenten Rahmen, bleibt sogar ein Teil des vorherigen Bildschirms sichtbar.

Es gibt verschiedene Arten von Dialogen, die Sie leicht in Ihre App einbauen können. Zum einfachen Anschauen und Ausprobieren finden Sie ein kleines Projekt namens *DialogDemos* auf der Buch-DVD, das ich Ihnen jetzt Schritt für Schritt erkläre.

### Standard-Dialoge

Android bringt eine Reihe von Standard-Dialogen mit, die Sie verwenden können. Der Vorteil: Sie müssen kein eigenes Layout anlegen. Die Kehrseite der Medaille ergibt sich sofort daraus: Der Flexibilität sind gewisse Grenzen gesetzt.

Vier Standard-Dialoge hat Android im Sonderangebot:

- `AlertDialog`: Zeigt einen Text sowie Buttons oder Auswahllisten an.
- `ProgressDialog`: Eine Art »ich arbeite dran ...«-Fensterchen
- `DatePickerDialog`: Lässt den Benutzer ein Datum auswählen.
- `TimePickerDialog`: Lässt den Benutzer eine Uhrzeit auswählen.

Es gibt eine einfache und eine kompliziertere Art, Dialoge anzuzeigen. Die kompliziertere hat den Vorteil, dass der Dialog sichtbar bleibt, selbst wenn die aktuelle Activity in den Hintergrund verschwindet und zurückgeholt wird. Ich beschränke mich hier auf die einfache Variante, die in den meisten Fällen genügt.

Um einen `AlertDialog` zu erzeugen, verwenden Sie nicht, wie vielleicht zu erwarten, den Konstruktor der Klasse (der ist `protected` und damit Ihrem Zugriff entzogen), sondern eine Hilfsklasse `AlertDialog.Builder`:

```
AlertDialog.Builder builder = new AlertDialog.Builder(this);
```

Der Dialog-Baumeister erzeugt intern den `AlertDialog`. Bevor Sie ihn anzeigen lassen, können Sie ihn durch passende `set`-Methoden nach Wunsch anpassen:

```
builder.setMessage("Ich bin ein Hinweis.");
```

Schließlich könnten Sie den Dialog anzeigen mit:

```
builder.show();
```

Wenn Sie Buttons anzeigen möchten, können Sie vor dem `show()`-Aufruf welche hinzufügen, und zwar inklusive Behandlungsmethoden für die Clicks. Leider bietet das Anlass zur Verwirrung, denn der geforderte `OnClickListener` ist nicht derselbe, den Sie von den Views her kennen, sondern folgendes:

```
DialogInterface.OnClickListener
```

Dieses Interface unterscheidet sich durch den Aufbau der `onClick()`-Methode. Schauen Sie sich ein Beispiel für einen Akzeptieren-Button an:

```
DialogInterface.OnClickListener positivListener =
new DialogInterface.OnClickListener() {
   @Override
   public void onClick(DialogInterface dialog, int which) {
      ergebnis.setText("Hinweis akzeptiert");
      dialog.dismiss();
   }
};
```

Sie sehen, dass diese `onClick()`-Methode eine Referenz auf den Dialog sowie eine Information über den geklickten Button als Parameter übergibt. Im Beispiel

habe ich einen `positivListener` als lokale, anonyme Klasse geschrieben, der einen kurzen Text in einen TextView namens `ergebnis` schreibt. Außerdem wird der Dialog mittels `dismiss()` geschlossen.

Sie können diesen Listener mitsamt einer Button-Beschriftung dank der Methode `setPositiveButton()` an den Builder übergeben:

```
builder.setPositiveButton("akzeptiert", positivListener)
```

Wenn Sie einen Negativ-Button hinzufügen möchten, können Sie eine Unterscheidung nach gedrücktem Button in den ersten Listener einbauen oder einen zweiten Listener schreiben.

Praktisch an den Builder-Methoden ist, dass sie alle eine Referenz auf den Builder selbst zurückgeben. Deshalb können Sie die Methodenaufrufe in einer Zeile verketten:

```
(new AlertDialog.Builder(this)).setMessage("Ich bin ein Hinweis.")
.setPositiveButton("akzeptiert", positivListener )
.setNegativeButton("nein", negativListener)
.show();
```

Dank dieses ungewohnten, aber übersichtlichen Codes können Sie leicht Dialoge mit einfachen Bedienelementen erzeugen (Abbildung 10.11).


**Abbildung 10.11** Der »AlertDialog« kann auf flexible Weise mit Texten, Buttons oder Auswahllisten daherkommen.

Noch viel einfacher als der `AlertDialog` gestaltet sich der Umgang mit dem `ProgressDialog`.

Die zugehörige Klasse bietet Ihnen eine statische Methode `show()` an, die sich um alles kümmert:

```
ProgressDialog dialog = ProgressDialog.show(this, "Bitte warten",
"Nur ein paar Sekunden...", true);
```

Sie können den Titel und einen Hinweistext übergeben (Abbildung 10.12). Um den Dialog zu schließen, rufen Sie einfach die `dismiss()`-Methode auf. Wenn Sie schon genau wissen, wann das geschehen soll, können Sie einen Handler verwenden, um den Schließen-Code beispielsweise drei Sekunden in die gewünschte Zukunft zu versetzen:

```
final ProgressDialog dialog = ProgressDialog.show(this, "Bitte warten",
"Nur ein paar Sekunden...", true);
handler.postDelayed(new Runnable() {
   @Override
   public void run() {
      dialog.dismiss();
   }
}, 3000);
```

Sie kennen ja bereits die Methode `postDelayed()` aus dem Mückenspiel. Im genannten Beispiel wird eine anonyme Klasse erzeugt, die ein Runnable implementiert und deren `run()`-Methode mit dem Schließen des Dialogs betraut ist. Vorsicht: Code wie dieser kann leicht unübersichtlich werden, weil Sie vor lauter Klammern kaum noch sehen, wo eine Methode aufhört und wo eine Klasse.


**Abbildung 10.12** Der animierte »ProgressDialog« eignet sich hervorragend dazu, dem Benutzer Wartezeiten zu versüßen.

Sie können den `ProgressDialog` alternativ mit einem horizontalen Fortschrittsbalken ausstatten:

```
dialog.setProgressStyle(ProgressDialog.STYLE_HORIZONTAL);
```

Wenn Sie das tun, können Sie mit der Methode `setProgress()` anzeigen, wie weit der Vorgang, auf dessen Ende der Benutzer wartet, bereits fortgeschritten ist. Die Skala läuft dabei immer von 0 bis 100.

Bleiben der `DatePickerDialog` und der `TimePickerDialog`. Sie haben beide vermutlich schon einmal gesehen, wenn Sie die innere Uhr oder den Wecker Ihres Handys gestellt haben. Da beide Dialoge gleich funktionieren, zeige ich Ihnen nur ein Beispiel für die Datumsauswahl:

```
DatePickerDialog dialog = new DatePickerDialog(this, listener, jahr,
 monat, tag);
dialog.show();
```

Wie gewohnt müssen Sie einen Listener angeben, diesmal lautet der Name des zugehörigen Interfaces `DatePickerDialog.OnDateSetListener`. Der Listener verfügt über eine Methode `onDateSet()`, die Android aufruft, wenn der Benutzer ein Datum ausgewählt und bestätigt hat. Um auf dieses Ereignis zu reagieren, können Sie wie gewohnt eine anonyme Klasse verwenden:

```
OnDateSetListener listener = new DatePickerDialog.OnDateSetListener(
) {
   @Override
   public void onDateSet(DatePicker view, int year, int monthOfYear,
     int dayOfMonth) {
      ergebnis.setText("Datum: " + dayOfMonth + "." + (monthOfYear+1)
      + "." + year);
   }
};
```

Wundern Sie sich über das `(monthOfYear+1)`? Das umgeht eine gemeine Falle, die in Javas Kalender eingebaut ist: Die Monate werden nämlich von 0 bis 11 durchgezählt. Denken Sie daran, sonst versäumen Sie garantiert die Geburtstage Ihrer Freunde (Abbildung 10.13).

**Die Klasse »Calendar«**

Wenn Sie mit Geburtstagen, Feiertagen und Weltuntergangsterminen hantieren möchten, kommen Sie an der in Java eingebauten Klasse `Calendar` nicht vorbei. Wichtig daran ist, dass diese Klasse auch mit Zeitzonen umgehen kann, denn 12 Uhr mittags ist nicht überall auf der Welt Essenszeit.

Holen Sie sich ein tagesaktuelles Kalender-Objekt, indem Sie die statische Methode `getInstance()` aufrufen:

```
Calendar heute = Calendar.getInstance();
```

Das Objekt `heute` enthält Datum, Uhrzeit und die Standard-Zeitzone Ihrer laufenden Java-Umgebung. Die einzelnen Komponenten des Datums können Sie mit der `get()`-Methode abrufen:

```
int jahr = heute.get(Calendar.YEAR);
int monat = heute.get(Calendar.MONTH);
int tag = heute.get(Calendar.DAY_OF_MONTH);
```

Achten Sie darauf, dass der Monat Januar dem Zahlenwert 0 entspricht.

Mit `Calendar`-Objekten können Sie rechnen, zum Beispiel ein Jahr in die Zukunft gehen:

```
heuteInEinemJahr = heute.add(Calendar.YEAR, 1);
```

Außerdem können Sie `Calendar`-Objekte miteinander vergleichen:

```
if(heute.after(weltuntergangsTermin)) {
}
```


**Abbildung 10.13** Der »DatePickerDialog« bietet alles, was ein Weltuntergangsesoteriker braucht.

## Eigene Dialoge

Wenn Ihnen die bisher beschriebenen Möglichkeiten nicht genügen, gibt Ihnen Android die Macht, ein eigenes Layout zu verwenden und jedes Dialogelement selbst zu kontrollieren.

Erstellen Sie dazu eine Layout-Datei, wie Sie es gewohnt sind, mit dem Wizard New Android XML File. Wählen Sie als Wurzelelement ein FrameLayout, das den gesamten Bildschirm füllt (`fill_parent`). Der eigentliche Dialog wird ein weiteres FrameLayout, dem Sie eine feste Größe verpassen sowie die `layout_gravity center`. Innerhalb des inneren FrameLayouts können Sie sich nach Belieben austoben. Fügen Sie beispielsweise einen TextView und einen Button ein, um eine Nachricht anzuzeigen und dem Benutzer die Möglichkeit zu geben, den Dialog zu schließen. Letzteres funktioniert übrigens auf jeden Fall mit der Back-Taste am Gerät, wenn Sie das nicht durch zusätzliche Maßnahmen verhindern.

Bei Dialogen ist es sinnvoll, den Hintergrund, den sie teilweise verdecken, abzudunkeln. Diesen Effekt können Sie mit einem einfachen Trick erreichen: Erzeugen Sie eine Resource-Datei *colors.xml*, und legen Sie darin eine halb transparente Farbe an:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <color name="halb transparent">#88000000</color>
</resources>
```

Weisen Sie dann dem Wurzel-FrameLayout Ihres Dialogs diese Farbe als `background`-Eigenschaft zu.

Eigene Dialoge sind besonders bei Spielen interessant, wo es auf ein einheitliches grafisches Erscheinungsbild ankommt. Verwenden Sie beispielsweise ein 9-Patch als Hintergrund des inneren Dialog-Layouts, um sich vom bekannten, grauen Android-Look abzuheben (Abbildung 10.14).


**Abbildung 10.14** Eigene Dialoge basieren auf selbst gebastelten Layout-Dateien, die Sie nach Belieben gestalten können.

Wie bringen Sie nun einen solchen, eigenen Dialog auf den Bildschirm?

Zunächst erzeugen Sie ein neues `Dialog`-Objekt:

```
Dialog dialog = new Dialog(this, android.R.style.
Theme_Translucent_NoTitleBar_Fullscreen);
```

Sehr wichtig ist es, das »leere« Theme namens

```
Theme_Translucent_NoTitleBar_Fullscreen
```

anzugeben. Ansonsten zeichnet Ihnen Android graue Rahmen an Stellen, an denen Sie sie nicht gebrauchen können.

Weisen Sie dann dem neuen `Dialog`-Objekt das selbst gebaute Layout zu:

```
dialog.setContentView(R.layout.eigenerdialog);
```

Holen Sie sich Referenzen auf TextViews, die sich im Layout befinden, wenn Sie deren Inhalt ändern möchten. Dazu dient die bekannte Methode `findViewById()`, diesmal allerdings die Version, die die Klasse `Dialog` mitbringt:

```
TextView inhalt = (TextView) dialog.findViewById(R.id.inhalt);
inhalt.setText("Beispieltext");
```

Fast immer werden Sie dem Dialog Buttons hinzufügen. Auch die holen Sie sich mit `findViewById()`, um ihnen `OnClickListener` hinzuzufügen:

```
Button button = (Button) dialog.findViewById(R.id.ok);
button.setOnClickListener(new OnClickListener() {
   @Override
   public void onClick(View v) {
      dialog.dismiss();
   }
});
```

Schließlich zeigen Sie den Dialog an:

```
dialog.show();
```

Das obige Beispiel verwendet einmal mehr eine anonyme Klasse, die `OnClickListener` implementiert – diesmal wieder die altbekannte Version, die Views zur Verfügung stellen, nicht `DialogInterface.OnClickListener`, die bei den Android-Dialogen zum Einsatz kommt. Wie Sie sehen, schließt die Behandlungsmethode `onClick()` den Dialog einfach.

Damit Java das hier präsentierte Codebeispiel schluckt, müssen Sie die Deklaration des Objekts `dialog` mit einem `final`-Modifier versehen:

```
final Dialog dialog = ...
```

Der `OnClickListener` wird zu einem späteren Zeitpunkt aufgerufen, zu dem die lokale Variable `dialog` an sich gar nicht mehr vorhanden ist. Damit sie noch existiert und mit Sicherheit ihren Originalwert besitzt, wenn `onClick()` aufgerufen wird, ist der `final`-Modifizierer erforderlich.

### Toasts

Manchmal ist selbst ein Dialog zu viel des Guten. Wenn Sie den Benutzer nur kurz auf etwas hinweisen möchten, das er aber genauso gut übersehen darf, können Sie einen **Toast** verwenden. Das sind kleine Textfensterchen, die für wenige Sekunden eingeblendet werden und automatisch wieder verschwinden.

Der Vorteil besteht darin, dass es extrem einfach ist, solche Toasts anzuzeigen. Eine einzige Zeile genügt:

```
Toast.makeText(this, "Na dann Prost!", Toast.LENGTH_SHORT).show();
```

`Toast.makeText()` ist eine statische Methode, die ein `Toast`-Objekt zurückgibt. Das ist ein spezielles `Dialog`-Objekt, das Sie mit dem Aufruf seiner Methode `show()` auf den Bildschirm bringen – es wird daraufhin am unteren Bildschirmrand weich ein- und wieder ausgeblendet (Abbildung 10.15).

Anstelle eines Textes können Sie als zweiten Parameter eine String-Resource-ID übergeben, außerdem gibt es alternativ zu `Toast.LENGTH_SHORT` noch `Toast.LENGTH_LONG`. Ersteres zeigt die Meldung für etwa drei Sekunden an, Letzteres für etwa fünf.

Genau wie Dialoge können Sie auch Toasts selbst gestalten.


**Abbildung 10.15** Hinterlässt keinen bleibenden Eindruck und ist gerade deshalb oft gern genommen: der Toast.

Allerdings können Sie einem Toast nicht direkt eine Layout-Resource-ID verabreichen, sondern nur einen View. Wenn Sie mehr als einen View anzeigen möch-

ten, müssen Sie den `LayoutInflater` bemühen, um aus einem Layout eine View-Hierarchie zu generieren:

```
Toast toast = new Toast(this);
toast.setGravity(Gravity.BOTTOM, 0, 0);
toast.setDuration(Toast.LENGTH_SHORT);
View layout = getLayoutInflater().inflate(R.layout.eigenertoast,
null);
toast.setView(layout);
toast.show();
```

Sie sehen: Android stellt für jeden Bedarf den richtigen Dialog zur Verfügung, und wenn nicht, können Sie selbst das Heft in die Hand nehmen.

## 10.4 Layout-Gefummel

Wenn Sie komplizierte Layouts bauen wollen, werden Sie schnell zig FrameLayouts und LinearLayouts ineinander verschachteln, bis Sie sich einen Ariadnefaden wünschen, der Sie durch dieses Gewirr wieder hinaus an die frische Luft führt.

Manche Wunschlayouts lassen sich nur mit schmerzhaften Kompromissen zusammenschrauben – häufig mit dem Resultat, dass sie auf deutlich größeren (oder kleineren Bildschirmen) unerträglich aussehen.

Es gibt kein Patentrezept, das alle Fälle berücksichtigt, aber neben den bereits in den zahlreichen Beispiel-Apps in diesem Buch erwähnten Layouts möchte ich Ihnen zwei Methoden zeigen, die Sie bei Gelegenheit sicher gut gebrauchen können. Beispielcode finden Sie auf der Buch-DVD im Projekt *LayoutDemos*.

### RelativeLayouts

Stellen Sie sich vor, Sie wünschen sich ein Layout mit Kopf- und Fußbereich und einem großen Inhaltsbereich dazwischen. Kopf und Fuß sollen immer gleich hoch sein, alles dazwischen soll aber je nach Größe des Bildschirms variieren.

Mit einem vertikalen LinearLayout funktioniert das nicht: Sie müssen die Höhe des Inhaltsbereichs auf `fill_parent` setzen, aber dann bleibt kein Platz mehr für die Fußzeile. Dieses Dilemma lässt sich auf unterschiedliche Weise beheben – ich verwende es jetzt als Paradebeispiel für einen Anwendungsfall eines RelativeLayouts. Darin dürfen Sie jeden einzelnen View relativ zu anderen positionieren, wobei die Reihenfolge im Objektbaum nebensächlich ist.

Um das Wunschlayout mit Inhaltsbereich, Kopf- und Fußzeile zu gestalten, fügen Sie diese Elemente zunächst einem RelativeLayout hinzu. Vergeben Sie sprechende IDs, setzen Sie die Breite in allen Fällen auf `fill_parent`. Dann befördern Sie den Inhaltsbereich in der Objekthierarchie nach unten und setzen die Properties `layout_below` und `layout_above` auf Kopf- bzw. Fußleiste. Alle Views erhalten als Höhenangabe `wrap_content`, aber der Inhaltsbereich bekommt die `layout_gravity` `fill_vertical` verpasst. Damit die Kopfzeile oben und die Fußzeile unten landen, setzen Sie `layout_alignParentTop` respektive `layout_alignParentBottom` (Abbildung 1.15).

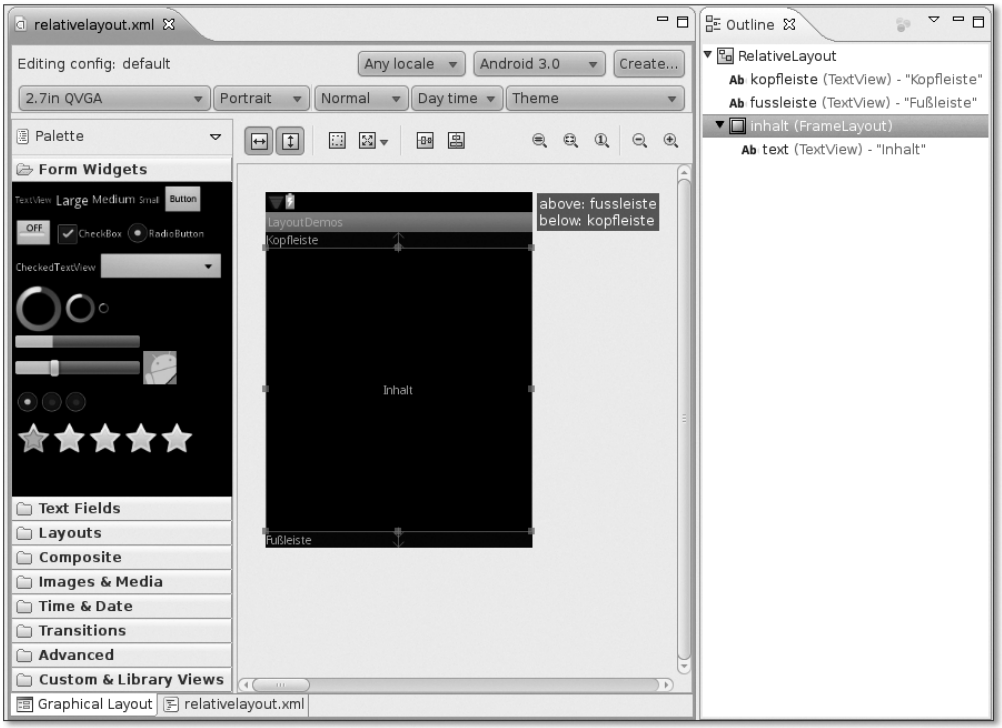



**Abbildung 10.16** Der Layout-Editor blendet Infoboxen ein, um Ihnen beim RelativeLayout den Überblick zu erleichtern.

Das RelativeLayout ermöglicht Ihnen außerdem, Views aneinander auszurichten: `layout_alignLeft` und Konsorten stehen zur Verfügung. Beachten Sie, dass alle genannten Properties nur im Kontextmenü eines Views auftauchen, wenn dieser Kind eines RelativeLayouts ist.

**Nicht jedes Eclipse-Plugin ist perfekt**

Wenn Sie versuchen, über das Kontextmenü einen Ausrichtungspartner für einen View einzustellen, könnte es passieren, dass Sie keine der zur Auswahl stehenden IDs zu sehen bekommen. Dann sind Sie Opfer eines Bugs im Android-Eclipse-Plugin, der mit etwas Glück behoben ist, bis Sie auf dieser Seite des Buches angekommen sind – ansonsten müssen Sie leider auf die XML-Ansicht umschalten und die Attribute von Hand eintippen. Immerhin steht dort wie üblich die Syntaxvervollständigung von Eclipse fehlerfrei zur Verfügung.

### Layout-Gewichte

Wenn Sie schon einmal mit HTML-Layouts gearbeitet haben, vermissen Sie möglicherweise eine wichtige Möglichkeit bei Android: prozentuale Größenangaben.

Dabei wäre es doch praktisch, einfach einem Layout-Bereich die Breite 50 % zu verpassen, damit er unabhängig von der Bildschirmgröße immer die halbe Breite einnimmt. Glücklicherweise hat Android eine ähnliche Funktion, die jedoch nicht mit Prozentwerten arbeitet, sondern mit **Gewichten**.

Wenn Sie zwei Elementen, die sich in einem horizontalen LinearLayout befinden, jeweils das `layout_weight 1` zuweisen, haben beide dasselbe Gewicht und werden gleich breit dargestellt (oder gleich hoch, wenn Sie ein vertikales LinearLayout verwenden). Allerdings müssen Sie dem äußeren Element, also dem LinearLayout, die Summe der Gewichte mitteilen – bei zwei enthaltenen Elementen also `weightSum 2`.

Das bedeutet, dass Sie immer auch die Summe ändern müssen, wenn Sie einzelne Gewichte ändern.

Auch nicht gleichmäßige Gewichtsverteilungen sind möglich. Beispielsweise können Sie einem Element ein Gewicht von 3 zuordnen und dem anderen 1. Tragen Sie eine Summe von 4 ins Eltern-LinearLayout ein, und das »schwerere« Element wird 75 % der Breite (bzw. Höhe) einnehmen, das leichtere 25 %.

Sie können diese Gewichtungen verschachteln, um eine gleichmäßige Skalierung auf verschiedenen Bildschirmgrößen zu erreichen (Abbildung 10.17).

Mit den bisher vorgestellten Layout-Konzepten und ihren möglichen Kombinationen können Sie beinahe jede vorstellbare Benutzeroberfläche bauen. Erst wenn Sie in die dritte Dimension expandieren oder mit frei beweglichen Elementen hantieren wollen, führt kein Layout am Fußweg vorbei. Aber die komplett selbst gebaute Bildschirmausgabe, die Sie in solchen Fällen bauen müssen, geht über den Dunstkreis dieses Buches weit hinaus.

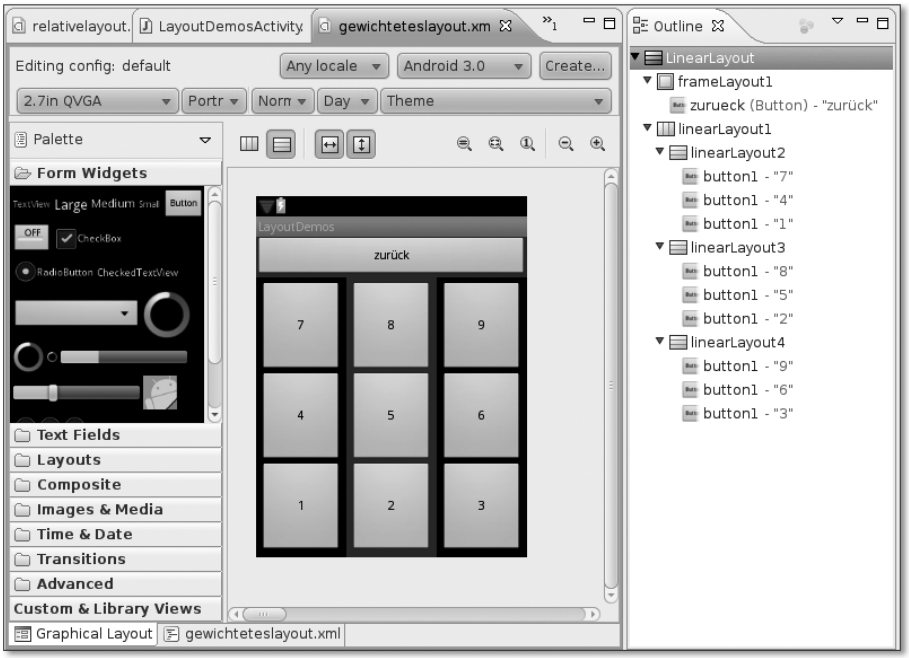



**Abbildung 10.17** Layout-Gewichte können Sie verschachteln, um sowohl vertikal als auch horizontal eine gleichmäßige Aufteilung zu erreichen. In diesem Beispiel haben alle Buttons ein Gewicht von 1, die vertikalen LinearLayouts ihrerseits ebenfalls.

## 10.5 Troubleshooting

Nichts funktioniert immer und ohne Probleme. Das gilt nicht nur in Politik, Sport oder Küche, sondern auch beim Entwickeln von Software. Damit Sie nicht ewig nach Lösungen für Probleme suchen, die andere (ich zum Beispiel) schon gefunden haben, liste ich Ihnen häufige Fälle auf – ohne Anspruch auf Vollständigkeit.

### Eclipse installiert die App nicht auf dem Handy

Erhalten Sie beim Starten über USB die folgende Meldung?

