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 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

[1] For example, it doesn't use the system's OEM widgets
[2] https://pub.dev/packages/provider

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.

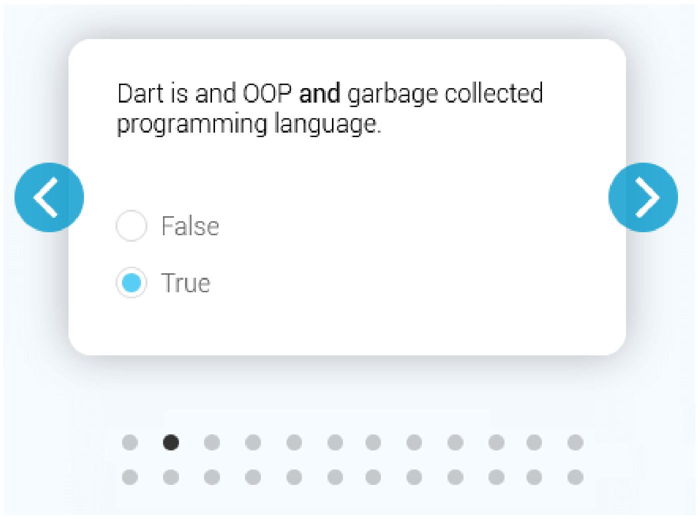
Special thanks to my friends Matthew Palomba and Alfred Schilken which carefully read the book improving the style and the quality of the contents.

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

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.



Dart is and OOP and garbage collected programming language.

False

True

[3]https://fluttercompletereference.com

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:

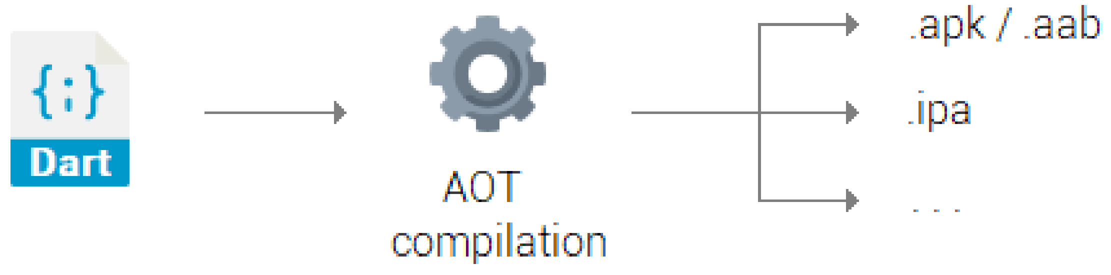
- **Stand-alone**. In the same way as a Java program can't be run without the Java Virtual Machine (JVM), a stand-alone Dart program can't be executed without the Dart Virtual Machine (DVM). There's the need to download and install the DVM which to execute Dart in a command-line environment. The SDK, 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.

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

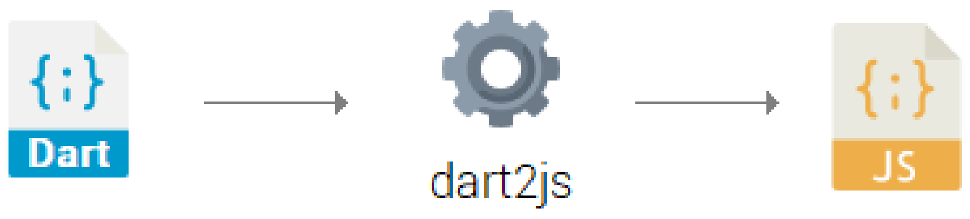


With AOT there is **NO** need to have the DVM installed because at the end you get a single binary file (an *.apk* or *.aab* for Android, an *.ipa* for iOS, an *.exe* for Windows...) that can be executed.

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

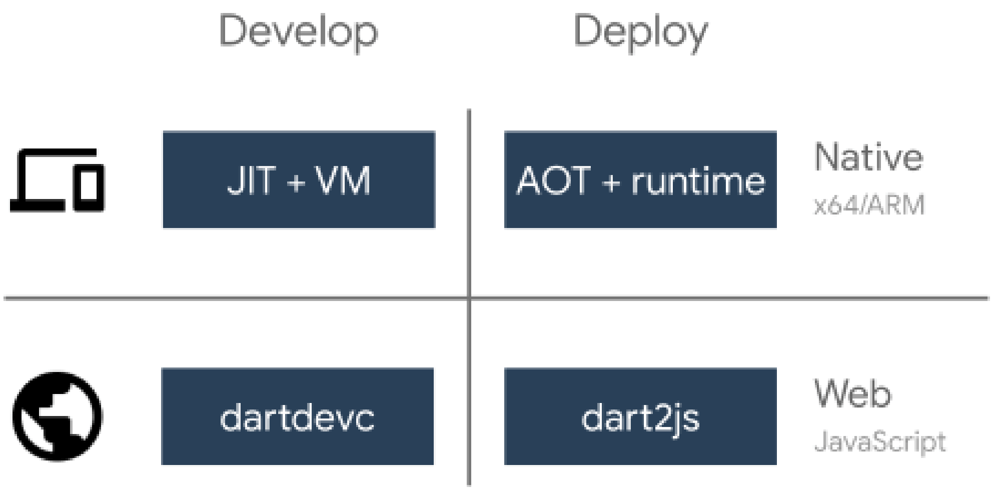
- **Web**. Thanks to the `dart2js` tool, your Dart project can be "transpiled" into fast and compact JavaScript code. By consequence Flutter can be run, for example, on Firefox or Chrome and the UI will be identical to the other platforms.



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!

[5] https://angulardart.dev/

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.



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.

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

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

packages created by the Dart team, the Flutter team or community users like you.

