Alles über Java und Android

1

In diesem Kapitel

- Wie Benutzer das Android-Ökosystem sehen
- Die Zehn-Cent-Tour durch die Java- und die Android-Technologien

Bis Mitte 2000 bezeichnete das Wort *Android* eine mechanische, menschenähnliche Kreatur – zum Beispiel einen überkorrekten, hundertprozentigen Gesetzesvertreter mit eingebautem Maschinengewehr oder einen hyperlogischen Weltraumreisenden, der alles außer der gesprochenen Kommunikation beherrschte. Und dann kaufte Google 2005 Android Inc. – eine 22 Monate alte Firma, die Software für Handys herstellte. Damit änderte sich alles.

2007 gründeten 34 Unternehmen die *Open Handset Alliance*. Deren Aufgabe ist es, Innovationen für mobile Geräte zu fördern und Verbrauchern eine reichhaltigere, preiswertere und bessere mobile Umgebung zu bieten. Ihr zentrales Projekt heißt *Android*, bei dem es sich um ein offenes, kostenlos zur Verfügung stehendes System handelt, das kerneltechnisch auf dem Betriebssystem Linux basiert.

Auch wenn HTC das erste kommerziell nutzbare Android-Handy bereits gegen Ende 2008 auf den Markt brachte, erregte Android erst Anfang 2010 die Aufmerksamkeit eines breiteren Publikums.

Während ich Mitte 2013 dieses Buch schreibe, berichtet Mobile Marketing Watch (`www.mobilemarketingwatch.com/google-play-tops-50-billion-app-downloads-34516`) von mehr als 50 Milliarden Downloads im Google Play Store für Apps. Im ersten Halbjahr 2013 haben Android-Entwickler mehr verdient als im gesamten Jahr 2012. Und Forbes (`www.forbes.com/sites/tristanlouis/2013/08/10/how-much-do-average-apps-make`) meldet, dass Google in den 12 Monaten seit Mitte 2012 ungefähr 900 Millionen Dollar an Android-Entwickler ausgezahlt hat, und alles nimmt immer mehr Fahrt auf.

Die Sicht der Benutzer

Ein Endbenutzer denkt über Alternativen nach:

- ✔ **Möglichkeit Nr. 1: Kein Handy**

  Vorteile: Preiswert; keine Störungen aufgrund von Anrufen.

  Nachteile: Kein unmittelbarer Kontakt mit Freunden und der Familie; keine Möglichkeit, in Notfällen Hilfe zu holen.

- ✔ **Möglichkeit Nr. 2: Ein mit zusätzlichen Funktionen ausgestattetes Telefon**

  Bei dieser Art von mobilen Telefonen handelt es sich nicht um Smartphones. Auch wenn es keine offizielle Regel gibt, die die Grenze zwischen einem mit zusätzlichen Funktionen

ausgestatteten Telefon und einem Smartphone beschreibt, so besitzt ein mit zusätzlichen Funktionen ausgestattetes Telefon ein starres Menü mit vordefinierten Wahlmöglichkeiten, während ein Smartphone im Vergleich dazu über einen »Desktop« verfügt, der aus heruntergeladenen Apps besteht.

Vorteil: Preiswerter als ein Smartphone.

Nachteile: Nicht so vielseitig einsetzbar wie ein Smartphone; auch nicht ansatzweise so cool wie ein Smartphone; macht auf keinen Fall so viel Spaß wie ein Smartphone.

- ✔ **Möglichkeit Nr. 3: Ein iPhone**

  Vorteil: Toll aussehende Grafik

  Nachteile: Weniger wandlungsfähig, da abhängig vom Betriebssystem iOS eines einzigen Herstellers; die Auswahl ist auf eine Handvoll Geräte beschränkt

- ✔ **Möglichkeit Nr. 4: Ein Windows Phone, ein BlackBerry oder ein anderes Gerät, das nicht mit Android läuft oder das von Apple ist**

  Vorteil: Ein Smartphone besitzen, ohne zur Menge zu gehören

  Nachteil: Die Möglichkeit, plötzlich mit einem verwaisten Produkt dazustehen, wenn sich der Krieg auf dem Markt der Smartphones ausweitet

- ✔ **Möglichkeit Nr. 5: Ein Android-Smartphone**

  Vorteile: Einsatz einer beliebten, offenen Plattform mit vielen Unterstützungsmöglichkeiten durch die Industrie und viel Bewegung in einem leistungsstarken Markt; Schreiben der eigenen Software, die auf dem eigenen Telefon installiert werden kann (ohne dass diese Software an die Website eines Unternehmens geschickt werden muss); Veröffentlichung der Software, ohne dass ein langwieriges Prüfverfahren notwendig ist.

  Nachteile: Sicherheitsfragen bei der Verwendung einer offenen Plattform; Frust, wenn sich iPhone-Besitzer über Ihr Telefon lustig machen

Ich bin der Meinung, dass die Vorteile, die Android bietet, die möglichen Nachteile bei Weitem überwiegen. Und da Sie ein Kapitel von *Java für die Android-Entwicklung für Dummies* lesen, ist die Wahrscheinlichkeit groß, dass Sie mir zustimmen.

