IN THIS CHAPTER

- What makes Flutter great
- Alternatives to Flutter
- Some boring terminology :-(

Chapter 1
What Is Flutter?

Several years ago, I won a smartphone in a raffle at an app developer conference. What a joy it was to win something! The experience made me feel that the entire cosmos favored me. Every time I used that phone, I felt like a big shot.

Eventually, the phone's battery became so weak that I had to charge it every hour. I didn't realize that the phone was still under warranty, so I tried to replace the phone's battery myself. I bought a new battery from an online vendor. The instructions told me how to take the case apart, unhook the circuit connections, and remove the old battery from its cradle.

Everything went nicely until the part about removing the old battery. The instructions said to pull on a little tab, but I couldn't find a tab. So, I tried for several minutes to get a grip on the battery.

The battery wasn't budging, so I found a little screwdriver and tried to pry the battery from its tight surroundings. That's when I heard a pop, smelled smoke, and realized that the phone's battery had caught fire.

Fast-forward to the next afternoon. I was wandering past an electronics shop, so I went in and asked whether the shopkeeper might be able to fix my phone. "Yes," he said. "Bring it in the next time you're in the neighborhood. I can fix any phone."

You should have seen the look on the shopkeeper's face when, later that day, I brought in the charred, bent-up, barely recognizable phone. I would have included a picture in this book but, alas, I couldn't take a picture. I had no phone.

I still remember this phone battery story from beginning to end. I remember the joy of winning a free phone, the shock of seeing it go up in flames, and the look of horror on the shopkeeper's face. But my most powerful memory comes from the moment I opened the phone's case: Inside that little case, I saw enough circuitry to make me dizzy. Having done some electrical work in my own home, I'd handled thick 10-gauge wires and hefty 220-volt connectors. I had replaced desktop computers' sound cards, laptop computers' hard drives, and the SSD inside a tightly packed MacBook Air. But this smartphone was amazing. The circuit board looked like a microchip in its own right. The connectors were so tiny that I wondered how signals could reliably squeeze through them.

No doubt about it: Mobile phones are complicated beasts. So how do they work? What makes them tick? What's going on inside each of those remarkable gadgets?

Hardware and Software (Things You May Already Know)

A mobile phone is really a small computer. And, like any computer, a mobile phone operates on several layers. Figure 1-1 shows you a few of those layers.

Hardware is the stuff you can touch. It's the bottom layer of the diagram in Figure 1-1. Hardware consists of items like circuitry, memory, and the battery.

Electrical signals that travel along the hardware's circuits make the hardware do what you want it to do. These signals encode instructions. Taken as a whole, these instructions are called *software*.

When people create software, they don't describe each electrical signal that travels through the hardware's circuitry. Instead, people write *source code* — instructions that look something like English-language instructions. One source code instruction can be shorthand for hundreds or thousands of electrical signals.

A collection of source code instructions that perform a particular task (word processing, web browsing, managing a smart thermostat, or whatever) is called a *program*. A person who writes these instructions is a *programmer* or — a fancier-sounding term — a *developer*. The person who runs a program on their own device is a *user*.

