# Table of Contents

**Welcome to the Warescription!**.................................................................xi

**Preface**........................................................................................................xiii

Welcome to the Book!............................................................................................xiii
What This Book Is All About............................................................................xiii
Prerequisites........................................................................................................xv
Warescription.......................................................................................................xvi
Book Bug Bounty................................................................................................xvii
Source Code.......................................................................................................xviii
Creative Commons and the Four-to-Free (42F) Guarantee...............................xviii
Lifecycle of a CommonsWare Book..................................................................xix
Acknowledgements.........................................................................................xx

**The Role of Alternative Environments**.........................................................1

In the Beginning, There Was Java.................................................................2
...And It Was OK..............................................................................................2
Bucking the Trend............................................................................................3
Support, Structure...........................................................................................4
Caveat Developer.............................................................................................4
Today’s Menu....................................................................................................5
About the Sample Applications.......................................................................6
# HTML5 Features for Android

<table>
<thead>
<tr>
<th>Feature</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offline Applications</td>
<td>7</td>
</tr>
<tr>
<td>Web Storage</td>
<td>14</td>
</tr>
<tr>
<td>Other Supported HTML5 Features</td>
<td>17</td>
</tr>
<tr>
<td>Geolocation</td>
<td>18</td>
</tr>
<tr>
<td>Video</td>
<td>18</td>
</tr>
<tr>
<td>Going To Production</td>
<td>18</td>
</tr>
<tr>
<td>Testing</td>
<td>18</td>
</tr>
<tr>
<td>Signing and Distribution</td>
<td>18</td>
</tr>
<tr>
<td>Updates</td>
<td>18</td>
</tr>
<tr>
<td>Issues You May Encounter</td>
<td>19</td>
</tr>
<tr>
<td>Android Device Versions</td>
<td>19</td>
</tr>
<tr>
<td>Screen Sizes and Densities</td>
<td>19</td>
</tr>
<tr>
<td>Limited Platform Integration</td>
<td>20</td>
</tr>
<tr>
<td>Performance and Battery</td>
<td>20</td>
</tr>
<tr>
<td>Look and Feel</td>
<td>21</td>
</tr>
<tr>
<td>Distribution</td>
<td>22</td>
</tr>
<tr>
<td>HTML5 and Alternative Android Browsers</td>
<td>22</td>
</tr>
<tr>
<td>Mozilla Fennec</td>
<td>22</td>
</tr>
<tr>
<td>Opera Mini</td>
<td>22</td>
</tr>
<tr>
<td>Skyfire</td>
<td>23</td>
</tr>
<tr>
<td>Dolphin</td>
<td>23</td>
</tr>
<tr>
<td>Steel</td>
<td>23</td>
</tr>
<tr>
<td>Chapter</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>HTML5: The Baseline</td>
<td>23</td>
</tr>
<tr>
<td>PhoneGap</td>
<td>25</td>
</tr>
<tr>
<td>What Is PhoneGap?</td>
<td>25</td>
</tr>
<tr>
<td>What Do You Write In?</td>
<td>25</td>
</tr>
<tr>
<td>What Features Do You Get?</td>
<td>26</td>
</tr>
<tr>
<td>What Do Apps Look Like?</td>
<td>27</td>
</tr>
<tr>
<td>How Does Distribution Work?</td>
<td>27</td>
</tr>
<tr>
<td>What About Other Platforms?</td>
<td>28</td>
</tr>
<tr>
<td>Using PhoneGap</td>
<td>28</td>
</tr>
<tr>
<td>Examining the Sample Application</td>
<td>29</td>
</tr>
<tr>
<td>Porting HTML5 to PhoneGap</td>
<td>29</td>
</tr>
<tr>
<td>Going to Production</td>
<td>29</td>
</tr>
<tr>
<td>Testing</td>
<td>29</td>
</tr>
<tr>
<td>Signing and Distribution</td>
<td>29</td>
</tr>
<tr>
<td>Updates</td>
<td>29</td>
</tr>
<tr>
<td>Issues You May Encounter</td>
<td>30</td>
</tr>
<tr>
<td>Security</td>
<td>30</td>
</tr>
<tr>
<td>Screen Sizes and Densities</td>
<td>30</td>
</tr>
<tr>
<td>Performance and Battery</td>
<td>30</td>
</tr>
<tr>
<td>Look and Feel</td>
<td>30</td>
</tr>
<tr>
<td>Rhodes</td>
<td>31</td>
</tr>
<tr>
<td>What Is Rhodes?</td>
<td>31</td>
</tr>
<tr>
<td>What Do You Write In?</td>
<td>32</td>
</tr>
<tr>
<td>What Features Do You Get?</td>
<td>32</td>
</tr>
<tr>
<td>What Do Apps Look Like?</td>
<td>33</td>
</tr>
<tr>
<td>How Does Distribution Work?</td>
<td>35</td>
</tr>
<tr>
<td>What About Other Platforms?</td>
<td>35</td>
</tr>
</tbody>
</table>

This document is licensed for Melvin Parinas’s exclusive use by CommonsWare, LLC
Using Rhodes.................................................................36
  Installation........................................................................36
  Creating and Installing Your Project..........................37
Examining the Sample Application.................................39
  Static Assets.................................................................39
  Configuration Files......................................................39
  Templates........................................................................39
  Application Class.........................................................39
  Helpers...........................................................................39
Porting HTML5 to Rhodes...............................................40
Going to Production.........................................................40
  Testing............................................................................40
  Signing and Distribution..............................................40
  Updates..........................................................................40
Issues You May Encounter................................................40
  Device Compatibility...................................................41
  Application Size..........................................................41
  Broken BACK Button..................................................41
  Slow Operation............................................................41
  Security..........................................................................41
  GPS and Battery..........................................................42
Flash and AIR....................................................................43
  Where We Are Today....................................................43
  Flash..............................................................................43
  Flex.................................................................................44
  AIR.................................................................................45
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>What Is Scala?</td>
<td>47</td>
</tr>
<tr>
<td>What Do You Write In?</td>
<td>48</td>
</tr>
<tr>
<td>What Features Do You Get?</td>
<td>48</td>
</tr>
<tr>
<td>What Do Apps Look Like?</td>
<td>48</td>
</tr>
<tr>
<td>How Does Distribution Work?</td>
<td>49</td>
</tr>
<tr>
<td>Using Scala</td>
<td>49</td>
</tr>
<tr>
<td>Installing Scala</td>
<td>49</td>
</tr>
<tr>
<td>Setting Up a Android Project for Scala</td>
<td>50</td>
</tr>
<tr>
<td>A Sample Scala Application</td>
<td>50</td>
</tr>
<tr>
<td>Going to Production</td>
<td>50</td>
</tr>
<tr>
<td>Testing</td>
<td>50</td>
</tr>
<tr>
<td>Signing and Distribution</td>
<td>50</td>
</tr>
<tr>
<td>Updates</td>
<td>50</td>
</tr>
<tr>
<td>Issues You May Encounter</td>
<td>50</td>
</tr>
<tr>
<td>Other Alternative Environments</td>
<td>51</td>
</tr>
<tr>
<td>Titanium Mobile</td>
<td>51</td>
</tr>
<tr>
<td>Bedrock</td>
<td>51</td>
</tr>
<tr>
<td>Corona</td>
<td>51</td>
</tr>
<tr>
<td>DroidScript</td>
<td>51</td>
</tr>
<tr>
<td>Flixeel</td>
<td>52</td>
</tr>
<tr>
<td>MobiForms</td>
<td>52</td>
</tr>
<tr>
<td>MonoDroid</td>
<td>52</td>
</tr>
<tr>
<td>MoSync</td>
<td>52</td>
</tr>
<tr>
<td>Squeak</td>
<td>52</td>
</tr>
<tr>
<td>SuperWaba</td>
<td>52</td>
</tr>
<tr>
<td>Other Alternative Environments</td>
<td>51</td>
</tr>
<tr>
<td>Titanium Mobile</td>
<td>51</td>
</tr>
<tr>
<td>Bedrock</td>
<td>51</td>
</tr>
<tr>
<td>Corona</td>
<td>51</td>
</tr>
<tr>
<td>DroidScript</td>
<td>51</td>
</tr>
<tr>
<td>Flixeel</td>
<td>52</td>
</tr>
<tr>
<td>MobiForms</td>
<td>52</td>
</tr>
<tr>
<td>MonoDroid</td>
<td>52</td>
</tr>
<tr>
<td>MoSync</td>
<td>52</td>
</tr>
<tr>
<td>Squeak</td>
<td>52</td>
</tr>
<tr>
<td>SuperWaba</td>
<td>52</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>The Role of the NDK</td>
<td>55</td>
</tr>
<tr>
<td>Dalvik: Secure, Yes; Speedy, Not So Much</td>
<td>55</td>
</tr>
<tr>
<td>Going Native</td>
<td>56</td>
</tr>
<tr>
<td>Speed</td>
<td>57</td>
</tr>
<tr>
<td>Porting</td>
<td>57</td>
</tr>
<tr>
<td>Skills</td>
<td>58</td>
</tr>
<tr>
<td>Knowing Your Limits</td>
<td>58</td>
</tr>
<tr>
<td>Android APIs</td>
<td>58</td>
</tr>
<tr>
<td>Cross-Platform Compatibility</td>
<td>59</td>
</tr>
<tr>
<td>Maturity</td>
<td>60</td>
</tr>
<tr>
<td>Available Expertise</td>
<td>60</td>
</tr>
<tr>
<td>Setting Up the NDK</td>
<td>61</td>
</tr>
<tr>
<td>Wrapping a Native Library</td>
<td>63</td>
</tr>
<tr>
<td>The Role of Scripting Languages</td>
<td>67</td>
</tr>
<tr>
<td>All Grown Up</td>
<td>67</td>
</tr>
<tr>
<td>Following the Script</td>
<td>68</td>
</tr>
<tr>
<td>Your Expertise</td>
<td>68</td>
</tr>
<tr>
<td>Your Users’ Expertise</td>
<td>69</td>
</tr>
<tr>
<td>Crowd-Developing</td>
<td>69</td>
</tr>
<tr>
<td>Going Off-Script</td>
<td>69</td>
</tr>
<tr>
<td>Security</td>
<td>69</td>
</tr>
<tr>
<td>Performance</td>
<td>69</td>
</tr>
<tr>
<td>Cross-Platform Compatibility</td>
<td>69</td>
</tr>
<tr>
<td>Maturity...On Android</td>
<td>69</td>
</tr>
<tr>
<td>The Android Scripting Environment</td>
<td>71</td>
</tr>
<tr>
<td>The Role of ASE</td>
<td>71</td>
</tr>
<tr>
<td>On-Device Development</td>
<td>71</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Rhino</td>
<td>88</td>
</tr>
<tr>
<td>Other JVM Scripting Languages</td>
<td>88</td>
</tr>
<tr>
<td>Binary Executable Engines</td>
<td>89</td>
</tr>
<tr>
<td>Scripting via the NDK</td>
<td>91</td>
</tr>
<tr>
<td>Checklist in Java</td>
<td>93</td>
</tr>
<tr>
<td>The Layout File</td>
<td>93</td>
</tr>
<tr>
<td>The Java Source</td>
<td>95</td>
</tr>
<tr>
<td>DatabaseHelper</td>
<td>95</td>
</tr>
<tr>
<td>The Activity</td>
<td>96</td>
</tr>
<tr>
<td>The List and Entries</td>
<td>96</td>
</tr>
<tr>
<td>The Menus</td>
<td>96</td>
</tr>
</tbody>
</table>
Welcome to the Warescription!