Die vielen Gesichter von Android

Eine Versionsnummerierung kann ganz schön trickreich sein. Die Modellnummer meines PCs lautet T420s. Als ich das Handbuch für Benutzer heruntergeladen habe, entschied ich mich für einen Leitfaden für Laptops der T400-Serie. (Es gab kein Handbuch für den T420, geschweige denn eines für den T420s.) Wenn es nun aber wirklich zu Problemen kommt, reicht es nicht aus, nur zu wissen, dass man einen T420s besitzt. Ich benötige Treiber, die genau zur siebenstelligen Modellnummer meines Laptops passen. Die Moral von dieser Geschichte ist: Was eine »Modellnummer« ist, hängt von demjenigen ab, der sie wissen will.

Mit diesem Wissen im Hinterkopf sollten Sie sich einmal die Android-Versionen in Abbildung 1.1 anschauen.

	Plattform	API-Level	Codename	Funktionen
2008	1.0	1		
	1.1	2		Ausgereifte Schnittstelle zum Marktplatz für Apps; bessere Werkzeuge für die Spracheingabe, 800 x 480
	1.5	3	Cupcake	
2009	1.6	4	Donut	
	2.0	5		Bessere Oberfläche für Benutzer, mehrere Bildschirmgrößen, zusätzliche Kamerafunktionen, Unterstützung von Bluetooth 2.1, Multitouch-Unterstützung
	2.0.1	6	Eclair	
	2.1	7		
2010	2.2	8	Froyo	Ein besseres Leistungsverhalten durch Just-in-time-Kompilierung, USB-Anbindung, 720p-Bildschirm, die Möglichkeit, Apps auf einer SD-Karte zu installieren
	2.3	9		Systemweites Kopieren, Multitouch-Bildschirmtastatur, bessere systeminterne Codeentwicklung, gleichzeitig ablaufende Speicherbereinigung
	2.3.3	10	Gingerbread	
	3.0	11		Entworfen für Tablets, neue Bildschirmtastatur, Browser mit Registerkarten, neu entworfene Widgets, »dreidimensionale« Benutzeroberfläche, Fragments für Oberflächen
2011	3.1	12	Honeycomb	
	3.2	13		
	4.0	14	Ice Cream	Benutzerdefinierter Launcher für das Starten von Programmen, Erstellen von Screenshots, Entsperren der Oberfläche, Chrome als Browser, Near-Field-Kommunikation, Robot als Schriftart
	4.0.3	15	Sandwich	
	4.1.2	16		Erweiterbare Benachrichtigungen, Google Now, glatteres Zeichnen, verbessertes Suchen über Spracheingabe
2012	4.2.2	17	Jelly Bean	
2013	4.3	18		
	4.4	19	KitKat	Lauffähig auf Geräte mit nur 512 MB RAM, neues Design, bessere Unterstützung von Goggle Now und Google +

Abbildung 1.1: Android-Versionen

Einige Anmerkungen sollen helfen, Abbildung 1.1 besser zu verstehen:

✔ **Die Plattformnummerierung ist für den Verbraucher und die Unternehmen von Bedeutung, die die Hardware vertreiben.**

Wenn Sie beispielsweise ein Telefon mit Android 4.2.2 erwerben, wollen Sie vielleicht doch wissen, ob der Hersteller das Gerät auch auf Android 4.3 oder 4.4 aktualisiert.

✔ **Der API-Level (der auch SDK-Version genannt wird) ist für den Entwickler von Android-Apps von Interesse.**

So hat zum Beispiel das Wort `MATCH_PARENT` ab Android-API-Level 8 eine besondere Bedeutung. Sie könnten `MATCH_PARENT` natürlich auch in Code eingeben, der API-Level 7 verwendet. Wenn Sie so etwas machen und erwarten, dass `MATCH_PARENT` eben mit dieser besonderen Bedeutung funktioniert, werden Sie Ihr blaues Wunder erleben.

Sie können in Kapitel 2 mehr über das *Application Programming Interface* (API; deutsch *Schnittstelle für die Anwendungsprogrammierung*) nachlesen. In Kapitel 4 finden Sie mehr darüber, wie Sie Android-API-Level (SDK-Versionen) in Ihrem Code verwenden.

✔ **Der Codename ist für den Hersteller von Android von Bedeutung.**

Ein *Codename* hat mit der Arbeit der Entwickler zu tun, die Android auf die nächste Versionsstufe heben. Stellen Sie sich die Mannschaft von Google vor, die monatelang hinter verschlossenen Türen am Projekt Cupcake arbeitet, und Sie sind auf der richtigen Spur *(Cupcake* bedeutet auf Deutsch *Törtchen)*.

Android-Versionen sind nicht starr. So verfügt zum Beispiel das gute alte Android 2.2 über einen Satz bewährter Funktionen. Sie können Android 2.2 Google-APIs (wie die Funktionalität von Google Maps) hinzufügen und verwenden immer noch 2.2 als Plattform. Sie könnten aber auch Funktionen hinzufügen, die zum Beispiel speziell für das Tablet Samsung Galaxy gemacht worden sind.

Ihre Aufgabe als Entwickler ist es, einen Ausgleich zwischen Portierbarkeit und reichhaltigen Funktionen zu finden. Wenn Sie eine App erstellen, geben Sie eine Android-Zielversion (die in der Entwicklungsumgebung *Target* genannt wird) und eine Minimalversion an. (Sie finden in Kapitel 4 mehr zu diesem Thema.) Je größer die Versionsnummer ist, desto mehr Funktionen kann Ihre App enthalten. Andererseits gilt aber auch, dass je größer die Versionsnummer ist, desto weniger Geräte existieren, auf denen Ihre App laufen kann.