```
java.io.IOException: Unable to open sync connection
```

Schalten Sie am Handy das USB-Debugging aus und wieder an (Einstellungen • Anwendungen).

### App vermisst existierende Ressourcen

Besonders seltsam: Manchmal baut Eclipse Ihnen eine App, »vergisst« aber, ein paar Dateien ins APK zu packen. Vor allem solche, die Sie lange nicht geändert haben.

Die Lösung ist meistens, das Projekt komplett neu bauen zu lassen: Wählen Sie in Eclipse PROJECT • CLEAN. Manchmal müssen Sie das sogar zwei- oder dreimal tun, bis Sie wieder ein funktionierendes APK haben.

### LogCat bleibt stehen

Wenn Ihr LOGCAT-View partout keine neuen Protokollzeilen ausspuckt, schalten Sie zum DEVICES-View um, und doppelklicken Sie auf Ihr angeschlossenes Handy.

Alternativ können Sie anstelle des LOGCAT-Views übrigens auch ein Terminal (unter Windows: Eingabeaufforderung) öffnen und einen einfachen Befehl eintippen:

```
adb logcat
```

Die Darstellung ist zwar weder bunt noch übersichtlich, hängt sich aber nicht auf. Außerdem können Sie im Terminal-Fenster Ausschnitte des LogCats markieren, kopieren und anderswo einfügen – das geht im LOGCAT-View von Eclipse nicht.

*»Fertig!«*
*(Ewiger Programmiererirrtum)*

# 11 Apps veröffentlichen

Haben Sie Ihre erste eigene App fertiggestellt, die reif ist, um auf potenziell Millionen von Android-Geräten in aller Welt installiert zu werden?

Nun, sehr wahrscheinlich werden Sie unmittelbar vor der Veröffentlichung feststellen, dass Ihre App doch nicht so ganz fertig ist: hier noch eine Kleinigkeit und da auch noch ...

Irgendwann aber ist es so weit. Wenn Sie selbst die maximal mögliche Qualität Ihrer App auf 99 % einschätzen, sollten Sie an die Öffentlichkeit gehen. Wenn Sie nicht das Pech haben, dass Ihre App völlig ignoriert wird, etwa, weil sie mit 9,95 Euro doch ein wenig teuer geraten ist, werden Sie schon bald erste Downloads und Bewertungen bekommen.

Bis es so weit ist, müssen Sie jedoch ein paar letzte Vorbereitungen treffen.

## 11.1 Vorarbeiten

Bevor Sie Ihre App im Android Market veröffentlichen können, müssen Sie eine spezielle Version des APK bauen. Das APK, das Sie beim Test auf dem Emulator oder Handy verwenden, ist nämlich nur eine Debug-Version, die vom Market rundweg abgelehnt wird. Sie müssen Ihre App mit einem eigenen Zertifikat signieren.

### Zertifikat erstellen

Eine digitale Signatur weist den Besitzer des zugehörigen privaten Schlüssels zweifelsfrei als den Unterzeichner aus. Da der private Schlüssel üblicherweise mit einem Passwort geschützt ist, kann niemand die Identität des Unterzeichners (das sind Sie!) annehmen. Außerdem gewährleistet die Signatur die Integrität von Daten. Im Fall einer App bedeutet das: Sie wurde nicht nachträglich geändert.

Leider ist eine solche, recht simple Anwendung starker Kryptographie alles andere als alltäglich im Umgang mit dem Internet. Stellen Sie sich vor, alle E-Mails wären auf diese Weise signiert (wie das übrigens bei klassischen Briefen auf Papier irgendwann mal üblich war): Vortäuschen falscher Identität zum Ein-

sammeln von Kreditkartennummern, Trojaner, ja sogar Spam – das alles wäre kein Problem mehr, denn Sie würden nur E-Mails mit gültigen Signaturen von Ihren Bekannten akzeptieren. Falls Sie dieser Blick in eine bessere Welt neugierig gemacht hat, recherchieren Sie mal zum Thema im Netz, denn gängige E-Mail-Programme wie Thunderbird bringen die nötige Funktionalität schon längst mit. Einstweilen müssen wir alle jedoch mit Spam leben und freuen uns, dass dergleichen im Android Market durch den vernünftigen Einsatz digitaler Signaturen unterbunden wird.

Damit Sie Ihre App signieren können, benötigen Sie einen sogenannten **Keystore**, und darin einen **privaten Schlüssel** samt zugehörigem Zertifikat. Während der Schlüssel lediglich eine mit Passwort geschützte, ziemlich lange Zahlenfolge ist, steht im Zertifikat Ihr Name oder der Name Ihrer Firma.

Sie können einen Keystore ohne Weiteres mit dem Android-Eclipse-Plugin erzeugen. Wählen Sie dazu im Kontextmenü Ihres Projekts Android Tools • Export Signed Application Package aus. Suchen Sie sich aus, wo Sie den Keystore speichern möchten, und wählen Sie ein Passwort (Abbildung 11.1).

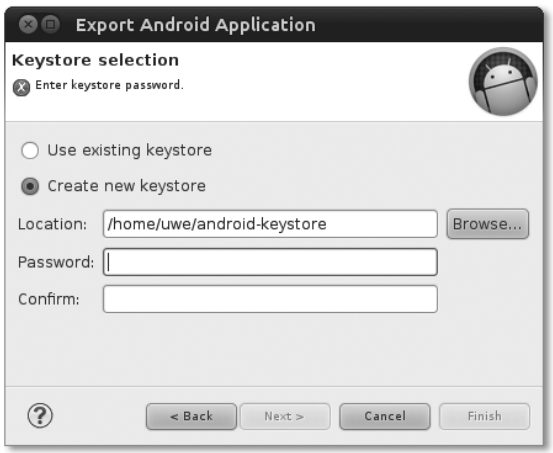



**Abbildung 11.1** Das Android-Plugin für Eclipse erzeugt Ihren Keystore samt Zertifikat und Schlüssel. Achten Sie darauf, alle Daten korrekt einzugeben – Sie können sie nicht nachträglich verändern.

Achten Sie darauf, dass es zwei verschiedene Passwörter gibt: Eines für den Keystore, und eines für den darin befindlichen Schlüssel. Natürlich können Sie dasselbe Passwort verwenden; falls Sie das nicht tun, müssen Sie aufpassen, welches Sie wo eingeben.

Wenn Sie alles richtig gemacht haben, erzeugt Ihnen Eclipse ein signiertes APK am gewünschten Ort. Sorgen Sie dafür, dass Sie es am Dateinamen von der

Debug-Version unterscheiden können, damit Sie nicht versehentlich die falsche Datei beim Android Market hochladen.

**Zwei Zertifikate, eine App?**

Zur Sicherheit erlaubt Android nicht, eine App, die mit Zertifikat A signiert ist, mit einer identischen App (d. h. mit dem gleichen Paketnamen) zu überschreiben, die mit Zertifikat B signiert ist.

Falls Sie also Ihre App mit Debug-Zertifikat installiert haben, können Sie sie nicht mit jener überschreiben, die Sie mit Ihrem endgültigen Zertifikat signiert haben. Sie müssen vorher deinstallieren.

Dasselbe Problem tritt sogar mit zwei verschiedenen Debug-Zertifikaten auf. Falls Sie mit zwei Eclipse-Installationen auf zwei Rechnern arbeiten, kann die vom ersten Eclipse erzeugte APK-Datei nicht mit der aus der anderen Installation überschrieben werden, weil jede beim ersten Benutzen ein eigenes Debug-Zertifikat generiert hat.

*Verlieren Sie auf keinen Fall Ihren Keystore und Ihre Passwörter, sonst können Sie Ihre App später nicht mehr durch ein Update aktualisieren!*

### Das Entwickler-Konto

Um im Android Market Apps zu veröffentlichen, benötigen Sie einen speziellen Google-Account. Anlegen können Sie den Account hier:

*https://market.android.com/publish/signup*

Diese Seite weist Sie direkt darauf hin, dass Kosten auf Sie zukommen, und zwar in Höhe von 25 US-Dollar. Da Sie diesen Betrag ausschließlich über Googles Bezahldienst Checkout begleichen können, benötigen Sie zwingend eine Kreditkarte.

Andere App-Veröffentlichungsplattformen sind kostenlos, haben aber natürlich bei Weitem nicht die Verbreitung wie der Google Market, daher kommen Sie kaum um die Registrierung herum.

Im Gegensatz zu den meisten anderen Lizenzvereinbarungen empfehle ich Ihnen, die des Android Markets zu lesen. Immerhin tragen Sie eine gewisse Verantwortung, wenn Sie selbst gebaute Software in die Welt setzen. Dass Sie keine gewaltverherrlichenden oder pornografischen Apps einstellen dürfen, mag noch naheliegend sein; Details im Hinblick auf die Ausschüttung von eventuellen Einnahmen oder das Recht Googles, Ihre App ohne Ihr Wissen von allen Handys zu deinstallieren, schon weniger.

Wenn Ihr Entwicklerkonto freigeschaltet ist, können Sie sofort damit beginnen, Ihre erste App hochzuladen.

## Die Entwicklerkonsole

Willkommen auf Ihrer persönlichen Entwicklerkonsole! Später, wenn Sie Apps veröffentlicht haben, sehen Sie alle in einer übersichtlichen Liste, komplett mit aktueller Bewertung durch Benutzer sowie einigen Statistiken. Anfangs haben Sie lediglich die Möglichkeit, Ihr Profil zu bearbeiten, und Sie sehen einen blauen Button, der es Ihnen erlaubt, Ihre erste App hochzuladen.

Der erste Schritt ist der einfachste: Wählen Sie die signierte Version Ihrer APK-Datei aus, und laden Sie sie hoch. Keine Sorge, damit ist sie nicht sofort veröffentlicht. Zuerst müssen Sie noch eine Reihe zusätzlicher Angaben machen (Abbildung 11.2).

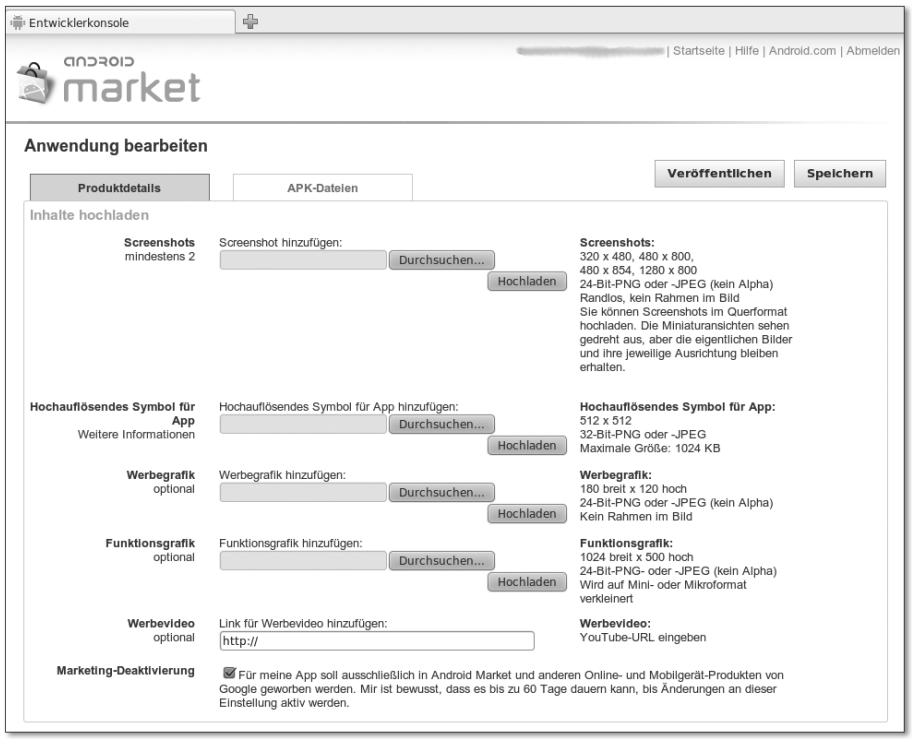



**Abbildung 11.2** Die Entwicklerkonsole erwartet von Ihnen einen Haufen Informationen zu Ihrer App.

Einige Daten sind erforderlich, andere Angaben freiwillig.

Im Webformular ganz oben haben Sie die Möglichkeit, Screenshots hochzuladen. Die sind so was wie die erweiterte Visitenkarte Ihrer App: Deshalb sind auch mindestens zwei Bilder Pflicht. Fertigen Sie die Screenshots an, indem Sie im DEVICES-View in Eclipse auf das Icon mit dem Fotoapparat klicken. Speichern Sie

hübsche oder aussagekräftige Screenshots im PNG- oder JPG-Format, und laden Sie sie hoch. Fotografieren Sie alle wichtigen Bildschirme Ihrer App, aber nicht jeden dreimal!

Die nächste Aufgabe ist eine der kniffligsten. Sie sollen ein Symbol für Ihre App hochladen. Nun, sicher haben Sie ein Icon für Ihre App entworfen und es bereits eingebaut. Aber der Market verlangt hier ernsthaft ein Icon in der Auflösung 512 × 512 Pixel! Deshalb ist es eine hervorragende Idee, App-Icons von Anfang an in dieser Auflösung anzulegen – oder gleich mit einem Vektorgrafikprogramm wie Inkscape. Alpha-Transparenz ist im Gegensatz zu den Screenshots beim Icon möglich und sinnvoll. Geben Sie sich Mühe mit Ihrem Icon, es ist im wahrsten Sinne des Wortes das erste Aushängeschild Ihrer App!

Sie können optional weitere Dateien hochladen: eine niedrig aufgelöste Werbegrafik, eine hochauflösende Funktionsgrafik oder ein YouTube-**Video.** Sie alle erscheinen an unterschiedlichen Stellen in den Handy- und Browser-Versionen des Android Markets. Beachten Sie, dass Sie zwingend weiter unten einen Promotext eingeben müssen, wenn Sie eine Werbegrafik hochladen.

Die zweite Sektion des Veröffentlichungsformulars ist den Texten vorbehalten (Abbildung 11.3).

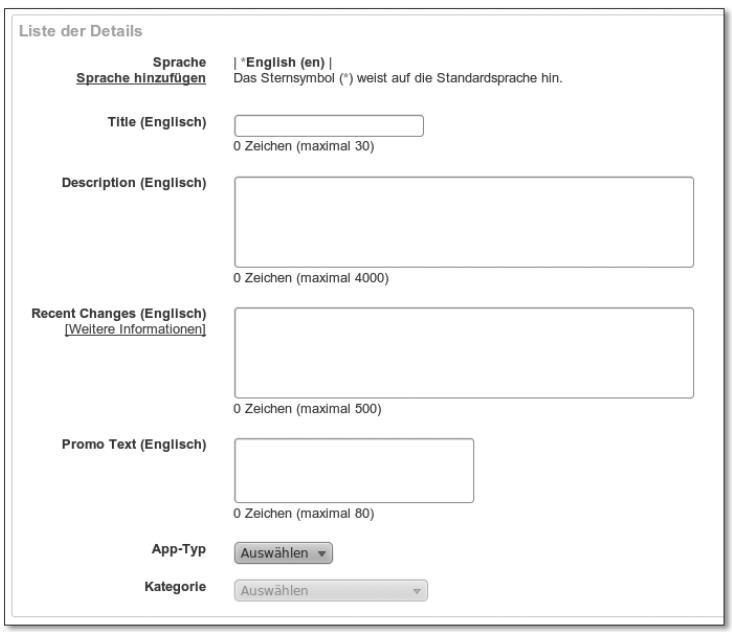



**Abbildung 11.3** Sie können Namen und Beschreibung Ihrer App für alle möglichen Sprachen eingeben.

Im Gegensatz zu Grafiken sind Texte sprachabhängig. Der Clou ist, dass eine deutsche Beschreibung im Grunde genügt: Fügen Sie weitere Sprachen per Klick hinzu, und wählen Sie die automatische Übersetzung.

Natürlich wissen wir alle nicht erst seit Spam aus Nigeria, dass computergenerierte Übersetzungen meist weniger zutreffend als amüsant sind. Sie sind daher gut beraten, eine vernünftige Übersetzung einzutragen. Überflüssig zu erwähnen, dass eine chinesische Beschreibung sinnlos ist, wenn Ihre App selbst nur deutsch spricht.

Wählen Sie schließlich Typ und Kategorie Ihrer App. Gerade bei Spielen ist die Auswahl leider sehr übersichtlich; immerhin hat Google neuerdings »Rennspiele« hinzugefügt (eine für »Einfache Puzzles« wäre sinnvoller gewesen). Kategorien für Rollenspiele, Adventures oder Multiplayer-Spiele gibt es leider bisher nicht.