We hope you enjoy this digital book and its updates – keep tabs on the Warescription feed off the CommonsWare site to learn when new editions of this book, or other books in your Warescription, are available.

Each Warescription digital book is licensed for the exclusive use of its subscriber and is tagged with the subscribers name. We ask that you not distribute these books. If you work for a firm and wish to have several employees have access, enterprise Warescriptions are available. Just contact us at enterprise@commonsware.com.

Also, bear in mind that eventually this edition of this title will be released under a Creative Commons license – more on this in the preface.

Remember that the CommonsWare Web site has errata and resources (e.g., source code) for each of our titles. Just visit the Web page for the book you are interested in and follow the links.

You can search through the PDF using most PDF readers (e.g., Adobe Reader). If you wish to search all of the CommonsWare books at once, and your operating system does not support that directly, you can always combine the PDFs into one, using tools like PDF Split-And-Merge or the Linux command `pdftk *.pdf cat output combined.pdf`.

Some notes for first-generation Kindle users:

- You may wish to drop your font size to level 2 for easier reading
• Source code listings are incorporated as graphics so as to retain the monospace font, though this means the source code listings do not honor changes in Kindle font size
Welcome to the Book!

Android is always looking for new developers, and developers wishing to try new avenues of creating applications. The fact that you are reading this suggests you are in one of those two groups, and it is great that you are interested in Android!

If you come to this book having read other CommonsWare Android books in the Warescription, thanks for sticking with the series! CommonsWare aims to have the most comprehensive set of Android development resources (outside of the Open Handset Alliance itself), and I appreciate your interest.

If you come to this book having learned about Android from other sources – or if you are new to Android entirely – thanks for joining the CommonsWare community! Android, while aimed at small devices, is a surprisingly vast platform, making it difficult for any given book, training, wiki, or other source to completely cover everything one needs to know.

And, most of all, thanks for your interest in this book! I sincerely hope you find it useful and at least occasionally entertaining.
What This Book Is All About

As the title suggests, this book is about developing for Android beyond the bounds of "traditional" Java-based development. Of course, since Android is a relative newcomer to the mobile OS scene, "traditional" is a bit...excessive, as only time will tell what the primary modes of working in Android turn out to be. Certainly today, if you survey the books and blogs and bunches of other materials, Java development is Android's current focus.

It does not have to be that way, at least not completely.

Unlike some mobile operating systems, Android has long been a playground for people interested in experimenting with this language or that. Some of these efforts are official; most are driven by the community or businesses interested in getting a piece of Android development ecosystem. Regardless, these efforts have borne fruit, and this book is your farmers' market, showing you what is possible.

The book is divided into three parts.

First, we look at alternative application development frameworks. Due to some peculiarities of the current Android architecture, many possible languages and coding techniques are not suitable for creating full-fledged Android applications, the way you can with Java. However, there are a few that let you create an Android application, front to back, without a stitch of Java code. This part examines several of these, so you can see how you can use them to create applications of your own.

Next, we examine the Android Native Development Kit, or NDK. This piece of technology, released by Google in the summer of 2009, allows one to write Android application code in C or C++. Previously, those languages were limited for use in modifying Android itself. The NDK cannot be used to create full applications – instead, you use C/C++ to create or port libraries that are attached to a standard Android Java application via the Java Native Interface (JNI). This part will walk you through how to set up the NDK toolchain and how to use it to integrate your desired C/C++ with your Android application.
Finally, we wrap by looking at **scripting languages**. For the purposes of this book, a "scripting language" is defined as one that is interpreted at runtime and cannot be used to create a standalone Android application. Rather, the scripting language is used as an extension to a Java-based application. This may be to enabled development by people less skilled in Java. This may be to allow device *users* to write scripts to do this or that, without their having to build everything using the standard Android Java tools.

Note that this book **surveys** the available options. For many of the technologies, it shows how to use them, but only enough to give you a sense of what is possible. Many of these technologies are worthy of a book in their own right – in the case of HTML5 and Adobe Flash/AIR, there will probably be many books. Hence, do not go into this book thinking that you will walk out being an expert on all of these technologies. What you will be an expert in are the options and their respective pros and cons.

**Prerequisites**

This book is a bit unusual, in that the three different parts require different skill sets.

The first part, on alternative application development frameworks, requires modest prior Android experience. You will, however, need something on which to run applications created with these frameworks, whether that is an Android device or the Android emulator that comes with the standard Java-based Software Development Kit (SDK).

The second part, on the Native Development Kit, requires both experience with writing Android applications in Java and experience in working with C/C++ code. You may, of course, be more comfortable with one over the other, but you will need some abilities in both areas.

The third part, on scripting languages, requires experience with Java-based Android application development and, ideally, experience with whatever scripting language you wish to integrate into your application. However, one chapter – covering the Android Scripting Environment (ASE) – requires no Java skills, only experience in a supported language, like Python or Perl.
If you picked this book up expecting to learn Android application development in Java, you really need another source first, since this book focuses on other topics. While we are fans of The Busy Coder's Guide to Android Development and the other books in the CommonsWare Warescription, there are plenty of other books available covering the Android basics, blog posts, wikis, and, of course, the main Android site itself. A list of currently-available Android books can be found on the Android Programming knol.

Some chapters may reference material in previous chapters, though usually with a link back to the preceding section of relevance.

In order to make effective use of this book, you will want to download the source code for it off of the book's page on the CommonsWare site.

You can find out when new releases of this book are available via:

- The cw-android Google Group, which is also a great place to ask questions about the book and its examples
- The commonsguy Twitter feed
- The CommonsBlog
- The Warescription newsletter, which you can subscribe to off of your Warescription page

**Warescription**

This book will be published both in print and in digital form. The digital versions of all CommonsWare titles are available via an annual subscription – the Warescription.

The Warescription entitles you, for the duration of your subscription, to digital forms of all CommonsWare titles, not just the one you are reading. Presently, CommonsWare offers PDF and EPUB.

Each subscriber gets personalized editions of all editions of each title: both those mirroring printed editions and in-between updates that are only available in digital form. That way, your digital books are never out of date
for long, and you can take advantage of new material as it is made available instead of having to wait for a whole new print edition. For example, when new releases of the Android SDK are made available, this book will be quickly updated to be accurate with changes in the APIs.

From time to time, subscribers will also receive access to subscriber-only online material, including not-yet-published new titles.

Also, if you own a print copy of a CommonsWare book, and it is in good clean condition with no marks or stickers, you can exchange that copy for a free four-month Warescription.

If you are interested in a Warescription, visit the Warescription section of the CommonsWare Web site.

**Book Bug Bounty**

Find a problem in one of our books? Let us know!

Be the first to report a unique concrete problem in the current digital edition, and we'll give you a coupon for a six-month Warescription as a bounty for helping us deliver a better product. You can use that coupon to get a new Warescription, renew an existing Warescription, or give the coupon to a friend, colleague, or some random person you meet on the subway.

By "concrete" problem, we mean things like:

- Typographical errors
- Sample applications that do not work as advertised, in the environment described in the book
- Factual errors that cannot be open to interpretation

By "unique", we mean ones not yet reported. Each book has an errata page on the CommonsWare Web site; most known problems will be listed there. One coupon is given per email containing valid bug reports.
NOTE: Books with version numbers lower than 0.9 are ineligible for the bounty program, as they are in various stages of completion. We appreciate bug reports, though, if you choose to share them with us.

We appreciate hearing about "softer" issues as well, such as:

- Places where you think we are in error, but where we feel our interpretation is reasonable
- Places where you think we could add sample applications, or expand upon the existing material
- Samples that do not work due to "shifting sands" of the underlying environment (e.g., changed APIs with new releases of an SDK)

However, those "softer" issues do not qualify for the formal bounty program.

Questions about the bug bounty, or problems you wish to report for bounty consideration, should be sent to bounty@commonsware.com.

Source Code

The source code samples shown in this book are available for download from a GitHub repository. All of the Android projects are licensed under the Apache 2.0 License, in case you have the desire to reuse any of it.

If you wish to use the source code from the CommonsWare Web site, bear in mind a few things:

1. The Java projects are set up to be built by Ant, not by Eclipse. If you wish to use the code with Eclipse, you will need to create a suitable Android Eclipse project and import the code and other assets.

2. For the Java projects, you should delete build.xml, then run `android update project -p ...` (where ... is the path to a project of interest) on those projects you wish to use, so the build files are updated for your Android SDK version.
Creative Commons and the Four-to-Free (42F) Guarantee

Each CommonsWare book edition will be available for use under the Creative Commons Attribution-Noncommercial-Share Alike 3.0 license as of the fourth anniversary of its publication date, or when 4,000 copies of the edition have been sold, whichever comes first. That means that, once four years have elapsed (perhaps sooner!), you can use this prose for non-commercial purposes. That is our Four-to-Free Guarantee to our readers and the broader community. For the purposes of this guarantee, new Warescriptions and renewals will be counted as sales of this edition, starting from the time the edition is published.

This edition of this book will be available under the aforementioned Creative Commons license on May 1, 2014. Of course, watch the CommonsWare Web site, as this edition might be relicensed sooner based on sales.

For more details on the Creative Commons Attribution-Noncommercial-Share Alike 3.0 license, visit the Creative Commons Web site.

Note that future editions of this book will become free on later dates, each four years from the publication of that edition or based on sales of that specific edition. Releasing one edition under the Creative Commons license does not automatically release all editions under that license.

Lifecycle of a CommonsWare Book

CommonsWare books generally go through a series of stages.