Die Sichtweise des Entwicklers

Android ist ein Wesen mit vielen Gesichtern. Wenn Sie für die Plattform Android entwickeln, greifen Sie auf viele Werkzeuge zu. Dieser Abschnitt gibt Ihnen darüber einen kurzen Überblick.

Java

James Gosling von Sun Microsystems entwickelte Mitte der 1990er die Programmiersprache Java. (Inzwischen ist Sun Microsystems von Oracle aufgekauft worden.) Javas kometengleicher Aufstieg hat etwas mit der Eleganz der Sprache und ihrer gut durchdachten Architektur zu tun. Nach einer kurzen Zeit voller Glanz und Gloria bei der Entwicklung von Applets und im Web etablierte sich Java als eine solide, allgemein nutzbare Sprache, die sich insbesondere für Server und Middleware eignet.

In der Zwischenzeit hat sich Java still und leise in eingebetteten Prozessoren breitgemacht. Sun Microsystems hat Java Mobile Edition (Java ME) entwickelt, mit dem sich kleine Apps erstellen lassen, die auf mobilen Geräten laufen. Java wurde zur zentralen Technologie bei Bluray-Playern. So überrascht die Entscheidung, Java zur primären Entwicklungssprache für Android zu machen, eigentlich nicht richtig.

Bei einem *eingebetteten Prozessor* handelt es sich um einen Computerchip, der sich als Teil eines Gerätes für besondere Zwecke vor den Benutzern verbirgt. So sind zum Beispiel die Chips in Autos eingebettete Prozessoren und auch das Silizium, das den Fotokopierer an Ihrer Arbeitsstelle mit Leistung versorgt, ist ein eingebetteter Prozessor. Und sicherlich werden auch die Blumentöpfe auf Ihrer Fensterbank bald mit eingebetteten Prozessoren ausgerüstet sein.

Abbildung 1.2 beschreibt die Entwicklung neuer Java-Versionen im Laufe der Zeit. Jede Java-Version hat, wie Android, mehrere Namen. Bei der *Produktversion* handelt es sich um eine offizielle Bezeichnung, die weltweit genutzt wird, während die *Entwicklerversion* eine Zahl ist, die Versionen identifiziert, damit Programmierer den Überblick behalten. (Im Alltag verwenden Entwickler für die verschiedenen Java-Versionen alle möglichen Bezeichnungen.) Beim *Codenamen* handelt es sich um eine eher spielerisch eingesetzte Bezeichnung, die die Version identifiziert, während sie entwickelt wird.

Jahr	Produktversion	Entwicklerversion	Codename	Funktionen
1995	(Beta)			
1996	**JDK* 1.0**	1.0		
1997	**JDK 1.1**	1.1		Innere Klassen, Java Beans, Reflexion
1998	**J2SE* 1.2**	1.2	Playground	Sammlungen, Swing-Klassen für das Erstellen von GUI-Interface
1999				
2000	**J2SE 1.3**	1.3	Kestrel	Java Naming and Directory Interface (JNDI)
2001				
2002	**J2SE 1.4**	1.4	Merlin	Neue Eingabe/Ausgabe, reguläre Ausdrücke, Analyse der XML-Syntax
2003				
2004	**J2SE* 5.0**	1.5	Tiger	Generische Typen, Annotations, Enum-Typen, varargs, Befehlserweiterungen, statische Importe, neue Concurrency-Klassen
2005				
2006	**Java SE* 6**	1.6	Mustang	Unterstützung der Skriptingsprache, verbessertes Leistungsverhalten
2007				
2008				
2009				
2010				
2011	**Java SE 7**	1.7	Dolphin	Strings in Switch-Anweisungen, Umgang mit mehreren Ausnahmen, Testanweisungen bei Ressourcen, Integration in JavaFX
2012				
2014	**Java SE 8**	1.8		Lambda-Ausdrücke

Abbildung 1.2: Java-Versionen

Die Sternchen markieren in Abbildung 1.2 Änderungen an der Formulierung des Namens der Java-Produktversion. 1996 hießen die Produktversionen *Java Development Kit 1.0* und *Java Development Kit 1.1*. 1998 entschied dann irgendjemand, das Produkt in *Java 2 Development Kit 1.2* umzutaufen, was natürlich zu einer heillosen Verwirrung führte. Damals wurde jeder, der den Ausdruck *Java Development Kit* verwendete, gebeten, nur noch von *Software Development Kit* (SDK) zu sprechen. 2004 wurde *1.* aus dem Plattformnamen entfernt und 2006 verlor der Java-Plattformname auch die *2* und das *.0*.

Zu den für Java-Entwickler bedeutendsten Änderungen kam es 2004. Die Java-Aufseher nahmen bei J2SE 5.0 Änderungen an der Sprache vor, indem neue Funktionen hinzugefügt wurden – zum Beispiel generische Typen, Annotations, varargs und die erweiterte Anweisung `for`.

Wenn Sie Abbildung 1.1 und Abbildung 1.2 miteinander vergleichen, fällt vielleicht auf, dass Android die Bühne betrat, als Java SE in der Version 6 vorlag. Das Ergebnis davon ist, dass Java für Android-Entwickler in der Version 6 eingefroren wurde. Wenn Sie eine Android-App entwickeln, können Sie entweder J2SE 5.0 oder Java SE 6 verwenden. Sie müssen auf Java SE 7 mit Strings in `switch`-Anweisungen oder auf Java SE 8 mit seinen Lambda-Ausdrücken verzichten. Das geht aber so in Ordnung: Sie werden als Android-Entwickler Funktionen dieser Art selten vermissen.