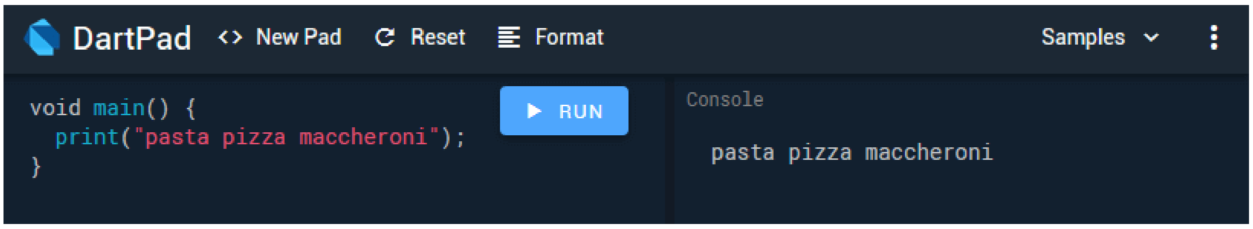
If you head to https://pub.dev you will find an endless number of packages for any purpose: I/O handling, XML serialization/de-serialization, localization, SQL/NoSQL database utilities and much more.

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).



```dart
void main() {
  print("pasta pizza maccheroni");
}
```

```
pasta pizza maccheroni
```

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

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

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.

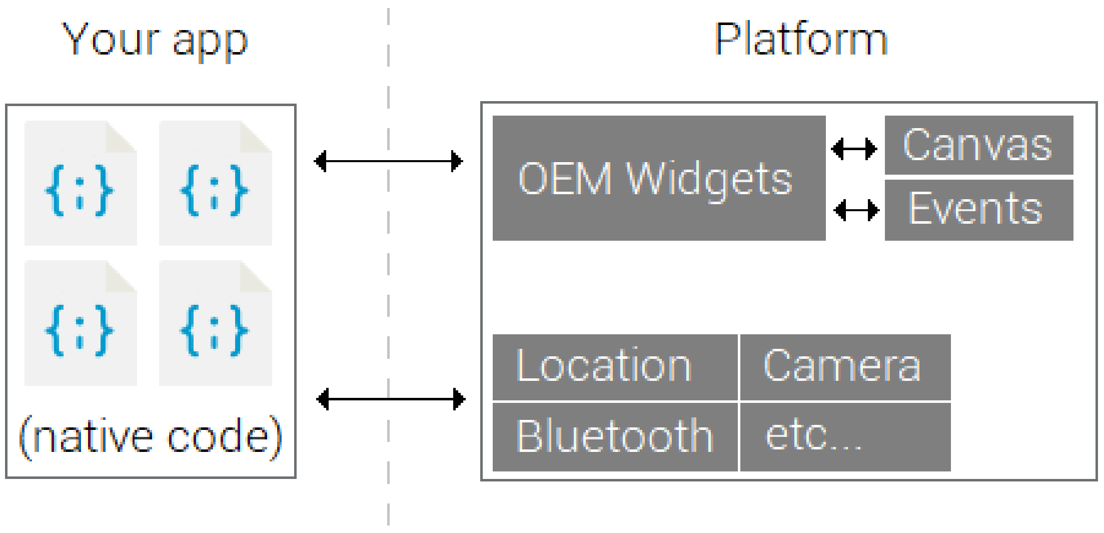
1.3 Intorduction to Flutter

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

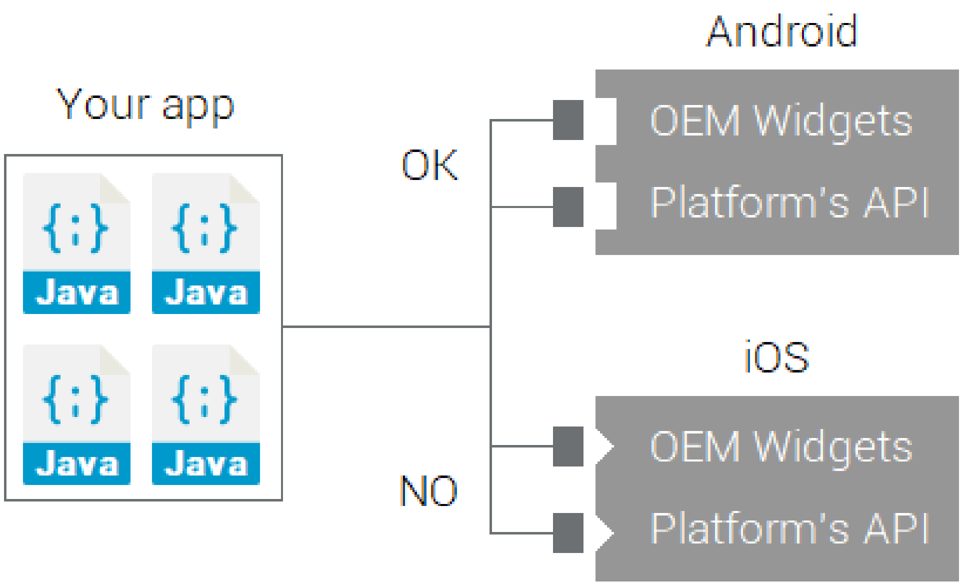
This picture shows how a native app interacts with the OS, whether it's been written in Kotlin (or Java) for Android or Swift (or Objective-C) for iOS. We're going to use these 2 platforms as examples in this section.



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.

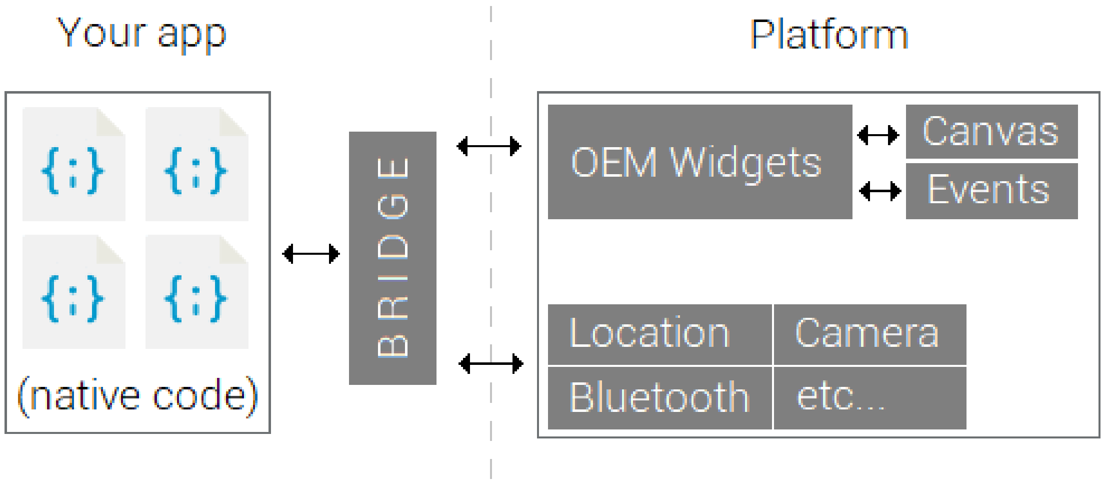
ARM is a family of RISC microprocessors (32 and 64 bit) widely used in embedded systems. It dominates the mobile world thanks to its qualities: low costs, good heat dissipation and a longer battery life thanks to a low power consumption.