First are the pre-release editions. These will have version numbers below 0.9 (e.g., 0.2). These editions are incomplete, often times having but a few chapters to go along with outlines and notes. However, we make them available to those on the Warescription so they can get early access to the material.
Release candidates are editions with version numbers ending in ".9" (0.9, 1.9, etc.). These editions should be complete. Once again, they are made available to those on the Warescription so they get early access to the material and can file bug reports (and receive bounties in return!).

Major editions are those with version numbers ending in ".0" (1.0, 2.0, etc.). These will be first published digitally for the Warescription members, but will shortly thereafter be available in print from booksellers worldwide.

Versions between a major edition and the next release candidate (e.g., 1.1, 1.2) will contain bug fixes plus new material. Each of these editions should also be complete, in that you will not see any "TBD" (to be done) markers or the like. However, these editions may have bugs, and so bug reports are eligible for the bounty program, as with release candidates and major releases.

A book usually will progress fairly rapidly through the pre-release editions to the first release candidate and Version 1.0 – often times, only a few months. Depending on the book's scope, it may go through another cycle of significant improvement (versions 1.1 through 2.0), though this may take several months to a year or more. Eventually, though, the book will go into more of a "maintenance mode", only getting updates to fix bugs and deal with major ecosystem events – for example, a new release of the Android SDK will necessitate an update to all Android books.

Acknowledgements

The author would like to thank Alex Gibson, author of the HTML5 MiniApps used as examples in this book, for making them available under an open source license and permitting their inclusion.
PART I – Alternative Application Environments
You might think that Android is all about Java. The official Android Software Development Kit (SDK) is for Java development, the build tools are for Java development, the discussion groups and blog posts and, yes, most books are for Java development.

However (and with apologies to William Goldman), it just so happens that Android is only mostly Java. There’s a big difference between mostly Java and all Java. Mostly Java is slightly not Java.

So, while Android’s "sweet spot" will remain Java-based applications for the near term, you can still create applications using other technologies. This first part of the book will review what those other technologies are.

This chapter starts with an examination of the pros and cons of Android's Java-centric strategy. It then enumerates some reasons why you might want to use something else for your Android applications. The downsides of alternative Android application environments – lack of support and technical challenges – are discussed, before reviewing what some of the players are in this space and setting up the chapters to come.
The core Android team made a fairly reasonable choice of language when they chose Java. It is a very popular language, and in the mobile community it had a clear predecessor in Java Micro Edition (J2ME). Lacking direct access to memory addresses (so-called "pointers"), a Java-based application will be less prone to developer errors leading to buffer overruns, resulting in possible hacks. And there is a fairly robust ecosystem around Java, in terms of educational materials, existing code bases, integrated development environments (IDEs), and so on.

However, while you can program Android in the Java language, an Android device does not run a Java application. Instead, your Java code is converted into something that runs on the "Dalvik virtual machine". This is akin to the technology used for regular Java applications, but Dalvik is specifically tuned for Android's environment. Moreover, it limits the dependency of Android on Java itself to a handful of programming tools, important as Java's stewardship moves from Sun to Oracle to wherever.

That Dalvik virtual machine is also capable of running code from other programming languages, a feature that makes possible much of what this book covers.

...And It Was OK

No mobile development environment is perfect, and so the combination of Java and Android has its issues.

At the time of this writing, Java, as implemented for the Dalvik virtual machine, is interpreted, without any of the "just-in-time compiler" tricks regular Java uses to boost performance. This is a bigger problem in mobile, since the devices Android runs upon tend to be less powerful than your average desktop, notebook, or Web server. Hence, there will be some things you just can't do on Android with Java because it is too slow.
Java uses garbage collection to save people from having to keep track of all of their memory allocations. That works for the most part, and it is generally a boon to developer productivity. However, it is not a cure-all for every memory and resource allocation problem. You can still have what amounts to "memory leaks" in Java, even if the precise mechanics of those leaks differ from the classic leaks you get in C, C++, etc.

Most importantly, though, not everybody likes Java. It could be because they lack experience with it, or perhaps they have experience with it and did not enjoy that experience. Certainly, Java is slowly being considered as a language for big enterprise systems and, therefore, is not necessarily "cool". Advocates of different languages will have their own pet peeves with Java as well (e.g., to a Ruby developer, Java is really verbose).

So, while Java was not a bad choice for Android, it was not perfect, either.

**Bucking the Trend**

However, just because Java is the dominant way to build apps for Android, that does not mean it is the only way, and for you, it may not even be the best way.

Perhaps Java is not in your existing skill set. You might be a Web developer, more comfortable with HTML, CSS, and Javascript. There are frameworks to help you with that. Or, maybe you cut your teeth on server-side scripting languages like Perl or Python – there are ways to sling that code on Android as well. Or perhaps you already have a bunch of code in C/C++, such as game physics algorithms, that would be painful to rewrite in Java – you should be able to reuse that code too.

Even if you would be willing to learn Java, it may be that your inexperience with Java and the Android APIs will just slow you down. You might be able to get something built much more quickly with another framework, even if you wind up replacing it with a Java-based implementation in the future. Rapid development and prototyping is frequently important, to get early feedback with minimal investment in time.
And, of course, you might just find Java programming to be irritating. You would not be the first, nor the last, to have that sentiment. Particularly if you are getting into Android as a hobby, rather than as part of your "day job", having fun will be important to you, and you might not find Java to be much fun.

Fortunately, Android is friendly towards alternative ways of building applications, unlike some mobile platforms.

**Support, Structure**

However, "friendly" and "fully supported" are two different things.

Some alternatives to Java-based development are officially supported by the core Android team, such as C/C++ development via the Native Development Kit (NDK) and Web-style development via HTML5.

Some alternatives to Java-based development are supported by companies. Adobe supports Flash, Appcelerator supports Titanium Mobile, Rhomobile supports Rhodes, and so on. Other alternatives are supported by standards bodies, like the World Wide Web Consortium (W3C) supporting HTML5. Still others are just tiny projects with only the backing of a couple of developers.

You will need to make the decision for yourself which of these levels of support will meet your requirements. For many things, support is not much of an issue, but there will always be cases where support becomes paramount (e.g., enterprise application development).

**Caveat Developer**

Of course, going outside the traditional Java environment for Android development has its issues, beyond just how much support might be available.
The Role of Alternative Environments

Some may be less efficient, in terms of processor time, memory, or battery life, than will development in Java. C/C++, on the whole, is probably better than Java, but HTML5 may be worse, for example. Depending on what you are writing and how heavily it will be used will determine how critical that inefficiency will be.

Some may not be available on all devices. Right now, Flash is the best example of this – some devices offer some amount of Flash support, while other devices have no Flash at all. Similarly, HTML5 support was only added to Android in Android 2.0, so devices running older versions of Android do not have HTML5 as a built-in option.

Every layer between you and officially supported environments makes it that much more difficult for you to ensure compatibility with new versions of Android, when they arise. For example, if you create an application using PhoneGap, and a new Android version becomes available, there may be incompatibilities that only the PhoneGap team can address. While they will probably address those quickly – and they may provide some measure of insulation to you from those incompatibilities – the response time is outside of your control. In some cases, that is not a problem, but in other cases, that might be bad for your project.

Hence, just because you are developing outside of Java does not mean everything is perfect. You simply have to trade off between these problems and the ones Java-based development might cause you. Where the balance lies is up to each individual developer or firm.

Today's Menu

This first portion of the book covers alternative development environments that allow you to create full-fledged Android applications. Here, "full-fledged" means that you do not have to write any Java code in order to make your applications usable and distributable.

Five such environments are covered in detail:
The Role of Alternative Environments

- HTML5, using techniques that work across all HTML5-capable browsers, mobile or desktop
- PhoneGap, for creating applications using HTML, CSS, and Javascript
- Rhodes, which uses HTML and Ruby for developing Android applications
- Flash and AIR, Adobe technologies popular on the desktop and now moving into mobile
- Scala, a compiled functional language that runs on the Dalvik virtual machine

Another chapter covers other options for development sans Java, and some of those tools will be "promoted" to full chapters in future editions of this book.

About the Sample Applications

For a somewhat apples-to-apples comparison between the environments, each chapter will examine two applications. The first will be whatever sort of "Hello, world" or "kitchen sink" application comes with the environment itself, to give you an idea of what the environment is capable of. The other will be a single small application ported to each environment. The exception will be HTML5, in which we only examine the latter application.

That latter application is Checklist, a very simple to-do list tracker. You can add items to the list, check them off to mark them as completed, and delete items from the list. You can also send the list of outstanding to-do items to you (e.g., via email). While this app pales in comparison to Remember the Milk or similar sites, it makes for a simple demonstration, including connecting to a data store and outbound communications.

For comparison purposes, you can find an implementation of Checklist in "traditional" Android Java in an appendix of this book.
Prior to the current wave of interest in mobile applications, the technology *du jour* was Web applications. A lot of attention was paid to AJAX, Ruby on Rails, and other techniques and technologies that made Web applications climb close to experience of a desktop application, and sometimes superior.

The explosion of Web applications eventually drove the next round of enhancements to Web standards, collectively called HTML5. Android 2.0 added the first round of support for these HTML5 enhancements. Notably, Android supports offline applications and Web storage, meaning that HTML5 becomes a relevant technique for creating Android applications, without dealing with Java.

**Offline Applications**

The linchpin for using HTML5 for offline applications – on Android or elsewhere – is the ability for those applications to be used when there is no connectivity, either due to problems on the client side (e.g., on an airplane sans WiFi) or on the server side (e.g., Web server maintenance).

**What Does It Mean?**

Historically, Web applications have had this annoying tendency to require Web servers. This led to all sorts of workarounds for offline use, up to and including shipping a Web server and deploying it to the desktop.
HTML5 solves this problem by allowing Web pages to specify their own caching rules. A Web app can publish a "cache manifest", describing which resources:

- Can be safely cached, such that if the Web server is unavailable, the browser will just use the cached copy
- Cannot be safely cached, such that if the Web server is unavailable, the browser should fail like it normally does
- Have a "fallback" resource, such that if the Web server is unavailable, the cached fallback resource should be used instead

For mobile devices, this means that a fully HTML5-capable browser should be able to load all its assets up front and keep them cached. If the user loses connectivity, the application will still run. In this respect, the Web app behaves almost identically to a regular app.

**How Do You Use It?**

For this chapter, we will use the Checklist "mini app" created by Alex Gibson. While the most up-to-date version of this app can be found at the MiniApps Web site, this chapter will review the copy found in HTML5/Checklist of the book’s source code repository. This copy is also hosted online on the CommonsWare site, or via a shortened URL: http://bit.ly/cw-html5.