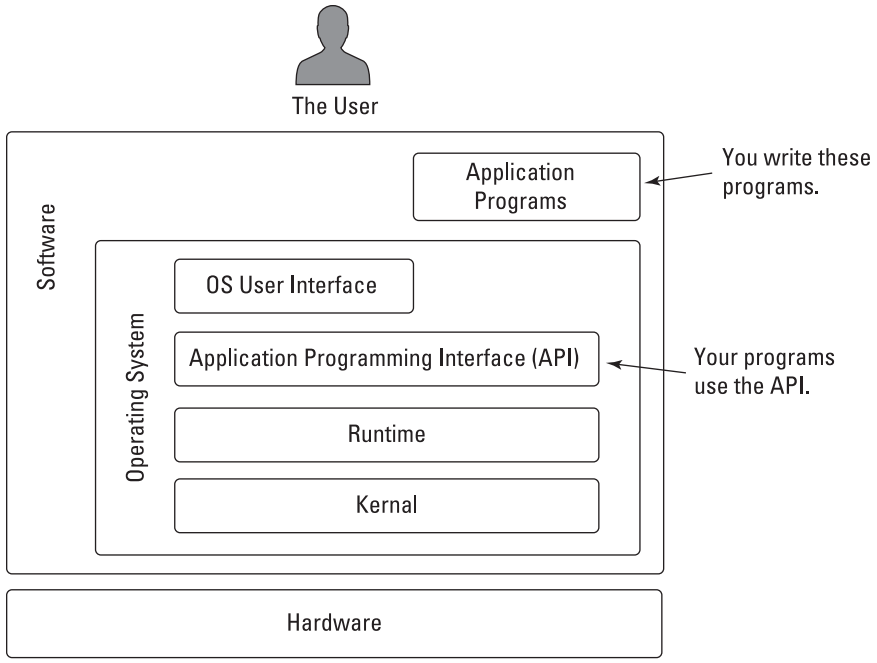

FIGURE 1-1: A conceptual view of your mobile phone.

Just as people communicate using many spoken languages, programmers write source code using many *programming languages*. If you create iPhone apps, you probably write code in either the Swift language or the Objective-C language. If you create Android apps, you're likely to write code in either Kotlin or Java.

When you create a Flutter app, you write code in the Dart programming language. Here's a complete Dart language program:

```
main() => print('Hello');
```

This program displays the word *Hello* on the screen. It's not very useful, but please be patient. This is only Chapter 1!

Figure 1-1 distinguishes between two kinds of software:

- ***Operating system*** **(OS) software runs whenever the device is turned on.**

  OS software manages the device and provides ways for the user to interact with the device. Devices made by Apple, such as iPhones and iPads, run the *iOS* operating system. Android phones and tablets run the *Android* operating system (of course).

- ***Application programs*** **do the work that users want done.**

  Apps to make phone calls, apps to read email, calendar apps, web browsers, and games are examples of application programs. As a Flutter developer, your job is to create application programs.

By one estimate, the popular operating system named Linux consists of nearly 28 million instructions. No one can deal with that much code, so operating systems are divided into layers of their own. Figure 1-1 shows only four of a typical operating system's many layers:

- **A *kernel* performs the operating system's most fundamental tasks.**

  The kernel schedules apps to be run, manages a device's memory and files, provides access to input and output, and does many other essential tasks.

- **A *runtime* is a bunch of code that does extra work in the background while your application program runs.**

  Runtimes come in many shapes and sizes. A runtime for the C programming language consists of a relatively small amount of code. In contrast, a Java language runtime (a *Java Virtual Machine,* or *JVM*) is a big piece of software with lots of moving parts.

  When you run an iOS app, the app uses the *Objective-C runtime*. When you run an Android app, that app uses the *Android runtime*, also known as *ART*.

- **An *application programming interface* (API) is a bunch of code that app developers use over and over again.**

  For example, Android's API has something named `toUpperCase`. If you apply `toUpperCase` to `"Flutter For Dummies"`, you get `"FLUTTER FOR DUMMIES"`. You don't have to write your own code to change each of the letters. Android's API provides this functionality for you. All you have to do is tell Android's API to apply its `toUpperCase` feature, and then you're all set.

  Here's some useful terminology: Rather than tell an API to "apply its `toUpperCase` feature," you *call* `toUpperCase`. This use of the word *call* dates back to the FORTRAN programming language of the 1950s.

  Operating systems haven't cornered the market on APIs. All kinds of software come with APIs. Flutter and Dart have their own APIs.

  Dart's API has general-purpose things, like `toUpperCase`, `isAtSameMomentAs`, and a bunch of others. Flutter's API has features that apply to visually oriented apps. For example, when you want to display a box where the user can type text, you don't have to describe every aspect of the box's appearance and behavior. Instead, you can call the API's `TextField` constructor and have Flutter do the hard work for you.

TECHNICAL STUFF

I sometimes refer to an API as a *library*. You borrow books from a public library, and you borrow existing code from the Dart and Flutter APIs.

In the Dart programming terminology, the word *library* has a slightly different meaning. You don't have to worry about that yet.

Throughout most of this book, I describe pieces of the Dart and Flutter APIs and then the way you use those pieces to create Flutter programs.

REMEMBER

A typical API has thousands of pieces. No one memorizes all of them. When you want to add an image to your app, you open Flutter's documentation and search for the word *`Image`*. The documentation's `Image` page tells you how to display an image, how to size an image, how to tile an image, and how to do all kinds of other good stuff.