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:

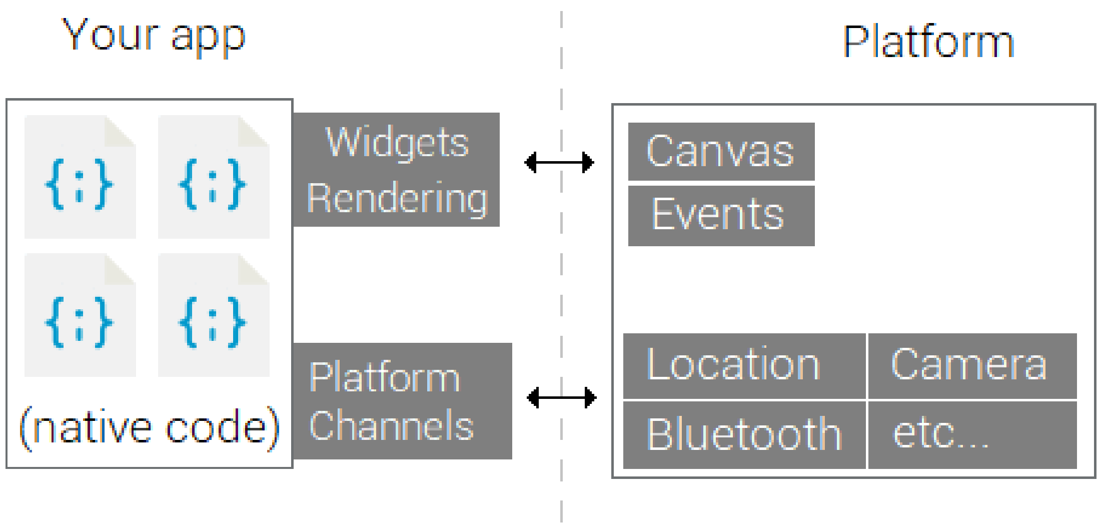


1. The bridge **always** exposes the same interface to the app so that it doesn't care anymore about the OS it's running on;
2. The bridge has an implementation of OEMs and APIs for each platform to allow the app to correctly work in many environments. In this way, you have a native app in the sense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

ℹ 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.

[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

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 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.

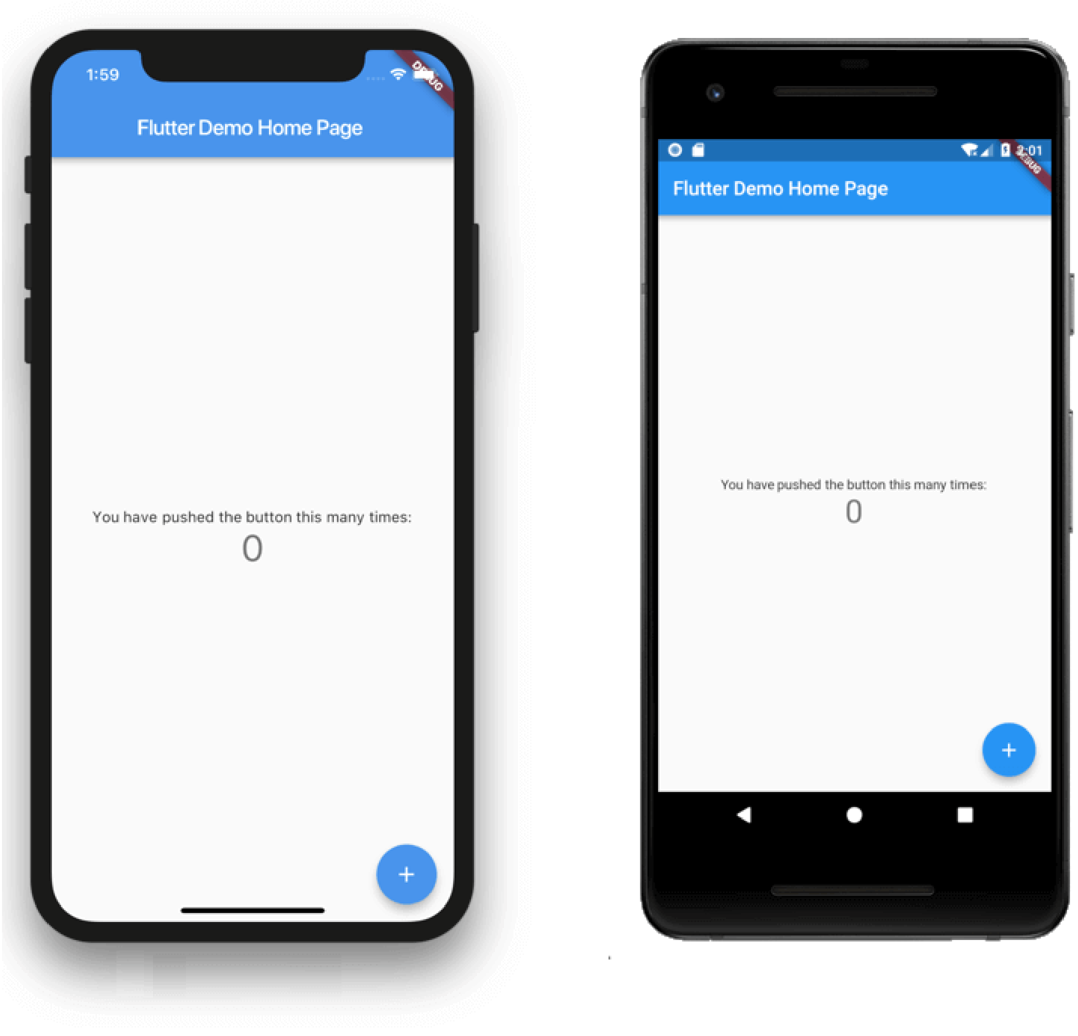
1.3.3 Hello world

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.

```dart
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.


1:59
DEBUG
Flutter Demo Home Page
You have pushed the button this many times:
0
+
2:01
DEBUG
Flutter Demo Home Page
You have pushed the button this many times:
0
+

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();
  ```

  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);
  ```

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

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).