**About the Sample App**

Checklist is, as the name suggests, a simple checklist application. When you first launch it, the list will be empty:
Figure 1. The Checklist, as initially launched

You can enter some text in the top field and click the Add button to add it to the list:
You can "check off" individual items, which are then displayed in strike-through:
You can also delete the checked entries (via the Delete Checked button) or all entries (via the Delete All button), which will pop up a confirmation dialog before proceeding:
"Installing" Checklist on Your Phone

To access Checklist on your Android device, visit one of the URLs above for the hosted edition using the Browser application – the shortened one may be easiest to enter into the browser on the device. You can then add a bookmark for it (More > Add bookmark from the browser's option menu) to come back to it later.

You can even set up a shortcut for the bookmark on your home screen, if you so choose – just long-tap on the background, choose Bookmark, then choose the Checklist bookmark you set up before.

Examining the HTML

All of that is accomplished using just a handful of lines of HTML:
For the purposes of offline applications, though, the key is the `manifest` attribute of our `html` element:

```html
<html lang="en" manifest="checklist.manifest">
```
Here, we specify the relative path to a manifest file, indicating what the rules are for caching various portions of this application offline.

**Examining the Manifest**

So, since the manifest is where all the fun is, here is what Checklist's manifest looks like:

```
CACHE MANIFEST #version 54 styles.css main.js splashscreen.png
```

<< TBD: fix formatting error with that snippet! >>

The HTML5 manifest format is extremely simple. It starts with a `CACHE MANIFEST` line, followed by a list of files (technically, relative URLs) that should be cached. It also supports comments, which are lines beginning with `#`.

The manifest can also have a `NETWORK:` line, followed by relative URLs that should never be cached. Similarly, the manifest can have a `FALLBACK:` line, followed by pairs of relative URLs: the URL to try to fetch off the network, followed by the URL of a cached resource to use if the network is not available.

In principle, the manifest should request caching for everything that the application needs to run, though the page that requested the caching (`index.html` in this case) is also cached.

**Web Storage**

Caching the HTML5 application's assets for offline use is all well and good, but that will be rather limiting on its own. In an offline situation, the application would not be able to use AJAX techniques to interact with a Web service. So, if the application is going to be able to store information, it will need to do so on the browser itself.
Google Gears and related tools pioneered this concept and blazed the trail for what is now variously called "Web Storage" or "DOM Storage" for HTML5 applications. An HTML5 app can store data persistently on the client, within client-imposed limits. That, in conjunction with offline asset caching, means an HTML5 application can deliver far more value when it lacks an Internet connection, or for data that just does not make sense to store "in the cloud".

Note that, technically, Web Storage is not part of HTML5, but is a related specification. However, it tends to get "lumped in with" HTML5 in common conversation.

**What Does It Mean?**

On a Web Storage-enabled browser, your Javascript code will have access to a `localStorage` object, representing your application's data. More accurately, each "origin" (i.e., domain) will have a distinct `localStorage` object on the browser.

The `localStorage` object is an "associative array", meaning you can work with it either via numerical indexes or string-based keys at your discretion. Values typically are strings. You can:

- Find out how many entries are in the array via `length()`
- Get and set items by key via `getItem()` and `setItem()`
- Get the key for a numerical index via `key()`
- Remove individual entries via `removeItem()` or remove all items via `clear()`

This means you do not have the full richness of a SQL database, like you might have with SQLite in a native Android application. But, for many applications, this should suffice.
How Do You Use It?

Checklist stores the list items as keys in the associative array, with a value of 0 for a regular item and 1 for a deleted item. Here, we see the code for putting a new item into the checklist:

```javascript
try {
    localStorage.setItem(strippedString, data);
} catch (e) {
    if (e == QUOTA_EXCEEDED_ERR) {
        alert('Quota exceeded!');
    }
}
```

Here is the code where those items are pulled back out of storage and put into an array for sorting and, later, display as DOM elements on the Web page itself:

```javascript
/*get all items from localStorage and push them one by one into an array.*/
for (i = 0; i <= listlength; i++) {
    var item = localStorage.key(i);
    myArray.push(item);
}
/*sort the array into alphabetical order.*/
myArray.sort();
```

When the user checks the checkmark next to an item, the storage is updated to toggle the checked setting persistently:

```javascript
/*toggle the check flag.*/
if (target.previousSibling.checked) {
    data = 0;
} else {
    data = 1;
}
/*save item in localStorage.*/
try {
    localStorage.setItem(name, data);
} catch (e) {
    if (e == QUOTA_EXCEEDED_ERR) {
        alert('Quota exceeded!');
    }
}
```
Checklist also has code to delete items from storage, either all those marked as checked:

```javascript
/*remove every item from localStorage that has the data flag checked.*/
while (i <= localStorage.length-1) {
    var key = localStorage.key(i);
    if (localStorage.getItem(key) === '1') {
        localStorage.removeItem(key);
    } else { i++; }
}

...or all items:

/*deletes all items in the list.*/
deleteAll: function() {

    /*ask for user confirmation.*/
    var answer = confirm("Delete all items?");

    /*if yes.*/
    if (answer) {

        /*remove all items from localStorage.*/
        localStorage.clear();
        /*update view.*/
        checklistApp.getAllItems();
    }
    /*clear up.*/
    delete checklistApp.deleteAll;
},
```

### Other Supported HTML5 Features

HTML5 is much broader than simply offline caching of assets and Web storage. The depth of Android’s support for HTML5 should continuously expand, as WebKit adds capabilities and Android rolls those into new versions of the operating system.

This section describes some additional features that may be relevant for an HTML5 application for Android.
Geolocation

Video

**Going To Production**

Creating a little test application requires nothing magical. Presumably, though, you are interested in others using your application – perhaps many others. Classic Java-based Android applications have to deal with testing, having the application digitally signed for production, distribution through various channels (such as the Android Market), and updates to the application by one means or another. Those issues do not all magically vanish because HTML5 is used as the application environment. However, HTML5 does change things significantly from what Java developers have to do.

Testing

Signing and Distribution

Updates
Issues You May Encounter

Unfortunately, nothing is perfect. While HTML5 may make many things easier, it is not a panacea for all Android development problems.

This section covers some potential areas of concern you will want to consider as you move forward with HTML5 applications for Android.

Android Device Versions

Not all Android devices support HTML5 – only those running Android 2.x or higher. Ideally, therefore, you do a bit of "user-agent sniffing" on your Web server and redirect older Android users to some other page explaining the limitations in their device.

Here is the user-agent string for a Nexus One device running Android 2.1:

```
Mozilla/5.0 (Linux; U; Android 2.1-update1; en-us; Nexus One Build/ERE27)
 AppleWebKit/530.17 (KHTML, like Gecko) Version/4.0 Mobile Safari/530.17
```

As you can see, it is formatted like a typical modern user-agent string, meaning it is quite a mess. It does indicate it is running Android 2.1-update1.

Eventually, somebody will create a database of user-agent strings for different device models, and from there we can derive appropriate regular expressions or similar algorithms to determine whether a given device can support HTML5 applications.

Screen Sizes and Densities

HTML5 applications can be run on a wide range of screen sizes, from QVGA Android devices to 1080p LCDs and beyond. Similarly, screen densities may vary quite a bit, so while a 48x48 pixel image on a smartphone may be an appropriate size, it may be too big for a 1080p television, let alone a 24" LCD desktop monitor.
Other than increasing the possible options on the low end of screen sizes, none of this is unique to Android. You will need to determine how best to design your HTML and CSS to work on a range of sizes and densities, even if Android were not part of the picture.

**Limited Platform Integration**

HTML5, while offering more platform integration than ever before, does not come close to covering everything an Android application might want to be able to do. For example, an ordinary HTML5 application cannot:

- Launch another application
- Work with the contacts database
- Raise a notification
- Do work truly in the background (though "Web workers" may alleviate this somewhat someday)
- Interact with Bluetooth devices
- Record audio or video
- Use the standard Android preference system
- Use speech recognition or text-to-speech
- And so on

Many applications will not need these capabilities, of course. And, one can expect that other application environments, like PhoneGap and Rhodes – profiled in upcoming chapters – will evolve into "HTML5 Plus" for Android. That way, you could create a stock application that works across all devices and an enhanced Android application that leverages greater platform integration, at the cost of some additional amount of programming.

**Performance and Battery**

There has been a nagging concern for some time that HTML-based user interfaces are inefficient compared to native Android UIs, in terms of
processor time, memory, and battery. For example, one of the stated reasons for avoiding BONDI-style Web widgets for the Android home screen is performance.

Certainly, it is possible to design HTML5 applications that will suck down the battery. For example, if you have a hunk of Javascript code running every second indefinitely, that is going to consume a fair amount of processor time. However, outside of that, it seems unlikely that an ordinary application would be used so heavily as to materially impact battery life. Certainly, more testing will need to be done in this area.

Also, an HTML5 may be a bit slower to start up than are other applications, if the Browser has not been used in a while, or if the network connection is there but has minimal bandwidth to your server.

Look and Feel

HTML5 applications can certainly look very slick and professional – after all, they are built with Web technologies, and Web apps can look very slick and professional.

However, HTML5 applications will not necessarily look like standard Android applications, at least not initially. Some enterprising developers will, no doubt, create some reusable CSS, Javascript, and images that will, for example, mirror an Android native Spinner widget (a type of drop-down control). Similarly, HTML5 applications will tend to lack option menus, notifications, or other UI features that a native Android application may well use.

This is not necessarily bad. Considering the difficulty in creating a very slick-looking Android application, HTML5 applications may tend to look better than their Android counterparts. After all, there are many more people skilled in creating slick Web apps than are skilled in creating slick Android apps.
However, some users may complain about the look-and-feel disparity, just because it is different.

**Distribution**

HTML5 applications can be trivially added to a user's device – browse, bookmark, and add a shortcut to the home screen.

However, HTML5 applications will not show up in the Android Market, so users trained to look at the Market for available applications will not find HTML5 applications, even ones that may be better than their native counterparts.

It is conceivable that, someday, the Android Market will support HTML5 applications. It is also conceivable that, someday, Android users will tend to find their apps by means other than searching the Android Market, and will be able to get their HTML5 apps that way. However, until one of those becomes true, HTML5 applications may be less "discoverable" than their native equivalents.

**HTML5 and Alternative Android Browsers**

While the built-in Android browser will be the choice of many Android users, there are other browsers available. Here is how some of the better-known alternatives stand in terms of HTML5 support.