XML

Wenn Sie zufällig in den Extras Ihres Webbrowsers auf SEITENQUELLTEXT ANZEIGEN (oder so ähnlich) stoßen, sehen Sie eine Menge Tags der *Hypertext Markup Language* (HTML). Bei einem *Tag* handelt es sich um Text, der in eckigen Klammern (< und >) steht und etwas beschreibt, das sich in seiner unmittelbaren Nachbarschaft befindet.

Um zum Beispiel auf einer Webseite fett gedruckte Zeichen zu erhalten, schreibt ein Webentwickler:

```
<b>Schaut euch das hier an!</b>
```

Das `b` in den spitzen Klammern schaltet Fettschrift (englisch *boldface*) ein und wieder aus.

Das *M* in HTML steht für *Markup* – was auf Deutsch *Auszeichnung* bedeutet und bei dem es sich um einen allgemein gebräuchlichen Ausdruck handelt, der zusätzlichen Text beschreibt, der die Inhalte eines Dokuments erläutert. Wenn Sie den Inhalt eines Dokuments beschreiben, betten Sie Informationen im Dokument selbst ein. So lautet zum Beispiel der Inhalt des oben stehenden Codebeispiels `Schaut euch das hier an!`. Die Auszeichnung (Information über den Inhalt) besteht aus den Tags `<b>` und `</b>`.

Der Standard HTML ist ein Gewächs der *Standard Generalized Markup Language* (SGML; deutsch *Normierte Verallgemeinerte Auszeichnungssprache*), bei der es sich um eine Technologie für alles und jeden handelt, um Dokumente für die Verwendung auf allen möglichen Computern auszuzeichnen, auf denen alle mögliche Software läuft, die von allen möglichen Herstellern verkauft wird.

Mitte der 1990er begann eine Arbeitsgruppe des World Wide Web Consortiums (W3C) damit, die *eXtensible Markup Language* (deutsch *Erweiterbare Auszeichnungssprache*) zu entwickeln, die im Allgemeinen nur als XML bekannt ist. Ziel der Arbeitsgruppe war es, eine Teilmenge von SGML zu erstellen, die verwendet wird, um Daten über das Internet zu übertragen. Die Arbeit war von Erfolg gekrönt: XML hat sich zu einem viel beachteten Standard für das Codieren von Informationen aller Art gemausert. In Kapitel 4 gibt es einen optisch hervorgehobenen Bereich, der XML näher beschreibt.

Java ist ideal, um Anweisungen schrittweise zu deklarieren, und XML eignet sich gut dafür zu beschreiben, wie die Dinge sind (oder sein sollten). Ein Java-Programm sagt: »Mache dies und dann das.« Im Gegensatz dazu sagt ein XML-Dokument: »Das geht auf diese und dies auf jene Weise.«

Android nutzt XML aus zwei Gründen:

- ✔ **Um die Daten einer App zu beschreiben**

  Die XML-Dokumente einer App beschreiben das Layout der Bildschirme der App, die Übersetzung der App in eine oder mehrere andere Sprachen und weitere Arten von Daten.

- ✔ **Um die App selbst zu beschreiben**

  Zu jeder Android-App gehört eine Datei mit dem Namen `AndroidManifest.xml`. Dies ist ein XML-Dokument, das Funktionen der App beschreibt. Das Betriebssystem eines Gerätes verwendet den Inhalt des `AndroidManifest.xml`-Dokuments, um das Ausführen der App zu verwalten.

  So kann zum Beispiel die Datei `AndroidManifest.xml` Code vorgeben, der dafür sorgt, dass die App dem Benutzer auch aus anderen Apps heraus zur Verfügung steht. Dieselbe Datei beschreibt auch die Berechtigungen, die die App vom System verlangt. Wenn Sie mit der Installation einer neuen App beginnen, zeigt Android diese Berechtigungen an und bittet um Ihre Genehmigung, damit es die Installation fortsetzen kann. (Ich weiß nicht, wie Sie vorgehen, aber ich lese diese Liste mit den Berechtigungen immer sehr sorgfältig durch.) Sie finden in Kapitel 4 weitere Informationen zur Datei `AndroidManifest.xml`.

Wenn es um XML geht, habe ich schlechte und gute Nachrichten für Sie. Die schlechte Nachricht ist, dass XML nicht ganz so einfach zu beherrschen ist. Das Schreiben des XML-Codes ist im besten Fall langweilig. Im schlechtesten Fall führt das Schreiben des XML-Codes zu einer totalen Verwirrung. Die gute Nachricht ist, dass XML-Code größtenteils über automatisch arbeitende Werkzeuge »hergestellt« wird. Bei Ihnen als Android-Entwickler stellt die Software auf Ihrem Entwicklungscomputer den größten Teil des XML-Codes einer App zusammen. Sie legen dann noch Feinheiten fest, lesen Teile des Codes, um Informationen über seine Herkunft zu erhalten, führen kleinere Anpassungen durch und fügen kleinere Zusätze hinzu. Aber nur in den seltensten Fällen erstellen Sie ganz XML-Dokumente manuell.

Linux

Bei einem *Betriebssystem* handelt es sich um ein umfangreiches Programm, das für das generelle Laufverhalten eines Computers oder eines Gerätes zuständig ist. Die meisten Betriebssysteme bestehen aus Ebenen. Die nach außen gerichteten Ebenen des Betriebssystems schauen in der Regel den Benutzer an. So haben sowohl Windows als auch Mac OS X einen standardmäßigen Desktop. Der Benutzer startet von diesem Desktop aus Programme, verwaltet Fenster und erledigt andere wichtige Dinge.

