Dedicated to Giorgia, my friends and my family.

Contents

1 | Welcome

1.1 Introduction

Thank you for having put your faith on this book. If you want to learn how to use a powerful tool that allows developers to quickly create native applications with top performances, you've chosen the right book. Nowadays companies tend to consider cross-platform solutions in their development stack mainly for three reasons:

1. **Faster development**: working on a single codebase;
2. **Lower costs**: maintaining a single project instead of many (N projects for N platforms);
3. **Consistency**: the same UI and functionalities on any platform.

All those advantages are valid regardless the framework being used. However, for a complete overview, there's the need to also consider the other side of the coin because a cross-platform approach also has some drawbacks:

1. **Lower performances**: a native app can be slightly faster thanks to the direct contact with the device. A cross-platform framework might produce a slower application due to a necessary *bridge* required to communicate with the underlying OS;
2. **Slower releases**: when Google or Apple announce a major update for their OS, the maintainers of the cross-platform solution could have the need to release an update to enable the latest features. The developers must wait for an update of the framework, which might slow down the work.

Every framework adopts different strategies to maximize the benefits and minimize or get rid of the drawbacks. The perfect product doesn't exist, and very likely we will never have one, but there are some high quality frameworks you've probably already heard:

- **Flutter**. Created by Google, it uses Dart;

- **React Native**. Created by Facebook, it is based on javascript;
- **Xamarin**. Created by Microsoft, it uses the C#;
- **Firemonkey**. Created by Embarcadero, it uses Delphi.

During the reading of the book you will see how Google tries to make the cross-platform development production-ready using the Dart programming language and the Flutter UI framework. You will learn that Flutter renders everything by itself [1] in a very good way and it doesn't use any intermediate *bridge* to communicate with the OS. It compiles directly to ARM (for mobile) or optimized JavaScript (for web).

1.1.1 Who is this book for

To get the most out of this book, you should already know the basics of object-oriented programming and preferably at least an "*OOP language*" such as Java or C#. Our goal is trying to make the contents of this book understandable for the widest possible range of developers. Nevertheless, you should already have a minimum of experience in order to better understand the concepts.

If you already know what is a class, what is inheritance and what is nullability, part 1 of this book is going to be a walk in the park. Foreknowledge aside, we will talk about both Dart and Flutter "from scratch" so that the reader can understand any concept regardless the expertise level.

1.1.2 Author

Alberto Miola is an Italian software developer that started working with Delphi (Object Pascal) for desktop development and Java for back-end and Android apps. He currently works in Italy where he daily uses Flutter for mobile and Java for desktop and back-end. Alberto graduated

[1] For example, it doesn't use the system's OEM widgets

in computer science at University of Padua with a thesis about cross-platform frameworks and OOP programming languages.

1.1.3 Acknowledgments

This book owes a lot to some people the author has to mention here because he thinks it's the minimum he can do to express his gratitude. They have technically supported the realization of this book with their fundamental comments and critiques that improved the quality of the contents.

- **Rémi Rousselet**. He is the author of the famous "provider" [2] package and a visible member in the Flutter/Dart community. He actively answers on stackoverflow.com helping tons of people and constantly works in the creation of open source projects.
- **Felix Angelov**. Felix is a Senior Software Engineer at Very Good Ventures. He previously worked at BMW for 3 years and is the main maintainer of the bloc state management library. He has been building enterprise software with Flutter for almost 2 years and loves the technology as well as the amazing community.
- **Matej Rešetár**. He is helping people get prepared for real app development on resocoder.com and also on the Reso Coder YouTube channel. Flutter is an amazing framework but it is easy to write spaghetti code in it. That's why he's spreading the message of proper Flutter app architecture.

Special thanks to my friends Matthew Palomba and Alfred Schilken which carefully read the book improving the style and the quality of the contents.

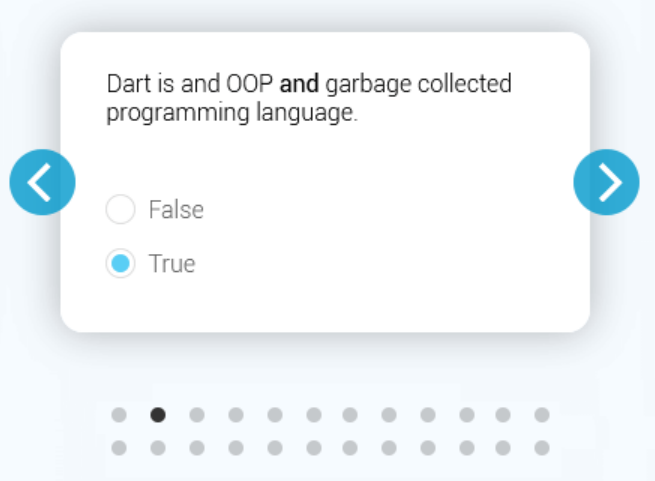
1.1.4 Online resources and the quiz

The official website of this book [3] contains the source code of the examples described in Part III. While reading the chapters you might encounter this box:

Resources > Chapter 16 > Files download

[2] https://pub.dev/packages/provider
[3] https://fluttercompletereference.com

It indicates that if you navigate to the *Resources* page of our website, you'll find the complete source code of the example being discussed at *Chapter 16 > Files download*. In addition, you can play the "Quiz game" which will test the Dart and Flutter skills you've acquired reading this book.



At the end, the result page will tell you the exact page of the book at which you can find an explanation of the answer.

1.2 Introduction to Dart

Dart is a client-optimized, garbage-collected, OOP language for creating fast apps that run on any platform. If you are familiar with an object oriented programming language such as Java or C# you might find many similarities with Dart. The first part of this book aims to show how the language can help you solving problems and the vastness of its API.

1.2.1 Supported platforms

Dart is a very flexible language thanks to the environment in which it lives. Once the source code has been written (and tested) it can be deployed in many different ways:

- **Stand-alone**. In the same way as a Java program can't be run without the Java Virtual Machine (JVM), a stand-alone Dart program can't be executed without the Dart Virtual Machine (DVM). There's the need to download and install the DVM which to execute Dart in a command-line environment. The SDK, other than the compiler and the libraries, also offers a series of other tools:
  - the `pub` package manager, which will be explored in detail in chapter 23;
  - `dart2js`, which compiles Dart code to deployable JavaScript;
  - `dartdoc`, the Dart documentation generator;
  - `dartfmt`, a code formatter that follows the official style guidelines.

  In other words, with the stand-alone way you're creating a Dart program that can only run if the DVM is installed. To develop Flutter apps for any platform (mobile, web and desktop), instead of installing the "pure" Dart SDK, you need to install Flutter [4] (which is basically the Dart SDK combined with Flutter tools).

- **AOT compiled**. The **A**head **O**f **T**ime compilation is the act of translating a high-level programming language, like Dart, into native machine code. Basically, starting from the Dart source code you can obtain a single binary file that can execute natively on a certain operating system. AOT is really what makes Flutter fast and portable.

  With AOT there is **NO** need to have the DVM installed because at the end you get a single binary file (an *.apk* or *.aab* for Android, an *.ipa* for iOS, an *.exe* for Windows...) that can be executed.

  - Thanks to the Flutter SDK you can AOT compile your Dart code into a native binary

[4] https://flutter.dev/docs/get-started/install

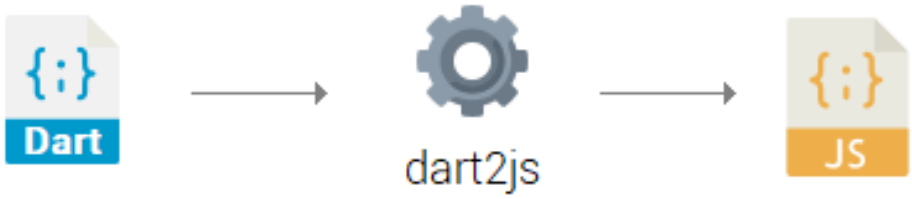
for mobile, web and desktop.

- As of Flutter 1.21, the Dart SDK is included in the Flutter SDK so you don't have to install them separately. They're all bundled in a single install package.

- Starting from version 2.6, the `dart2native` command (supported on Windows, macOS and Linux) makes AOT compiles a Dart program into x64 native machine code. The output is a standalone executable file.

AOT compilation is very powerful because it natively brings Dart to mobile desktop. You'll end up having a single native binary which doesn't require a DVM to be installed on the client in order to run the application.

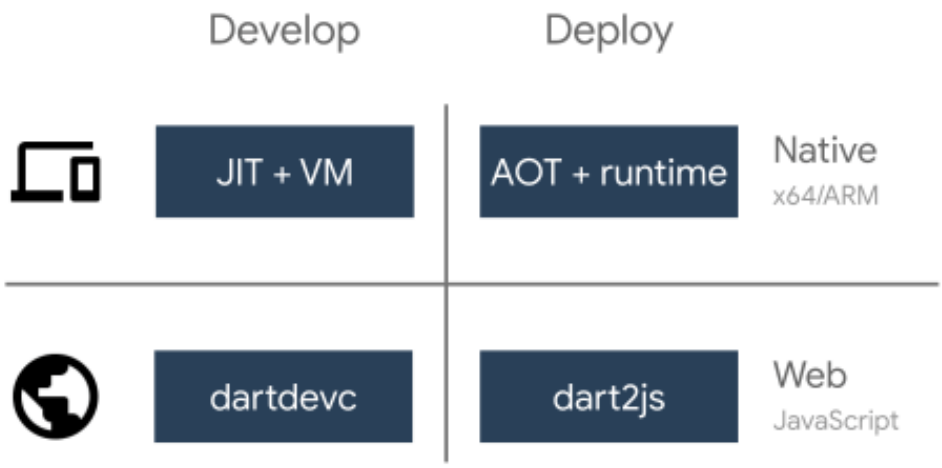
- **Web**. Thanks to the `dart2js` tool, your Dart project can be "transpiled" into fast and compact JavaScript code. By consequence Flutter can be run, for example, on Firefox or Chrome and the UI will be identical to the other platforms.



AngularDart [5] is a performant web app framework used by Google to build some famous websites, such as "AdSense" and "AdWords". Of course it's powered by Dart!

So far we've covered what you can do with Dart when it comes to deployment and production-ready software. When you have to debug and develop, both for desktop/mobile and web, there are useful some tools coming to the rescue.