- **The *OS user interface* is the area that includes the home screen, the launch icons, a file explorer, and any other stuff users see when they're not working with a particular application program.**

  On your laptop computer, you probably have a *desktop* instead of a home screen. One way or another, the OS presents options to help users launch application programs and perform other maintenance tasks. These options are part of the OS user interface.

Each layer in Figure 1-1 contains a collection of related components. This helps programmers focus on the components that concern them the most — for example:

- **The API has code to help developers write application programs.**

  A developer who's creating an online purchasing app looks for components in the API.

- **The Runtime layer has code to run programs efficiently.**

  To make everyone's code run faster, engineers at Apple make improvements to the iOS Runtime layer.

In addition to separating parts of the code from one another, the layers form organized paths of communication among parts of the system. In general, a layer's code communicates only with the layers immediately above and below it. For example, a user taps a button belonging to a weather app. The app responds by calling on functionality provided by the API. Communication works its way down the diagram in Figure 1-1 until it reaches the hardware, which responds by changing the pixels on the device's screen. A user never communicates directly with the API, and application programs have no direct access to the operating system's kernel.

CODE YOU CAN USE

During the early 1980s, my cousin-in-law Chris worked for a computer software firm. The firm wrote code for word processing machines. (At the time, if you wanted to compose documents without a typewriter, you bought a "computer" that did nothing but word processing.) Chris complained about being asked to write the same old code over and over again. "First, I write a search-and-replace program. Then I write a spell checker. Then I write another search-and-replace program. Then a different kind of spell checker. And then a better search-and-replace program."

How did Chris manage to stay interested in his work? And how did Chris's employer manage to stay in business? Every few months, Chris had to reinvent the wheel — toss out the old search-and-replace program and write a new program from scratch. That's inefficient. What's worse, it's boring.

For years, computer professionals were seeking the holy grail — a way to write software so that it's easy to reuse. Don't write and rewrite your search-and-replace code. Just break the task into tiny pieces. One piece of code searches for a single character, another piece looks for blank spaces, and a third piece substitutes one letter for another. When you have all the pieces, just assemble these pieces to form a search-and-replace program. Later on, when you think of a new feature for your word processing software, you reassemble the pieces in a slightly different way. It's sensible, it's cost-efficient, and it's much more fun.

The late 1980s saw several advances in software development, and by the early 1990s, many large programming projects were being written from prefabricated components. For a particular project or a particular programming language, these prefab components formed a library of reusable code. This was the birth of the modern API.

When you create a Flutter app, you use the Dart programming language. Dart and Flutter have separate APIs:

- **Dart's API deals with the tasks that every programming language should be able to do, no matter what programmers want to do with that language.**

  For example, Dart's API helps programmers round a number, trim a string of characters, describe a time interval, reverse a list, and so on.

- **Flutter's API deals with the presentation of components and images on a device's screen.**

  One part of Flutter's API deals with buttons, text fields, check boxes, and the like. Another part handles a user's gestures. Yet another covers animation.

Every Dart program, even the simplest one, calls on code in the Dart API, and every Flutter app calls on both the Dart and Flutter APIs. These APIs are both useful and formidable. They're useful because of all the things you can do with the API code. They're formidable because both APIs are extensive. No one memorizes all the features made available by the Dart and Flutter APIs. Programmers remember the features that they use often and look up the features that they need in a pinch. They look up these features on a website called the *Flutter API reference documentation*.

The API documentation (see `https://api.flutter.dev`) describes the features in both the Dart and Flutter APIs. As a Flutter developer, you consult this API documentation on a daily basis. You can bookmark the website and revisit the site whenever you need to look up something.

Where Does Flutter Fit In?

The heart of Flutter is an API for creating apps. Most Flutter apps run on mobile devices, but Flutter apps can run on laptop and desktop computers, too. Flutter certainly wasn't the first API for mobile devices, so why should anyone consider using Flutter to create apps?

Cross-platform development

My favorite burger joint advertised a new mobile ordering app. I needed the app so that I could quickly hop off a commuter train, grab a burger, and run to a nearby tech meeting. I did this several times each month. But I had a problem: The app ran only on iPhones, and I had an Android phone.