Die internen Ebenen eines Betriebssystems können (zumindest meistens) vom Benutzer nicht gesehen werden. Während der Benutzer zum Beispiel Solitär spielt, jongliert das Betriebssystem mit Prozessen, verwaltet Dateien, behält die Sicherheit im Auge und erledigt insgesamt die Dinge, um deren Einzelheiten sich der Benutzer nicht kümmern kann.

Auf der untersten Ebene eines Betriebssystems befindet sich der sogenannte Kernel (deutsch: *Kern*) des Systems. Der Kernel läuft direkt auf der Prozessorhardware und führt die Arbeiten aus, die den Prozessor ans Laufen bringen. In einem System, das sauber in Ebenen aufgeteilt worden ist, erledigen die oberen Ebenen die Arbeit, indem sie Aufrufe an die unteren Ebenen tätigen. Eine App mit besonderen Anforderungen an die Hardware sendet diese Anforderungen (direkt oder indirekt) über den Kernel.

Die bekanntesten und beliebtesten überall einsetzbaren Betriebssysteme sind Windows, Mac OS X (bei dem es sich um echtes Unix handelt) und Linux. Sowohl Windows als auch Mac OS X sind Eigentum der entsprechenden Unternehmen. Demgegenüber ist Linux Open Source. Dies ist einer der Gründe dafür, dass die Schöpfer von Android ihr System auf dem Linux-Kernel aufgebaut haben.

Als Entwickler haben Sie den engsten Kontakt mit dem Android-Betriebssystem über die Befehlszeile, die auch *Linux-Shell* genannt wird. Die Shell verwendet Befehle wie `cd`, um in ein anderes Verzeichnis zu wechseln, `ls`, um die Dateien und Unterverzeichnisse eines Verzeichnisses aufzulisten, `rm`, um Dateien zu löschen, und viele weitere Befehle.

In Googles Android Market gibt es eine Menge kostenloser Terminal-Apps. Eine *Terminal-App* ist eine App, bei der die Oberfläche nur aus einem Textbildschirm besteht, auf dem Sie Linux-Shell-Befehle eingeben. Und indem Sie eines der Android-Entwicklerwerkzeuge, die Android Debug Bridge, verwenden, sind Sie in der Lage, einem Android-Gerät über Ihren Entwicklungscomputer Befehle zu erteilen. Wenn Sie sich gerne die virtuellen Hände schmutzig machen, ist die Linux-Shell genau das Richtige für Sie.

Mit Java von der Entwicklung bis zur Ausführung

Bevor es Java gab, musste ein Computerprogramm übersetzt werden, damit es ablaufen konnte. Irgendjemand (oder irgendetwas) übersetzte den Code, den ein Entwickler geschrieben hatte, in einen Code, der einer Geheimsprache ähnelte. Dieser Code konnte vom Computer ausgeführt werden. Aber dann tauchte Java auf und fügte eine eigene Übersetzungsebene hinzu, die letztendlich von Android um eine weitere Ebene ergänzt wurde. Dieser Abschnitt hier beschreibt diese Ebenen.

Was ist ein Compiler?

Ein Java-Programm (zum Beispiel eine Android-Anwendung) durchläuft auf seinem Weg, der mit dem ersten Schreiben des Codes durch Sie beginnt und zum Schluss vom Prozessor ausgeführt wird, verschiedene Übersetzungsschritte. Einer der Gründe hierfür ist recht einfach: Für die meisten Menschen ist es sehr schwer, Anweisungen zu schreiben, mit denen ein Computer problemlos umgehen kann.

Menschen können den Code in Listing 1.1 schreiben und verstehen:

```
public void checkVacancy(View view) {
  if (room.numGuests == 0) {
    label.setText("Verfügbar");
  } else {
    label.setText("Belegt :–(");
  }
}
```

Listing 1.1: Java-Quellcode

Der Java-Code in Listing 1.1 prüft nach, ob es in einem Hotel noch freie Zimmer gibt. Sie können diesen Code nicht ausführen, ohne noch ein paar zusätzliche Zeilen hinzuzufügen. Aber hier in Kapitel 1 interessieren diese zusätzlichen Zeilen nicht wirklich. Wichtig ist, dass Sie ein wenig blinzeln, wenn Sie auf den Code starren, und versuchen, hinter die ungewohnte Interpunktion zu kommen und herauszufinden, was der Code eigentlich will:

```
Wenn der Raum nicht durch Gäste belegt ist,
  soll der Text der Meldung auf "Verfügbar" gesetzt werden,
anderenfalls
  legen Sie "Belegt :–(" als Meldungstext fest
```

Bei dem Inhalt von Listing 1.1 handelt es sich um *Java-Quellcode*. Die Prozessoren in Computern, Telefonen und anderen Geräten folgen aber normalerweise Anweisungen wie denen in Listing 1.1 nicht. Oder anders ausgedrückt: Prozessoren befolgen keine Java-Anweisungen, die im Quellcode vorliegen. Stattdessen folgen Prozessoren kryptischen Anweisungen wie denen in Listing 1.2.

```
0 aload_0
1 getfield #19 <com/allmycode/samples/MyActivity/room
Lcom/allmycode/samples/Room;>
4 getfield #47 <com/allmycode/samples/Room/numGuests I>
7 ifne 22 (+15)
10 aload_0
11 getfield #41 <com/allmycode/samples/MyActivity/label
Landroid/widget/TextView;>
14 ldc #54 <Available>
16 invokevirtual #56
<android/widget/TextView/setText
(Ljava/lang/CharSequence;)V>
19 goto 31 (+12)
22 aload_0
23 getfield #41 <com/allmycode/samples/MyActivity/label
Landroid/widget/TextView;>
26 ldc #60 <Taken :–(>
28 invokevirtual #56
```

```
<android/widget/TextView/setText
(Ljava/lang/CharSequence;)V>
31 return
```