```dart
// 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.

```dart
// 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:

```dart
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.

[2]https://dart.dev/guides/libraries/library-tour#strings-and-regular-expressions

```
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();

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.

[3] https://dart.dev/guides/language/effective-dart/usage#prefer-using-interpolation-to-compose-strings-and-values

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>`.

ℹ `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

3 | Control flow and functions

3.1 If statement

This is probably the most famous statement of any programming language and in Dart it works exactly as you would expect. The `else` is optional and it can be omitted when not needed. You can avoid using brackets in case of one-liner statements.

```
void main() {
    final random = 13;

    if (random % 2 == 0)
        print("Got an even number");
    else
        print("Got an odd number");
}
```

Conditions must be boolean values. In C++ for example you can write `if (0) {...}` where zero is evaluated to `false` but in Dart it doesn't compile; you have to write `if (false) {...}`.

3.1.1 Conditional expressions

In Dart there are two shorthands for conditional expressions that can replace the if-else statement:

- `valueA ?? valueB`. If `valueA` is non-null, `valueA` is returned; otherwise `valueB` is evaluated and then returned. If the definition is too verbose, you can understand this syntax by looking at the following example.

  ```
  String? status;  // This is null

  // isAlive is a String declared somewhere before
  if (status != null)
      isAlive = status;
  else
      isAlive = "RIP";
  ```

  Basically we want to know whether `status` is null or not and then decide the proper value to assign. The same logic can be expressed in another, more concise way:

```
String? status; // This is null
String isAlive = status ?? "RIP";
```

In the example `isAlive` doesn't need to be nullable as it's guaranteed to be initialized with a string. The `??` operator automatically checks if `status` is null and decides what to do:

- `status` is **not** null: return `status`;
- `status` is null: return the provided "default value" at the right of `??`

It's a very helpful syntax because it guarantees that a variable is properly initialized avoiding unwanted operations with `null`.

- `condition ? A : B;`. If `condition` is true A is returned, otherwise you get B. It's a pretty common pattern among modern languages so you might already be familiar with it.

```
String status;

if (correctAns >= 18)
    status = "Test passed!";
else
    status = "You didn't study enough..."
```

If it looks a bit too verbose, you can rewrite the logic in a more concise way:

```
String status = (correctAns >= 18) ?
    "Test passed!" : "You didn't study enough...";
```

We could call this the "*shorter if*" syntax in which you replace the `if` with the question mark (?) and the `else` with the colon (:). You can omit parenthesis.

3.1.2 Good practices

Simple boolean expressions are easy to read but complicated ones might require documentation and might also not fit well inside a single `if` statement like the following:

```
if ( (A && B || C && !A) || (!(A && C) || B) ) { ... }
```

You might get a headache while trying to figure out what's going on and there are also no comments at all. In such cases, you probably want to make the code more readable by splitting the conditions:

```
final usefulTestName1 = A && B || C && !A;
final usefulTestName2 = !(A && C)

if (usefulTestName1 || usefulTestName2 || B) { ... }
```

For sure it's more understandable and another programmer, or yourself in the future, will be very grateful. We also recommend to not underestimate the usefulness of variable names.

The point is that you have to keep expressions short and easy to read. Break down long conditions into smaller pieces and give the variables good names to better understand what you want to check.

We also recommend the usage of the *short if* syntax only when there's one condition or at maximum two short ones. The longer the line is the harder it is to understand.

3.2 switch statement

When you have a series of cases to take into account, instead of using a long chain of if-else*s* you should go for the `switch` statement. It can compare many types:

1. compile-time constants
2. enums
3. integers
4. strings
5. classes

Classes must not override == if they want to be compared with this statement. At the bottom there's a `default` label used as fallback if none of the previous cases matches the item being compared.

```
enum Status { Ready, Paused, Terminated }

void main() {
    final status = Status.Paused;

    switch (status) {
        case Status.Ready:
            run();
            break;
        case Status.Paused
            pause();
            break;
        case Status.Terminated
            stop();
            break;
        default
            unknown();
    }
}
```

If the body of the `case` is **NOT** empty you must put a `break` otherwise your code won't compile.

When you just want a fall-through to avoid code-replication, leave the body empty. Here's a few examples:

- This code is not going to compile because the first `case` has a body, containing `start()`, but there isn't a `break`.

  ```
  switch (status) {
      case Status.Ready:
          start();
          //missing "break;" here
      case Status.Paused
          pause();
          break;
      }
  }
  ```

- This code instead is fine because the `case` doesn't have a body; the method `pause()` is going to be called when `status` is ready or paused.

  ```
  switch (status) {
      case Status.Ready:
      case Status.Paused
          pause();
          break;
      }
  }
  ```

  The above code is equivalent to...

  ```
  switch (status) {
      case Status.Ready:
          pause();
          break;
      case Status.Paused
          pause();
          break;
      }
  }
  ```

  ... but you should avoid code duplication which is always bad in terms of code maintenance. When you have two or more cases that must execute the same action, use the fall-through approach.

3.3 for and while loops

The iteration with a `for` loop is the most traditional one and doesn't need many explanations. You can omit brackets in case of one-liner statements. The index cannot be nullable (using `int? i = 0` doesn't work).

```
for(var i = 0; i <= 10; ++i)
    print("Number $i");
```

As you'd expect, the output prints a series of "*Number (i)*" in the console. An equivalent version can be written with a classic `while` loop:

```
var i = 0;

while (i <= 10) {
    print("Number $i");
    ++i;
}
```

The language also has the `do while` loop that always executes at least **one** iteration because the condition is evaluated only at the end of the cycle.