**Mozilla Fennec**

**Opera Mini**
HTML5: The Baseline

HTML5 is likely to become rather popular for conventional application development. It gives Web developers a route to the desktop. It may be the only option for Google’s Chrome OS. And, with ever-improving support on popular mobile devices – Android among them – developers will certainly be enticed by another round of "write once, run anywhere" promises.

It is fairly likely that HTML5 will be the #2 option for Android application development, after the conventional Java application written to the Android SDK. That will make HTML5 the baseline for comparing alternative Android development options – not only will those options be compared to using the SDK, they will be compared to using HTML5.
CHAPTER 3

PhoneGap

PhoneGap is perhaps the original alternative application framework for Android, already on the scene in early 2009. PhoneGap is open source, backed by Nitobi, who offers a mix of open source and commercial products, along with consulting and training services.

What Is PhoneGap?

As the PhoneGap About page puts it:

*The PhoneGap mission is to Web-enable native device functionality with open standards like HTML, CSS and JavaScript so that you can focus on the app you’re building, not on authoring complex platform compatibility layers.*

PhoneGap, today, focuses on bridging the gap between Web technologies and native mobile development, with access to more features than HTML5 applications have.

What Do You Write In?

A PhoneGap application is made up of HTML, CSS, and JavaScript, no different than a mobile Web site or HTML5 application, except that the
Web assets are packaged with the application, rather than downloaded on the fly.

A pre-installed PhoneGap application, therefore, can contain comparatively large assets, such as complex JavaScript libraries, that might be too slow to download over slower EDGE connections. However, PhoneGap will still be limited by the speed of mobile devices and how quickly WebKit can load and process those assets.

Also, development for WebKit-for-mobile has its differences over development for WebKit-for-desktops, particularly with respect to touch versus mouse events. You may want to develop using mobile layers of JavaScript frameworks (e.g., jqTouch versus plain jQuery) where practical.

What Features Do You Get?

As with an HTML5 application, you get the basic capabilities of a Web browser, including AJAX support. Beyond that, PhoneGap adds a number of JavaScript APIs to allow you to get at the underlying features of the Android platform. At the time of this writing, that includes:

- Accelerometer access, for detecting movement of the device
- Audio recording and playback
- Camera access, for taking still pictures
- Database access, both to databases of your creation (SQLite) or others built into Android (e.g., contacts)
- File system access, such as to the SD card
- Geolocation, for determining where the device is
- Vibration, for shaking the phone (e.g., force-feedback)

Since some of these are part of the HTML5 specification (e.g., geolocation), you have your choice of APIs. Also, this list changes over time, so you may have access to more than what is described here.
What Do Apps Look Like?

They will look like Web pages, more so than native Android apps. You can use CSS and images to mimic the Android look and feel to some extent, but only for those sorts of widgets that are readily able to be created in both Android and HTML. For example, the Android Spinner widget – which resembles a drop-down list – may be difficult to mimic in HTML.

Here is a screenshot of the example application that ships with PhoneGap:

![Figure 5. The example application that comes with PhoneGap](image)

How Does Distribution Work?

Distributing a PhoneGap application is pretty much identical to distributing any other standard Android application. After testing, you will create a standard APK file with the Android build tools, from an Android project generated for you by PhoneGap. This project will contain the Java, XML, and other necessary bits to wrap around your HTML, CSS, and
PhoneGap

JavaScript to make up your application. Then, you digitally sign the application and upload it to the Android Market or any other distribution mechanism you wish to use.

What About Other Platforms?

PhoneGap is not just for Android. You can create PhoneGap applications for iPhone, Blackberry, some flavors of Symbian, and Palm’s WebOS. In theory, at least, you can create one application using HTML, CSS, JavaScript, and the PhoneGap JavaScript APIs, and have it run across many devices.

There are a couple of limitations that will hamper your progress to that goal:

1. The Web browsing component used by PhoneGap across all those platforms is not identical. Even multiple platforms using WebKit will have different WebKit releases, based upon what was available when WebKit was integrated into a given device's firmware. Hence, you will want to test and ensure your CSS, in particular, works as you would expect on as many devices as possible.

2. Not all PhoneGap JavaScript APIs are available on all devices as yet, due to a variety of factors (e.g., not exposed in the platform's native APIs, lack of engineering time to hoist the capability into the PhoneGap APIs). There is a table on the PhoneGap wiki that will keep you apprised of what works and what does not across the devices. You will want to restrict your feature use to match your desired platforms, or restrict your platforms to match your desired features.

Using PhoneGap

Now, let's look at more of the mechanics for using PhoneGap.

PhoneGap's installation and usage, as of the time of this writing, normally requires an expert in Java-based Android development. You need to install a
whole bunch of tools, edit configuration files by hand, and so forth. If you want to do all of that, documentation is available on the PhoneGap site.

If you are reading this chapter, there's a decent chance that you would rather skip all of that.

<< TBD >>

Examining the Sample Application

Porting HTML5 to PhoneGap

Going to Production

Testing

Signing and Distribution

Updates
To the App

To PhoneGap Itself

Issues You May Encounter

Security

permissions required – always READ_PHONE_STATE and INTERNET

Screen Sizes and Densities

Performance and Battery

Look and Feel
Rhodes is another alternative application framework that focuses on using Web technologies – HTML, CSS, and JavaScript – for the presentation. It differs from PhoneGap in how you write your business logic. PhoneGap has you write everything in JavaScript, leveraging their own JavaScript libraries for device functions. Rhodes, on the other hand, gives you a Ruby runtime that you use, much like you might write a Ruby Web application using Rails, Sinatra, etc. Rhodes is an open source project run by Rhomobile.

What Is Rhodes?

As the Rhodes product page puts it:

*Rhodes is an open source framework to rapidly build NATIVE apps for all major smartphone operating systems (iPhone, Windows Mobile, RIM, Symbian and Android). These are true native device applications (NOT mobile web apps) which work with synchronized local data and take advantage of device capabilities such as GPS, PIM contacts and camera.*

While PhoneGap has the feel of an HTML5 offline Web application with extra features, Rhodes has the feel of a Rails app, squeezed into a mobile device.
What Do You Write In?

Your UI is created using HTML, CSS, and JavaScript, akin to PhoneGap or an HTML5 application. The difference is that the HTML you create is actually a set of ERB templates. ERB templates will be familiar to those of you who have done work with Rails or similar Ruby frameworks in the past. In a nutshell, they allow you to intersperse Ruby code with your HTML markup, with the Ruby bits providing your gateway to device-specific functionality, databases, etc.

As with PhoneGap, an installed Rhodes application can have larger JavaScript libraries, images, and the like than you might ordinarily want to download at runtime. However, you are still limited by WebKit’s speed in parsing and executing the JavaScript, plus the idiosyncrasies of a mobile version of a browser. The Rhodes project recommends the use of jqTouch "for styling your interfaces and providing native-like effects such as sliding transitions".

Rhodes attempts to enforce a model-view-controller (MVC) pattern, much like Rails does. Hence, if you are a Rails aficionado, you should have a relatively easy time adjusting to Rhodes development. On the other hand, newcomers to Ruby and Rails might want to experiment with some small Rails applications first, to get comfortable with the environment, before tackling Rhodes.

What Features Do You Get?

As with PhoneGap or HTML5, you get everything you might expect to get with a mobile WebKit-based browser for your GUI.

The Rhodes Ruby class library, baked into every application, gives you access to:

- Audio playback
- Camera access, for taking still pictures
Rhodes

- Database access, both to databases of your creation (SQLite) or others built into Android (e.g., contacts)
- Geolocation, for determining where the device is
- Google Maps access, if needed

Databases you create for Rhodes, using the Rhom object-relational mapping library, can be optionally synced to a RhoSync server, for both local and "in the cloud" storage.

What Do Apps Look Like?

As with HTML5 or PhoneGap applications, Rhodes apps tend to look like Web apps more than they look native apps. This is not inherently bad, but it is different from what users might ordinarily expect.

Here is the default "Loading..." screen for a stub project you create with Rhodes:
And here is the initial "home page" of that stub project, reached when the "loading" process is complete:
How Does Distribution Work?

Distributing a Rhodes application, as with PhoneGap, is pretty typical for Android application in general. After testing, you will create a standard APK file via some commands supplied to you by Rhodes. Those commands can even handle digitally signing your APK for production distribution via the Android Market or other channels.

While Rhodes used to be licensed under the GPLv3 – with ramifications for proprietary applications – the recent Rhodes 2.0 release changed the license to the MIT License, which should cause few, if any, challenges.

What About Other Platforms?

As noted above, Rhodes supports iPhone, Blackberry, Symbian, and Windows Mobile in addition to Android.
Rhodes

The ability of any of these tools to target certain platforms, like iPhone, is subject to the license agreement for developing for that platform. Apple's 2010 modifications to their developer agreement contains some specific language that prohibits certain tools and may impact using Rhodes for iPhone. Please consult with qualified legal counsel if you have any concerns over Apple's developer agreement.

As with PhoneGap, Rhodes does not support all device-specific features across all platforms. There is a table on the Rhodes wiki that outlines the current state of what is and is not supported.

And, as with any browser-based GUI framework, the browser versions vary somewhat between mobile platforms, so you will want to test your application on whichever platforms you are targeting.

Using Rhodes

Now, let's look at the steps for actually using Rhodes.

Note that, at the time of this writing, setting up Rhodes may require the assistance of somebody skilled at Java-based Android development, to get all

Installation

There are several prerequisites you should install on your development machine first:

- Ruby, RubyGems, and Rake, such as the Windows "One-Click Installer" or as provided for you from your Linux package manager
- A GNU-compatible make program, such as the one from your Linux package manager or the GnuWin32 edition
- A working Android development setup, including Java JDK, the Android SDK, and the Android NDK
Rhodes

You will not be using most of those directly, but the Rhodes environment will expect all of them.

Then, you can install Rhodes by executing the following command:

```
gem install rhodes
```

This may take a few minutes, as there are yet more dependencies that Rhodes will take care of as part of this installation process. **NOTE:** Linux (and perhaps OS X) users may need to run this command via `sudo`, depending on how your RubyGems environment is set up.