Behind the scenes, the burger joint's app developers were hard at work converting their iPhone app to an Android app. This was no minor task, because Android's API doesn't recognize the same commands as iPhone's API. Going from one API to the other isn't straightforward. It's not a matter of making routine code changes. To convert from one kind of phone to another, developers rewrite thousands (and maybe even millions) of lines of code. The process is time-consuming and expensive.

So I waited and waited for the restaurant to have an Android app. I was so desperate for a delicious cheeseburger that I finally broke down and bought a second phone. But that turned out to be a bad idea. As soon as my new iPhone arrived, the burger place released its shiny, new Android app.

The whole story comes down to things called platforms. People throw around the word *platform* as if the word means everything and nothing. But to my mind, a *platform* is a particular operating system along with the hardware the OS runs on.

What makes the Android platform different from its iOS counterpart? To create radio buttons in Android's API, you write code of the following kind:

```
<RadioGroup>

    <RadioButton
        android:id="@+id/radioButton1"
        android:text="Red"
        android:onClick="onRadioButtonClicked"/>

    <RadioButton
        android:id="@+id/radioButton2"
        android:text="Yellow"
        android:onClick="onRadioButtonClicked"/>

    <RadioButton
        android:id="@+id/radioButton3"
        android:text="Green"
        android:onClick="onRadioButtonClicked"/>

</RadioGroup>
```

Try converting that code to work on an iPhone. The iOS API doesn't have radio buttons, so, to adapt an Android app with radio buttons for iOS, you write code to make things that look like radio buttons. You also code rules for the radio buttons to follow — rules like "only one button at a time can be selected." If you don't want to create radio buttons from scratch, you can replace Android's radio buttons with an iOS picker component, a thing that looks like an old automobile odometer. One way or another, replacing an app's components takes time and costs money.

Some companies give up and create apps for only one platform — iPhone or Android. Other companies hire two teams of programmers — one for iPhone development and another for Android development. Still other companies have one team of programmers that work on both versions of the code. For the companies' managers, the problem is exasperating. Why spend nearly twice the money and create two apps that do almost the same things?

The developer community has names for this ugly situation:

» Software written for one platform isn't *compatible* with other platforms.

» The mobile phone arena suffers from *fragmentation*: The market is divided between two different operating systems, and the Android half is divided among many vendors' phones.

A program that makes direct use of either the Android or iOS API is called *native code*, and native code written for Android can't run on an iOS device. In the same way, native code written for iOS is meaningless to an Android device. What's a developer to do?

A framework is a second-level API. What the heck does that mean? A *framework* is an API that serves as an intermediary between the developer and some other API. If direct use of the Android or iOS API is problematic, you switch to a framework's API. The framework's API deals head-on with Android's and iOS's problems.

Frameworks like Flutter offer an alternative to native app development. When you write a Flutter program, you don't write code specifically for Android or iOS. Instead, you write code that can be translated into either system's API calls. Here's how you create radio buttons in the Flutter framework:

```
Radio(
  value: TrafficLight.Red,
  groupValue: _trafficLightValue,
  onChanged: _updateTrafficLight,
),
Radio(
  value: TrafficLight.Yellow,
  groupValue: _trafficLightValue,
  onChanged: _updateTrafficLight,
),
Radio(
  value: TrafficLight.Green,
  groupValue: _trafficLightValue,
  onChanged: _updateTrafficLight,
)
```

Your computer translates code of this kind into either Android API calls or iOS API calls — or both. That's cool!

A quick-and-easy development cycle

You may have heard stories about the early days of computer programming. I worked for a few summers at the University of Pennsylvania Physics department. I wrote FORTRAN programs and typed them myself on a big deck of punch cards. A 600-line program weighed about 1400 grams (roughly 3 pounds).

A BRIEF HISTORY

130,000 years ago: Humans first walk the earth.

10,000 years ago: Humans begin farming.

1752: Ben Franklin discovers electricity.

1760: The Industrial Revolution begins.

March 10, 1876: Alexander Graham Bell makes the first telephone call.

April 3, 1973: Martin Cooper makes the first mobile phone call.

August 16, 1994: BellSouth Cellular releases IBM Simon — the first smartphone.

June 29, 2007: Apple releases the first iPhone.

November 5, 2007: Google releases the first public beta of Android.

Both the iOS and Android are native development technologies. With native development, the programmer makes calls directly to the system's API.