[5] https://angulardart.dev/



This picture sums up very well how the Dart code can be used in development and deployment. We've just covered the "Deploy" side in the above part, so let's analyze the "Develop" column:

- **Desktop/mobile**. The **J**ust **I**n **T**ime (JIT) technique can be seen as a "real time translation" because the compilation happens while the program is executing. It's a sort of "dynamic compilation" which happens while the program is being used.

  JIT compilation, combined with the DVM (JIT + VM in the picture), allows the dispatch of the code dynamically without considering the user's machine architecture. In this way it's possible to smoothly run and debug the code everywhere without having to mess up with the underlying architecture.

- **Web**. The Dart development compiler, abbreviated with **dartdevc**, allows you to run and debug Dart web apps on Google Chrome. Note that dartdevc is for development only: for deployment, you should use **dart2js**. Using special tools like *webdev* [6] there's the possibility to edit Dart files, refreshing Chrome and visualizing changes almost immediately.

As you've just seen, Dart can run literally everywhere: desktop, mobile and web. This book will give you a wide overview of the language (Dart version 2.10, with null safety support) and all the required skills to create easily maintainable projects.

[6] https://dart.dev/tools/webdev#serve

1.2.2 Package system

Dart's core API offers different packages, such as `dart:io` or `dart:collection`, that expose classes and methods for many purposes. In addition, there is an official online repository called *pub* containing packages created by the Dart team, the Flutter team or community users like you.

If you head to https://pub.dev you will find an endless number of packages for any purpose: I/O handling, XML serialization/de-serialization, localization, SQL/NoSQL database utilities and much more.

1. Go to https://pub.dev, the official repository;
2. Let's say you're looking for an equation solving library. Type "equations" in the search bar and filter the results by platform. Some packages are available only for Dart, others only for Flutter and a good part works for both;
3. The page of the package contains an installation guide, an overview and a guide so that you won't get lost.

You should check the amount of *likes* received by the community and the overall *reputation* of the package because those values indicate how mature and healthy the product is. You will learn how to properly write a library and how to upload it to the pub.dev repository in order to give your contribution to the growth of the community.

1.2.3 Hello World

The simplest way you have to run your Dart code is by opening DartPad [7], an open-source compiler that works in any modern browser. Clicking on "New Pad" you can decide whether creating a new Dart or Flutter project (with latest stable version of the SDK).

[7] https://dartpad.dartlang.org/

It's the perfect tool for the beginners that want to play with Dart and try the code. If you're new to the language, start using DartPad (which is absolutely **not** and IDE). It always has the latest version of the SDK installed and it's straightforward to use.

```
void main() {
    // Best food worldwide!
    print("pasta pizza maccheroni");
}
```

Like with Java and C++, any Dart program has to define a function called `main()` which is the entry point of the application. Very intuitively the `print()` method outputs to the console, on the right of the DartPad, a string. Starting from chapter 2, you'll begin to learn the syntax and the good practices that a programmer should know about Dart.

When you develop for real world applications, you're going to download the whole SDK and use an IDE like IntelliJ IDEA, Android Studio or VS Code. DartPad doesn't give you the possibility to setup tests, import external packages, add dependencies and test your code.

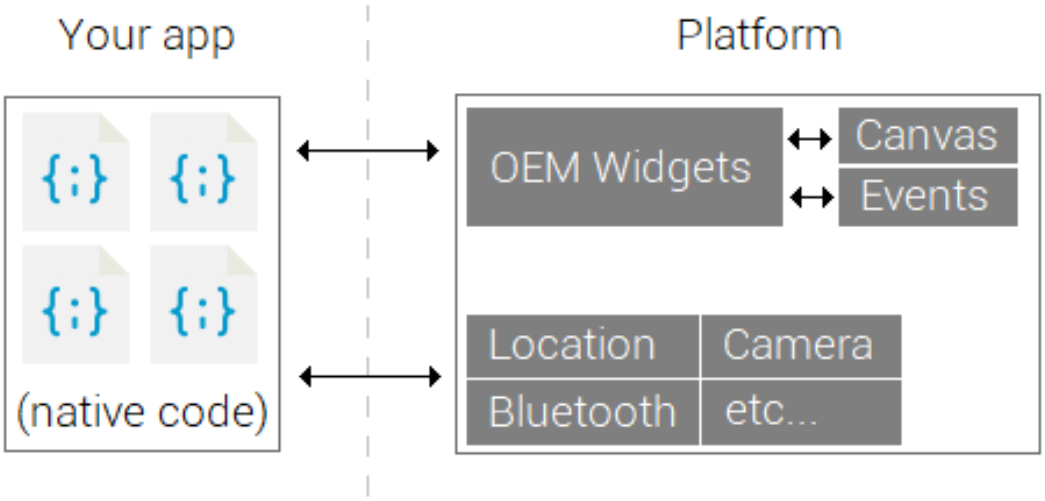
1.3 Intorduction to Flutter

Flutter is an UI toolkit for building natively compiled applications for mobile, desktop and web with a single codebase. At the time of writing this book, only Flutter for mobile is stable and ready for production. Web support is currently in beta while desktop (macOS, Linux and Windows) is in early alpha: they will be covered in a future release of this reference once they will be officially released as stable builds.

Being familiar with *Jetpack compose* or *React Native* is surely an advantage because the concepts of reactive views and "components tree" are the fundamentals of the Flutter framework.

1.3.1 How does it work

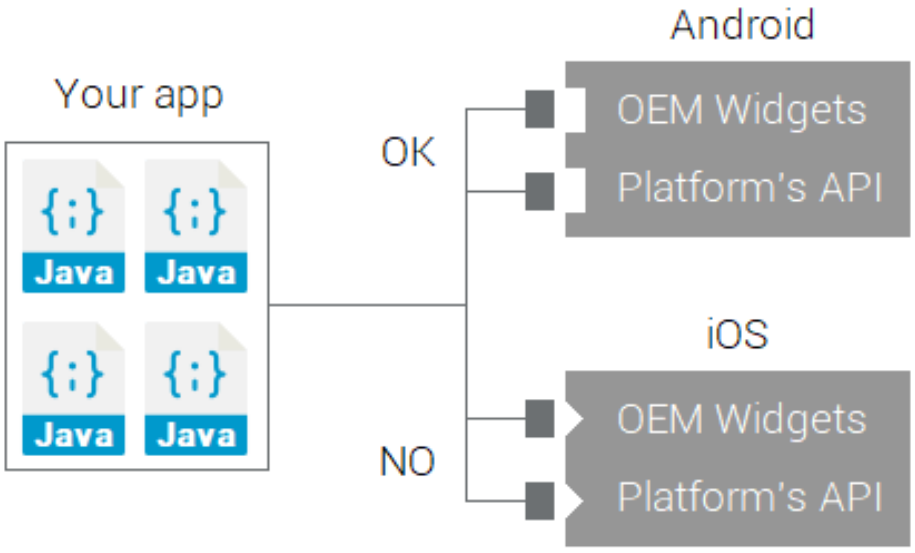
This picture shows how a native app interacts with the OS, whether it's been written in Kotlin (or Java) for Android or Swift (or Objective-C) for iOS. We're going to use these 2 platforms as examples in this section.



1. The platform, which can be Android or iOS, exposes a series of OEM widgets used by the app to build the UI. Those widgets are **fundamental** because they give our app the capabilities to paint the UI, use the canvas and respond to events such as finger taps.
2. If you wanted to take a picture from your app or use the bluetooth to send a file, there would be the need to communicate with the native API exposed by the platform. For example, using OS-specific APIs, you could ask for the camera service, wait for a response and then start using it.

The cross-platform approach is different and it **has** to be like so. If you want your app to run on both Android and iOS with the same codebase, you can't directly use OEM widgets and their API because they come from different architectures. They are **NOT** compatible. On the hardware side however, both are based on the ARM architecture (precisely, v7 and v8) and the most recent versions have 64-bit support. Flutter AOT compiles the Dart code into native ARM libraries.

> **ARM** is a family of RISC microprocessors (32 and 64 bit) widely used in embedded systems. It dominates the mobile world thanks to its qualities: low costs, good heat dissipation and a longer battery life thanks to a low power consumption.

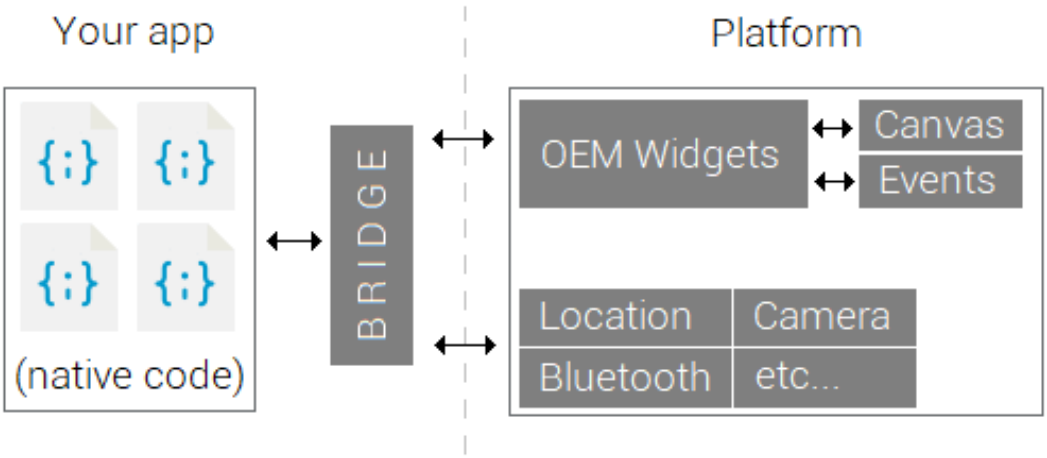


The picture has rectangles on Android and triangles on iOS to indicate that OEM widgets and

APIs have differences in how they are structured, in how they interact with the app and in how you have to use them. For this reason, cross-platform apps cannot directly "talk" to the underlying environment: they must speak a language that everyone can understand.

> Try to only think about the runtime environment for a moment. If you wrote a Java Android app, it would be compiled to work with the ART ecosystem (**A**ndroid **R**un**T**ime): how could the same binary file work with iOS architecture which is completely different and has no ART?
>
> In the above image, squares represents calls made by Java to interact with the ART which is available only in Android and not on iOS. This compatibility problem is solved by cross-platform frameworks.