Listing 1.2: Java-Bytecode

Bei den Anweisungen in Listing 1.2 handelt es sich nicht um Java-Quellcodeanweisungen. Es sind Anweisungen, die in Form eines *Java-Bytecodes* vorliegen. Wenn Sie ein Java-Programm schreiben, schreiben Sie Quellcodeanweisungen (siehe Listing 1.1). Nachdem Sie den Quellcode geschrieben haben, führen Sie mit dem Quellcode ein Programm aus (das heißt, Sie wenden auf diesen Code ein Werkzeug an). Bei diesem Programm handelt es sich um einen Compiler (der im Deutschen auch *Kompiler* geschrieben und vom englischen *to compile* abgeleitet wird, was auf Deutsch *zusammentragen* bedeutet). Er übersetzt Ihre Quellcodeanweisungen in Java-Bytecode. Oder anders ausgedrückt: Der Compiler übersetzt Code, den Sie lesen und verstehen können, in Code, den Computer ausführen können.

Jetzt könnte der Zeitpunkt gekommen sein, an dem Sie fragen: »Was muss ich tun, damit der Compiler ausgeführt wird?« Die aus einem Wort bestehende Antwort auf Ihre Frage lautet: »Eclipse.« Alle Übersetzungsschritte, die in diesem Kapitel beschrieben werden, lassen sich auf den Einsatz von Eclipse reduzieren – einer Software, die Sie mithilfe der Anleitung in Kapitel 2 kostenlos herunterladen können. Wenn Sie also in diesem Kapitel vom Kompilieren und von anderen Übersetzungsschritten lesen, sollten Sie ganz ruhig bleiben. Sie müssen nicht in der Lage sein, eine Lichtmaschine zu reparieren, um ein Auto fahren zu können, und Sie müssen auch nicht verstehen, wie Compiler funktionieren, damit Sie Eclipse nutzen können.

Niemand (vielleicht bis auf ein paar verrückte Entwickler, die irgendwo in isolierten Computerräumen hocken) schreibt Java-Bytecode. Sie lassen Software (einen Compiler) ablaufen, um Java-Bytecode zu erzeugen. Der einzige Grund, sich näher mit Listing 1.2 zu beschäftigen, ist, dass Sie verstehen, wie viel Schwerstarbeit Ihr Computer leisten muss.

Wenn das Kompilieren als solches schon eine gute Sache ist, sollte doch zweimal Kompilieren noch besser sein. 2007 entwickelte Dan Bornstein von Google *Dalvic Bytecode*. Diese Art von Code stellt einen anderen Weg für Prozessoren dar, Anweisungen zu folgen. (Wenn Sie wissen wollen, woher Bornsteins Vorfahren stammen, nehmen Sie sich eine Karten-App und suchen dort nach Dalvik in Island.) Dalvik-Bytecode ist für die begrenzten Ressourcen eines Handys oder Tablets optimiert worden.

Listing 1.3 enthält ein Beispiel mit Dalvik-Anweisungen. Damit ich diesen Code sehen konnte, habe ich das Programm *Dedexer* verwendet. Sie finden es auf der Website `dedexer.source for ge.net`.

```
.method public checkVacancy(Landroid/view/View;)V
.limit registers 4
; this: v2 (Lcom/allmycode/samples/MyActivity;)
; parameter[0] : v3 (Landroid/view/View;)
.line 30
iget–object
```

```
v0,v2,com/allmycode/samples/MyActivity.room
Lcom/allmycode/samples/Room;
; v0 : Lcom/allmycode/samples/Room; , v2 :
Lcom/allmycode/samples/MyActivity;
    iget    v0,v0,com/allmycode/samples/Room.numGuests I
; v0 : single-length , v0 : single-length
    if-nez    v0,l4b4
; v0 : single-length
.line 31
iget-object
v0,v2,com/allmycode/samples/MyActivity.label
Landroid/widget/TextView;
; v0 : Landroid/widget/TextView; , v2 :
Lcom/allmycode/samples/MyActivity;
const-string    v1,"Available"
; v1 : Ljava/lang/String;
invoke-virtual
{v0,v1},android/widget/TextView/setText
; setText(Ljava/lang/CharSequence;)V
; v0 : Landroid/widget/TextView; , v1 : Ljava/lang/String;
l4b2:
.line 36
return-void
l4b4:
.line 33
iget-object
v0,v2,com/allmycode/samples/MyActivity.label
Landroid/widget/TextView;
; v0 : Landroid/widget/TextView; , v2 :
Lcom/allmycode/samples/MyActivity;
const-string    v1,"Taken :-("
; v1 : Ljava/lang/String;
invoke-virtual
{v0,v1},android/widget/TextView/setText ;
setText(Ljava/lang/CharSequence;)V
; v0 : Landroid/widget/TextView; , v1 : Ljava/lang/String;
    goto    l4b2
.end method
```

Listing 1.3: Dalvik-Bytecode

Wenn Sie eine Android-App erstellen, führt Eclipse mindestens zwei Kompilierungen durch:

✔ **Ein Kompilierungslauf erstellt aus Ihrer Java-Quelldatei Java-Bytecode.** Die Namen der Quelldateien haben die Erweiterung `.java`; die Java-Bytecode-Dateien haben die Erweiterung `.class`.