December 2007: Articles and blog posts about fragmentation in mobile phone technologies start appearing in large numbers.

March 13, 2009: Nitobi Software introduces a framework that uses HTML, CSS, and JavaScript to create mobile phone apps.

October 4, 2011: Adobe acquires Nitobi, rebrands its framework with the name PhoneGap, and spins off an open-source version that eventually becomes Apache Cordova.

Cordova and its cousins are hybrid app development frameworks. With *hybrid app development*, an app runs in a window that's essentially a web browser. Because web browser technology is standard across all platforms, a hybrid app can run on both Android and iOS devices, or even on a desktop computer.

What's "hybrid" about hybrid apps? The code to display text and images in a web browser doesn't vary much from one environment to another, so a browser page on an iPhone looks more or less like the same page on an Android phone. But communicating with hardware devices, such as the GPS receiver and vibration motor, is another story entirely.

Web pages aren't designed to talk directly to a device's hardware. In fact, you don't want to visit `awfulwebsite.com` and have the site's code quietly take pictures with your laptop's built-in camera. To make a hybrid app interact with hardware, you have to backpedal and make calls to the iPhone's API, the Android API, or whatever other API you can use. That's why frameworks like Apache Cordova have *plug-ins* — additional programs whose code is specific to either iOS or Android. The bottom line is, a typical hybrid app does some of its work in a web browser and the rest of its work with native API calls.

What's the downside with hybrid apps? Frameworks like Apache Cordova are like foreign language interpreters: While the app runs, the device must constantly translate web browser instructions into native code instructions. When you talk through an interpreter, the conversation can become sluggish. Hybrid apps aren't always as responsive as native apps. In addition, hybrid apps can't do all the things that native apps can do. It's the same when you talk through a foreign language interpreter. You can say most of the things you want to say, but some ideas simply can't be translated.

Returning to the history lesson . . .

Summer 2013: A hackathon for Facebook employees gives birth to *React Native* — a cross-platform framework based on the React.js JavaScript framework.

February 24, 2016: Microsoft acquires Xamarin — a cross-platform mobile development framework based indirectly on Microsoft's own .NET framework.

With a *cross-platform framework*, a programmer writes one program that targets neither iOS nor Android. When the programmer says, "Test this code," the framework translates the whole program into native code for either Android or iOS, whichever platform the programmer chooses. When the program is ready for public distribution, the framework translates it into two different native apps — one for iOS and the other for Android.

But why stop there? If you can translate code into both iOS and Android apps, you can translate the code into web pages and desktop apps. A developer can create one piece of code and have it run on all kinds of phones, tablets, PCs, Macs, watches, toasters, or whatever.

This brings me to the subject of my book:

December 4, 2018: Google announces Flutter 1.0 for cross-platform development.

(continued)

(continued)

Flutter differs from Xamarin and React Native in some significant ways. First and foremost, Xamarin isn't entirely free. Using Xamarin for professional projects costs between $300 and $1900 a year, depending on the size and scope of the projects under development.

In addition, Flutter's way of displaying components is different from the React Native and Xamarin way. When you run a React Native app on an iPhone, the app calls on the iOS API to create iOS buttons, text fields, and other visual components. The same is true for Android development. React Native gets the Android API to display Android-specific components. Components created by the iOS and Android APIs don't look alike. The two APIs use different shapes, different color palettes, and different navigation schemes. The differences can lead to unexpected results and can occasionally sabotage the whole cross-platform development effort.

Flutter doesn't call on the iOS or Android APIs to display an app's components. Instead, Flutter specifies all the tiny pixels required to draw a button or a text field and calls on the iOS or Android API to paint those pixels. If you want an app to look the same on both iOS and Android devices, Flutter is your natural choice.

What if you want your app to have that special, iPhone look when it runs on iOS devices? Can you do that with Flutter? Of course, you can. (I wouldn't pose the question if the answer were "No.") The Flutter framework has two special libraries — one for Android and another for iOS. Flutter's Material Design library draws things that look like Android components, and Flutter's Cupertino library makes objects look like iOS components. This book emphasizes the Material library, but almost everything in it has a Cupertino counterpart.