```
var i = 0;

do {
    print("Number $i");
    ++i;
} while (i <= 10)
```

The difference is that the `while` evaluates the condition at the beginning so the loop could never start. The `do while` instead runs at least once because the condition check is placed at the end. If you wanted to alter the flow of the loop you could use:

- `break`. It immediately stops the loop in which it is called. In case of nested loops, only the one whose scope contains `break` is stopped. For example:

  ```
  for (var i = 0; i <= 3; ++i) {     // 1.
      for(var j = 0; j <= 5; ++j) { // 2.
          if (j == 5)
              break;
      }
  }
  ```

  In this case only loop *2* is terminated when `j` is 5 but loop *1* executes normally until `i` reaches 3. In practical terms, we can say `break` stops only 1 loop.

- `continue`. It skips to the next iteration and, like we've seen before, in case of nested loops it does the jump only for the loop containing it, not the others.

3.3.1 for-in loop

There are some cases in which you want to completely traverse a string or a container and you don't care about the index. Look at this very easy example:

```
final List<String> friendsList = ["A", "B", "C", "D", "E"];

for(var i = 0; i < friendsList.length; ++i)
    print(friendsList[i]);
```

That's perfectly fine but you're using `i` just to retrieve the element at the *i*-th position and nothing more. There are no calculations based on the index as it's just used to traverse the list. In such cases you should do the following:

```
List<String> friendsList = ["A", "B", "C", "D", "E"];

for(final friend in friendsList)
    print(friend);
```

This version is less verbose and clearer. You're still traversing the entire list but now, instead of the index `i`, you have declared `final friend` that represents an item at each iteration.

3.4 Assertions

While writing the code you can use assertions to throw an exception [1] if the given condition evaluates to false. For example:

```
// the method returns a json-encoded string
final json = getJSON();

// if length > 0 is false --> runtime exception
assert(json.length > 0, "String cannot be empty");

// other actions
doParse(json);
```

The first parameter of `assert` must be an expression returning a boolean value. The second parameter is an optional string you can use to tell what's gone wrong; it will appear in the IDE error message window if the condition evaluates to `false`.

ℹ In release mode, every `assert` is **ignored** by the compiler and you're guaranteed that they won't interfere with the execution flow. Assertions work only in debug mode.

[1]Exceptions will be discussed in detail in chapter 5

When you hit **Run** on Android Studio or VS Code your Flutter app is compiled in debug mode so assertions are enabled.

3.5 Good practices

Sometimes you have to implement a complicated algorithm and you don't want to make it even more complex by writing code hard to understand. Here's what we recommend.

- Try to always use brackets, even if they can be omitted, so that you can avoid unexpected behaviors. Imagine you had written this code...

  ```
  // Version 1
  if ("A" == "A")
      if ("B" == "B")
          print("Oh well!");
  else
      print("Oops...");
  ```

  ... but in reality you wanted to write this, with a better indentation:

  ```
  // Version 2
  if ("A" == "A")
      if ("B" == "B")
          print("Oh well!");
      else
          print("Oops...");
  ```

  There's a high possibility that, at first glance, in version 1 you associated the `else` to the first `if` but it'd be wrong! While it may seem obvious, in a complex architecture with thousands of lines you might misread and get tricked.

- When you have to traverse an entire list and you **don't** care about the position in which you are during the iteration, use a `for-in` loop. As we've already said, it's less verbose and so more understandable.

Use assertions, in particular when you create Flutter apps, to control the behavior of your software. Don't remove them when you're ready to deploy the code to the production world because they will be automatically discarded.

3.6 The basics of functions

Functions in Dart have the same structure you're used to see in the most popular programming languages and so you'll find this example self-explanatory. You can mark a parameter with `final` but in practice it does nothing.

```
bool checkEven(int value) {
    return value % 2 == 0
}
```

When the body of the function contains only one line, you can omit the braces and the `return` statement in favor of the "arrow syntax". It works with **expressions** and not with statements.

```
// Arrow syntax
bool checkEven(int value) => value % 2 == 0;

// Arrow syntax with method calls
bool checkEven(int value) => someOtherFunction(value);

// Does NOT work
bool checkEven(int value) => if (value % 2 == 0) ... ;
```

The second example is a conditional statement so it doesn't work with the arrow syntax. The first example instead is still a condition but it's written as **expression** and so it works fine.

```
// 1. This function does not return a value
void test() {}

// 2. No return type so this function returns dynamic. Don't do this.
test() {}
```

When you don't need a function to return a value, simply make it `void` like you'd do in Java for example. If you omitted the return type like in (*2.*), the compiler would automatically append `return dynamic` at the end of the body.

```
void test() => print("Alberto");
```

Interestingly, if you have a void function with an one-liner body, you can use the arrow syntax. The function doesn't return anything because of the `void` but you're allowed to do it anyway.

ℹ Try to **always** specify the return type or use `void`. Avoid ambiguity; you could avoid the return type for laziness (you just don't want to write `void`) but someone else could think you're returning `dynamic` on purpose.

3.6.1 The Function type

Dart is truly an OOP language because even functions are objects and the type is called... `Function`! A return type is required while the parameters list is optional:

```
// Declare a function
bool checkEven(int value) => value % 2 == 0;

void main() {
  // Assign a function to a variable
  bool Function(int) checker = checkEven;

  // Use the variable that represents the function
  print(checker(8)); // true
}
```

It's nothing new: you're just writing a type (`bool Function(int)`), its name (`checker`) and then you're assigning it a value (`checkEven`). You may find this declaration a bit weird because it's made up of many keywords but it's a simple assignment. This is a comparison to clarify the idea:

- `28`: It's an integer and its type is `int`.
- `"Pizza"`: It's a string and its type is `String`.
- `bool checkEven(int value) => ...`: It's a function and its type is `bool Function(int)`.

This particular syntax is very expressive; you have to declare the return type and the **exact** order of the type(*s*) it takes. In other words, signatures must match. If you think it's too verbose, you can use the typical automatic type deduction you're getting used to see:

```
bool checkEven(int value)  => value % 2 == 0;

void main() {
  final checker1 = checkEven;
  var checker2 = checkEven;

  print(checker1(8)); // true
  print(checker2(8)); // true
}
```

Both `var` and `final` will be evaluated to `bool Function(int)`. There's still something to say about this type but you'll have to wait until the next chapter where we'll talk about classes and the special `call()` method.

When you declare a variable you can only write `Function(int) name` without the return type. However, automatic type deduction is generally the best choice because it reduces a lot the verbosity.

It might not seem very useful and we'd agree with you because there is no usage context, at the moment. When you'll arrive at part 2 of the book you'll see that the `Function` type is super handy in Flutter because it's used to create "function callbacks".

3.7 Anonymous functions

So far you've only seen *named functions* such as `bool checkEven(int value)` where *checkEven* is the name. Dart gives you the possibility to create nameless functions called anonymous functions.