✔ **Ein weiterer Kompilierungslauf erzeugt aus den Java-Bytecode-Dateien Dalvik-Bytecode-Dateien.** Die Namen der Dalvik-Bytecode-Dateien haben die Erweiterung `.dex`.

Das ist aber noch nicht alles! Eine Android-App enthält zusätzlich zum Java-Code auch XML-Dateien, Bilddateien und alle möglichen anderen Elemente. Bevor Sie eine App auf einem Gerät installieren, fasst Eclipse diese Elemente in einer Datei zusammen, die dann die Erweiterung `.apk` erhält. Wenn Sie die App in einem App-Store veröffentlichen, kopieren Sie die `.apk`-Datei auf die Server des App-Stores. Und zum Schluss besucht ein Benutzer den App-Store und installiert die App, indem er die `.apk`-Datei herunterlädt.

Um Java-Quellcode zu Java-Bytecode zu kompilieren, verwendet Eclipse ein Programm mit dem Namen *javac*, das auch als »der Java-Compiler« bekannt ist. Um aus dem Java-Bytecode Dalvik-Code zu machen, verwendet Eclipse ein Programm, das den Namen *dx* trägt (und im Allgemeinen als *dx-Tool* bezeichnet wird). Und damit dann letztendlich die `.apk`-Datei zu erzeugen, verwendet Eclipse das Programm *apkbuilder*.

Was ist eine virtuelle Maschine?

Ich mache weiter vorn in diesem Kapitel im Abschnitt *Was ist ein Compiler?* ein ziemliches Getue um Telefone und Geräte, die Anweisungen wie denen in Listing 1.3 folgen. Wenn sich die Aufregung darüber gelegt hat, sehen Sie, dass eigentlich nur Positives übrig bleibt. Sie müssen dazu aber jedes Wort genau lesen, weil Sie sich ansonsten in die falsche Richtung bewegen. Es geht hier insbesondere um die Passage »folgen Prozessoren kryptischen Anweisungen *wie* denen in Listing …«. Die Anweisungen in Listing 1.3 ähneln schon stark denen, die ein Telefon oder ein Tablet ausführen kann. Aber Computer führen Java-Bytecode generell nicht direkt aus, und auch Telefone können Anweisungen in Form von Dalvik-Bytecode nicht direkt verwenden. Stattdessen besitzt jeder Prozessor seinen eigenen Satz an ausführbaren Anweisungen, die dann jedes Betriebssystem passgenau verwendet.

Stellen Sie sich vor, dass Sie zwei unterschiedliche Geräte besitzen: ein Smartphone und ein Tablet. Diese Geräte weisen unterschiedliche Prozessortypen auf. Das Telefon verfügt über einen ARM-Prozessor, und das Tablet besitzt einen Intel-Atom-Prozessor. (Das Akronym ARM stand ursprünglich für *Advanced RISC Machine*. Heutzutage bedeutet ARM einfach nur noch *ARM Holdings*. Dabei handelt es sich um ein Unternehmen, dessen Mitarbeiter Prozessoren entwerfen.) Auf einem ARM-Prozessor lautet die Anweisung zum Multiplizieren `000000`. Bei einem Intel-Prozessor lauten dieselbe Anweisungen `D8`, `DC`, `F6`, `F7` und anders. Für viele ARM-Anweisungen gibt es in der Atom-Architektur keine Entsprechung, und vielen Atom-Anweisungen fehlt ein Äquivalent auf der ARM-Seite. Letztendlich bedeutet dies, dass eine ARM-Anweisung für den Atom-Prozessor eines Tablets sinnlos ist und dass Anweisungen für den Atom-Prozessor bei einem ARM-Prozessor nur zu virtuellen Kopfschmerzen führen.

Was muss ein Entwickler tun? Muss er Übersetzungen aller Apps für alle denkbaren Prozessoren zur Verfügung stellen?

Nein. Es sind virtuelle Maschinen, die Ordnung in dieses Chaos bringen. Dalvik-Bytecode ähnelt zwar dem Code in Listing 1.3, aber er ist nicht auf einen einzelnen Prozessortyp oder ein einzelnes Betriebssystem begrenzt. Es gibt einen Satz von Dalvik-Anweisungen, der auf jedem

Prozessor laufen kann. Wenn Sie ein Java-Programm schreiben und durch Kompilieren zu Dalvik-Bytecode machen, kann Ihr Android-Telefon den Bytecode ausführen, Ihr Android-Tablet kann den Bytecode ausführen und selbst der Supercomputer Ihrer Großmutter ist in der Lage, den Bytecode auszuführen. (Damit das klappt, muss Ihre Großmutter auf der Intel-basierten Maschine *Android-x86* installieren.)

Sie müssen Dalvik-Bytecode niemals schreiben oder entschlüsseln. Es ist Aufgabe des Compilers, Bytecode »herzustellen«. Und es ist Aufgabe der virtuellen Maschine, Bytecode zu entschlüsseln.

Sowohl der Java- als auch der Dalvik-Bytecode haben virtuelle Maschinen. Bei der *Dalvik Virtual Machine* können Sie eine Bytecode-Datei nehmen, die Sie für ein Android-Gerät erstellt haben, den Bytecode auf ein anderes Android-Gerät kopieren und ihn dort problemlos ablaufen lassen. Dies ist einer der vielen Gründe, warum Android so schnell so beliebt geworden ist. Diese unglaubliche Funktion, die es Ihnen erlaubt, Code auf vielen verschiedenen Arten von Computern ablaufen zu lassen, wird *Portierbarkeit* genannt.