I end this sidebar with one more historical nougat:

May 10, 2239: Technology historian Alice Touge publishes a paper on the origin of the word *phone,* which is Latin for "sound." She explains that 23rd century phones came from devices whose original purpose was solely to transmit sound. This surprising fact goes viral on all the direct-to-mind streaming outlets.

I'd carry my program from the punch card machine to the computer operator's counter, where a permanently surly operator would tell me about the unusually long job-turnaround time.

Four hours later, I'd get back an inch-thick pile of paper with an error message somewhere in the middle of it. I'd go back to the punch card machine, make another card with an added comma in the 23rd column, and do the whole business again.

There's no doubt about it — a long and arduous development cycle hinders productivity. These days, shaving a few seconds off the turnaround time can make a huge difference.

Here's what happens when you create an app for mobile devices:

1. You write some code, or you modify some existing code.


REMEMBER

   You don't write Android or iOS code on a phone of any kind. Phones aren't powerful enough for all the editing and other stuff you need to do. Instead, you create an app's code on a laptop or desktop computer. This laptop or desktop computer is called your *development computer*.

2. You issue a command for your development computer to build the code.

   Building the code takes place in several stages, one of which is called *compiling*. *Compiling* means automatically translating your program from the source code you wrote to detailed object code instructions. Think of object code as a bunch of zeros and ones. It's very detailed and extremely unintuitive. Humans hardly ever read or write object code but, at the heart of things, processors respond only to object code instructions.


CROSS REFERENCE

   For a detailed look at compiling code, see this section's "What is a compiler?" sidebar.

   In addition to the translation step, the build process connects the program you wrote with additional code that your program needs in order to run. For example, if your program accesses the Internet, the build process integrates your code with existing network code.

   What happens next?

3. The development computer *deploys* your code to a target device.

   This so-called "device" may be a real phone connected to your computer or a picture of a phone on your computer's screen. One way or another, your program starts running.

4. You press buttons, type text, and otherwise test your app to find out whether it's doing the things you want it to do.

   Of course, it's not doing all those things. So you return to Step 1 and keep trying.

Steps 2 and 3 can be painfully slow. For some simple iPhone and Android apps, I've watched for several minutes as my computer prepares code for the program's next run. This sluggishness reduces my productivity considerably.

But along with Flutter comes some good news. Flutter uses the Dart programming language, and Dart comes with these two (count 'em — two) compilers:

- **Ahead-of-time (AOT) compiler**

  With an *AOT compiler*, your development computer translates an entire program and makes the translated code available for devices to run. No further translation takes place when the devices run your program. Each target device devotes its processing power to the efficient running of your code.

  An app running on AOT-compiled code runs smoothly and efficiently.

- **Just-in-time (JIT) compiler**

  With a *JIT compiler*, your development computer translates enough code to start the app running. It feeds this code to a test device and continues translating while the test device runs the app. If the developer presses a button on the test device's screen, the JIT compiler hurries to translate that button's code.

  An app running on a JIT compiler may appear to be sluggish because the compiler translates code while the app runs. But using a JIT compiler is a great way to test an app.

Here's what happens when you develop a Flutter app:

1. You write some code.
2. You issue a command for your development computer to build the code.

   The first time around, building code can take some time.
3. The development computer deploys your code to a target device.

   Again, you face a noticeable time lag.
4. In testing your code, you find out that it's not doing all the things you want it to do.
5. You modify your existing code, and then . . .
6. You issue a command for your development computer to rebuild the code.

   Here's where Flutter's magic happens. Dart's JIT compiler recompiles only the part of the app that you've modified and sends the change straight to the target device. The modified code starts running in a fraction of a second. You save hours of time every day because you're not waiting for code changes to take effect.

REMEMBER

WHAT IS A COMPILER?

You're a human being. (Sure, every rule has exceptions. But if you're reading this book, you're probably human.) Anyway, humans can write and comprehend the following Flutter source code:

```
import 'package:flutter/widgets.dart';

main() => runApp(SizedBox`());
```

When you paraphrase the source code in English, here's what you get:

Get some code (code from the Flutter API) named `widgets.dart`.

Run an application whose only component is a box widget.

If you don't see the similarities between the Flutter code and its English equivalent, don't worry. You're reading *Flutter For Dummies*, and, like most human beings, you can learn to read and write the Flutter code. In case you're wondering, this source code contains the world's simplest and most useless Flutter app. When the app runs, you see a completely black screen. It's not what you'd call a "killer app."