```
void main() {
    bool Function(int) isEven = (int value) => value % 2 == 0;

    print(isEven(19)); //false
}
```

This syntax allows you to create functions "*on the fly*" that are immediately assigned to a variable. If you want an anonymous function with no parameters, just leave the parenthesis blank `()`. Of course you can use `final` and `var` to automatically deduce the type.

- **Single line**. You can use the arrow syntax when you have one-liner statements. This example declares a function with no parameters that returns a double.

  ```
  final anon = () => 5.8 + 12;
  ```

- **Multiple lines**. Use brackets and `return` when you have to implement a logic that's longer than one line.

  ```
  final anon = (String nickname) {
      var myName = "Alberto";
      myName += nickname;

      return myName;
  };
  ```

We recommend to always write down the type of the parameter even if it's not required by the compiler. You can decide whether the type has to appear or not. Using `final` and `var` is allowed but it doesn't make much sense.

```
String Function(String) printName = (String n) => n.toUpperCase();
String Function(String) printName = (final n) => n.toUpperCase();
String Function(String) printName = (var n) => n.toUpperCase();
String Function(String) printName = (n) => n.toUpperCase();
```

Any variant compiles with success but none of them is the best option, the decision is up to your discretion. Before moving on, we're going to show a simple scenario you'll encounter many times in Flutter.

```
// 1.
void test(void Function(int) action) {
    // 2.
    final list = [1, 2, 3, 4, 5];
```

```
        // 3.
        for(final item in list)
            action(item);
    }

    void main() {
        // 4.
        test(
            // 5.
            (int value) { print("Number $value"); }
        );
    }
```

The `action` parameter commonly known as *callback* because it executes an action given from the outside.

1. This function doesn't return a value because of the `void`. The parameter, called `action`, accepts a `void` function with a single integer value.
2. It's a simple list of integer values
3. We iterate through the entire list and, for each item, we call the function.
4. `test(...);` is how you normally call a function
5. This is an anonymous function returning nothing (`void`) and asking for a single integer parameter.

The flexibility of callbacks lies on the fact that you can reuse the same function `test()` with different implementations. The caller doesn't care about the body of the anonymous function, it just invokes it as long as the signature matches.

```
    // The same method (test) outputs different values
    // because anonymous functions have different bodies
    test( (int value) => print("$value") );
    test( (int value) => print("${value + 2}") );
```

You will often encounter the `forEach()` method on collections, which accepts a callback to be executed while elements are traversed. Again, the same function (`forEach()`) is reused multiple times regardless the implementation (thanks to callbacks).

```
void main() {
    // Declare the list
    final list = [1, 2, 3, 4, 5];
    // Iterate
    list.forEach((int x) => print("Number $x"));
}
```

This is an even shorter way that doesn't use a `for-in` loop. You pass an anonymous function to the method and it executes the given action for every item. Pay attention because the documentation suggests to avoid using anonymous functions in `forEach()` calls.

ℹ A very handy feature you'll see very often is the possibility to put an underscore when one or more parameters of a function aren't needed. For example, in Flutter the `BuildContext` object is often given as a callback param but it's not always essential.

```
builder: (BuildContext context) {
    return Text("Hello");
}
```

Since the variable `context` isn't used, but it must be there anyway to match the method signature, you can use an underscore to "hide" it:

```
builder: (_) {
    return Text("Hello");
}
```

It's less code for you to write and the reader focuses more on what's really important. In case of multiple values that you don't use, just chain a series of underscores.

```
builder: (_, value, __) {
    return Text("$value");
}
```

3.8 Optional parameters

In Dart function parameters can be **optional** in the sense that if you don't provide them, the compiler will assign `null` or a default value you've specified.

3.8.1 Named parameters

In the simplest case, a function can have optional parameters whose names must be explicitly written in order to be assigned. Pay attention to null-safety in case you don't plan to give the variables a default value.

Declaration

```
void test({int? a, int? b}) {
    print("$a");
    print("$b");
}
```

Calling

```
void main() {
    // Prints '2' and '-6'
    test(a: 2, b: -6);
}
```

When calling a function with optional named parameters, the order **doesn't** matter but the names of the variables names must be explicit. For example, you could have called `test(b: -6, a: 2);` and it would have worked anyway. When a parameter is missing, the default value is given:

Declaration

```
void test({int? a, int? b}) {
    print("$a");
    print("$b");
}
```

Calling

```
void main() {
    // Prins '2' and 'null'
    test(a: 2);
}
```

Calling `test(a: 2);` initializes only `a` because `b`, which is omitted, is set to `null` by the compiler. `null` is the default value of nullable types. You can manually give a default value to an optional named parameter just with a simple assignment:

Declaration

```
void test({int? a, int b = 0}) {
    print("$a");
    print("$b");
}
```

Calling

```
void main() {
    // Prints '2' and '0'
    test(a: 2);
}
```

Note that `b` doesn't need to be nullable anymore thanks to the default value. In Dart 2.9 (an lower) nnbd was not enabled so you were able to successfully compile this code, which initializes both `a` and `b` to `null`:

Declaration

```
// Dart 2.9 and lower
void test({int a, int b}) {
    print("$a");
    print("$b");
}
```

Calling

```
// Dart 2.9 and lower
void main() {
    // Prints 'null' and 'null'
    test();
}
```

The `required` modifier, introduced in Dart 2.10 with nnbd, forces an optional parameter to be set. You won't be able to compile if a required parameter is not used when calling the function.