Stellen Sie sich vor, dass Sie der Vertreter von Intel beim United Nations Security Council wären (siehe Abbildung 1.3). Rechts neben Ihnen sitzt der Vertreter von ARM und links von Ihnen hat der Vertreter von Texas Instruments Platz genommen. Auf dem Podium befindet sich der Vertreter von Dalvik und hält einen Vortrag in Dalvik-Bytecode und weder Sie noch die anderen (von Intel und Texas Instruments) verstehen auch nur ein Wort Dalvik-Bytecode.

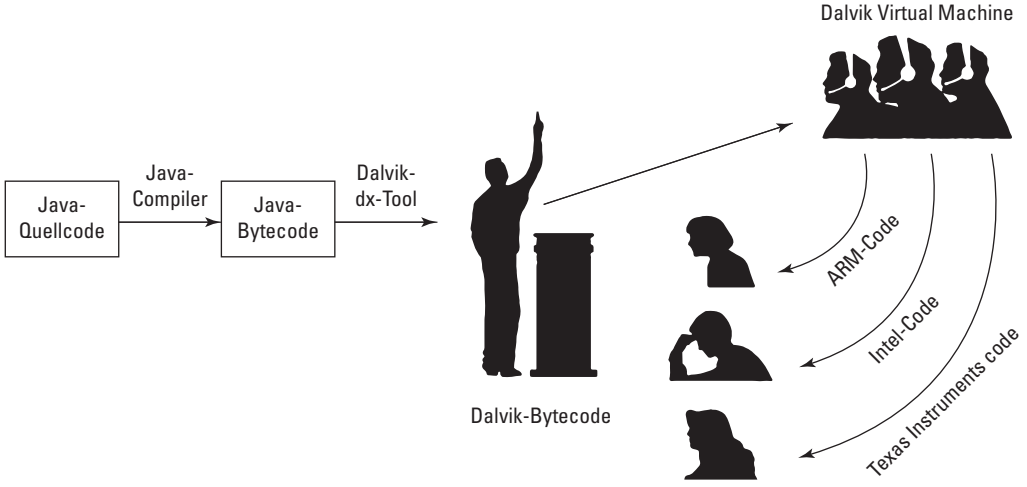

Abbildung 1.3: Ein imaginäres Treffen des UN Security Councils

Aber jeder von Ihnen hat einen Dolmetscher (der im Umfeld der Programmierung auch im Deutschen mit dem englischen Ausdruck für *Dolmetscher* als *Interpreter* bezeichnet wird). Ihr Interpreter übersetzt, während der Vertreter von Dalvik spricht, Dalvik-Bytecode in Intel-Anweisungen. Ein anderer Interpreter übersetzt Dalvik in »ARMisch«. Und der dritte Interpreter (Dolmetscher) übersetzt Bytecode in die Sprache von Texas Instruments.

Stellen Sie sich Ihren Interpreter als einen virtuellen Botschafter vor. Der Interpreter repräsentiert Ihr Land nicht wirklich, aber er erledigt eine wichtige Aufgabe, um die sich auch ein echter Botschafter kümmern muss: Er hört dem Dalvik-Bytecode in Ihrem Auftrag zu. Der Interpreter, der so tut, als wäre er der Intel-Botschafter, kümmert sich um die langweilige Bytecode-Sprache, nimmt jedes einzelne Wort auf und verarbeitet es irgendwie.

Sie besitzen einen Interpreter – diesen virtuellen Botschafter. Auf dieselbe Weise führt ein Intel-Prozessor seine eigene, den Bytecode interpretierende Software aus. Bei dieser Software handelt es sich um die *Dalvik Virtual Machine*. Sie dient zwischen Dalviks Code, der irgendwo laufen soll, und dem System Ihres Gerätes als Dolmetscher (Interpreter). Während die virtuelle Maschine läuft, leitet sie Ihr Gerät durch die Ausführung der Anweisungen des Bytecodes. Sie untersucht den Bytecode Bit für Bit und bringt Anweisungen ans Tageslicht, die im Bytecode beschrieben werden. Die virtuelle Maschine interpretiert Bytecode für ARM-Prozessoren, Intel-Prozessoren, Texas-Instruments-Chips oder beliebige andere Prozessoren. Und genau das macht Java-Code und Dalvik-Code so viel portierbarer als Code, der in einer anderen Sprache geschrieben worden wäre.

Java, Android und Gartenbau

»Du siehst den Wald vor lauter Bäumen nicht«, sagte mein Onkel Harvey. Und meine Tante Clara ergänzte: »Und du siehst die Bäume nicht, die den Wald bilden.« Die beiden diskutierten dann dieses Thema so lange, bis sie müde wurden und ins Bett gingen.

Als Autor möchte ich gerne sowohl den Wald als auch die Bäume darstellen. »Wald« ist dabei der große Überblick, der Ihnen hilft zu verstehen, warum Sie verschiedene Schritte ausführen müssen. Und »Bäume« entspricht den einzelnen Schritten, die Sie von Punkt A zu Punkt B bringen, damit Sie eine Aufgabe vollenden können.

Dieses Kapitel zeigt Ihnen den Wald. Der Rest des Buches zeigt Ihnen die Bäume.