Source code is nice, but source code isn't for everyone and everything. The processors in computers and mobile devices aren't human beings. Processors don't follow source code instructions. Instead, they follow cryptic instructions of the following kind:

```
1100100 1100101 1111000 00001010 00110000 00110011 00110101 00000000 10000100
```

These zeros and ones are, in fact, the first few words in an Android phone's version of the Black Screen app's code. Here's the Black Screen app after a processor interprets the zeros and ones:

```
.class public com/allmycode/dexperiment/MainActivity
.super io/flutter/embedding/android/FlutterActivity
.source MainActivity.java


.method public <init>()V
.limit registers 1
; this: v0 (Lcom/allmycode/dexperiment/MainActivity;)
.line 8
   invoke-direct    {v0},io/flutter/embedding/android/FlutterActivity/<init>
   ; <init>()V
   return-void
.end method
```

(continued)

(continued)

```
.method public configureFlutterEngine(Lio/flutter/embedding/engine/
   FlutterEngine;)V
.limit registers 2
; this: v0 (Lcom/allmycode/dexperiment/MainActivity;)
; parameter[0] : v1 (Lio/flutter/embedding/engine/FlutterEngine;)
.line 11
   invoke-static   {v1},io/flutter/plugins/GeneratedPluginRegistrant/
   registerWith
   ; registerWith(Lio/flutter/embedding/engine/FlutterEngine;)V
.line 12
   return-void
.end method
```

What a mess! Humans don't want to read or write instructions of this kind. These instructions aren't Dart source code instructions. They're *Dalvik bytecode* instructions. When you write a Flutter program, you write Dart source code instructions. If you test your program on an Android device, your development computer translates the source code into bytecode. If you test your program on an iPhone, the computer translates your source code into something that's even more obscure than bytecode.

The tool that performs the translation is a compiler. The *compiler* takes code that you can write and understand and translates it into code that a processor has a fighting chance of carrying out.

You might put your source code in a file named `main.dart`. To run your app on Android devices, the compiler creates other files named `MainActivity.dex` and `app.apk`. Normally, you don't bother looking at these compiled files. You can't even examine `.dex` files or `.apk` files with an ordinary editor. If you try to open `MainActivity.dex` with Notepad, TextEdit, or even Microsoft Word, you'll see nothing but dots, squiggles, and other gobbledygook.

No one (except for a few crazy programmers in some isolated labs in faraway places) writes Dalvik bytecode or any other kind of code that processors actually understand. When you ask your development computer to run your code, the computer uses its own software (a compiler) to create processor-friendly instructions. The only reason to look at the bytecode in this sidebar is to understand what a hard worker your development computer is.

Flutter gives you two ways to apply changes to a running app:

- With *hot restart*, the app begins its run anew, removing any data that you've entered during the most recent test, displaying the app as if you're running it for the first time.

- With *hot reload*, the app takes up from where it left off, with the data you last entered intact, if possible. The only changes are the ones dictated by your modifications to the code.

Flutter's hot restart and hot reload are both blazingly fast. They turn the app development cycle into a pleasure rather than a chore.

CROSS REFERENCE

Chapter 2 tells you more about building, testing, and rerunning apps.

A great way to think about app development

The language you speak influences the way you think. If you don't believe me, look up the Sapir-Whorf hypothesis. You're bound to find it on your favorite linguistics website.

Spoken languages are neither good nor bad, but programming languages can have good qualities and bad qualities. Most hybrid apps are written in the JavaScript programming language. Yes, JavaScript is one of the world's most widely used languages. But, no, JavaScript doesn't encourage good software design. It's easy to write confusing code in JavaScript because its rules are quite permissive. You can write sloppy JavaScript code, and the code runs just fine. That is, it runs fine until someone enters unexpected input. When that happens, you have trouble figuring out how your code was working in the first place. Even when you're not busy fixing errors, adding new features to JavaScript code can be difficult and frustrating. JavaScript aficionados will argue with every word in this paragraph but, one way or another, JavaScript has its downsides.

Apple's iOS platform uses the Swift and Objective-C languages, whereas Android uses Kotlin and Java. Objective-C dates back to the early 1980s and, like me, it's showing its age. The other three languages fare pretty well on the scale of good language features, but none of them is as straightforward and intuitive as Dart.