ReactJS is a Reactive web framework which tries to solve the above problem by adding a *bridge* in the middle that takes care of the communication with the platform. With this approach, the bridge becomes the real starring of the scene it acts like a translator:

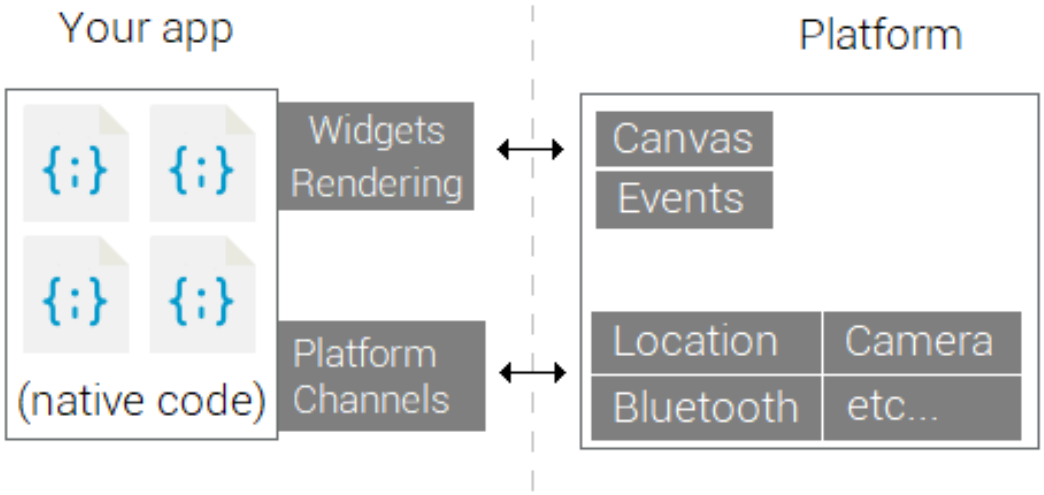


1. The bridge **always** exposes the same interface to the app so that it doesn't care anymore about the OS it's running on;
2. The bridge has an implementation of OEMs and APIs for each platform to allow the app to correctly work in many environments. In this way, you have a native app in the sense that it uses the native tools given by the OS, but there's still an "adapter" in the middle.

As you can see from the picture, the *bridge* is an abstraction layer between the app the OS in which it's hosted. Of course, there has to be a bridge for each supported platform but that's not something you have to deal with because the developers of the framework will take care of creating all of them.

> If you used a cross-platform framework, you'd just need to care about creating the app with the code and the API exposed by the framework. The implementation of the bridge is already in the internals of the SDK and it's automatically "attached" to the app in the build phase. You don't have to create the bridge.

The *bridge* approach is quite popular, but it could be a potential bottleneck that slows down the execution and thus the performances might drop. If you think about animations, swipes or transitions, widgets are accessed very often and many of them running at the same time could slow down the app. Flutter adopts a completely different strategy:



It uses its own very efficient rendering engine, called *Skia*, to paint the UI so that OEM widgets are not needed anymore. In this way, the app doesn't rely on the instruments the OS exposes to draw the interface and you can freely control each single pixel of the screen.

- Flutter produces **native** ARM code for the machine;

- when launched, the app loads the Flutter library. Any rendering, input or event handling, and so on, is delegated to the compiled Flutter and app code. This is much faster than having a bridge.
- A minimal Flutter app is about 4.4 MB on Android and 10.9 MB on iOS (depending on the architecture, whether it be ARM 32 or 64 bit) [8]

The true power of Flutter lies on the fact that apps are built with their own rendering stuff and they are not constrained to paint the UI following the rules "imposed" by OEM widgets. You're free to control the screen and manipulate every single pixel.

1.3.2 Why Flutter uses Dart

There are many reasons behind the decision made by Google to choose Dart as language for the Flutter framework. At the time of writing this book, the latest **stable** version of Dart is 2.9.2 (Dart 2.10 is on beta, but downloadable anyway). Here's a summary [9] of what brought them to make this choice.

1. **OOP style**. The vast majority of developers have object-oriented programming skills and thus Dart would be easy to learn as it adopts most of the common OOP patterns. The developer doesn't have to deal with a completely new way of coding; he can reuse what he already knows and integrate it with the specific details of Dart.
2. **Performances**. In order to guarantee high performances and avoid frame dropping during the execution of the app, there's the need of a high performance and predictable language. Dart can guarantee to be very efficient and it provides a powerful memory allocator that handles small, short-lived allocations. This is perfect for Flutter's functional-style flow.
3. **Productivity**. Flutter allows developers to write Android, iOS, web and desktop apps with a single codebase keeping the same performances, aspect and feeling in each platform. A highly productive language like Dart accelerates the coding process and makes the framework more attractive.
4. Both Flutter and Dart are developed by Google which can freely decide what to do with them listening to the community as well. If Dart was developed by another company, Google probably wouldn't have the same freedom of choice in implementing new features and and the language couldn't evolve at the desired pace.

Another important aspect is that Dart is **strongly** typed, meaning that the compiler is going

[8] https://flutter.dev/docs/resources/faq#how-big-is-the-flutter-engine
[9] https://flutter.dev/docs/resources/faq#why-did-flutter-choose-to-use-dart

to be very strict about types; you'll have both less runtime surprises and an easier debugging process. In addition, keep in mind that Dart is a complete swiss-knife because it has built-in support for:

- tree-shaking optimization;
- hot reload feature;
- a package manager with mandatory documentation and the possibility to play with the code using DartPad;
- *DevTools*, a collection of debugging and performance tools;
- code documentation generator tool;
- support for JIT and AOT compilation.

By owning two home-made products, Google can keep the entire projects under control and decide how to integrate them in the best way possible with quick development cycles. Dart evolves together with Flutter and as time goes by: they help each other maximizing productivity and performances.

1.3.3 Hello world

When creating Flutter apps for the production world, you should really consider using Android Studio or VSCode and install the respective plugins. They offer a debugger, hints, a friendly UI and powerful optimization tools we will explore in detail.

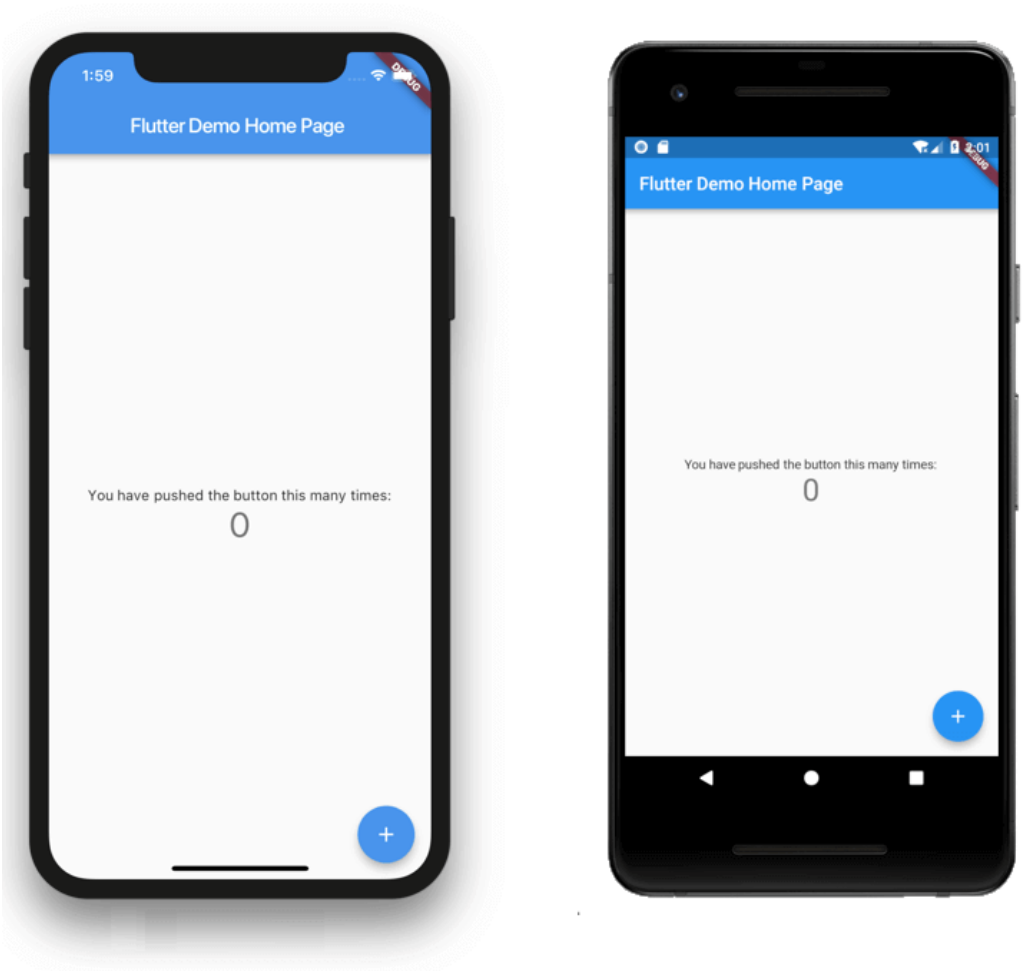
```
void main() {
    runApp(MyApp());
}

class MyApp extends StatelessWidget {
    const MyApp();

    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                body: Center(
                    child: Text("Flutter app!"),
                ),
            ),
```

```
            );
        }
    }
```

This is a very simple example of a minimal Flutter application. You can notice immediately that there is a `void main() { ... }` function, required by Dart to define the entry point of the program. An UI is a composition of *widgets* that decorate the screen with many objects; you will learn how to properly use them to create efficient and beautiful designs.



This is an example of how a simple Flutter app looks identical in two different platforms. In this book we will focus on Android and iOS mobile apps but **everything** you're going to learn is also valid for web and desktop because it's always Flutter. Once you have the code ready, open the console...

```
$ flutter build appbundle

$ flutter build ios

$ flutter build web

$ flutter build macos

$ flutter build windows

$ flutter build linux
```

...and it's just a matter of running different *build* commands to get different native binaries of the same app. For more info on Flutter for web and desktop, see the appendix B at the bottom of the book.

Part I

The Dart programming language

"I'm not a great programmer; I'm just a good programmer with great habits."