Once that completes, you can run the `rhodes-setup` command to provide configuration details about your development machine. In particular, it will be looking for the paths to the Java JDK, Android SDK, and Android NDK (plus paths to other platform development tools, should be developing for other mobile environments as well). Here is the list of questions you will be asked:

```
$ rhodes-setup
We will ask you a few questions below about your dev environment.

JDK path (required) (/usr/lib/jvm/java-6-sun):
Android 1.5 SDK path (blank to skip) (/opt/android-sdk-linux_x86-1.6_r1):
Android NDK path (blank to skip) ():
Windows Mobile 6 SDK CabWiz (blank to skip) ():
BlackBerry JDE 4.6 (blank to skip) ():
BlackBerry JDE 4.6 MDS (blank to skip) ():
BlackBerry JDE 4.2 (blank to skip) ():
BlackBerry JDE 4.2 MDS (blank to skip) ():

If you want to build with other BlackBerry SDK versions edit:
/var/lib/gems/1.8/gems/rhodes-1.5.4/rhobuild.yml
```

Creating and Installing Your Project

Next, you need to create a Rhodes project in which to put your HTML, CSS, JavaScript, and related Ruby code. To do that, run the `rhogen app` command. This takes the name of the project as a command-line parameter, such as:

```
rhogen app Checklist
```
This will create a directory tree containing your Rhodes project. It bears no resemblance to a traditional Android project – if anything, it looks a lot like a Rails project. Notably, your CSS, JavaScript, and images reside in subdirectories under `public/`, and your ERB templates and static HTML are in the `app/` directory. We will take a look at those files and see what you need to manipulate later in this chapter.

![Figure 8. The default directory tree of a Rhodes project](image)

When you are ready to try out your modifications to your project, the simple answer is to execute the following command from your project's root directory:

```
rake run:android
```

This will compile your application, create an Android 1.5 emulator, launch the emulator, and install the application in that emulator. You will then find your application in the Android home screen launcher under whatever name you gave the project (e.g., Checklist).

If you wish to test your application on another emulator (e.g., Android 2.1) or on a device, run:

```
rake device:android:debug
```
Rhodes

This will compile your application to an Android executable (APK), located in `bin/target/` in your project, which you can then install on your Android device or emulator via:

```
adb install bin/target/YourProjectName-debug.apk
```

(substituting in your project name for `YourProjectName` above)

**Examining the Sample Application**

If you create a boilerplate Rhodes application, using the above commands, you will get a project directory pre-populated with a bunch of files, designed for you to edit, replace, or in some cases, remove.

**Static Assets**

**Configuration Files**

**Templates**

**Application Class**

**Helpers**
Porting HTML5 to Rhodes

Going to Production

Testing

Signing and Distribution

Updates

To the App

To Rhodes Itself

Issues You May Encounter

No application development framework is perfect. Cross-platform frameworks frequently are less-than-perfect, as perfection sometimes gets in the way of the cross-platform goal. Hence, it is not surprising that
Rhodes has issues, ones that may or may not affect you in your development.

Device Compatibility

Rhodes relies upon the Android NDK to provide the Ruby runtime used to interpret the Ruby code interspersed in your Rhodes application. At the time of this writing, the NDK only supports ARM development. Since ARM is the chipset used in most Android-powered smartphones, this means Rhodes applications can be widely deployed. However, Android does run on other chipsets, such as MIPS and Intel's Atom. Until such time as the NDK supports these chipsets, your Rhodes applications will not work on devices powered by non-ARM chips. This may include new "sexy" areas like Android-powered televisions, set-top boxes, and tablets.

Application Size

1.5MB for hello, world; double that upon installation (almost 4MB)

Broken BACK Button

Slow Operation

Security

Waaaaaaaaaaaaaaaaaaaaaay too many permissions required, not configurable
GPS and Battery
Outside of the mobile realm, Adobe’s Flash is perhaps the #2 cross-platform tool in use today, based on popularity (after HTML-based Web technologies). Adobe’s AIR – allowing Flash-based applications to run "on the desktop" – is an increasingly popular choice as well. Hence, it is no surprise that Android advocates have waited with bated breath for Flash and AIR to arrive on Android.

That time has come...at least in part.

**Where We Are Today**

Official Android ports of these core Adobe technologies were announced at the Google I|O 2010 conference in May 2010. However, these ports are not final and have significant limitations at this time.

**Flash**

Adobe’s Flash Player 10.1 is available for Android 2.2 in a public beta, and only for actual Android hardware – not the emulator. This has two major ramifications:

1. Only supporting Android 2.2 means millions of existing devices will not be able to run Flash any time soon, if ever. Hence, you are
s

significantly limiting your target audience if you elect to use Flash today

2. Not supporting the emulator increases testing costs, as Flash developers will need to purchase an Android 2.2 (or higher) device

Preliminary indications, based on the published release notes, are that the Flash Player 10.1 is designed primarily to work inside a Web browser. Whether this will solely be the built-in Android browser or whether this will include other third-party browsers (e.g., Mozilla Fennec, Opera Mini) has yet to be determined.

Note that some devices had Flash before Android 2.2. However, these were based on an older Flash specification ("Flash Lite") and only appeared on select models.

Flex

Flash, in its raw form, is designed for playing back animations. While many developers have created Flash-based applications and widgets this way, many developers have also had problems figuring out how to adapt their existing GUI skills to Flash.

Adobe released Flex to attempt to address this. Flex allows developers to create Flash applications using an XML-based markup language, coupled with Actionscript (a scripting language based on Javascript).

However, according to Adobe's site, "Flex 4 is not yet optimized for mobile devices", suggesting that using Flex is not necessarily ideal. Actionscript itself is usable, as one "can also create Actionscript only projects using Flash Builder 4 to produce high performance Android applications." However, without the Flex class library, Actionscript itself may be insufficient to dramatically simplify Flash-based application development.
AIR

AIR is a piece of technology designed to create native applications using Flash and Flex. On the desktop, AIR uses WebKit in conjunction with a Flash player to achieve this end. On Android, presumably it does much the same thing, just resulting in an APK file instead of, say, a Windows EXE. A pre-release version of AIR for Android is now available.

However, at the time of this writing, this book can say no more on the subject, because in order to learn more, one has to sign an NDA:

You will be subject to terms of an NDA which allows you to display your application publicly, but you will not be permitted to disclose anything outside of Licensee’s own personal development experiences of a beta Android application(s) using this beta AIR SDK.

This chapter will be significantly extended sometime in the future, after this NDA restriction is lifted.
Many programming languages exist that are designed to run on a Java virtual machine. These generally come in two flavors: those that generate Java classes (in source or bytecode) ahead of time, and those that do not. Those that do stand a very good chance of working with Android, with only minor tweaks to the build process used to build regular Java-based Android applications.

Perhaps the most popular of these languages, in terms of its use with Android, is Scala.

**What Is Scala?**

As the Scala site describes it:

> Scala is a general purpose programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented and functional languages, enabling Java and other programmers to be more productive. Code sizes are typically reduced by a factor of two to three when compared to an equivalent Java application.
Scala

In particular, Scala is a functional programming language. For a long time, functional programming languages were considered fairly esoteric. Now, such languages are considered very important for scalable applications spanning multiple cores of a CPU. In effect, multi-core CPUs are forcing developers to improve the parallelism of their applications, and functional languages fit in well with that.

While the issue of scaling is less critical in Android (typically running fairly limited CPUs), the increase in interest in Scala for other fields naturally causes Scala developers to want to do their Android work in Scala. And, of course, you may find functional programming Scala-style to be to your liking, even if you only use Scala for Android.

What Do You Write In?

As one might expect, you write Android applications in Scala using Scala.

Scala, however, is a replacement for Java only in the Android programming model. While HTML5, PhoneGap, and Rhodes all replace the Java-based model, Scala replaces only the language. Hence, you will still be writing a fair amount of XML – for menus, preferences, UI layouts, etc. – and using Scala for the balance.

What Features Do You Get?

Scala integrates fully with class libraries supported on the same JVM. Hence, Scala has access to just about everything from the Android SDK. Unlike the other frameworks, you do not "lose" features by switching to Scala. At the same time, though, your only change is the programming language, whereas the other frameworks tend to simplify even more of the application.
What Do Apps Look Like?

Scala applications will be indistinguishable, visually, from Java-based Android applications. Of course, it is perfectly possible to make a Scala Android application look nothing like traditional Android applications, but the same can be said for Java.

How Does Distribution Work?

Scala applications result in standard APK files, no different than their Java counterparts. However, there is an additional JAR file – scala-library.jar – which will need to be included as part of your APK for the Scala-created code to function properly.

Using Scala

Most of the other alternative application frameworks described so far in this book cater to developers who really would rather not know much of anything about the Android SDK and its tools. They may be used to developing Web apps (HTML5, PhoneGap, Rhodes) or Flash animations.

Scala developers, on the other hand, need to be reasonably proficient today in Java-based Android application development. In effect, Scala developers will be mentally translating the samples and techniques shown for Java to Scala equivalents, and tweaking the Android SDK build chain to use Scala as the base language instead of Java.

Installing Scala

Android integration works best with Scala 2.8.0 or newer. At the time of this writing, Scala 2.8.0 is at RC2 and therefore may ship as a final release before this book does.
Scala

You can get Scala from the Scala download page on the project's Web site. Eventually, Linux users may also be able to get a sufficiently-recent Scala package from their distro's package manager.

Scala is just a ZIP or TGZ archive – no fancy installer is necessary. Just unpack it into some likely directory. You will also want to:

- Add a SCALA_HOME environment variable, pointing to where you have Scala unpacked
- Update your PATH to include the bin/ directory underneath SCALA_HOME

Setting Up a Android Project for Scala

A Sample Scala Application

Going to Production

Testing

Signing and Distribution
Updates

Issues You May Encounter
CHAPTER 7

Other Alternative Environments

The alternative application environments described in the preceding chapters are but the tip of the iceberg. Here, we will take a look at other contenders.

**Titanium Mobile**

**Bedrock**

**Corona**

**DroidScript**
Other Alternative Environments

Flixel

MobiForms

MonoDroid

MoSync

Squeak

SuperWaba
PART II – The Native Development Kit
CHAPTER 8

The Role of the NDK

When Android was first released, many a developer wanted to run C/C++ code on it. There was little support for this, other than by distributing a binary executable and running it via a forked process. While this works, it is a bit cumbersome, and the process-based interface limits how cleanly your C/C++ code could interact with a Java-based UI. On top of all of that, the use of such binary executables is not well supported.

In June 2009, the core Android team released the Native Development Kit (NDK). This allows developers to write C/C++ for Android applications in a supported fashion, in the form of libraries linked to a hosting Java-based application via the Java Native Interface (JNI). This offers a wealth of opportunities for Android development, and this part of the book will explore how you can take advantage of the NDK to exploit those opportunities.

We start by examining Dalvik’s primarily limitation – speed. Next, we look at the reasons one might choose the NDK, speed among them. We wrap up with some reasons why the NDK may not be the right solution for every Android problem, despite its benefits.