On top of that, both iOS and Android divide an app's code into two separate parts:

- **Layout:** How the app looks.
- **Logic:** The sequence of instructions that the app performs.

The Android radio button example in this chapter's earlier section "Cross-platform development" is neither Kotlin nor Java code. It's XML code (a term that I don't bother to define here). It has a different format and lives in a different file from the code that responds to radio button choices.

In my Android books, I argue that separating layout from logic is a good thing. It puts distinct aspects of an app into different parts of the code. Developers can maintain each part independently. For an Android developer, that's a good thing.

But this isn't an Android book. It's a Flutter book. So, in this book, I claim that separating layout from logic is *not* optimal. Here's why:

You may have heard the all-encompassing mantra of Flutter app development:

> *In Flutter, almost everything is a widget.*

And what is a widget? In a mobile app, every button is one of the app's widgets. Every text field is a widget. The app itself is a widget. The positioning of buttons and text fields is a widget. The animating of objects from one part of the screen to another is a widget. When you create a Flutter app, you put widgets inside of other widgets, which in turn are inside even more widgets. Listing 1-1 has some fake code that illustrates the point:

LISTING 1-1: **Like a Wheel Within a Wheel**

```
// Don't fall for my trickery. This isn't real Flutter code!
Application(
  Background(
    CenterWhateverIsInsideThis(
      Button(
        onPressed: print("I've been clicked."),
        Padding(
          Text(
            "Click Me"
          ),
        ),
      ),
    ),
  ),
)
```

Listing 1-1 has a `Text` widget inside of a `Padding` widget, which is inside of a `Button` widget inside a `CenterWhateverIsInsideThis` widget. That `CenterWhateverIsInsideThis` widget is inside a `Background` widget, which is inside an `Application` widget. When I created Listing 1-1, I modeled it after real Flutter code. The real Flutter code creates the app shown in Figure 1-2. When the user presses the `Button` in Figure 1-2, the words *I've been clicked* appear.

FIGURE 1-2: An app with a button.

ON THE WEB

To see the real code that inspired this chapter's fake code, visit `www.allmycode.com/flutter` and look for the big download link. The real code is in a file named `app0101`.

Compare Figures 1-2 and 1-3. Figure 1-2 shows the app as the user sees it. Figure 1-3 shows the same app as the Flutter developer codes it.

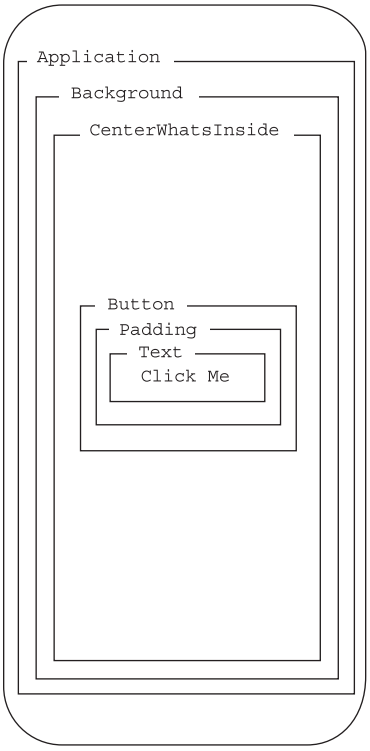

FIGURE 1-3: Widgets within widgets.

If you're not already a Flutter developer, the word *widget* might suggest a visible component, such as a button, a slider, an icon, or some other such thing. But in Flutter, things that aren't really visible are also widgets. For example, in Listing 1-1, `CenterWhateverIsInsideThis` is a widget. Having layout features like `CenterWhateverIsInsideThis` be widgets is a powerful idea. It means that Flutter developers can focus their attention on one overarching task — stuffing widgets inside other widgets. Flutter has a certain simplicity and elegance that other app development frameworks don't have.


TIP

Flutter has no built-in widget named `CenterWhateverIsInsideThis`. But don't be disappointed. Flutter's `Center` widget does what my fictitious `CenterWhateverIsInsideThis` widget is supposed to do.

Enough New Terminology! What's Next?

You may have read this chapter from start to finish but not one word in the chapter prompted you to touch your computer keyboard. What a shame! If you'll read the next chapter, I can rectify that awful omission.