Martin Fowler

2 | Variables and data types

2.1 Variables

As in any programming language, variables are one of the basics and Dart comes with support for type inference. A typical example of creation and initialization of a variable is the following:

```
var value = 18;
var myName = "Alberto"
```

In the example, `value` is an integer while `myName` is a string. Like Java and C#, Dart is able to **infer** the type of the variable by looking at the value you've assigned. In other words, the Dart compiler is smart enough to figure out by itself which is the correct type of the variable.

```
int value = 18;
String myName = "Alberto"
```

This code is identical to the preceding example with the only difference that here the types have been typed explicitly. There would also be a third valid way to initialize variables, but you should almost never use it.

```
dynamic value = 18;
dynamic myName = "Alberto"
```

`dynamic` can be used with any type, it's like a "jolly": any value can be assigned to it and the compiler won't complain. The type of a `dynamic` variable is evaluated at runtime and thus, for a proper usage, you'd need to work with checks and type casts. According with the Dart guidelines and our personal experience you should:

1. Prefer initializing variables with `var` as much as you can;
2. When the type is not so easy to guess, initialize it explicitly to increase the readability of the code;

3. Use `Object` or `dynamic` only if it's really needed but it's almost never the case.

Actually, we could say that `dynamic` is not really a type: it's more of a way to turn off static analysis and tell the compiler you know what you're doing. The only case in which you'll deal with it will come in the Flutter part in regard to JSON encoding and decoding.

2.1.1 Initialization

The official Dart guidelines [1] state that you should prefer, in most of the cases, the initialization with `var` rather than writing the type explicitly. Other than making the code shorter (programmers are lazy!) it can increase the readability in various scenarios, such as:

```
// BAD: hard to read due to nested generic types
List<List<Toppings>> pizza = List<List<Toppings>>();
for(List<Toppings> topping in pizza) {
    doSomething(topping);
}

// GOOD: the reader doesn't have to "parse" the code
// It's clearer what's going on
var pizza = List<List<Toppings>>();
for(var topping in pizza) {
    doSomething(topping);
}
```

Those code snippets use generics, classes and other Dart features we will discuss in depth in the next chapters. It's worth pointing out two examples in which you want to explicitly write the type instead of inferring it:

- When you don't want to initialize a variable immediately, use the `late` keyword. It will be explained in detail later in this chapter.

```
// Case 1
late List<String> names;

if (iWantFriends())
    names = friends.getNames();
else
    names = haters.getNames();
```

[1] https://dart.dev/guides/language/effective-dart/design#types

If you used `var` instead of `List<String>` the inferred type would have been `null` and that's **not** what we want. You'd also lose the type safety and readability.

- The type of the variable is not so obvious at first glance:

```
// Is this a list? I guess so, "People" is plural...
// but actually the function returns a String!
var people = getPeople(true, 100);

// Ok, this is better
String people = getPeople(true, 100);
```

However, there isn't a golden rule to follow because it's up to your discretion. In general `var` is fine, but if you feel that the type can make the code more readable you can definitely write it.

2.1.2 final

A variable declared as `final` can be set only once and if you try to change its content later, you'll get an error. For example, you won't be able to successfully compile this code:

```
final name = "Alberto";
name = "Albert"; // 'name' is final and cannot be changed
```

You can also notice that `final` can automatically infer the type exactly like `var` does. This keyword can be seen as a "restrictive var" as it deduces the type automatically but does not allow changes.

```
// Very popular - Automatic type deduction
final name = "Alberto";
// Generally unnecessary - With type annotation
final String nickName = "Robert";
```

If you want you can also specify the type but it's not required. So far we've only shown examples with strings, but of course both `final` and `var` can be used with complex data types (classes, enums) or methods.

```
final rand = getRandomInteger();

// rand = 0;
// ^ doesn't work because the variable is final
```

The type of `rand` is deduced by the return statement of the method and it cannot be re-assigned in a second moment. The same advice we've given in "2.1.1 Initialization" for `var` can be applied here as well.

> Later on in the book we will analyze in detail the `const` keyword, which is the "brother" of `final`, and it has very important performance impacts on Flutter.

While coding you can keep this rule in mind: use `final` when you know that, once assigned, the value will **never** change in the future. If you know that the value might change during the time use `var` and think whether it's the case to annotate the type or not. Here's an example in which a `final` variable fits perfectly:

```
void main() {
    // Assume that the content of the file can't be edited
    final jsonFile = File('myfile.json').readAsString();

    checkSyntax(jsonFile);
    saveToDisk(jsonFile, 'file.json');
}
```

In this example the variable `jsonFile` has a content that doesn't have to be modified, it will always remain the same and so a `final` declaration is good:

- it won't be accidentally edited later;
- the compiler will give an error if you try to modify the value.

If you used `var` the code would have compiled anyway but it wouldn't have been the best choice. If the code was longer and way more complicated, you could accidentally change the content of `jsonFile` because there wouldn't be the "protection" of `final`.

2.2 Data types

Types in Dart can be initialized with "literals"; for example `true` is a boolean literal and `"test"` is a string literal. In chapter 6 we will analyze *generic* data types that are very commonly used for collections such as lists, sets and maps.

2.2.1 Numbers

Dart has two type of numbers:

- `int`. 64-bit at maximum, depending on the platform, integer values. This type ranges from -2^{63} to 2^{63}-1.
- `double`. 64-bit double-precision floating point numbers that follow the classic IEEE 754 standard definition.

Both `double` and `int` are subclasses of `num` which provides many useful methods such as:

- `parse(string)`,
- `abs()`,
- `ceil()`,
- `toString()`...

You should always use `double` or `int`. We will see, with generic types, a special case in which `num` is needed but in general you can avoid it. Some examples are always a good thing:

```
var a = 1; // int
var b = 1.0; // double

int x = 8;
double y = b + 6;
num z = 10 - y + x;

// 7 is a compile-time constant
const valueA = 7;
// Operations among constant values are constant
const valueB = 2 * valueA;
```

From Dart 2.1 onwards the assignment `double a = 5` is legal. In 2.0 and earlier versions you were forced to write 5.0, which is a *double* literal, because 5 is instead an *integer* literal and the compiler didn't automatically convert the values. Some special notations you might find useful are:

1. The exponential representation of a number, such as `var a = 1.35e2` which is the equivalent of 1.35 * 10^2;

2. The hexadecimal representation of a number, such as `var a = 0xF1A` where 0xF1A equals to F1A in base 16 (3866 in base 10).

2.2.1.1 Good practices

Very likely, during your coding journey, you'll have at some point the need to parse numbers from strings or similar kinds of manipulations. The language comes to the rescue with some really useful methods:

```
String value = "17";

var a = int.parse(value); // String-to-int conversion
var b = double.parse("0.98"); // String-to-double conversion
var c = int.parse("13", radix: 6); // Converts from 13 base 6
```

You should rely on these methods instead of writing functions on your own. In the opposite direction, which is the conversion into a string, there is `toString()` with all its variants:

```
String v1 = 100.toString(); // v1 = "100";
String v2 = 100.123.toString(); // v2 = "100.123";
String v3 = 100.123.toStringAsFixed(2); // v3 = "100.12";
```

Since we haven't covered functions yet you can come back to this point later or, if you're brave enough, you can continue the reading. When converting numbers from a string, the method `parse()` can fail if the input is malformed such as `"12_@4.49"`. You'd better use one of the following solutions (we will cover nullable types later):

```
// 1. If the string is not a number, val is null
double? val = double.tryParse("12@.3x_"); // null
double? val = double.tryParse("120.343"); // 120.343

// 2. The onError callback is called when parsing fails
var a = int.parse("1_6", onError: (value) => 0); // 0
var a = int.parse("16", onError: (value) => 0); // 16
```

Keep in mind that `parse()` is deprecated: you should prefer `tryParse()`. What's important to keep in mind is that a plain `parse("value")` call is risky because it assumes the string is already well-formed. Handling the potential errors as shown is safer.

2.2.2 Strings

In Dart a string is an ordered sequence of UTF-16 values surrounded by either single or double quotes. A very nice feature of the language is the possibility of combining expressions into strings by using `${expr}` (a shorthand to call the `toString()` method).

```
// No differences between s and t
var s = "Double quoted";
var t = 'Single quoted';

// Interpolate an integer into a string
var age = 25;
var myAge = "I am $age years old";

// Expressions need '{' and '}' preceeded by $
var test = "${25.abs()}"

// This is redundant, don't do it because ${} already calls toString()
var redundant = "${25.toString()}";
```

A string can be either single or multiline. Single line strings are shown above using single or double quotes, and multiline strings are written using triple quotes. They might be useful when you want to nicely format the code to make it more readable.

```
// Very useful for SQL queries, for example
var query = """
    SELECT name, surname, age
    FROM people
    WHERE age >= 18
    ORDER BY name DESC
""";
```

In Dart there isn't a `char` type representing a single character because there are only strings. If you want to access a particular character of a string you have to use the `[]` operator:

```
final name = "Alberto";

print(name[0]); // prints "A"
print(name[2]); // prints "b";
```

The returned value of `name[0]` is a `String` whose length is 1. We encourage you to visit [2] the online Dart documentation about strings which is super useful and full of examples.

```
var s = 'I am ' + name + ' and I am ' + (23).toString() + ' y.o.';
```

You can concatenate strings very easily with the + operator, in the classic way that most programming languages support. The official Dart guidelines [3] suggest to prefer using interpolation to compose strings, which is shorter and cleaner:

```
var s = 'I am $name. I am ${25} years old';
```

In case of a string longer than a single line, avoid the + operator and prefer a simple line break. It's just something recommended by the Dart for styling reasons, there are no performance implications at all. Try to be as consistent as possible with the language guidelines!

```
// Ok
var s = 'I am going to the'
        'second line';

// Still ok but '+' can be omitted
var s = 'I am going to the' +
        'second line';
```

Since strings are immutable, making too many concatenations with the + operator might be inefficient. In such cases it'd be better if you used a `StringBuffer` which efficiently concatenates strings. For example:

```
var value = "";

for(var i = 0; i < 900000; ++i) {
    value += "$i ";
}
```

Each time the + operator is called, `value` is assigned with a **new** instance which merges the old value and the new one. In other words, this code creates for 900000 times a new `String` object, one for each iteration, and it's not optimal at all. Here's the way to go:

```
var buffer = StringBuffer();
```

[2] https://dart.dev/guides/libraries/library-tour#strings-and-regular-expressions
[3] https://dart.dev/guides/language/effective-dart/usage#prefer-using-interpolation-to-compose-strings-and-values

```
for(var i = 0; i < 900000; ++i)
    buffer.write("$i ");

var value = buffer.toString();
```

This is much better because `StringBuffer` doesn't internally create a new string on each iteration; the string is created only **once** at the moment in which `toString()` is called. When you have to do long loops that manipulate strings, avoid using the + operator and prefer a buffer. The same class can also be found in Java and C# for example.