Dalvik: Secure, Yes; Speedy, Not So Much

Dalvik was written with security as a high priority. Android’s security architecture is built around Linux’s user model, with each application
The Role of the NDK

going its own user ID. With each application’s process running under its own user ID, one process cannot readily affect other processes, helping to contain any single security flaw in an Android application or subsystem. This requires a fair number of processes. However, phones have limited RAM, and the Android project wanted to offer Java-based development. Multiple processes hosting their own Java virtual machines simply could not fit in a phone. Dalvik’s virtual machine is designed to address this, maximizing the amount of the virtual machine that can be shared securely between processes (e.g., via "copy-on-write").

Of course, it is wonderful that Android has security so woven into the fabric of its implementation. However, inventing a new virtual machine required tradeoffs, and most of those are related to speed.

A fair amount of work has gone into making Java fast. Standard Java virtual machines do a remarkable job of optimizing applications on the fly, such that Java applications can perform at speeds near their C/C++ counterparts. This borders on the amazing and is a testament to the many engineers who put countless years into Java.

Dalvik, by comparison, is very young. Many of Java’s performance optimization techniques – such as advanced garbage collection algorithms – simply have not been implemented to nearly the same level in Dalvik. This is not to say they will never exist, but it will take some time. Even then, though, there may be limits as to how fast Dalvik can operating, considering that it cannot "throw memory at the problem" to the extent Java can on the desktop or server.

If you need speed, Dalvik is not the answer today, and may not be the answer tomorrow, either.

Going Native

Java-based Android development via Dalvik and the Android SDK is far and away the option with the best support from the core Android team. HTML5 application development is another option that was brought to you by the
The Role of the NDK

core Android development team. The third leg of the official Android development triad is the NDK, provided to developers to address some specific problems, outlined below.

Speed

Far and away the biggest reason for using the NDK is speed, pure and simple. Writing in C/C++ for the device’s CPU will be a major speed improvement over writing the same algorithms in Java, even given the just-in-time compiler added to Android 2.2.

<< insert benchmarks here >>

There is overhead in reaching out to the C/C++ code from a hosting Java application, and so for the best performance, you will want a coarse interface, without a lot of calls back and forth between Java and the native opcodes. This may require some redesign of what might otherwise be the "natural" way of writing the C/C++ code, or you may just have to settle for less of a speed improvement. Regardless, for many types of algorithms – from cryptography to game AI to video format conversions – using C/C++ with the NDK will make your application perform much better, to the point where it can enable applications to be successful that would be entirely too slow if written solely in Java.

Porting

You may already have some C/C++ code, written for another environment, that you would like to use with Android. That might be for a desktop application. That might be for another mobile platform, such as iPhone or WebOS, where C/C++ is an option. That might be for mobile platform, such as Symbian, where C/C++ is the conventional solution, rather than some other language. Regardless, so long as that code is itself relatively platform-independent, it should be usable on Android.

This may significantly streamline your ability to support multiple platforms for your application, even if down-to-the-metal speed is not really
something you necessarily need. This may also allow you to reuse existing C/C++ code written by others, for image processing or scripting languages or anything else.

Skills

Even if you do not have much in the way of existing C/C++ code you are looking to bring over to Android, it may be that is where your skills and "comfort zone" lie. This will be particularly true for developers coming to Android from, say, Symbian, used to C/C++ as the primary development language. Or, perhaps you are used to Linux application development in C/C++ and are looking at Android as the closest thing to pure Linux on a phone. While you may still need to deal with Java for some things, putting more of your logic in C/C++ would make your transition to Android just a bit easier.

Knowing Your Limits

Developers love silver bullets. Developers are forevermore seeking The One True Approach to development that will be problem-free. Sisyphus would approve, of course, as development always involves tradeoffs. So while the NDK’s speed may make it tantalizing, it is not a solution for general Android application development, for several reasons, explored in this section.

Android APIs

The biggest issue with the NDK is that you have very limited access to Android itself. There are a few libraries bundled with Android that you can leverage, and a few other APIs offered specifically to the NDK, such as the ability to render OpenGL 3D graphics. But, generally speaking, the NDK has no access to the Android SDK, except by way of objects made available to it from the hosting application via JNI.
Beyond that, you cannot create an application only through the NDK. You cannot set up an Android manifest that only uses the NDK. You cannot create a subclass of Activity or Service solely through the NDK. And without access to the SDK, you cannot do all sorts of things that a regular Android application might do, from raising notifications to finding out when the WiFi connection is available.

In principle, you could get at all of those capabilities via some sort of JNI bridge. However, crossing the language barrier is fairly expensive, and it will not take much for that to swamp your hard-won performance gains.

As such, it is best to view the NDK as a way of speeding up particular pieces of an SDK application – game physics, audio processing, OCR, and the like. All of those are algorithms that need to run on Android devices with data obtained from Android, but otherwise are independent of Android itself.

Cross-Platform Compatibility

While C/C++ can be written for cross-platform use, often it is not.

Sometimes, the disparity is one of APIs. Any time you use an API from a platform (e.g., iPhone) or a library (e.g., Qt) not available on Android, you introduce an incompatibility. This means that while a lot of your code – measured in terms of lines – may be fine for Android, there may be enough platform-specific bits woven throughout it that you would have a significant rewrite ahead of you to make it truly cross-platform.

Android itself, though, has a compatibility issue, in terms of CPUs. Android mostly runs on ARM devices today, since Android's initial focus was on smartphones, and ARM-powered smartphones at that. However, the focus on ARM will continue to waver, particularly as Android moves into other devices where other CPU architectures are more prevalent, such as Atom for tablets or MIPS for set-top boxes. While your code may be written in a fashion that works on all those architectures, the binaries that code produces will be specific to one architecture. The NDK will, over time, give you additional assistance in managing that, so that your application can
simultaneously support multiple architectures. Right now, though, the NDK is for ARM, and so using the NDK may limit your application's ability to be used on some Android devices.

Maturity

The Dalvik VM is young. The NDK is younger still. It is barely a year old as of the time of this writing, since its debut in mid-2009. Fewer developers have been using the NDK than have been using the SDK. The combination of age and usage gives the NDK a fairly short track record, meaning that there may be more NDK problems than are presently known.

Available Expertise

If you are seeking outside assistance for your Android development efforts, there will be fewer people available to assist you with NDK development, compared to SDK development. The NDK is newer than the SDK, so many developers started with what was originally available. Many applications do not need the NDK, and so many developers will not have taken the time to learn how to use it. Furthermore, many Android developers may be far more fluent in Java than they are in C/C++, based on their own backgrounds, and so they would tend to stick with tools they are more comfortable with. To top it off, few books on Android development cover the NDK, though this is being incrementally improved, via books such as this one.

If you are looking for somebody with NDK experience, ask for it – do not assume that Android developers know the NDK nearly as well as they know the SDK.
CHAPTER 9

Setting Up the NDK
CHAPTER 10

Wrapping a Native Library
PART III – Scripting Languages
CHAPTER 11

The Role of Scripting Languages

A scripting language, for the purpose of this book, has two characteristics:

- It is interpreted from source and so does not require any sort of compilation step
- It cannot (presently) be used to create a full-fledged Android application without at least some form of custom Java-based stub, and probably much more than that

Of course, some of the alternative application frameworks from the preceding chapters come close to meeting these characteristics – tools like PhoneGap and Rhodes are interpreted but can create full Android applications.

In this part of the book, we will look at scripting languages on Android and what you can accomplish with them, despite any limitations inherent in their collective definition.

All Grown Up

Interpreted languages have been a part of the programming landscape for decades. The language most associated with the desktop computer revolution – BASIC – was originally an interpreted language. However, the advent of MS-DOS and the IBM PC (and clones) led developers in the
The Role of Scripting Languages

direction of C for "serious programming", for reasons of speed. While interpreted languages continued to evolve, they tended to be described as "scripting" languages, used to glue other applications together. Perl, Python, and the like were not considered "serious" contenders for application development.

The follow-on revolution, for the Internet, changed all of that. Most interactive Web sites were written as CGI scripts using these "toy" languages, Perl first and foremost. Even in environments where Perl was unpopular, such as Windows, Web applications were still written using scripting languages, such as VBScript in Active Server Pages (ASP). While some firms developed Web applications using C/C++, scripting languages ruled the roost. That remains to this day, where you are far more likely to find people writing Web applications in PHP or Ruby than you will find them writing in C or C++. The most likely compiled language for Web development – Java – is still technically an interpreted language, albeit not usually considered a scripting language.

Nowadays, writing major components of an application using a scripting language is not terribly surprising. While this is still most common with Web applications, you can find scripting languages used in the browser (Javascript), games (Lua), virtual worlds (Second Life language), and so on. Even though these languages execute more slowly than their C/C++ counterparts, they offer much greater flexibility, and faster CPUs make the performance of scripts less critical.

**Following the Script**

arguments for using a scripting language

**Your Expertise**
The Role of Scripting Languages

Your Users' Expertise

Crowd-Developing

(CollabCalc-style script libraries developed from hither and yon)

**Going Off-Script**

problems with scripting

Security

Performance

Cross-Platform Compatibility

Maturity...On Android

(double-edged sword – JRuby and BeanShell)
When it comes to scripting languages on Android, the first stop should always be the Android Scripting Environment. Led by Damon Kohler, this project is rather popular, both among hardcore Android developers and those people looking to automate a bit more of their Android experience.

The Role of ASE

What started as an experiment to get Python and Lua going on Android, back in late 2008, turned into a more serious endeavor in June 2009, when the Android Scripting Environment (ASE) was announced on the Google Open Source blog and the Google Code site for it was established. Since then, ASE has been a magnet for people interested in getting their favorite language working on Android or advancing its support.

On-Device Development

Historically, the primary role of ASE was as a tool to allow people to put together scripts, often written on the device itself, to take care of various chores. This appealed to developers who were looking for something lightweight compared to the Android SDK and Java. For those used to tinkering with scripts on other mobile Linux platforms (e.g., the Nokia N800 running Maemo), ASE promised a similar sort of capability.
Over time, ASE’s scope in this area has grown, and there are stated intentions to eventually support ASE scripts packaged as APK files, much like an Android application written in Java or any of the alternative frameworks described in this book.

Coordinator of Scripting Language Implementations

Beyond that, ASE is expanding its scope to help establish some standards for in-Android scripting languages, so their implementations can be used beyond ASE itself. ASE is not particularly designed to be used by other applications seeking to embed scripting capabilities, but appropriately-packaged ASE interpreters might be. In this respect, ASE will be taking on the role held by the Bean Scripting Framework (BSF) and JSR-223 in traditional Java development.