In der dritten großen Box namens Veröffentlichungsoptionen (Abbildung 11.4) können Sie den Verkaufspreis für Ihre App festlegen. Ich gebe Ihnen dazu einen kleinen Tipp: Kostenlose Apps werden schätzungsweise *zehntausendmal* öfter heruntergeladen als kostenpflichtige. Leider ist auch zehntausendmal null immer noch null – das bedeutet vor allem eines: Wenn Sie mit Ihrer App Geld verdienen möchten, müssen Sie es sehr, sehr geschickt anstellen. Eine kostenlose Demo-Version ist dabei die einfachste Verkaufsmasche, In-App-Payment die weitaus kompliziertere (dazu kommen wir noch).

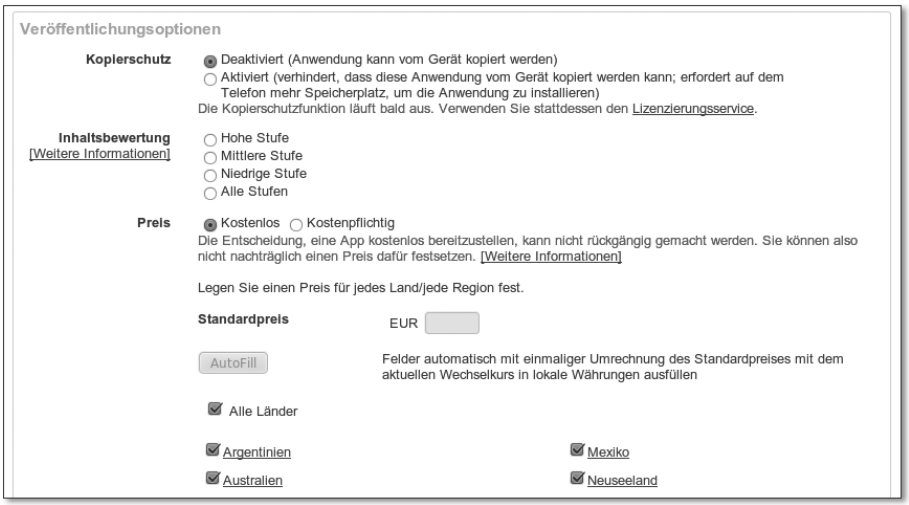



**Abbildung 11.4** Bei den Veröffentlichungsoptionen geht es um Verkaufspreis und -regionen. Die komplette Länderliste erspare ich Ihnen auf diesem Screenshot.

Verzichten Sie auf den veralteten Kopierschutzmechanismus, der laut Google irgendwann »nicht mehr unterstützt« wird (das schreiben sie allerdings schon seit einem Jahr). Wenn Sie sicherstellen wollen, dass eine kostenpflichtige App nicht von Gaunern vom Gerät gezogen und irgendwo ins Internet gestellt wird, müssen Sie den Lizensierungsservice verwenden. Der prüft durch eine Abfrage beim Android Market, ob der aktuelle Benutzer die App irgendwann tatsächlich gekauft hat, sonst verweigert die App den Start.

Wie man den Lizensierungsservice verwendet, erkläre ich Ihnen in diesem Buch nicht, aber wenn Sie so weit sind, dass Sie glauben, ihn zu brauchen, verstehen Sie sicher auch die zugehörige Google-Dokumentation. Allzu kompliziert ist die Sache jedenfalls nicht.

Preise können Sie für jedes Land separat festlegen. Achten Sie darauf, dass es länderabhängige Mindestpreise gibt. Da Google außerdem etwa 30 % des Verkaufspreises behält, wären Verkaufspreise unter 0,79 Euro ohnehin wenig rentabel.

Wenn Sie das Formular korrekt ausgefüllt haben, quittiert die Entwicklerkonsole das mit einem lapidaren Hinweis »Applikation veröffentlicht«. Von wegen lapidar! Sie haben gerade den letzten und entscheidenden Schritt auf dem Weg des App-Entwicklers getan. Herzlichen Glückwunsch!

Und anstatt jetzt minütlich die F5-Taste zu drücken, um die ersten Downloads nicht zu versäumen, bereiten Sie sich lieber auf die Arbeiten vor, die noch vor Ihnen liegen. Denn mit der Geburt fängt das Leben erst an.

Außerdem aktualisiert Google Download-Statistiken ohnehin nur alle paar Stunden – gehen Sie also für heute ruhig schlafen, und schauen Sie morgen wieder rein!

## 11.2 Hausaufgaben

Nach der Veröffentlichung ist vor der Veröffentlichung: Spätestens nach den ersten Rückmeldungen von Benutzern kommen Sie auf großartige Ideen, wie Sie Ihre App verbessern können. Aber das ist nicht alles: Software braucht Pflege, dann haben Sie länger was davon. In dieser Hinsicht ähnelt sie einem Haustier. Denken Sie jetzt aber bitte nicht an einen braven Hund, sondern besser an eine besonders dickköpfige Katze ...

### Updates

Noch bevor die ersten Benutzer Ihre App heruntergeladen haben, ist Ihnen eine großartige Verbesserung eingefallen. Minuten später haben Sie sie auch schon eingebaut und getestet – was nun?

Bevor Sie ein weiteres, signiertes APK Ihrer App erzeugen, müssen Sie unbedingt die Versionsnummer im Manifest erhöhen. Es genügt, den Versionscode um 1 zu erhöhen. Bewährt hat es sich, die in der IT meist übliche dreistellige Versionszählung zu verwenden: Version 1.0.0 erhält den Versionscode 100, Version 1.0.1 Code 101. Natürlich können die Unter- und Unterunterversionsnummern bei diesem Spielchen nicht zweistellig werden, aber mit dieser Einschränkung kann man leben. Wenn nicht, erweitern Sie einfach den Versionscode um eine Stelle: Aus Version 1.0.10 wird der Code 1010. Bloß einen Weg zurück zu dreistelligen Versionscodes gibt es dann nicht mehr, denn neue Versionen sollten immer einen höheren Code haben als vorherige.

Achten Sie drauf, dass Sie das hochzuladende APK mit demselben Schlüssel signieren wie das vorherige, ansonsten lehnt die Entwicklerkonsole es rundweg ab.

Wenn Sie ein neues APK hochgeladen haben, können Sie es sofort aktivieren (Abbildung 11.5) oder vorher Beschreibung oder Screenshots ändern, damit alles zueinander passt. Wenn Sie damit fertig sind, müssen Sie auf jeden Fall den Speichern-Button oben rechts anklicken.

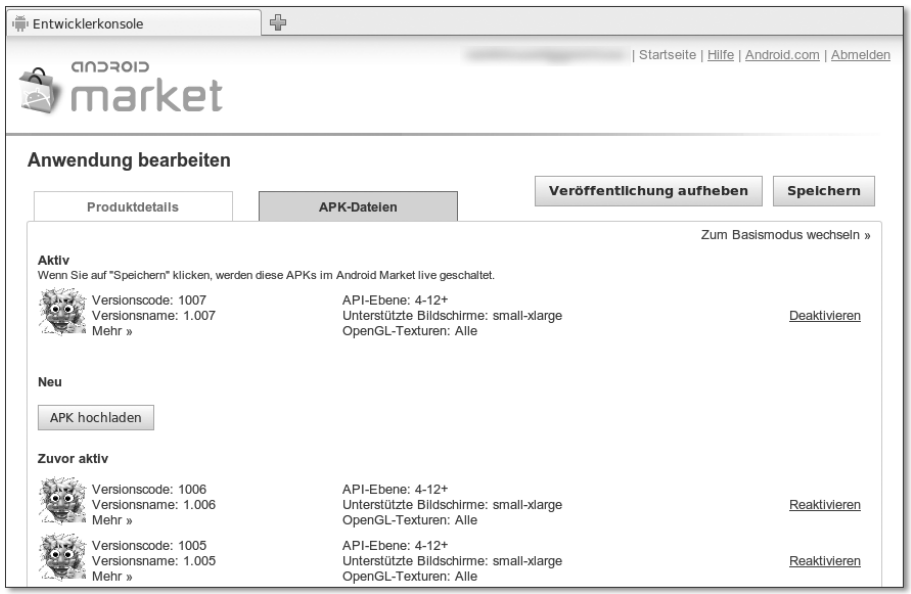



**Abbildung 11.5** Im erweiterten Modus können Sie nicht nur neue App-Versionen hinzufügen, sondern auch ältere reaktivieren.

Immer wenn Sie ein Update veröffentlichen, landet Ihre App in den »Neue-Apps«-Listen des Android Markets ganz oben. Es hängt von der Tageszeit und der Anzahl gleichzeitiger Neuheiten ab, wie lange Ihre App auf einem einigermaßen hohen Platz steht. Tatsache ist, dass ein geringer Prozentsatz aller Android-Benutzer in dieser Liste alles ausprobiert, was ihnen annähernd interessant vorkommt.

Die Folge ist, dass ein Update nichts anderes ist als kostenlose Werbung. Sie werden später an den Download-Statistiken sehen, dass die Kurven zu jenen Zeitpunkten steil ansteigen, an denen Sie ein Update veröffentlicht haben. Leider fallen die Kurven danach auch wieder ab, denn wenn Ihre App nicht mehr zu den neuesten gehört, hat sie eine Sichtbarkeit nahe null.

Es müsste schon jemand zufällig nach Ihrer App oder einem Stichwort in Ihrer Beschreibung suchen, um sie zu finden. Aber wie wahrscheinlich ist das?

Die einfachste Möglichkeit, viele neue Benutzer zu bekommen, liegt auf der Hand: häufige Updates. Wenn Sie die neugierigen Downloader dazu bringen, Ihre App nicht wieder zu deinstallieren, sondern möglichst sogar ihren Freunden zu empfehlen, werden Sie einen stetigen Aufwärtstrend Ihrer Zahlen beobachten.

Womit wir beim nächsten Thema wären.

### Statistiken

In der Übersicht aller Ihrer Apps verrät Ihnen die Entwicklerkonsole nur das, was auch im Market selbst für jedermann sichtbar ist: die Anzahl der Installationen und die Anzahl der aktiven Installationen. Wenn Sie die beiden Zahlen voneinander abziehen, wissen Sie, wie viele Leute Ihre App wieder vom Handy geworfen haben. Das Verhältnis zwischen aktiven Installationen zu deren Gesamtzahl ist wiederum ein direktes Maß für die Qualität Ihrer App: Je mehr Leute Ihrer App einen Teil ihres wertvollen Handyspeichers gönnen, umso besser kommt sie offensichtlich an. Gerade bei kostenlosen Apps ist jedoch die Anzahl der Nur-mal-eben-Ausprobierer sehr groß. Das Resultat ist, dass ein Anteil aktiver Installationen um 20 % schon ganz gut ist. Aussagekräftig ist der Wert allerdings erst nach einigen Wochen bis Monaten, oder wie häufig räumen Sie Ihr Handy auf?

Die Entwicklerkonsole bietet Ihnen zu jeder Ihrer Apps einen ausführlichen Statistik-Bildschirm an. Der zeigt Ihnen zum Beispiel den genauen zeitlichen Verlauf der Anzahl aktiver Installationen (Abbildung 11.6).

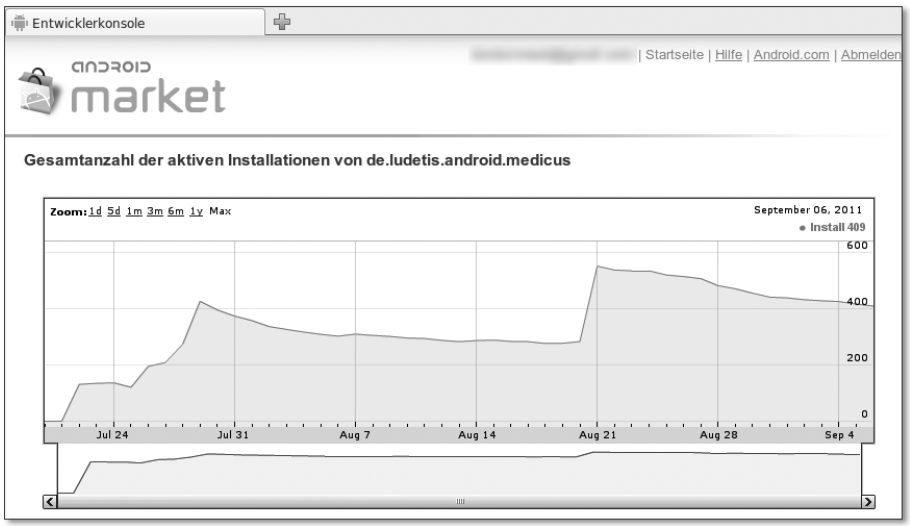


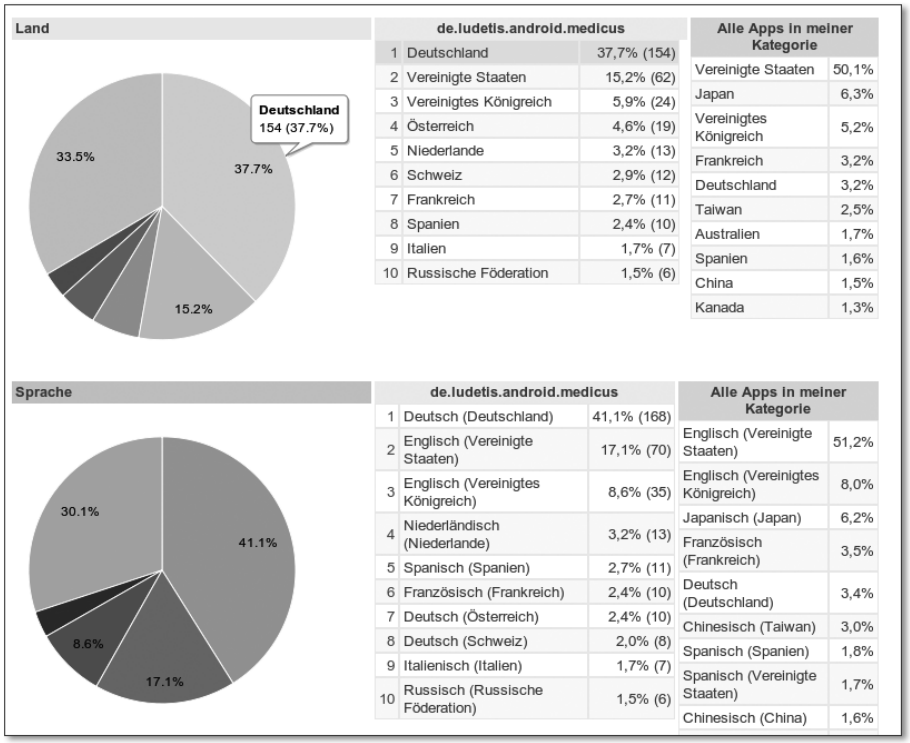

**Abbildung 11.6** Der Verlauf der Anzahl der Installationen lässt einige Rückschlüsse zu.

Die Zahlen werden zwar nur zeitversetzt (um Stunden, manchmal Tage) zur Verfügung gestellt, zeigen aber sehr genau die Auswirkung von Updates: Jedes erzeugt automatisch einen Anstieg.

Und jedem Anstieg folgt ein Abfall. Je steiler der ist, desto schneller wird den Leuten Ihre App langweilig. Gerade bei Spielen ist das eine interessante Messgröße. Sie können leicht abschätzen, wie oft Sie Ihre App updaten müssen, um die Anzahl der Benutzer zu steigern. Wenn das Ergebnis allerdings lautet »dreimal täglich«, sollten Sie das Konzept Ihrer App grundsätzlich überdenken.

Die weiteren Statistiken sind ebenfalls hochinteressant. In Tortendiagrammen sehen Sie die Verteilung Ihrer Benutzer auf Android-Versionen, häufige Geräte, Herkunftsländer und Sprachen. Spannend sind die Vergleichsangaben für die gesamte Kategorie (Abbildung 1.15).

Beispielsweise können Sie sehen, dass die Vereinigten Staaten über die Hälfte aller Android-Spieleinstallationen verfügen (Stand September 2011), gefolgt von Japan und Großbritannien. Sie werden Japan aber nicht ohne Weiteres bei Ihrer eigenen App finden, es sei denn, sie spricht japanisch oder kommt ohne Worte aus. Ähnliches gilt für Taiwan, China und Korea.




| | de.ludetis.android.medicus | |
|---|---|---|
| 1 | Deutschland | 37,7% (154) |
| 2 | Vereinigte Staaten | 15,2% (62) |
| 3 | Vereinigtes Königreich | 5,9% (24) |
| 4 | Österreich | 4,6% (19) |
| 5 | Niederlande | 3,2% (13) |
| 6 | Schweiz | 2,9% (12) |
| 7 | Frankreich | 2,7% (11) |
| 8 | Spanien | 2,4% (10) |
| 9 | Italien | 1,7% (7) |
| 10 | Russische Föderation | 1,5% (6) |

| Alle Apps in meiner Kategorie | |
|---|---|
| Vereinigte Staaten | 50,1% |
| Japan | 6,3% |
| Vereinigtes Königreich | 5,2% |
| Frankreich | 3,2% |
| Deutschland | 3,2% |
| Taiwan | 2,5% |
| Australien | 1,7% |
| Spanien | 1,6% |
| China | 1,5% |
| Kanada | 1,3% |

| | de.ludetis.android.medicus | |
|---|---|---|
| 1 | Deutsch (Deutschland) | 41,1% (168) |
| 2 | Englisch (Vereinigte Staaten) | 17,1% (70) |
| 3 | Englisch (Vereinigtes Königreich) | 8,6% (35) |
| 4 | Niederländisch (Niederlande) | 3,2% (13) |
| 5 | Spanisch (Spanien) | 2,7% (11) |
| 6 | Französisch (Frankreich) | 2,4% (10) |
| 7 | Deutsch (Österreich) | 2,4% (10) |
| 8 | Deutsch (Schweiz) | 2,0% (8) |
| 9 | Italienisch (Italien) | 1,7% (7) |
| 10 | Russisch (Russische Föderation) | 1,5% (6) |

| Alle Apps in meiner Kategorie | |
|---|---|
| Englisch (Vereinigte Staaten) | 51,2% |
| Englisch (Vereinigtes Königreich) | 8,0% |
| Japanisch (Japan) | 6,2% |
| Französisch (Frankreich) | 3,5% |
| Deutsch (Deutschland) | 3,4% |
| Chinesisch (Taiwan) | 3,0% |
| Spanisch (Spanien) | 1,8% |
| Spanisch (Vereinigte Staaten) | 1,7% |
| Chinesisch (China) | 1,6% |

**Abbildung 11.7** Die Nationen-Torten sagen einiges über die Vorlieben in manchen Ländern aus.

Vergessen Sie nicht, dass Asien ein riesiger Markt ist, der überproportional wächst. Wenn Sie feststellen, dass Ihre App gut läuft, aber in Asien überhaupt nicht, schauen Sie, ob Sie irgendjemanden finden, der Ihre App übersetzt. Das ist natürlich umso aufwendiger, je mehr Texte Sie verwenden – für Spiele bedeutet eine internationale Symbolsprache beispielsweise am Ende eine echte Geldersparnis.

## Fehlerberichte

Stellen Sie sich vor, ein Android-Benutzer hat Ihre App nicht nur heruntergeladen, sondern auch einen Kommentar samt miesepetriger Bewertung veröffentlicht: Die App stürzt auf seinem Handy vom Typ Niegehört XL nämlich sofort beim Start ab.

Es gibt jetzt drei Möglichkeiten:

- Sie kennen rein zufällig jemanden, der das gleiche Handy besitzt, und überreden denjenigen, es an Ihr Eclipse zu stöpseln.

- Sie haben noch nie von dem Handy gehört, finden im Internet keine Hinweise auf irgendwelche ungewöhnlichen Eigenschaften und haben keine Chance, den Fehler zu finden. Schade.