2.2.3 Enumerated types

Also known as "enums", enumerated types are containers for constant values that can be declared with the `enum` keyword. A very straightforward example is the following:

```
enum Fruits { Apple, Pear, Grapes, Banana, Orange }

void main() {
    Fruits liked = Fruits.Apple;
    var disliked = Fruits.Banana;

    print(liked.toString()); // prints 'Fruits.Apple'
    print(disliked.toString()); // prints 'Fruits.Banana'
}
```

Each item of the enum has an associated number, called **index**, which corresponds to the zero-based position of the value in the declaration. You can access this number by using the `index` property.

```
enum Fruits { Apple, Pear, Grapes, Banana, Orange }

void main() {
    var a = Fruits.Apple.index;  // 0
    var b = Fruits.Pear.index;  // 1
    var c = Fruits.Grapes.index;  // 2
}
```

Note that when you need to use an `enum` you always have to fully qualify it. Using the name only doesn't work.

2.2.3.1 Good Practices

When you need a predefined list of values which represents some kind of textual or numeric data, you should prefer an `enum` over a primitive data type. In this way you can increase the readability of the code, the consistency and the compile-time checking. Look at these 2 ways of creating a function (*///* is used to document the code):

```
enum Chess { King, Queen, Rook, Bishop, Knight, Pawn }

/// METHOD 1. Checks if the piece can move in diagonal
bool diagonalMoveC(Chess item) { ... }

/// METHOD 2. Checks if a piece can move in diagonal: [item] can only be:
///  1. King
///  2. Queen
///  3. Rook
///  4. Bishop
///  5. Knight
///  6. Pawn
/// Any other number is not allowed.
bool diagonalMoveS(int item) { ... }
```

This example should convince you that going for the first method is for sure the right choice.

- `diagonalMoveC(Chess item)`. There's a big advantage here: we're guaranteed by the compiler that `item` can only be one of the values in `Chess`. There's no need for any particular check and we can understand immediately what the method wants us to pass.
- `diagonalMoveS(int item)`. There's a big disadvantage here: we can pass any number, not only the ones from 1 to 6. We're going to do extra work in the body because we don't have the help of the compiler, so we need to manually check if `item` contains a valid value.

In the second case, we'd have to make a series of `if` conditions to check whether the value ranges from 1 to 6. Using an `enum`, the compiler does the checks for us (by comparing the types) and we're guaranteed to work with valid values.

2.2.4 Booleans

You can assign to the `bool` type only the literals `true` or `false`, which are both compile-time constants. Here there are a few usage examples:

```
bool test = 5 == 0; // false
bool test2 = !test; // has the opposite value of test

var oops = 0.0 / 0.0; // evaluates to 'Not a Number' (NaN)
bool didIFail = oops.isNaN;
```

2.2.5 Arrays

Probably you're used to create arrays like this: `int[] array = new int[5];` which is the way that Java and C# offer. In Dart it doesn't really work like that because you can only deal with collections: an "array" in Dart is represented by a `List<T>`.

> `List<T>` is a generic container where `T` can be any type (such as `String` or a class). We will cover generics and collections in detail in chapter 6. Basically, Dart doesn't have "arrays" but only generic containers.

If this is not clear, you can look at this comparison. In both languages there is a generic container for the given type but only Java has "primitive" arrays.

- **Java**

```
// 1. Array
double[] test = new test[10];
// 2. Generic list
List<double> test = new ArrayList<>();
```

- **Dart**

```
// 1. Array
// (no equivalent)
// 2. Generic list
List<double> test = new List<double>();
```

In Dart you can work with arrays but they are intended to be instances of `List<T>`. Lists are 0-indexed collections and items can be randomly accessed using the `[]` operator, which will throw an exception if you exceed the bounds.

```
//use var or final
final myList = [-3.1, 5, 3.0, 4.4];
final value = myList[1];
```

A consequence of the usage of a `List<T>` as container is that the instance exposes many useful methods, typical of collections:

- `length`,
- `add(T value)`,
- `isEmpty`,
- `contains(T value)`

... and much more.

2.3 Nullable and Non-nullable types

Starting from Dart 2.10, variables will be **n**on-**n**ullable **b**y **d**efault (nnbd) which means they're not allowed to hold the `null` value. This feature has been officially introduced in June 2020 as *tech preview* in the dev channel of the Dart SDK.

```
// Trying to access a variable before it's been assigned will cause a
// compilation error.
int value;
print("$value"); // Illegal, doesn't compile
```

If you don't initialize a variable, it's automatically set to `null` but that's an error because Dart has non-nullability enabled by default. In order to successfully compile you have to initialize the variable as soon as it's declared:

```
// 1.
int value = 0;
print("$value");

// 2.
int value;
value = 0;
print("$value");
```

In the first case the variable is assigned immediately and that's what we recommend to do as much as possible. The second case is still valid because `value` is assigned **before** it's ever accessed. It wouldn't have worked if you had written this:

```
// OK - assignment made before the usage
```

```
int value;
value = 0;
print("$value");

// ERROR - usage made before assignment
int value;
print("$value");
value = 0;
```

Non-nullability is very powerful because it adds another level of type safety to the language and, by consequence, lower possibilities for the developer to encounter runtime exceptions related to `null`. For example, you won't have the need to do this:

```
String name = "Alberto";

void main() {
    if (name != null) {
        print(name)
    }
}
```

The compiler guarantees that it can't be `null` and thus no null-checks are required. To sum up, what's important to keep in mind while writing Dart 2.10 code (and above) is:

- By default, variables cannot be `null` and they must **always** be initialized before being used. It would be better if you immediately initialized them, but you could also do it in a second moment before they ever get utilized.
- Don't do null-checks on "standard" non-nullable variables because it's useless.

In Dart you can also declare *nullable* types which doesn't require to be initialized before being accessed and thus they're allowed to be `null`. Nullables are the counterpart of non-nullable types because the usage of `null` is allowed (but the additional type safety degree is lost).

```
int? value;
print("$value"); // Legal, it prints 'null'
```

If you append a question mark at the end of the type, you get a nullable type. For safety, they would require a manual null checks in order to avoid undesired exceptions but, in most of the cases, sticking with the default non-nullability is fine.

```
// Non-nullable version - default behavior
```

```
int value = 0;
print("$value"); // prints '0'

// Nullable version - requires the ? at the end of the type
int? value;
print("$value"); // prints 'null'
```

Nullable types that support the index operator `[]` need to be called with the `?[]` syntax. `null` is returned if the variable is also `null`.

```
String? name = "Alberto";
String? first = name?[0]; // first = 'A';

String? name;
String? first = name?[0]; // first = 'null';
```

We recommend to stick with the defaults, which is the usage of non-nullable types, as they're safer to use. Nullables should be avoided or used only when working with legacy code that depends on `null`. Last but not least, here are the only possible conversions between nullables and non nullables:

- When you're sure that a nullable expression isn't null, you can add a `!` at the end to convert it to the non-nullable version.

  ```
  int? nullable = 0;
  int notNullable = nullable!;
  ```

  The `!` (called "bang operator") converts a nullable value (`int?`) into a non-nullable value (`int`) of the same type. An exception is thrown if the nullable value is actually `null`.

  ```
  int? nullable;
  // An exception is thrown
  int notNullable = nullable!;
  ```

- If you need to convert a nullable variable into a non-nullable subtype, use the typecast operator `as` (more on it later):

  ```
  num? value = 5;
  int otherValue = value as int;
  ```

  You wouldn't be able to do `int otherValue = value!` because the bang operator works only when the type is the same. In this example, we have a `num` and an `int` so there's the need for a cast.

- Even if it isn't a real conversion, the operator `??` can be used to produce a non-nullable value from a nullable one.

```
int? nullable = 10;
int nonNullable = nullable ?? 0;
```

  If the member on the left (`nullable`) is non-null, return its value; otherwise, evaluate and return the member of the right (`0`).

Remember that when you're working with nullable values, the member access operator (`.`) is not available. Instead, you have to use the **null-aware** member access operator (`?.`):

```
double? pi = 3.14;

final round1 = pi.round();   // No
final round2 = pi?.round();  // Ok
```

2.4 Data type operators

In Dart expressions are built using operators, such as `+` and `-` on primitive data types. The language also supports operator overloading for classes as we will cover in chapter 4.

2.4.1 Arithmetic operators

Arithmetic operators are commonly used on `int` and `double` to build expressions. As you already know, the `+` operator can also be used to concatenate strings.

Symbol	Meaning	Example
+	Add two values	`2 + 3 //5`
-	Subtract two values	`2 - 3 //-1`
*	Multiply two values	`6 * 3 //18`
/	Divide two values	`9 / 2 //4.5`
~ /	Integer division of two values	`9 ~/ 2 //4`

%	Remainder (modulo) of an int division	`5 % 2 //1`

Prefix and postfix increment or decrement work as you're used to see in many languages.

```
int a = 10;
++a; // a = 11
a++; // a = 12

int b = 5;
--b; // b = 4;
b--; // b = 3;

int c = 6;
c += 6 // c = 12
```

As a reminder, both postfix and prefix increment/decrement have the same result but they work in a **different** way. In particular:

- in the prefix version (`++x`) the value is first incremented and then "returned";
- in the postfix version (`x++`) the valie is first "returned" and then incremented

2.4.2 Relational operators

Equality and relational operators are used in boolean expression, generally inside `if` statements or as a stop condition of a `while` loop.

Symbol	Meaning	Example
==	Equality test	`2 == 6`
!=	Inquality test	`2 != 6`
>	Greather than	`2 > 6`

<	Smaller than	`2 < 6`
>=	Greater or equal to	`2 >= 6`
<=	Smaller or equal to	`2 <= 6`

Testing the equality of two objects *a* and *b* always happens with the == operator because, unlike Java or C#, there is no `equals()` method. In chapter 6 we will analyze in detail how classes can be properly compared by overriding the equality operator. In general here's how the == works:

1. If *a* or *b* is null, return `true` if both are null or `false` if only one is null. Otherwise...
2. ... return the result of == according with the logic you've defined in the method override.