```
void test({int a = 0, required int b}) {
    print("$a");
    print("$b");
}

void main() {
     test(a: 5, b: 3); // Ok
     test(a: 5);       // Compilation error, 'b' is required
 }
```

Even if you had written `required int? b` you'd have to assign `b` anyway because it's required. Version 2.9 of Dart and lower didn't have this keyword: you had to use instead an *annotation* which just produced a warning (and not a compilation error) by default.

```
// Dart 2.9 and lower
void test({int a = 0, @required int b}) {...}
```

In Flutter, for a better readability, some methods only use named optional params together with `required` to force the explicit name in the code.Y ou can mix optional named parameters with "classic" ones:

Declaration

```
void test(int a, {int b = 0}) {
    print("$a");
    print("$b");
}
```

Calling

```
void main() {
    // Prints '2' and '3'
    test(2, b: 3);
}
```

Optional parameters must stay at the end of the list.

```
// it compiles
void test(int a, {int? b}) { }

// it doesn't compile
void test({int? a}, int b) { }
```

3.8.2 Positional parameters

You can also use optional parameters without being forced to write down the name. Optional positional parameters follow the same rules we've just seen for named params but instead of using curly braces (`{ }`) they're declared with square brackets (`[ ]`).

Declaration

```
void test([int? a, int? b]) {
    print("$a");
    print("$b");
}
```

Calling

```
void main() {
    // Prints '2' and '-6'
    test(2, -6);
}
```

All the examples we've made for named parameters also apply here. They really have the same usage but the practical difference is that, in this case, the name of the parameter(s) doesn't have to be written in the function call.

3.9 Nested functions

The language allows you to declare functions inside other functions visible only within the scope in which they're declared. In other words, nested functions can be called only inside the function containing them; if you try from the outside, you'll get a compilation error.

From a practical side, the **scope** is the "area" surrounded by two brackets `{ }`

This example shows how you can nest two functions, where `testInner()` is called "**outer** function" and `randomValue()` is called "**inner** function".

```
void testInner(int value) {
    // Nested function
    int randomValue() => Random().nextInt(10);

    // Using the nested funcion
    final number = value + randomValue();
    print("$number");
}
```

As we've just seen, functions are types in Dart so a "nested function" is nothing more than a `Function` type assignment. Given this declaration, we're able to successfully compile the following:

```
void main() {
    // testInner internally calls randomValue
    testInner(20);
}
```

An error is going to occur if we try to directly call `randomValue` from a place that's not inside the scope of its **outer** function.

```
void main() {
    // Compilation error
    var value = randomValue();
}
```

3.10 Good practices

Following the official Dart guidelines [2] we strongly encourage you to follow these suggestions in order to guarantee consistency with what the community recognizes as a good practice.

- Older versions of Dart allow the specification of a default value using a colon (:); don't do it, prefer using =. In both cases, the code compiles successfully.

  ```
  // Good
  void test([int a = 0]) {}

  // Bad
  void test([int a : 0]) {}
  ```

  The colon-initialization *might* be removed in the future.

- When no default values are given, the compiler already assigns `null` to the variable so you don't have to explicitly write it.

  ```
  // Good
  void test({int? a}) {}

  // Bad
  void test({int? a = null}) {}
  ```

  In general, you should **never** initialize nullables with `null` because the compiler already does that by default.

Dart gives the possibility to write only the name of a function, with no parenthesis, and automatically pass proper parameters to it. It's a sort of "method reference". We are going to convince you with this example:

```
void showNumber(int value) {
    print("$value");
}

void main() {
    // List of values
    final numbers = [2, 4, 6, 8, 10];

    // Good
    numbers.forEach(showNumber);

    // Bad
```

[2] https://dart.dev/guides/language/effective-dart/usage#functions

```
        numbers.forEach((int val) { showNumber(val); });
    }
```

The bad example compiles but you can avoid that syntax in favor of a shorter one.

- The `forEach()` method asks for a function with a single integer parameter and no return type (`void`).
- The `showNumber()` function accepts an integer as parameter and returns nothing (`void`).

The signatures match! If you pass the function name directly inside the method, the compiler automatically initializes the parameters. This **tear-off** is very useful and you might already have seen it somewhere else under the name of "method reference" (Java).

3.11 Using typedefs

The `typedef` keyword simply gives another name to a function type so that it can be easily reused. Imagine you had to write a callback function for many methods:

```
    void printIntegers(void Function(String msg) logger) {
        logger("Done.");
    }

    void printDoubles(void Function(String msg) logger) {
        logger("Done.");
    }
```

Alternatively, rather than repeating the declaration every time, which leads to code duplication, you can give it an alias using the `typedef` keyword.

```
    typedef LoggerFunction = void Function(String msg);

    void printIntegers(LoggerFunction logger) {
        logger("Done int.");
    }

    void printDoubles(LoggerFunction logger) {
        logger("Done double.");
    }
```

You are going to encounter this technique very often, especially in Flutter, in callbacks for classes or methods. For instance, `VoidCallback` [3] is just a function alias for a void function taking no parameters.

```
    typedef VoidCallback = void Function();
```

[3]https://api.flutter.dev/flutter/dart-ui/VoidCallback.html

In a future version of Dart, probably later than 2.10, `typedef` will also be used to define new type names. At the moment it's not possible, but in the future there will be the possibility to compile the following code:

```
typedef listMap = List<Map<int,double>>;
```

The reason is that generic types can become very verbose and so an alias could improve the readability. Currently, `typedef` only works with functions.

4 | Classes

Up to now you've seen us saying many times the claim that Dart is an OOP language and now we're finally going to prove it. There are a lot of similarities with the most popular programming language so you probably are already familiar with the concepts.