- Der Benutzer hat nach dem Crash auf den richtigen Knopf gedrückt, sodass Sie in der Entwicklerkonsole einen Fehlerbericht vorfinden.

Werfen wir einen genaueren Blick auf den letzten Fall, denn in den beiden anderen erübrigt sich eine Diskussion.

Grundsätzlich gibt es zwei Kategorien von Fehlern, zu denen Berichte existieren können: Hänger und Abstürze. Erstere sind Fälle, in denen die App mehrere Sekunden nicht reagierte und der Benutzer sie ungeduldig abgeschossen hat. Da Sie aber die Hinweise in diesem Buch zum Thema Handler und Threads genau beachtet haben, kommen Hänger in Ihrer App so gut wie nie vor, richtig?

Bei Abstürzen sieht die Sache anders aus. Man kann nie alle möglichen Fälle berücksichtigen. Beschränken wir uns also auf die Absturzfehler (Abbildung 11.8).

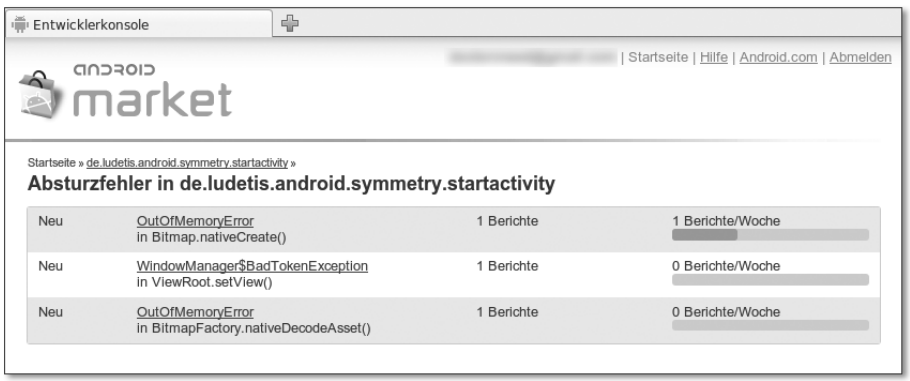



**Abbildung 11.8** Sie sollten jeden gemeldeten Absturz genau untersuchen.

Jeder Absturzfehler enthält einen Stacktrace, Datum, Uhrzeit und Versionsnummer. Freundlicherweise fasst die Entwicklerkonsole gleiche Fehler zusammen, aber nur pro Version. Das ist sehr vernünftig, denn die Stacktraces enthalten Zeilenangaben, die in verschiedenen Versionen unterschiedlich sein können. Achten Sie darauf, wenn Sie die Stacktraces auf ihre Ursache hin untersuchen.

Manchmal hat tatsächlich ein Benutzer Ihre App auf eine Art und Weise verwendet, die Sie überhaupt nicht bedacht hatten, geschweige denn getestet. Das Resultat ist ein Crash, den Sie nun analysieren müssen. Oft sehen Sie die Ursache schnell und können zumindest einen Workaround einbauen.

Andere Fehler sind nicht so leicht zu identifizieren, weil sie tief im Android-System liegen. Manchmal hilft es, im Internet zu recherchieren. Möglicherweise hatte ein anderer Entwickler ein ähnliches Problem, und wenn Sie eine Menge Glück haben, hat er es gelöst – und die Lösung ebenfalls ins Netz gestellt.

Bei wieder anderen Fehlern haben Sie keine Chance: In Abbildung 11.8 sehen Sie beispielsweise einen OutOfMemoryError. Offensichtlich war die App zu speicherhungrig, oder der Garbage Collector konnte nicht schnell genug aufräumen. Ob Sie an der Angelegenheit etwas ändern können, müssen Sie abwägen: Vielleicht ist Ihre App wirklich sehr speicherhungrig. Verwenden Sie eine Hilfs-App wie Android System Info, um den Speicherbedarf zur Laufzeit abzulesen. Da es meist große Bilder sind, die zu viel Speicher verbrauchen, ziehen Sie in Erwägung, niedriger aufgelöste Versionen für ältere Geräte im Resource-Verzeichnis *drawable-ldpi* mitzuliefern.

Natürlich müssen Sie die Häufigkeit von Fehlern in Relation zu Ihrer Benutzeranzahl sehen: Ein Absturzbericht pro Woche fällt bei 1.000 Benutzern nicht ins Gewicht – aber 10 bei 100 Benutzern durchaus.

Die Fehlersuche mag anstrengend sein, aber stellen Sie sich vor, Sie wären nicht der Entwickler, sondern der Benutzer: Sie würden eine App, die öfter als ein-, zweimal abstürzt, wahrscheinlich sehr bald deinstallieren. Sie können das verhindern, indem Sie jedem Fehler nachgehen und ihn, wenn möglich, korrigieren. Das fällige Update bringt mehr Qualität und automatisch neue Benutzer – es lohnt sich also doppelt.

## 11.3 In-App-Payment

Falls Sie noch keine App kennengelernt haben, die In-App-Payment anbietet, ist das kein Wunder: Es ist eine relativ neue Art, mit einer App Geld einzunehmen. Freilich gibt es das System im Sektor der Browser-Spiele schon lange – fast alle ernähren sich davon.

Entscheidend ist: Das Spiel selbst ist kostenlos, und man kann es auch ohne Weiteres mit gewissem Spaß und Erfolg genießen, ohne einen Cent zu zahlen. Wozu soll der Spieler dann überhaupt Geld ausgeben?

Das Prinzip besteht meist darin, Zeit gegen Geld zu tauschen. Vorgänge, die ziemlich lange dauern, z. B. das Bauen von Infrastruktur, lassen sich mit Spielgeld oder einer speziellen Währung beschleunigen. Und die zugehörigen Münzen oder Diamanten bekommt man eben gegen echtes Geld, oft in verschiedenen

Chargen (10, 50, 100 Diamanten) zu steigenden Preisen. So haben Zahlungswillige keinen grundsätzlichen Vorteil, den andere als unfair auffassen könnten, sondern sie sparen nur Zeit.

Seit Frühjahr 2011 bietet der Google Market die Möglichkeit, neben Bezahl-Apps auch Artikel innerhalb einer App zu verkaufen, und zwar über eine einheitliche Schnittstelle, die an Google Checkout geknüpft ist, das zahlungswillige Benutzer ohnehin kennen (Abbildung 11.9). Genau wie bei Apps bleiben 30 % des Geldes bei Google, den Rest kassieren Sie. Dass Sie solche Einnahmen genau wie die aus App-Verkäufen gegebenenfalls versteuern müssen, werde ich hier nicht weiter diskutieren – wenden Sie sich dazu an den Steuerberater Ihres Vertrauens.

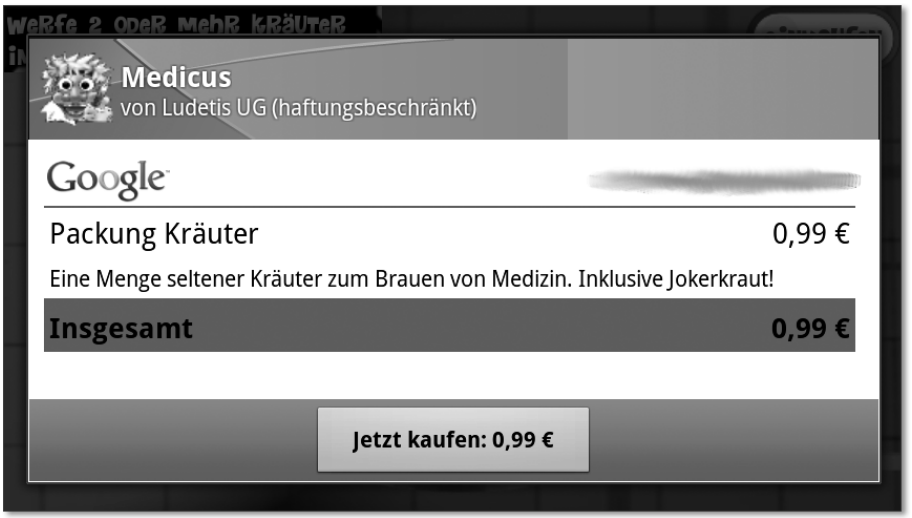



**Abbildung 11.9** In-App-Payment ist praktisch und genauso sicher wie eine Bezahl-App.

Wie Sie diese Schnittstelle verwenden können, erkläre ich Ihnen in den folgenden Abschnitten. Leider ist das Thema relativ umfangreich, sodass ich Ihnen keine kompletten Codebeispiele zeigen werde. Die bisher in diesem Buch als Beispiele vorgestellten Apps eignen sich ohnehin nicht für In-App-Payment, daher werde ich das Beispiel aus der öffentlichen Google-Dokumentation (zu finden auf *http://developer.android.com*) für die Erklärung heranziehen.

Lassen Sie uns zunächst darüber sprechen, welche Artikel Sie innerhalb einer App anbieten können.

### In-App-Produkte

Natürlich eignet sich nicht jede App dafür, In-App-Produkte anzubieten. Bei Spielen fällt Ihnen sicher sofort Spielgeld ein oder ein oder mehrere Pakete mit zusätzlichen Levels.

Prinzipiell ist In-App-Payment aber bei jeder App eine Überlegung wert: Stellen Sie sich vor, dass Sie eine kostenlose Standardversion und eine Premium-Variante zu einer App zusammenfassen. Die Premium-Features werden dann via In-App-Payment freigeschaltet.

Grundsätzlich gibt es zwei unterschiedliche Arten von In-App-Produkten:

- Produkte, die jeder Benutzer höchstens einmal kaufen kann (z. B. Premium-Freischaltung, Level-Packs)
- Produkte, die jeder Benutzer beliebig oft kaufen kann (z. B. Spielgeld)

Wenn Sie entschieden haben, welche Produkte Sie verfügbar machen wollen, müssen Sie sie zunächst im Market anlegen. Dazu gibt es eine Benutzeroberfläche in der Entwicklerkonsole. Hier beißt sich leider die Katze in den Schwanz: Solange Sie Ihre App nicht mit In-App-Payment versehen haben, können Sie in der Entwicklerkonsole keine Produkte hinzufügen. Um dieses Riff zu umschiffen, schlagen Sie folgenden Kurs ein:

- Fügen Sie dem Manifest Ihrer App die Permission `com.android.vending.BILLING` hinzu.
- Laden Sie die App als Update hoch (mit neuer Versionsnummer natürlich).

Sobald die Entwicklerkonsole begriffen hat, dass Ihre App In-App-Payment zu unterstützen wünscht, können Sie auf den Link In-App Produkte klicken. Fügen Sie dann Ihr Produkt hinzu (Abbildung 11.10).

Legen Sie zunächst eine Produkt-ID fest. Das ist eine eindeutige Bezeichnung, die Sie später nicht mehr ändern können, ähnlich einer Bestellnummer. Anhand dieser ID wird Ihre App nach dem Kauf wissen müssen, welche Inhalte sie für den Kunden freischalten muss.

Von entscheidender Bedeutung ist, dass Sie den »Kauftyp« richtig setzen, denn diese Einstellung trägt gewissermaßen den unsichtbaren Java-Modifier `final`: Einmal festgelegt, können Sie den Kauftyp nicht mehr ändern.

- Pro Nutzerkonto verwaltet: Kann nur einmal gekauft werden.
- Nicht verwaltet: Kann mehrfach gekauft werden.

Übrigens: Wenn Sie eine eigene Kaufbegrenzung benötigen, etwa »höchstens einmal im Monat«, müssen Sie NICHT VERWALTET wählen und die Begrenzung selbst implementieren, indem Sie sich jeden einzelnen Kauf irgendwo merken.

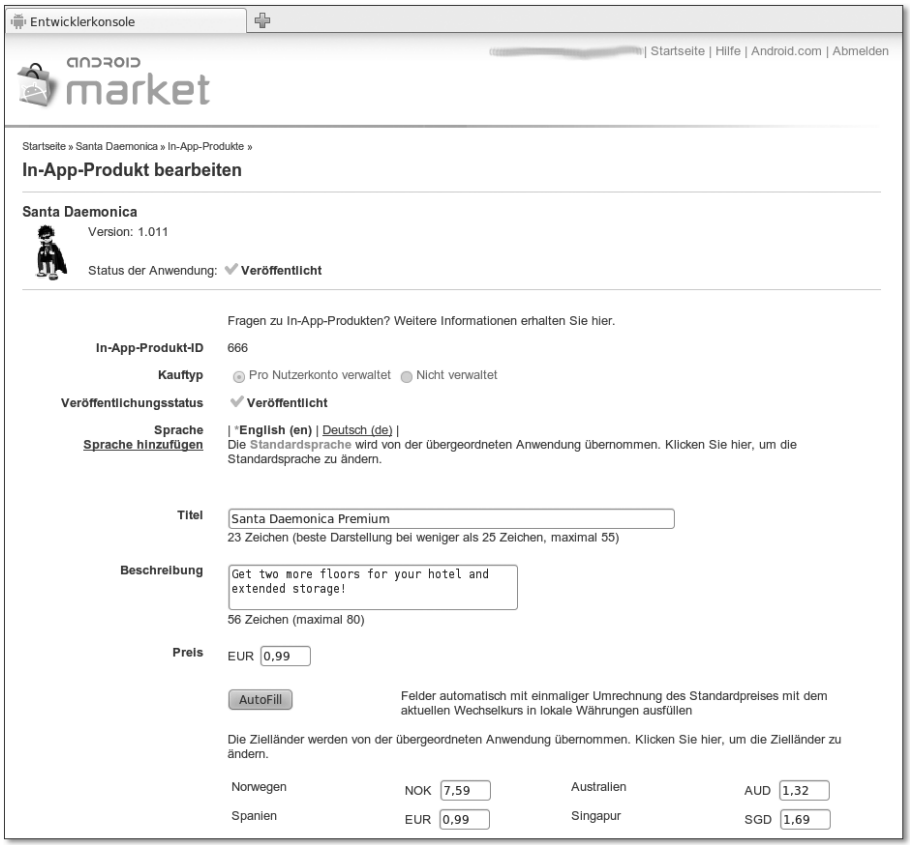



**Abbildung 11.10** Jedes In-App-Produkt hat ein eigenes Formular, in dem Sie alle Details erfassen können.

Wie schon bei den Apps selbst können Sie hier Titel und Beschreibung in verschiedenen Sprachen eingeben. Viel Platz haben Sie dafür nicht, aber Sie sind ohnehin gut beraten, Ihren potenziellen Käufern schon vor dem eigentlichen Aufruf des In-App-Payment-Dialogs genau zu erklären, wofür sie ihr Geld ausgeben.

Preise können Sie wie für Apps für jedes Land separat festlegen. Auch bei In-App-Produkten ist der niedrigste sinnvolle Preis 0,79 Euro.

Für pro Nutzerkonto verwaltete Einkäufe gilt übrigens, dass der Käufer das Produkt selbst nach einer Neuinstallation auf einem anderen Handy erneut erhalten

kann. Voraussetzung ist, dass derjenige denselben Google-Account verwendet. Wie das funktioniert, erkläre ich Ihnen im Zusammenhang mit dem BillingService in den folgenden Abschnitten.

**Offizielle Dokumentation**

Im Wesentlichen erkläre ich Ihnen das In-App-Payment hier so, wie Google es in seiner offiziellen Dokumentation tut, bloß etwas übersichtlicher. Wenn Sie möchten, können Sie sich die Original-Dokumentation im Internet anschauen:

*http://developer.android.com/guide/market/billing/index.html*

## Der BillingService-Apparat

Wie ich Ihnen bereits angedroht habe, ist es relativ kompliziert, In-App-Payment in eine App einzubauen. Ihre App muss dabei mit einem externen Service sprechen, der Teil der Android-Market-App ist. Diese wiederum unterhält sich mit dem Market-Server, um den Sie sich glücklicherweise nicht selbst kümmern müssen (Abbildung 11.11).


**Abbildung 11.11** Der Gesprächspartner Ihrer App ist der BillingService der Market-App, die wiederum mit dem Market-Server konferiert.

Bisher habe ich Ihnen noch nicht erklärt, wie eine App mit einem Service kommunizieren kann, der sich in einer anderen App befindet. Dazu benögigt Ihre App eine Definition des verwendeten Interfaces, das im Fall des Market Billings `IMarketBillingService` heißt. Die Interface-Definition befindet sich in einer Datei namens *IMarketBillingService.aidl*. Sie finden diese Datei in der Beispiel-App und können sie einfach übernehmen. Das Android-Eclipse-Plugin erzeugt automatisch die benötigte Java-Klasse `IMarketBillingService` und platziert sie im *gen*-Verzeichnis. Sie müssen sich nicht weiter darum kümmern.

Manche Entwickler stellen Interfaces ein `I` voran, um sie anhand des Namens von Klassen unterscheiden zu können – daher der sonderbare Name dieses Services.

Die Beispiel-App, die Google bereitstellt, verfügt über einen lokalen Service, der für die Kommunikation mit dem `MarketBillingService` zuständig ist. Ich habe

diesen BillingService bislang immer übernommen und empfehle Ihnen, dasselbe zu tun. Sie müssen lediglich die nötigen Dateien aus dem Beispielprojekt kopieren. Um den Service benutzen zu können, müssen Sie ihn in Ihrer App deklarieren und starten. Danach können Sie – beispielsweise nach einem Klick des Benutzers auf einen EINKAUFEN-Button – den Kauf eines Produkts anstoßen:

```
billingService.requestPurchase(productId, developerPayload);
```

Natürlich müssen Sie die gewünschte `productId` übergeben, außerdem können Sie im Parameter `developerPayload` einen String mitgeben, der Ihnen nach erfolgreichem Einkauf zurückgegeben wird. Denn der Einkauf läuft asynchron: Die `requestPurchase()`-Methode kehrt sofort zurück. Vom erfolgreichen Kauf erfahren Sie anderweitig. Wie? Darauf kommen wir erst im nächsten Abschnitt zu sprechen.

Eine Warnung zur `developerPayload`: Ich habe schon Fälle erlebt, in denen der Parameter *nicht* wie versprochen durchgereicht wurde. Verlassen Sie sich nicht auf ihn; merken Sie sich anderweitig alles, was Sie zum Kauf wissen müssen, beispielsweise in einer statischen Klasse.

Der Aufruf von `requestPurchase()` zaubert nach wenigen Sekunden den Einkaufs-Dialog auf den Bildschirm. Wenn der Benutzer den Einkauf beendet hat, teilt der externe `MarketBillingService` Ihnen das mit (Abbildung 11.12).

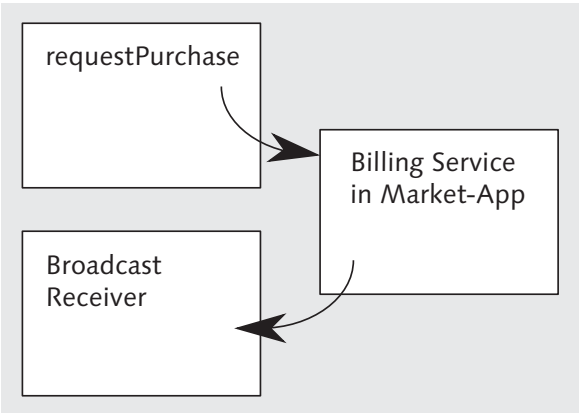



**Abbildung 11.12** Das Ergebnis des Kaufs erfahren Sie, wenn es so weit ist. Bis dahin läuft Ihre App normal weiter.

Genau genommen, besteht das Gespräch zwischen der Market-App und Ihrer App aus mehr als diesen zwei Schritten. Die Details können Sie jedoch guten Gewissens dem Code überlassen, den Sie in der Beispielanwendung finden.

Lesen Sie den Code ruhig, auch wenn er kompliziert ausschaut. Es stehen sogar Kommentare drin, die leidlich gut erklären, wie die Angelegenheit funktioniert.

Sie müssen jetzt nur noch dafür sorgen, dass der Käufer sein Produkt erhält. Dafür zuständig ist der `BillingReceiver`.

### BillingReceiver und BillingResponseHandler

Die Market-App muss Ihre App ansprechen können, um ihr den Verlauf des Verkaufs bekanntzugeben. Dazu benötigen Sie einen Receiver, den Sie im Manifest Ihrer App deklarieren müssen. Wenn Sie sich die Beispiel-App ansehen, können Sie sehen, wie das funktioniert (Abbildung 11.13).

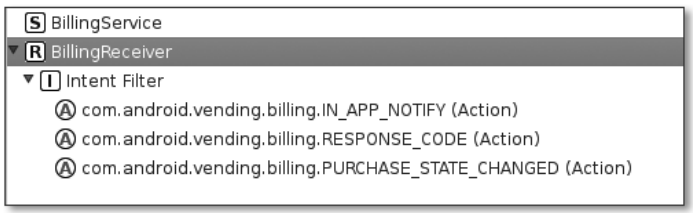



**Abbildung 11.13** Der BillingReceiver muss samt Intent Filter im Manifest deklariert werden.

Ähnlich wie eine Activity kann ein `Receiver` Intents verarbeiten, die an ihn gerichtet sind. Das funktioniert auch von außerhalb einer App. Auf diese Weise kann die Market-App Ihrer App Informationen über den Kauf schicken.

Die Beispielimplementierung des `BillingReceivers` können Sie ohne Änderung übernehmen. Sie reicht die Behandlung aller Nachrichten an den `BillingService` weiter. Auch hier müssen Sie sich nicht um die Details kümmern. Entscheidend ist: Sobald die Kasse geklingelt hat, wendet sich der Service an die Klasse `BillingResponseHandler`. Er ruft darin die Methode `purchaseResponse()` auf – und deren Aufgabe ist es nun, auf den Kauf zu reagieren. Sie können die Klasse aus der Beispielanwendung übernehmen, müssen aber die genannte Methode selbst mit Leben füllen. Das funktioniert so:

```
public static void purchaseResponse(Context context, PurchaseState
purchaseState, String productId, String orderId, long purchaseTime,
String developerPayload) {
   if(purchaseState.equals(PurchaseState.PURCHASED)) {
      // Erfolgreicher Kauf von productId
   }
}
```

Googles Beispielanwendung wird an dieser Stelle etwas kompliziert, indem sie einen Thread startet und einen `PurchaseObserver` verwendet, der hauptsächlich dazu da ist, die Benutzereberfläche zu aktualisieren. Notwendig ist diese Komplikation nur, wenn die Methode `purchaseResponse()` möglicherweise etwas länger braucht, bis sie fertig ist. Denn der Aufruf erfolgt synchron, und wir wollen die Market-App doch nicht warten lassen, oder? Sie wissen ja, dass nach drei Sekunden der böse »... antwortet-nicht«-Dialog erscheint. Was geschieht, wenn der Benutzer an dieser Stelle auf »gewaltsam beenden« drückt, möchten Sie gar nicht wissen. Erfreulich ist es sicher nicht.

Wie Sie auf den erfolgreichen Kauf reagieren, hängt vollkommen von der Natur Ihrer App ab. Möglicherweise speichern Sie einfach eine Information in den Shared Preferences. Das ist beispielsweise für Premium-Versionen oder Level-Packs hilfreich und könnte wie folgt aussehen:

```
private void aktiviereLevelPack(int productId) {
   SharedPreferences prefs = getPreferences(0);
   Editor editor = prefs.edit();"
   editor.putBoolean("LEVELPACK_
          + Integer.toString(productId), true);
   editor.commit();
}
...
private boolean istLevelPackInstalliert(int productId) {
   SharedPreferences prefs = getPreferences(0);
   return prefs.getBoolean("LEVELPACK_"
          + Integer.toString(productId), false);
}
```

Sobald Sie das In-App-Payment komplett implementiert haben, müssen Sie es testen. Leider müssen Sie dazu Ihre App veröffentlichen, denn die Bezahlung funktioniert nicht, wenn sich die aktuell publizierte Version von jener unterscheidet, die die Bezahlung initiieren möchte – und schon gar nicht mit einer Signatur vom Debug-Zertifikat. Sicherheitsmaßnahmen, die wir gut verstehen können, oder?

Versuchen Sie einen Trick: Machen Sie den EINKAUFEN-Button nur auf Ihrem Gerät sichtbar. Dazu können Sie dessen Android-ID mit Ihrer eigenen vergleichen:

```
String androidId = Secure.getString(getContentResolver(),
Secure.ANDROID_ID);
if("meineandroidid".equals(androidId)) {
   einkaufenButton.setVisibility(View.VISIBLE);
}
```

Die Android-ID ist theoretisch bei jedem Gerät unterschiedlich. Ihre können Sie beispielsweise mit einer App wie Android System Info auslesen oder mit der obigen Methode. Leider gibt es einige Hersteller, die darauf verzichten, ihre Geräte mit eindeutigen Android-IDs auszustatten. In dem Fall müssen Sie sich einen anderen Trick ausdenken.

Wie bei kostenpflichtigen Apps können Sie alle Einkäufe in der Entwicklerkonsole beobachten. Davon zeige ich Ihnen leider keinen Screenshot, weil ich ohnehin alle Einträge darin aus Datenschutzgründen unkenntlich machen müsste. Ich drücke Ihnen einfach die Daumen, dass Ihre App dank In-App-Payment erfolgreich wird – dann sehen Sie die Liste der Einkäufe selbst. Und ich versichere Ihnen: Sie werden die Liste dreizehnmal täglich aufrufen.

Der Vollständigkeit halber sei erwähnt, dass das In-App-Payment auch das Stornieren von Einkäufen vorsieht (`PurchaseState.REFUNDED`). Allerdings weist Google den Kunden vor dem Kauf darauf hin, dass eine Rückgabe überhaupt nicht möglich ist, und die meisten Apps ignorieren solche Versuche folglich. Stellen Sie sich vor, jemand kauft in einem Fantasy-Spiel eine Familienpackung Heiltränke, leert sie in einem Zug und möchte anschließend den Kauf stornieren ...

## 11.4 Alternative Markets

Alles hat seine Vor- und Nachteile, das gilt für Konjunkturprogramme genau wie für den Android Market. Natürlich ist er die größte Quelle an Apps und auf so gut wie jedem Android-Gerät vorinstalliert. Aus Entwicklersicht könnte man ihn jedoch genauso gut als Schrotthaufen astronomischer Größe bezeichnen, in dem Normalsterbliche nur mit übernatürlicher Begabung auf brauchbare Apps aufmerksam werden. Oder durch die Top-Listen, auf denen eine neue App naturgemäß nicht so bald landet.

Nicht zuletzt kostet es Sie ein paar Euro, Apps im Android Market veröffentlichen zu dürfen; und ich glaube, in Googles Geschäftsbedingungen irgendwas vom Verkauf meiner Seele gelesen zu haben, aber in dieser Hinsicht bin ich nicht ganz sicher ...

Werfen wir einen Blick auf die Alternativen zum Android Market. Vorweg: Sie haben alle einen gravierenden Nachteil – sie sind längst nicht auf jedem Gerät vorinstalliert.

**Nicht-Spezialisten**

Nicht vorstellen werde ich Shops, die nicht auf Android spezialisiert sind. Wenn Sie Interesse haben, Ihre App in möglichst vielen Stores zu verteilen, werfen Sie einen Blick auf folgende Liste:

- *http://www.handster.com/*
- *http://www.handango.com/*
- *http://www.mobihand.com/*
- *http://www.pocketgear.com/*

### Amazon AppStore

Amazon verkauft längst nicht nur Bücher, MP3-Musik, sondern auch Kühlschränke und – Android-Apps, wenn auch zum Zeitpunkt der Drucklegung dieses Buches nur in den USA.

Gerüchteküchen aller Welt glauben aber bereits zu wissen, dass Android mit einem eigenen Tablet auf den Markt drängen wird, auf dem natürlich der Amazon AppStore vorinstalliert sein wird. Im Moment hat Apple noch was dagegen, aber um die wenig zufällige Namensgleichheit sollen sich Juristen streiten. Es wäre jedenfalls kaum verwunderlich, wenn ein Anbieter mit einer Marktmacht wie Amazon mittelfristig eine Alternative zum Original wird.

Bis dahin müssen Sie sich auf andere Shops konzentrieren, aber behalten Sie Amazons Android-Aktivitäten im Auge.

### AppsLib

Versuchen Sie nicht, die AppsLib-App im Android Market zu finden oder von der Webseite zu installieren: Sie läuft nur auf bestimmten Tablets, hauptsächlich des französischen Herstellers **Archos**. Dafür ist AppsLib auf vielen Tablets vorinstalliert (Abbildung 11.14), der Android Market aber nicht. Es gibt eine Reihe von Geräten, die von Google nicht zertifiziert sind, sodass man den Original-Market nicht ohne Weiteres installieren kann. Benutzer solcher Geräte sind auf die vorinstallierte AppsLib angewiesen.

AppsLib ist also eine Anlaufstelle für (laut AppsLib) zwei Millionen Tablet-Besitzer. Das ist keine ganz kleine Zielgruppe. Dementsprechend sollten Sie durchaus in Erwägung ziehen, dort Apps zu veröffentlichen, die auf Tablets besonders gut zur Geltung kommen – und Sie können sicher sein, dass Ihre App nicht völlig untergeht, weil das Angebot viel überschauberer ist. Die Registrierung ist kostenlos, die zugehörige Entwicklerkonsole aber bei Weitem nicht so komfortabel wie jene des Originals (Abbildung 11.15).


**Abbildung 11.14** AppsLib ist unter anderem auf Archos-Tablets vorinstalliert.

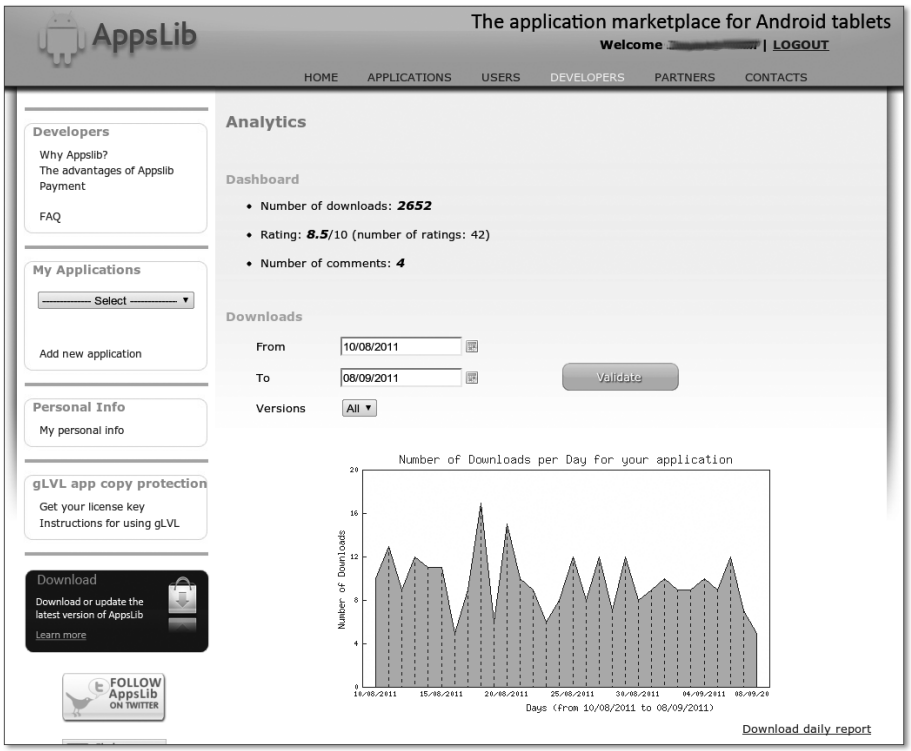



**Abbildung 11.15** AppsLib ist auf bestimmte Tablets spezialisiert. Die Download-Zahlen liegen typischerweise unter 5 % jener des Android Markets.

Auch AppsLib schüttet bei kostenpflichtigen Apps 70 % des Verkaufspreises an Sie aus. Voraussetzung für den Kauf ist ein PayPal-Konto – noch ein Vorteil gegenüber dem Android Market, wo meist nur eine Kreditkarte hilft.

Da Archos als Hersteller in Frankreich recht bekannt ist, ist es hilfreich, Ihre App auch auf Französisch anzubieten, wenn Sie größere Download-Zahlen erreichen wollen.

### AndroidPIT App Center

Das App Center der deutschen Website *Androidpit.de* ist inzwischen ebenfalls auf einer ganzen Reihe von Geräten vorinstalliert. AndroidPIT zitiert Statistiken, lautet derer App Center der fünftgrößte Android-App-Store der Welt sei. Im Gegensatz zu AppsLib läuft App Center auf jedem Android-Gerät, nachdem man die zugehörige App von *http://androidpit.de* installiert hat. App Center bringt auch gleich die Inhalte der Webseite bequem aufs Handy: Tests, Blog und Forum sind direkt eingebunden (Abbildung 11.16).


**Abbildung 11.16** Das AndroidPIT App Center verfügt über annähernd dieselben Kategorien wie das Original.

Sie brauchen lediglich einen kostenlosen Entwicklerzugang bei *Androidpit.de*, um Ihre App einzustellen. AndroidPIT übernimmt sogar die Daten aus dem An-

droid Market, falls Ihre App dort ebenfalls verfügbar ist. Dann müssen Sie nur noch das signierte APK hochladen.

Auch kostenpflichtige Apps unterstützt App Center, dazu existiert ein eigener Lizensierungsmechanismus, der Raubkopien verhindern soll. Ihre Einnahmen schiebt Ihnen AndroidPIT auf ein deutsches Girokonto oder einen PayPal-Account.

Ein großer Vorteil von AndroidPIT ist, dass die ganze Webseite deutsch spricht. Außerdem garantiert die relativ hohe Verbreitung bei einer gleichzeitig überschaubaren Anzahl verschiedener Apps (schätzungsweise 5.000) eine ganze Menge Downloads.

Da es kaum Aufwand ist, eine App hier zu veröffentlichen, sollten Sie nicht lange zögern und es einfach probieren.

### SlideME.org

Auch SlideME ist ein großer, webbasierter App-Katalog, der eine eigene App mitbringt. Die hört auf den Namen SAM und wirkt im Vergleich zum Original-Market oder auch dem App Center etwas lieblos (Abbildung 11.17).


**Abbildung 11.17** Der SlideME-Anwendungsmanager spricht zwar deutsch, lässt aber einige Wünsche offen.

SlideME hat eigene Kategorien, aber keine Unterkategorien. Ich wünsche Ihnen viel Spaß dabei, die 3.000 Apps in der Kategorie »Spaß & Spiel« nach etwas von Interesse zu durchsuchen. Trotzdem kann ich Ihnen versprechen, dass Sie über SlideME einige Downloads generieren werden. Ähnlich wie beim App Center ist die Anzahl der Apps überschaubar und die Zahl der User relativ hoch. Da es wie im Original-Market eine Liste von Neuzugängen gibt, bleibt Ihre App nicht unbemerkt.

Auch Bezahl-Apps können Sie bei SlideME einstellen, die Bedingungen entsprechen jenen der anderen Stores. Benutzer können die SlideME-Geldbörse (**Wallet**) via PayPal oder Kreditkarte aufladen und das Geld dann in Apps investieren. Allerdings erscheinen mir die Auszahlungsbedingungen für Entwickler etwas ungünstig – wenn ich die unübersichtlichen Hinweise richtig verstehe, müssen Sie mindestens 100 US-Dollar einnehmen, bevor man sich die Mühe macht, Ihnen etwas zu überweisen.

Für kostenpflichtige Apps steht ein Kopierschutzmechanismus zur Verfügung.

Sie sehen: Es gibt einige Alternativen zum Android Market, die die nähere Betrachtung lohnen. Natürlich bedeutet es zusätzlichen Aufwand, Ihre App an mehreren Stellen hochzuladen, wenn Sie ein Update fertiggestellt haben. Beobachten Sie die Download-Zahlen, um zu erkennen, ob sich die Arbeit lohnt.

Da Sie jetzt ein fähiger Android-Entwickler sind, bin ich jedenfalls sicher, dass Ihre Apps hinreichend gewürdigt werden – egal, wo Sie sie veröffentlichen.

# Die Buch-DVD

Die Software auf der Buch-DVD wird ohne irgendwelche Gewährleistung zur Verfügung gestellt. Den Sourcecode dürfen Sie beliebig verwenden und verändern, bloß versuchen Sie bitte nicht, ihn zu verkaufen. Programme, die wir Ihnen zur Verfügung stellen, um Ihnen den Download zu ersparen, unterliegen ihren jeweiligen Lizenzen, die jeweils beiliegen.

Die Eclipse-Projekte können Sie direkt von Eclipse aus in Ihren Workspace übernehmen, wählen Sie dazu im Menü File • Import... • Import Existing Projects Into Workspace. Setzen Sie einen Haken bei Copy projects into workspace.

Hier die Ordnerstruktur:

- *eclipse-projekte*
  - *SagHallo*
  - *Mueckenfang*
  - *Mueckenfang2*
  - *Mueckenfang3*
  - *Mueckenfang4*
  - *Kompass*
  - *Schrittzaehler*
  - *Schrittzaehler2*
  - *WegInsBuero*
  - *DialogDemos*
  - *LayoutDemos*
- *software*
  - *windows* (Eclipse, JDK, Android SDK, Audacity, Inkscape, GIMP)
  - *linux* (Eclipse, JDK, Android SDK, Inkscape)
  - *mac os* (Eclipse, Android SDK)

# Index


@Override 88
9-Patches 330

## A

AAC+ 176
above 342
abstract 60, 301
Accelerometer 292
Activity 86
ActivityNotFoundException 319
Adapter 240
ADB 78
adb 84
add() 311
addView() 156
ADT 70, 74, 85
AlertDialog 333
AlertDialog.Builder 333
Alpha 182
Alpha-Transparenz 142
Amazon 368
andengine 34
Android Debug Bridge 78, 122
Android Debug-Bridge 83
Android Development Tool 70, 101
Android Market 23
Android Resource Manager 84, 102
Android SDK 76
Android Virtual Device 77
AndroidHttpClient 223
Android-ID 366, 367
Android-Manifest 94, 99, 130, 133, 302
AndroidPI 370
AndSMB 123
Animation 181
AnimationListener 188
Annotation 62
anonyme innere Klasse 190
Apache 124
API-Key 311
APK 102, 345, 347
apkbuilder 76
Apotheke 31
App Center 370
Application Nodes 96
AppsLib 368
AppStore 23
ArrayAdapter 244
ArrayList 56
ArrayWayOverlay 314
Atomreaktoren 39
Attribut 45
AttributeSet 287
Audacity 174
Audio-Formate 176
Augmented Reality 32, 253, 279
Auswahlliste 244
AVD 77–79
Azimuthwinkel 280

## B

Background 135, 326
Background-Thread 229
barcoo 29
Basisklasse 60
Baumstruktur 72
Beans 37
Bedingung 50
Beschleunigungssensor 20, 26
Bildschirmausrichtung 97
BillingReceiver 365
BillingResponseHandler 365
BillingService 363–364
bin 121
Bit 45
boolean 45, 47, 151
Boolsche Operatoren 150
Breakpunkt 323
Browserspiele 39
Build Target 85
Bump 26
Button 107
byte 47
Bytecode 38

## C

C 37
C++ 37
c:geo 27
cacheColorHint 242
Calendar 336
Camera 254
CameraView 254
Canvas 275
Cast 119
Casting 153
Caused by 320
char 47
CheckBox 107
checked 330
Checked Exceptions 228
Checkout 349
Children 157
Chrominanz 264
class 40
clear() 311
colors.xml 338
Compiler 37
Console View 121
Constructor 42
Content Assist 131
Context 154
Countdown 146
CPU 37
Culicidae 127
Custom View 275, 291
Cut the Rope 34

## D

Dalvik VM 38
Date 156
DatePicker 117
DatePickerDialog 333, 336–337
DDMS 74
Debug-Perspektive 323
Debug-View 324
Debug-Zertifikat 349
Denglisch 41
device independant pixels 136
Dialog 161, 332
DialogInterface 333
Digicam 29, 253
Digitale Signatur 347
dismiss() 334
DisplayMetrics 145
doGet() 217
double 47
draw9patch 331
drawable 100, 103, 105
drawArc() 289
Dropbox 124

## E

Eclipse 39, 45, 69, 81
Editable 120
EditText 107
einblenden 182
else 197
Emulator 66, 70, 76, 78
enabled 330
Enterprise 32
Entwicklerkonsole 350, 358
Entwickler-Konto 349
Entwicklungsumgebung 69
Erdanziehungskraft 292
Erdbeben 20
Erdbeschleunigung 20
Ereignis-Warteschlange 165
Event 164
Eventqueue 164
Exceptions 226
extends 60

## F

Farb-Resourcen 141
Fehlerberichte 321
FILL 276
FILL_AND_STROKE 276
final 103, 339
findViewById() 119, 153, 249, 339
finish() 132
flickr 31
float 47, 284
focused 330
Form Widgets 107
Fortran 37

Fragezeichen 83
FrameLayout 141
fromHtml() 237

## G

Galaxy of Fire 2 173
Game Engine 137
Garbage Collection 84
Garbage Collector 42, 179
Geocaching 21, 27
Geokoordinaten 21
getAnimation() 190
getChildAt() 157
getChildCount() 157
getDefaultSensor() 272
getDrawable() 238
getIdentifier() 170, 200
getRessources() 213
getSharedPreferences() 208
getSystemService() 272, 310
getter 62
getText() 212
getWriter() 221
GIMP 195
GONE 211
Google App Engine 214
Google Maps 21, 23
Google Sky Map 25
GPS 21, 307
Grafiken 100
Gravity 114, 136, 156

## H

handleMessage() 296
Handler 163, 165, 295–296, 305
Hänger 84
Hexadezimale Farbwerte 141
Hexadezimalzahl 104, 106
Hierarchy Viewer 75
High Replication Datastore 218
Hintergrundfarbe 242
HorizontalScrollView 116
HTML 234
HTML-Farbcodes 142
HTTP-Client 223
HttpEntity 224
HttpGet 223
HttpServletResponse 218
Hubble-Teleskop 25
HVGA 79
Hypertext Transfer Protocol 216

## I

Icon 97, 100, 351
icon.png 100
ids.xml 156
if 50
IllegalArgumentException 228
ImageButton 116
ImageGetter 238
ImageView 116
IMarketBillingService 363
implements 91, 118
importieren 44, 89
In-App-Payment 359
initialisieren 46
Inkscape 134, 325, 351
innere Klasse 184
InputStreamReader 224
Installation 355
Instanz 40
instanziieren 41
int 47
Integer 46
Intent 132
Intent Filter 96–97
Interface 91, 118
interface 91
Interpolation 186
Interpolator 187
invalidate() 275
INVISIBLE 211
IOException 226, 256, 345

## J

jar 312
Java 37
Java Development Kit 67
Java Runtime Environment 38
Java-Kompatibilität 83

Java-Perspective 71
Java-Runtime 40–41
JDK 67
JRE 38

## K

Kaffeemaschine 67
kartesisches Koordinatensystem 279
Keystore 348
Klasse 40
Kommentar 88
Konstante 156
Konstruktor 42
Kopierschutzmechanismus 353
Kriegshammer 14
Kugelkoordinatensystem 279

## L

Labyrinth 20
Lagesensoren 35
landscape 97
Laufvariable 157
Launch Options 80
Launcher 97
Layout 106, 129, 131, 342
Layout gravity 141
Layout Weight 143
layout_weight 343
Layout-Datei 161
Layout-Editor 328
Layouteditor 105
LayoutInflater 248, 251, 341
LayoutParams 145, 155, 194
LinearLayout 113
lineTo() 277
ListView 116, 240
Lizensierungsservice 353
loadAnimation() 183
Location 310
LocationManager 310
Log Level 318
Log.e 321
LogCat 318
Logcat 257
Log-Filter 322
Logging 257
LOGTAG 322
lokale Klasse 296
long 47
Loop 157
Luminanz 264

## M

Magnetfeldsensor 21, 271
main.xml 106
MapActivity 312
MapController 315
MapDroyd 23
mapsforge 311
MapView 313
MapViewMode 313
Maßstab 145
match_parent 114
Math 145, 153
Math.min() 145
MediaPlayer 178
Methode 49
Mikrofon 21, 33, 174
Mindestpreise 353
modifier 44
Modifizierer 44
moveTo() 277
mp3 176
Mücke 18, 128
Multitasking 163

## N

Namespace-Attribut 183
network based location 307
netzwerkbasierte Ortsbestimmung 307
notifyDataSetChanged() 250
NullPointerException 228, 306, 320
NV21 263
NV21Image.java 269

## O

Objekte 40
objektorientiert 40

Öffi 30
Ogg Vorbis 176
onActivityResult() 207
onClick() 333
onClickListener 118, 155, 294, 303, 308, 339
onClickListener() 132
onCreate() 88
onDateSet() 336
onDateSetListener 336
onDestroy() 179
onDraw() 275
onInit() 90
onKeyDown() 251
onLocationChanged() 310
onPreviewFrame() 262
onResume() 206
onSensorChanged() 274, 283, 295
Openstreetmaps.org 311
Operator 48
Organize Imports 90, 94
OSM 24
Outline 72, 113, 171
OutOfMemoryError 359
Overlays 314
OverlayWay 314
Override 62

## P

Package 43
Package Builder 102
Package Explorer 72, 87, 94, 99, 105
Padding 136
Paint 276
Panoramico 31
Parameter 42, 50
parseInt() 218
Pascal 37
Path 276
PayPal 370–372
Permission 97, 222
Perspective 71
Pfad 276
Physik-Engine 34
Pivotpunkt 185
Pizzeria 31
Plugins 70
PNG 100, 326
Polarwinkel 280
Polymorphie 63
portrait 97
postDelayed() 166, 335
PreviewCallback 262
primitive Datentypen 46–47
private 51, 61, 89
Privater Schlüssel 348
Problems 72
Programmbibliothek 312
ProgressBar 107
ProgressDialog 333–334
protected 61
public 44
purchaseResponse() 365
Pythagoras 201

## Q

qualified name 44
qualifizierter Name 44
Query 220

## R

R.java 102, 119
Radar 286
Random 148
raw 176
Receiver 365
Refactor 151, 169, 246
Refactoring 152
Reference Chooser 135
RegionalExpress 42
registerListener() 273
RelativeLayout 341
removeCallbacks() 186
removeUpdates() 311
removeView() 160
replace() 226
Request 216
requestLocationUpdates() 310
requestPurchase() 364
res 99, 103
Resource Chooser 108
Resource Manager 102, 106

Response 216
Ressource Chooser 136
Ressourcen 99
Restore 72
rotate() 277
round() 145, 196
Router 21
run() 165, 229
Runescape 39
Runnable 165, 229
runOnUiThread() 231
RuntimeException 228

## S

Sampling 176
scale-independant pixels 136
Schatztruhen 16
Schleife 58, 157
Screen Orientation 294
Screenshots 350
ScrollView 116
SDK Platform Tools 76
SD-Karte 79
selected 330
selector 329
sendEmptyMessage() 306
SensorEvent 274
SensorEventListener 273, 295
SensorManager 272, 294
Serveranwendung 27, 30, 39
Service 300, 307
Servlet 216
setAnimationListener() 190
setAntiAlias() 276
setBackgroundResource() 170
setColor() 276
setContentView 106
setContentView() 131, 243, 278, 313, 339
setDisplayOrientation() 256
setImageResource() 170, 198
setLayoutParams() 194
setOneShotPreviewCallback() 262
setPreviewDisplay() 255
setProgress() 336
setResult() 207
setStrokeWidth() 287
setStyle() 276
setTag() 280
setter 62
setVisibility() 211
setWayData() 315
Shaky Tower 35
Shared Preferences 207
SharedPreferences 366
SharesFinder 123
short 47
SimpleStringSplitter 235, 249
Single Processing 163
SlideME 371
SMB 123
SO-8859-1 225
Software Development Kit. Dahinter 76
Software Sites 74
Sound 173
Sound-Formate 176
Speicher 41
sphärische Koordinaten 279
Spiel 34
Spielregeln 137
Spinner 245
Sprachausgabe 85, 90
Sprachsuche 32
src 87
Stacktrace 319, 358
Standard-Dialoge 333
Standard-Konstruktor 54
startActivity 87, 106
startActivity() 132
startActivityForResult() 207, 246
startAnimation() 184
startPreview() 258
startService() 304
state_pressed 330
static 304, 306
statische Attribute 304
Statische Methoden 145
Statistik 355
Steuerrad 20
stopService() 304
Stream 224
String 48
String-Konstante 111
String-Ressource 109
strings.xml 111, 213
String-Verweise 111
STROKE 276

Stromverschwendung 50
Styles 326
super 89
surfaceChanged() 256
surfaceCreated() 255
SurfaceHolder 254
SurfaceView 117, 254
svg 134
Synonymwörterbuch 18

## T

Tags 156
Task List 72
Telepathie 106
Text Style 141
Text to speech 85
TextColor 141, 327
TextToSpeech 90
TextView 107
Theme 327
Thesaurus 18
this 90
Threads 229
TimePicker 117
TimePickerDialog 333, 336
Toast 340
Traceview 75
translate() 277
Transparenz 100
Tricorder 19, 297
trim() 212
try-catch 227
Tutorials 73
Type-Casting 153

## U

Überschreiben 62
Ubuntu 71
UI-Thread 229, 231
Unchecked Exceptions 228
Unicode 225
Update 355
URLEncoder 233
URL-Parameter 217
USB-Debugging 81
USB-Kabel 65, 81, 85, 92
User Agent 223
Uses Permission 98
UTF-8 225

## V

Vampir 127
Variable 54
Variablennamen 41–42, 46
Vektor 191
Venus 25
Verantwortung 51
Vererbung 59–60
Vergleichsoperator 51
Versionscode 95
Versionsname 95
Versionsnummer 354
VideoView 117
View 119, 153
ViewGroup 157
Views bewegen 191
virtuelle Kamera 281
virtueller Raum 279
void 49
Vollbild-Modus 170

## W

Wahrheitswert 50
Wahrscheinlichkeit 147
Wasserkocher 67
WAV 176
Webcams 31
WebView 116
while() 267
while(bedingung) 157
Wikipedia 32
Wikitude 31
Wizard 128
workspace 70
Wrap in Container 258
wrap_content 114, 136

## X

XE 124, 188
XML 258, 329
xmlns 183

## Y

YCbCr_420_SP 263
Youtube 351

## Z

Zeitmaschine 16
Zentrifugalkraft 292
Zertifikat 348
Zufallsgenerator 147–148, 192
Zugspitzbahn 17
Zuweisungsoperator 46, 51
zweidimensionales Array 198