Of course, == works only with objects of the same type.

2.4.3 Type test operators

They are used to check the type of an object at runtime.

Symbol	Meaning	Example
`as`	Cast a type to another	`obj as String`
`is`	True if the object has a certain type	`obj is double`
`is!`	False if the object has a certain type	`obj is! int`

Let's say you've defined a new type like `class Fruit {}`. You can cast an object to `Fruit` using the `as` operator like this:

```
(grapes as Fruit).color = "Green";
```

The code compiles but it's unsafe: if `grapes` was `null` or if it wasn't a `Fruit`, you would get an exception. It's always a good practice checking whether the cast is doable before doing it:

```
if (grapes is Fruit) {
    (grapes as Fruit).color = "Green";
}
```

Now you're guaranteed the cast will happen only if it's possible and no runtime exceptions can happen. Actually, the compiler is smart enough to understand that you're doing a type check with `is` and it can do a *smart cast*.

```
if (grapes is Fruit) {
    grapes.color = "Green";
}
```

You can avoid writing the explicit cast `(grapes as Fruit)` because, inside the scope of the condition, the variable `grapes` is automatically casted to the `Fruit` type.

2.4.4 Logical operators

When you have to create complex conditional expressions you can use the logical operators:

Symbol	Meaning
`!expr`	Toggles true to false and vice versa
`expr1 && expr2`	Logical AND (true if both sides are true)
`expr1 || expr2`	Logical OR (true if at least one is true)

2.4.5 Bitwise and shift operators

You'll never use these operators unless you're doing some low level data manipulation but in Flutter this never happens.

Symbol	Meaning
`a & b`	Bitwise AND
`a | b`	Bitwise OR
`a ^ b`	Bitwise XOR
`~ a`	Bitwise complement
`a >> b`	Right shift
`a << b`	Left shift

Part II

The Flutter framework

"Programs must be written for people to read, and only incidentally for machines to execute."

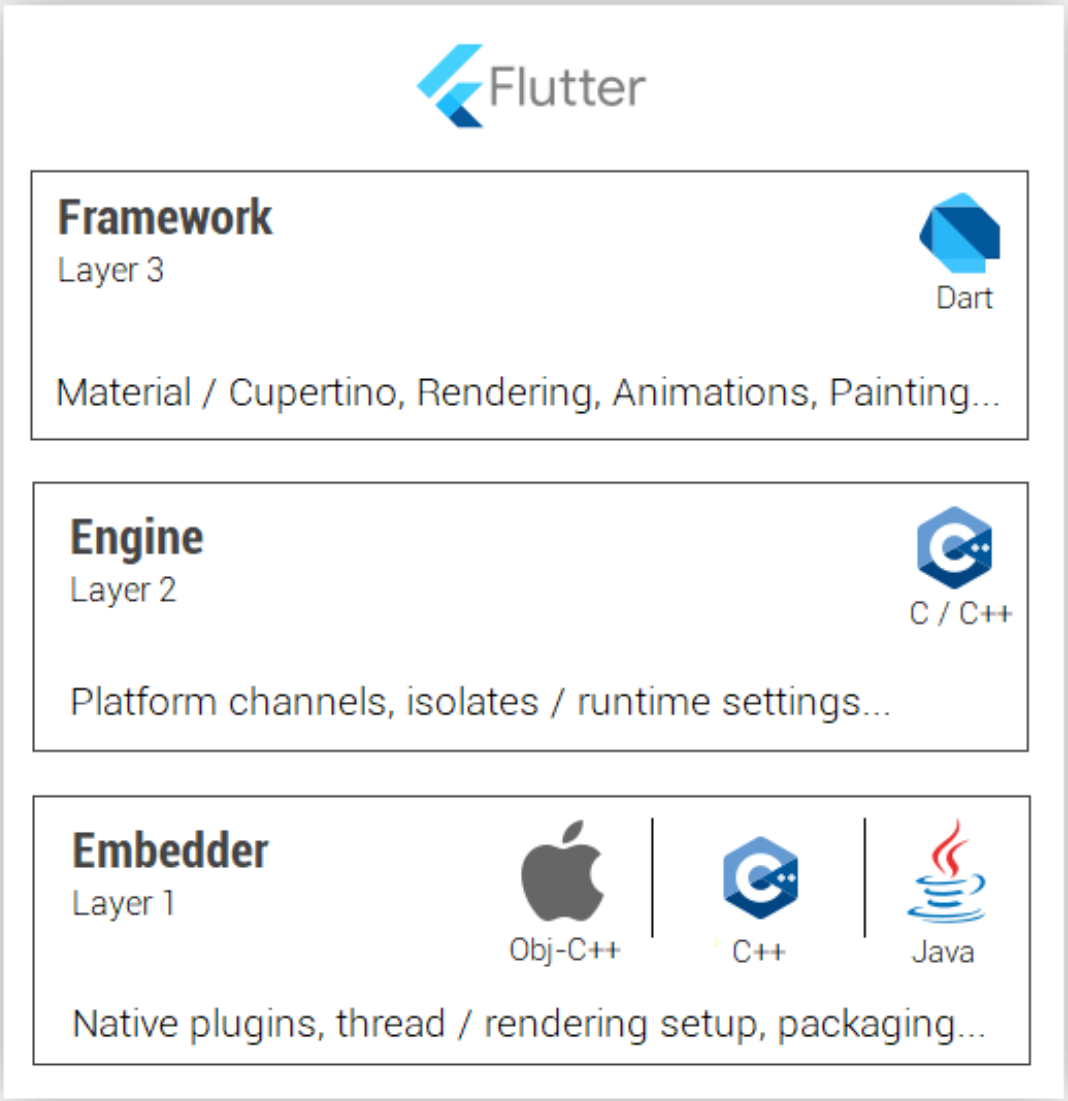
Abelson and Sussman

3 | Basics of Flutter

You are reading the **last** section of chapter 9. This is **not** the full chapter.

3.1 Architecture

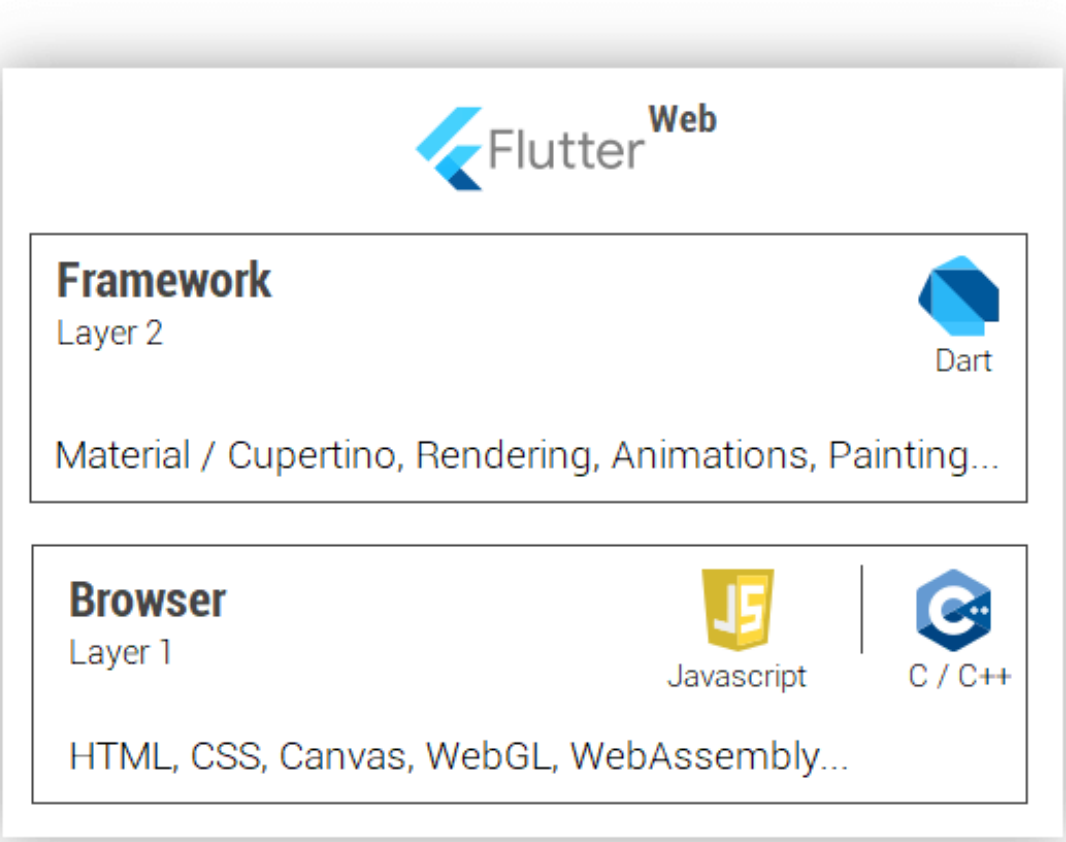
In this section we're giving a general overview of the architecture of the framework digging a bit more into the details. Flutter is divided into three layers (it's said to be a **layered system**) where each depends on the one below. Layers are made up of libraries written in different languages.



The **embedder** is written in different languages according with the platform in which Flutter has to run: Objective C++ for iOS / macOS, Java / C++ for Android and C++ for Linux / Windows. It's a native application that takes care of "hosting" your Flutter contents on the OS. When the app is started, the embedder provides a valid entrypoint, obtains threads for UI and rendering, starts the Flutter engine and much more.

The embedder is at the lowest layer and it directly interacts with the operating system providing entry points for access to services. The developer mostly works on the third layer and sometimes on the second, but never on the first.

The **engine** is the heart of Flutter, it's mostly written in C++ and it's always packaged in the binary produced by the `flutter build` tool. It's a portable runtime for hosting Flutter application which includes core libraries for network I/O, file, animations and graphics. The engine is exposed to the developer via `import "dart:ui"`, which basically wraps C++ sources into Dart classes. For the web world, the situation is different:



The C++ engine is designed to work with the operating system but for the web Flutter has to

deal with a browser. For this reason, the approach has to be different. Dart can be compiled to JavaScript thanks to the highly-optimized **dart2js** compiler so, by consequence, Flutter apps can be ported as well. There are 2 ways to deploy an application for the web:

1. **HTML** mode. Flutter uses HTML, CSS, JavaScript and Canvas.
2. **WebGL** mode. Flutter uses CanvasKit, which is Skia compiled to WebAssembly.