Getting Started with ASE

The Android Scripting Environment is a bit more difficult to install than is the average Android application, due to the various interpreters it uses and their respective sizes. That being said, none of the steps involved with getting ASE set up are terribly difficult, and most are just part of the application itself.

Installing ASE

At the time of this writing, ASE is not distributed via the Android Market. Instead, you can download it to your device off of the ASE Web site. Perhaps the easiest way to do that is to scan the QR code on the ASE home page using Barcode Scanner or a similar utility.

Installing Interpreters

When you first install ASE, the only available scripting language is for shell scripts, as that is built into Android itself. If you want to work with other interpreters, you will need to download those. That is why the base ASE
download is so small (~200KB) – most of the smarts are separate downloads, largely due to size.

To add interpreters, launch ASE from the launcher, then choose View > Interpreters from the option menu. You will be presented with the (presently short) list of installed interpreters:

![Figure 9. The initial list of installed ASE interpreters](image)

Then, to install additional interpreters, choose Add from the option menu. You will be given a roster of ASE-compatible interpreters to choose from:
Choose an interpreter from the list, and ASE will download and install the interpreter's component parts:
This may take one or several downloads, depending on the interpreter. When done, and after a few progress dialogs' worth of unpacking, the interpreter will appear in the list of interpreters:
Note that the interpreters will be installed on your device's "external storage" (typically some flavor of SD card), due to their size. You will find an `ase/` directory on that card with the interpreters and scripts.

### Running Supplied Scripts

Back on the Scripts activity (e.g., BACK button from the list of interpreters), you will be presented with a list of the available scripts. Initially, these will be ones that shipped with the interpreters, as examples for how to write ASE scripts in that language:
The Android Scripting Environment

Figure 13. The list of ASE scripts

Tapping on any of these scripts will run it, showing its terminal output along the way:
Writing ASE Scripts

While the scripts supplied with the interpreters are...entertaining, they only scratch the surface of what an ASE script can accomplish. Of course, to go beyond what is there, you will need to start writing some scripts.

Editing Options

Since scripts are stored on your SD card (or whatever the "external storage" is for your device), you can create scripts using some other computer – one with fancy things like "mice" and "ergonomic keyboards" – and transfer it over via USB, like you would transfer over an MP3 file. This eases typing, but it will make for an awkward development cycle, since your computer and the Android device cannot both have access to the SD card simultaneously. The mount/unmount process may get a bit annoying. On the other hand, this is a great way to transfer over a script you obtained from somebody else.
The Android Scripting Environment

Another option is to edit your scripts on the device. ASE has a built-in script editor designed for this purpose. Of course, the screen may be a bit small and the keyboard may be a bit...soft, but this is a great answer for small scripts.

To add a new script, from the Scripts activity, choose Add from the option menu. This will bring up a roster of available scripting languages:

![Add Script Language Selection Dialog](image)

Figure 15. The add-script language selection dialog

(the "Scan Barcode" option gives you an easy route to install a third-party script, one encoded in a QR code)

Tap the language you want, and you will be taken into the script editor:
The field at the top is for the script name, and the large text area at the bottom is for the script itself. A file extension and boilerplate code will be supplied for you automatically.

In fact, that boilerplate code is rather important, as you will see momentarily.

To edit an existing script, long-tap on the script in the list and choose Edit from the context menu.

To save your changes to a new or existing script, choose the Save option from the script editor option menu. You can also "Save & Run" to test the script immediately.
Calling Into Android

In the real world, Perl knows nothing about Android. Neither does Python, BeanShell, or most of the other scripting languages available for ASE. This would be rather limiting, as most of what you would want a script to do will have to deal with the device to some level: collect input, get a location, say some text using speech synthesis, dial the phone, etc.

Fortunately, ASE has a solution, one of those "so crazy, it just might work" sorts of solutions: ASE has a built-in RPC server. While implementing a server on a smartphone is not something one ordinarily does, it provides an ingenious bridge from the scripting language to the device itself.

Each scripting language is given a local object proxy that works with the RPC server. For example, here is a Python script that speaks the current time:

```
***Speak the time.***

__author__ = 'T.V. Raman <raman@google.com>'
__copyright__ = 'Copyright (c) 2009, Google Inc.'
__license__ = 'Apache License, Version 2.0'

import android
import time

droid = android.Android()
droid.speak(time.strftime('%I %M %p on %A, %B %d, %Y'))
```

Figure 17. The script editor, showing the say_time.py script
The import android and droid=android.Android() statements establish a connection between the Python interpreter and the ASE RPC server. From that point, the droid object is available for use to access Android capabilities – in this case, speaking a message.

Python does not strictly realize that it is accessing local functionality. It simply makes RPC calls, ones that just so happen to be fulfilled on the device rather than via some remote RPC server accessed over the Internet.

**Browsing the API**

Therefore, ASE effectively exposes an API to each of its scripting languages, via this RPC bridge. While the API is not huge, it accomplishes a lot and is ever-growing.

If you are editing scripts on the device, you can browse the API by choosing the API Browser option menu from the script editor. This brings up a list of available methods on your RPC proxy (e.g., droid) that you can call:
Tapping on any item in the list will "unfold" it to provide more details, such as the parameter list. Long-tapping on an item brings up a context menu where you can:

- insert a template call to the method into your script at the cursor position
- "prompt" you for the parameter values for the method, then insert the completed method call into your script

It is also possible to browse the API in a regular Web browser, if you are developing scripts off-device.

**Running ASE Scripts**

Scripts are only useful if you run them, of course. We have seen two options for running scripts: tapping on them in the scripts list, or choosing "Save & Run" from the script editor. Those are not your only options, however.
Background

If you long-tap on a script in the script list, you will see a context menu option to "Start in Background". As the name suggests, this kicks off the script in the background. Rather than seeing the terminal window for the script, the script just runs. A notification will appear in the status bar, with the ASE icon, indicating that the RPC server is in operation and that script(s) may be running.

Shortcuts

Rather than have to open up ASE every time, you can set up shortcuts on your home screen to run individual scripts. Just long-tap on the home screen background and choose Shortcuts from the context menu, then ASE from the available shortcuts. This brings up the scripts list, but this time, when you choose a script, you are presented with a menu of options for how to start it: in a terminal or in the background:

![Figure 19. Configuring an ASE shortcut](image)
At this point, a shortcut, with the ASE icon and the name of the script, will appear on your home screen. Tapping it runs the script.

Locale

<< TBD >>

IntentBuilders

<< TBD >>

**Potential Issues**

As the ASE Web site indicates, ASE is "alpha-quality". It is not without warts. How much those warts are an issue for you, in terms of crafting and running utility scripts, is up to you.

**Security...From Scripts**

ASE itself hold a long list of Android permissions, including:

- The ability to read your contact data
- The ability to call phone numbers and place SMS messages
- Access to your location
- Access to your received SMS/MMS messages
- Bluetooth access
- Internet access
- The ability to write to the SD card
- The ability to record audio and take pictures
- The ability to keep your device awake
• The ability to retrieve the list of running applications and restart other applications
• And so on

Hence, its scripts – via the RPC-based API – can perform all of those actions. For example, a script you download from a third party could read all your contacts and send that information to a spammer. Hence, you should only run scripts that you trust, since ASE effectively "wires open" many aspects of Android’s standard security protections.

Security...From Other Apps

<< TBD >>
CHAPTER 13

JVM Scripting Languages

Languages on Languages

caption of languages running on top of Java (vs. impl in machine opcode)

A Brief History of JVM Scripting

milestones in the development of JVM scripting: BSF, JSR-227(?)

Limitations

issues with JVM scripting on Android

Android SDK Limits

Not JavaSE

classloading/reflection not quite what you'd expect
Wrong Bytecode

Age

BeanShell

The ASE Interpreter Interface

BeanShell

JRuby

Rhino

Other JVM Scripting Languages
CHAPTER 14

Binary Executable Engines
CHAPTER 15

Scripting via the NDK
This book's first part is focused on comparing different ways to build an Android application without using Java.

This, of course, makes Java sad.

Hence, in the interests of fairness, let us take a peek at a port of the Checklist sample application to the conventional Java-based Android development model. You can find the source code for this in the Java/Checklist directory of the book's source code repository.

Note that this Java port takes advantage of some capabilities available to Java that are not necessarily available to the alternative frameworks. For example, the Java port uses option menus and context menus instead of extra on-screen buttons.

This appendix describes what the Java port looks like. It does not attempt to cover the concepts behind this implementation. There are plenty of other Android books – some published by this publisher – that cover this topic.

The Layout File

First, we need a layout file that describes the Checklist user interface. This needs to support:
Checklist in Java

- the checklist itself
- the field and button to add a new item to the checklist
- the "total" and "remaining" counts

The rest, as you will see, will be handled in other ways.

Here is the res/layout/main.xml file from the Java/Checklist project:

```xml
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent" >
    <Button android:id="@+id/add"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/add"
        android:layout_alignParentTop="true"
        android:layout_alignParentRight="true"
        android:layout_marginTop="8dip"
        android:layout_marginRight="8dip"
        android:layout_marginLeft="8dip"
        android:onClick="onAdd" />
    <EditText android:id="@+id/item"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_toLeftOf="@id/add"
        android:layout_alignBaseline="@id/add"
        android:layout_marginLeft="8dip"
        android:hint="@string/item_hint" />
    <TextView android:id="@+id/total_label"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_below="@id/add"
        android:layout_marginTop="8dip"
        android:layout_marginLeft="8dip"
        android:text="@string/total" />
    <TextView android:id="@+id/total"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_toRightOf="@id/total_label"
        android:layout_alignBaseline="@id/total_label" />
    <TextView android:id="@+id/remaining"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignBaseline="@id/total_label" />
</RelativeLayout>
```
This layout is fairly conventional (and, hence, bland), using a Button and EditText for the new item, a set of TextView widgets for the counts (and their labels), and a ListView for the checklist itself. Note that the ListView is set up with android:choiceMode="multiple", which means the ListView will, indeed, be a checklist.

**The Java Source**

The Java source code for the Checklist port comprises two public classes: Checklist and DatabaseHelper. The former is an Activity (for the user interface); the latter helps mediate our connection to a SQLite database used for storing the the checklist itself.

**DatabaseHelper**

DatabaseHelper extends SQLiteOpenHelper and teaches Android how to create our database schema:

```java
package com.commonsware.android.abj;
```
Right now, there is only the one schema version, so the `onUpgrade()` method will not be used. This is good, considering that its current implementation wipes out the user's data.

The Activity

<< TBD >>

The List and Entries

<< TBD >>

The Menus

<< TBD >>