For the web, there's no need for the Dart runtime because your Flutter app is compiled to JavaScript as we've already said. The produced code is already minified and it can be deployed to any server. At the time of publishing this book (September 2020), web support for Flutter is only available in the beta channel.

> In case you didn't know, **WebAssembly** is recognized [1] by the W3C as the 4th language to natively run on browsers along with HTML, CSS, and JavaScript. WebAssembly can be both AOT and JIT compiled.

3.1.1 Element and RenderObject

You've already seen that, to build the UI, the developer has to create a **widget tree** by nesting widgets one inside the other. In reality, Flutter doesn't only rely on widgets because internally there are two other kinds of trees maintained in parallel [2]. Through this section, we're assuming that `SomeText` is just a simple widget showing some text.

```
class MyWidget extends StatelessWidget {
    const MyWidget();

    @override
    Widget build(BuildContext context) {
        return Container(
            decoration: BoxDecoration(),
            child: SomeText(
                text: "Hello"
            ),
        );
```

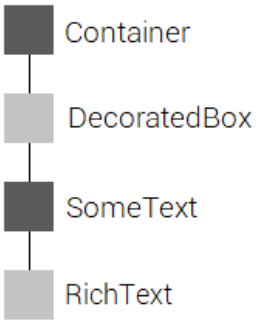
[1] https://www.w3.org/TR/wasm-core-1/
[2] See "The Layer Cake" by Frederik Schweiger on Medium

```
        }
    }
```

When it's time to render, Flutter calls the `build()` method of the widget which might introduce some new widgets, in case of nesting. In our case, the widget tree will contain `Container`, `SomeText` plus some more you actually don't see. In fact, if you looked at the definition of a `Container`...

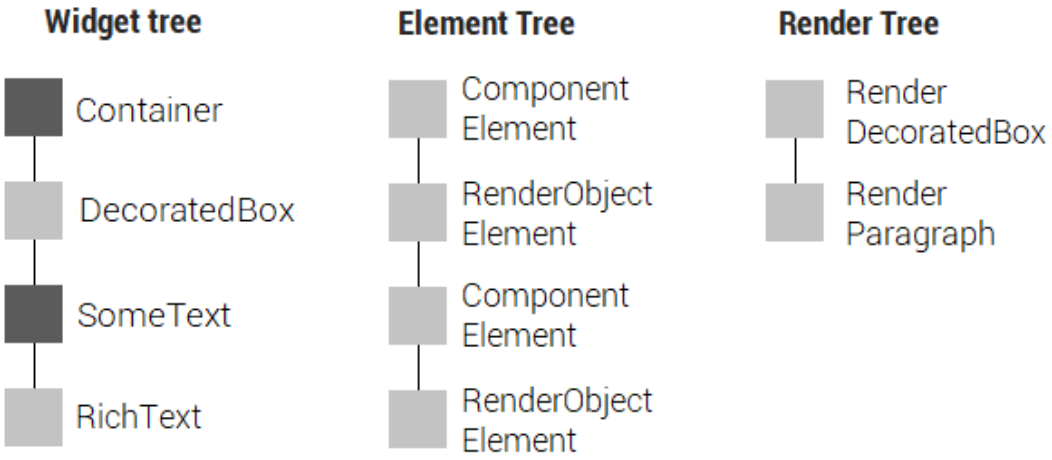
```
    if (decoration != null)
        current = DecoratedBox(decoration: decoration, child: current);
```

... you'd notice that an instance of `DecoratedBox` is added under the hood if a `decoration` is given. For this reason, if you made a DevTools [3] inspection you'd see more children than you actually inserted. It's because widgets might insert other widgets inside but you just don't see it; the tree actually looks like this:



Some boxes are in grey to visualize the fact they haven't been added by you. Along with the widget tree, Flutter also builds in parallel the **element** tree and the **render** tree. They are created calling respectively `createElement()` and `createRenderObject()` on the widget being traversed. Note that `createElement()` is always called on widgets but `createRenderObject()` is only called on elements whose type is `RenderObjectElement`. So yes, at the end Flutter works with 3 trees.

[3] More on it in chapter 16



An `Element` can hold a reference to a widget and the respective `RenderObject`. There are a lot of new things you've never seen up to now so let's carefully analyze the trees to understand how Flutter really works.

- **Render tree**. A `RenderObject` contains all the logic to render the corresponding widget and it's expensive to create. They take care of the layout, the constraints, hit testing and painting. The framework keeps them in memory as much as possible, changing their properties whenever there's a chance. They can be of many types:
  - `RenderFlex`
  - `RenderParagraph`
  - `RenderBox` ...

  During the build phase, the framework updates or creates a new type of `RenderObject` only when a `RenderObjectElement` is encountered in the element tree.
- **Element tree**. An `Element` is the link between a `Widget` and its respective `RenderObject` so it holds references inside. `Elements` are very good at comparing items and looking for changes but they don't perform rendering. They can be of two types:
  - `ComponentElement`. An element that contains other elements. It's associated to a widget that can nest other widgets inside.

    ```
    abstract class ComponentElement extends Element { ... }
    ```

  - `RenderObjectElement`. An element that takes part in painting, layout and hit testing phases.

```
abstract class RenderObjectElement extends Element { ... }
```

  The element tree is basically a series of `ComponentElement` or `RenderObjectElement`, depending on the widget they refer to. In our example, a `Container` is a `ComponentElement` because it can host other widgets inside.

- **Widget tree**. It's made up of classes extending `StatelessWidget` or `StatefulWidget`. They're used by the developer to build the UI and are not expensive to be created (much less than a `RenderObject`).

Whenever the widget tree is changed (by a state management library for example), Flutter uses the element tree to make a comparison between the new widget tree and the render tree. An `Element` is a "middle way" between a `Widget` and a `RenderObject` used to make quick comparisons needed to keep the trees updated.

1. A `Widget` is "light" and it's instantiated quickly so frequent rebuilds aren't a problem at all. Widgets are **all** immutable and that's why the state of a `StatefulWidget` is implemented in another separated class. A stateful widget itself is immutable but the state it returns can mutate.

```
class Example extends StatefulWidget {
    const Example();

    @override
    _ExampleState createState() => _ExampleState();
}

class _ExampleState extends State<Example> {
    @override
    Widget build(BuildContext context) { ... }
}
```

   The widget itself (`Example`) is immutable and so its mutable state (`_ExampleState`) is implemented in another class. A `StatelessWidget` is immutable as well.

2. A `RenderObject` is relatively "expensive" and it takes time to instantiate so it's recreated only when really needed. Most of the times they're internally modified (reusability is the key).

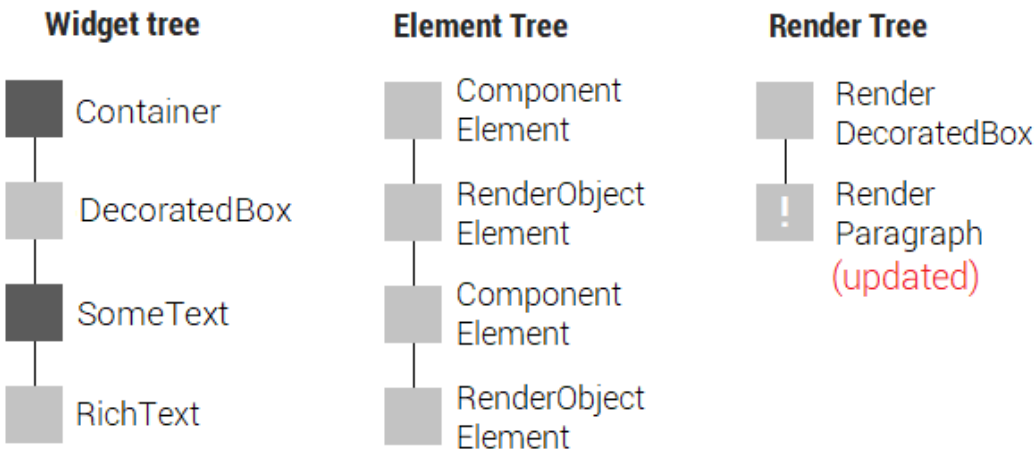
For each rebuild, Flutter traverses the entire tree looking for changes on widgets. If the type of the `Widget` changed, then it'd be removed and replaced **together** with its associated `Element` and `RenderObject`. All the 3 subtrees would also be recreated. If the `Widget` were of the same type and just some properties changed, the `Element` would stay untouched and the `RenderObject` would be updated (and **not** recreated). Let's see an example:

```
Widget build(BuildContext context) {
    return Container(
        decoration: BoxDecoration(),
        child: SomeText(
            text: "Hello"
        ),
    );
}
```

This is what we had earlier. Of course, on the first build the 3 trees are entirely created but from now on, the framework will try to recreate the render tree as less as possible. Let's say our state management library changed the text of `SomeText`.

```
Widget build(BuildContext context) {
    return Container(
        decoration: BoxDecoration(),
        child: SomeText(
            text: "Hello world!"
        ),
    );
}
```

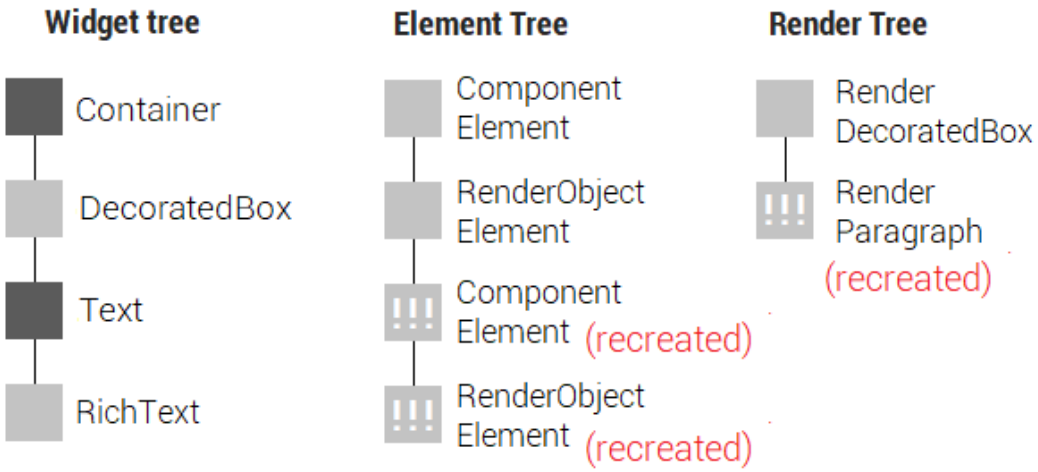
When a rebuild happens, thanks to the element tree, Flutter notices that the type is still the same (`SomeText`) but an internal property (`text`) has changed. By consequence, the associated `RenderObject` just needs an update, which is cheap.



This process is very fast because the `RenderObject` is not recreated but it's just modified. Widgets and elements are also quick to update so this is a good situation. Let's now say that our library replaces `SomeText` with Flutter's `Text` widget.

```
Widget build(BuildContext context) {
    return Container(
        decoration: BoxDecoration(),
        child: Text("Hello world!"),
    );
}
```

While traversing the tree, the framework notices again the change thanks to the element tree. In particular, this time the type of the widget is completely different so there's the need to rebuild the entire subtrees (widgets, elements and renders).



The associated `RenderObject` is not updated: it has to be entirely **recreated** because the widget has a different type and thus there's no way to reuse the old instance. In summary, Flutter relies on 3 trees to efficiently handle the rendering and tries to reuse `RenderObjects` as much as possible. Thanks to `Elements`, the framework knows when something has changed on `Widgets`.

> ℹ The `BuildContext` parameter you see in any `build()` method basically represents the `Element` associated to the widgets. In reality, `BuildContext` objects are `Element` objects. The Flutter team created `BuildContext` to avoid the direct interaction with `Element`, which should be used by the framework and not by you.

The render tree is the one that actually takes care of painting elements to the UI. The widget tree is manually built by you, the developer. The element tree is maintained by the framework to decide whether it's time to update or recreate a `RenderObject`.

3.1.2 Foreign Function Interface

Thanks to the `dart:ffi` library, also known as **F**oreign **F**unction **I**nterface, your Dart code can directly bind to native APIs written in C. FFI is very fast because there's no serialization required to pass data since calls are made to dynamically or statically linked libraries. Here's a an example:

```
// demo.h
void print_demo() {};
```

```
// demo.c
#include <stdio.h>
#include "demo.h"

void print_demo() {
    printf("Dart FFI demo!");
}

int main() {
    print_demo();
    return 0;
}
```

We're going to call `void print_demo()` written in C inside a Dart app thanks to FFI. To keep the example simple, we assume that every file is in the same folder and the following Dart code is all inside `main.dart`. Let's start with the fundamentals:

```
import "dart:ffi" as FFI;

// Signature of the function in C
typedef print_demo_c = FFI.Void Function();
// Signature of the function in Dart
typedef PrintDemo = void Function();
```

The first `typedef` uses FFI to represent the signature of the C function we're going to call. It's basically used to represent the C function into its Dart counterpart, identified by `PrintDemo`. Of course, you have to declare two `typedef` whose signatures match.

```
import "dart:ffi" as FFI;

typedef print_demo_c = FFI.Void Function();
typedef PrintDemo = void Function();

void main() {
    // Open the library
    final path = "demo_lib.dll"; // On Windows
    final lib = FFI.DynamicLibrary.open(path);

    // Create a "link" from C to Dart
```

```
    final PrintDemo demo = lib
        .lookup<FFI.NativeFunction<print_demo_c>>('print_demo')
        .asFunction();

    // Call the function
    demo();
}
```

In general, when working with FFI you always have to create two `typedef`: one for the "C side" and the other for the "Dart side". When building the C code, various files are created but you're only interested in the one with the following extension: `.dll` on Windows, `.so` on Linux and `.dylib` on macOS. On Windows, be sure that your compiler properly exports to the DLL the functions Dart has to use.

```
int sum(int a, int b) {
    return a + b;
}
```

The above code can easily be used by Dart in the same way we did earlier in the demo function. Inside `dart:ffi` you'll find many types representing the C primitive ones, such as `Int32`, `Double`, `UInt32`, `Handle` and much more.

```
typedef sum_c = FFI.Int32 Function(FFI.Int32 a, FFI.Int32 b);
typedef Sum = int Function(int a, int b);
```

Check out the official documentation [4] for some nice examples on how to interact with `structs`, strings and SQLite databases.

3.1.3 Method channels

Available only for mobile and desktop, *method channels* allow Dart to call platform-specific code of your hosting app. Data are serialized from Dart and then deserialized in Java, Kotlin, Swift or Objective-C. Look how easy it is:

```
const channel = MethodChannel("person");
final name = await channel.invokeMethod<String>("getPersonName");
print(name); // 'name' is a regular Dart string
```

As example, let's say the above code is going to call the `getPersonName(): String` function declared in a native Android app written in Kotlin. There's a similar setup to do in the native

[4] https://api.dart.dev/stable/2.9.2/dart-ffi/dart-ffi-library.html

part as well but it's very simple to understand:

```
// Initialization
val channel = MethodChannel(flutterView, "person")
channel.setMethodCallHandler {call, result ->
    when (call.method) {
        "getPersonName" -> result.success(getPersonName())
        else -> result.notImplemented()
    }
}

// This function is defined somewhere
fun getPersonName(): String {
    return "Alberto"
}
```

In both cases, the `MethodChannel` instance has to be created with the same name (`"person"`) otherwise the "link" between Dart and Kotlin won't work. The name of the function matches the actual name on the native side just for convenience but it's not required. With `invokeMethod<T>()` you can also pass parameters in case the function were asking for some. For example, if you called this in Dart...

```
const channel = MethodChannel("random");
final random = await channel.invokeMethod<int>("getRandom", 60);
```

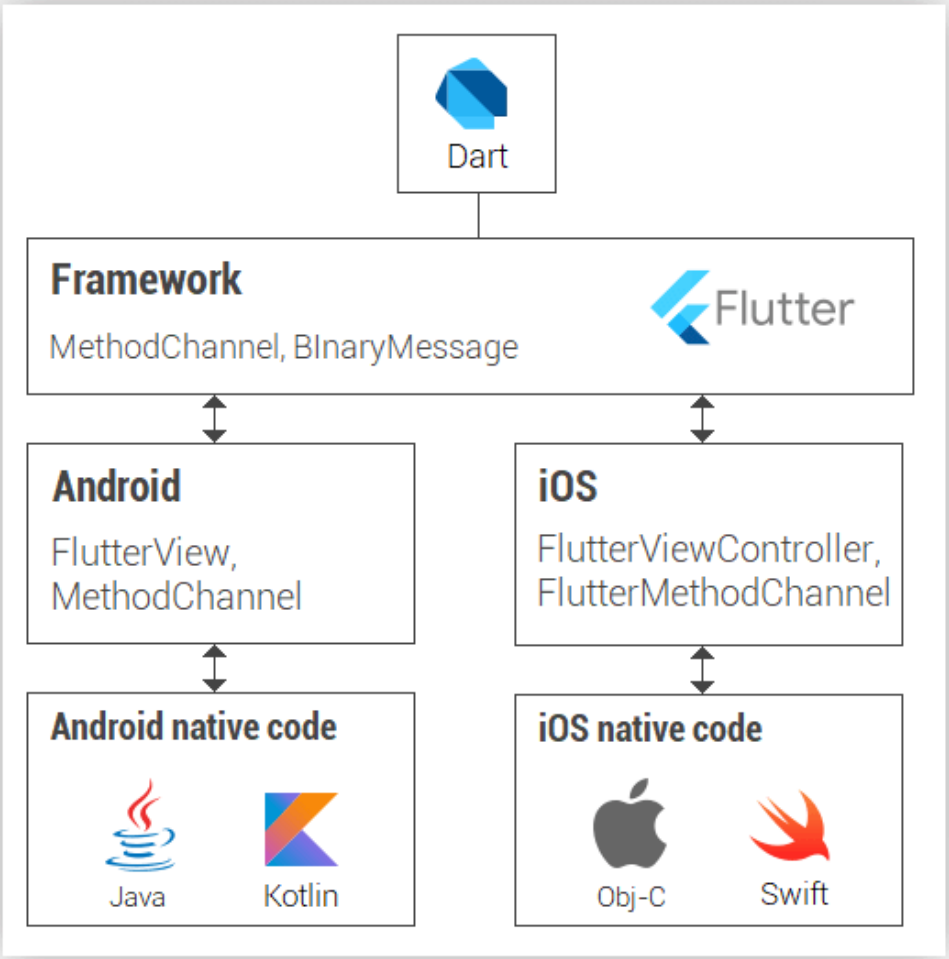
... it would mean that you're expecting a method on the native language called `getRandom` asking for a single integer parameter. This example instead is written in Swift but the logic is always the same (just a different syntax):

```
// Initialization
let chl = FlutterMethodChannel(name: "random", binaryMessenger: flutterView)
chl.setMethodCallHandler {
    (call: FlutterMethodCall, result: FlutterResult) -> Void in
        switch (call.method) {
            case "getRandom": result(getRandom(call.arguments as! Int))
            default: result(FlutterMethodNotImplemented)
        }
}

// This function is defined somewhere
```

```
func getRandom(value: Int) -> Int {
    return Int.random(in: 0...value);
}
```

Thanks to `call.arguments` you access the argument passed via method channel which could be, for example, a primitive type or a map. A `MethodChannel` is a common interface for both Dart and the other native language that allows Flutter to send/receive messages. This is a scheme of how method channels are implemented:



In the native code, method channels must be called in the main thread and not in a background

one (in Android, the "main" thread is actually called **UI** thread). To sum up, the communication flow works like this:

1. Flutter sends a message to the iOS or Android part of the app using a method channel;
2. the underlying system listens on the method channel and so the message is received;
3. one or more platform-specific APIs are called, using the native programming language;
4. a response is sent back to the client (Flutter) which processes the result.

You can't do the same with FFI because there are no libraries to be linked and data need serialization/deserialization: method channels work differently and require a native implementation as well.

Index

Special thanks to Felix Angelov, Matej Rešetár, Rémi Rousselet, Matthew Palomba and Alfred Schilken