```
class Person {
    // Instance variables
    String name;
    String surname;

    // Constructor
    Person(String name, String surname) {
        this.name = name;
        this.surname = surname;
    }
}
```

This syntax is almost identical to Java, C# or C++ and that's very good: if you're going to learn the language, there's nothing you've never seen before. Some keywords might be different but the essence is always the same.

Every object is an instance of a class. Dart classes, even if it's not explicitly written in the declaration, descend from `Object` and in the next chapter you will see the benefits. In Delphi and C# as well, any class implicitly derives from `Object`.

In any class you have methods (it's the OOP way to call *functions*) which can be public or private. The keyword `this` refers to the current instance of the class. Dart has **NO** method overload so you cannot have more than a function with the same name. For this reason, you'll see how named constructors come to the help.

```
class Example {
    // Doesn't compile; you have to use different names
    void test(int a) {}
    void test(double x, double y) {}
}
```

You might be able to write this in other programming languages because methods have the same name but different signature. In Dart it's not possible, every function name (in the same class) must be unique. Before going into the details of classes, look at the *cascade notation*.

```
class Test {
  String val1 = "One";
  String val2 = "Two";

  int randomNumber() {
    print("Random!");
    return Random().nextInt(10);
  }
}
```

Given this class, you have two ways to give a value to the variables:

```
// first way, the "classic" one
test.val1 = "one";
test.val2 = "two";

// second way, using the cascade operator
test..val1 = "one"
    ..val2 = "two";
```

It's just a shorthand version you can use when there are multiple values of the same objects that has to be initialized. You can do the same even with methods but the returned value, if any, will be **ignored**. For this reason, the cascade notation is useful when calling a series of `void` methods on the same object.

```
Test()..randomNumber()
      ..randomNumber()
      ..randomNumber();
```

Here the integer returned by `randomNumber()` is discarded but the body is executed. If you run the snipped, you'll get `Random!` printed three times in the console. In case of nullable values...

```
MyClass? test = MyClass();

test?..one()
    ..two()
    ..three();
```

... the cascade notation has to start with `?..` in order to be null-checked before dereferencing. In Dart there **cannot** be nested classes.

4.1 Libraries and visibility

In the Dart world, when you talk about a "library" you're referring to the code inside a file with the .dart extension. If you want to use that particular library, you have to reference its content with the import keyword.

If you aren't sure you've understood the above statement, here's an example. Let's say there's the need to handle fractions in the form *numerator / denominator*: you are going to create a file named fraction.dart.

```
// === Contents of fraction.dart ===
class MyFraction {
    final int numerator;
    final int denominator;

    //... other code
}
```

Congratulations, you have just created the fraction library! It can be used by any other .dart file that references it via import.

```
import 'package:fraction.dart';

void main() {
    // You could have written 'new Fraction(1, 2)' but
    // starting from Dart 2.0 'new' is optional
    final frac = Fraction(1, 2);
}
```

You can also use the library keyword to "name" your library as you prefer. This keyword really just names the library as you like, nothing more, and it's not required.

```
library super_duper_fraction;

class MyFraction {
    final int numerator;
    final int denominator;
    const MyFraction(this.numerator, this.denominator);
}
```

The import directive accepts a string which must contain a particular scheme. For built-in libraries you have to use the dart: prefix followed by the library name:

```
import 'dart:math';
import 'dart:io';
import 'dart:html';
```

Everything else that doesn't belong to the Dart SDK, such as a custom library created by you or another developer in the community, must be prefixed by `package`.

```
import 'package:fraction.dart';
import 'package:path/to/file/library.dart';
```

It could happen that two different libraries have implemented a class with the same name; the only possible technique to avoid ambiguity is called "library aliases". It's a way to reference the contents of a library under a different name.

```
// Contains a class called 'MyClass'
import 'package:libraryOne.dart';
// Also contains a class called 'MyClass'
import 'package:libraryTwo.dart' as second;

void main() {
    // Uses MyClass from libraryOne
    var one = MyClass();

    //Uses MyClass from libraryTwo.
    var two = second.MyClass();
}
```

You can selectively import or exclude types using the `show` and `hide` keywords:

- `import 'package:libraryOne.dart' show MyClass;` Imports only `MyClass` and discards all the rest.
- `import 'package:libraryTwo.dart' hide MyClass;` Imports everything except `MyClass`.

At the moment, you have the basics of libraries (simple creation and usage). In chapter 23 you'll learn how to create a *package* (a collection of libraries and tools) and how to publish it at https://pub.dev.

4.1.1 Encapsulation

You may be used to hide implementation details in your classes using *public*, *protected* and *private* keywords but there's no equivalent in Dart. Every member is public by default unless you append an underscore (_) which makes it private to its library. If you had this file...

```
// === File: test.dart ===
class Test {
    String nickname = "";
    String _realName = "";
}
```

... you could later import the library anywhere:

```dart
// === File: main.dart ===
import 'package:test.dart';

void main() {
    final obj = Test();

    // OK
    var name = obj.nickname;
    // ERROR, doesn't compile
    var real = obj._realName;
}
```

The variable `nickname` is public and everyone can see it but `_realName` can be seen **ONLY** inside `test.dart`. In other words, if you put the underscore in front of the name of a variable, it is visible only within that file.

```dart
// === File: main.dart ===
class Test {
    String nickname = "";
    String _realName = "";
}

void main() {
    final obj = Test();

    // Ok
    var name = obj.name;
    // Ok, it works because 'Test' is in the same file
    var real = obj._realName;
}
```

We've moved everything in the same file: now both `Test` and `main()` belong to the **same** library and so `_realName` is not private anymore.

The same rules on package private members also apply to classes and functions. For example, `void something()` is visible from the outside while `void _something()` is private to its library.

```dart
// Inside a file called users.dart
class Users { }

class _UsersHelper { }
```

In this case, `Users` is visible while `_UsersHelper` is package-private (exactly as it happens with variables and methods).

In Dart everything is "public" by default; if you append an underscore at the front, it becomes "private". There is no way to define "protected" members or variables (where `protected` is a typical OOP keyword that makes a member or variable accessible only by subclasses of a certain type.).

4.1.2 Good practices

We strongly recommend to **NOT** put everything in a single file (or a few ones) for the following reasons:

- Dealing with thousands of lines of code containing literally everything is not good at all. Maintenance is going to be hard and you're on the good way to become a professional "*spaghetti code*" writer!
- If you placed everything in the same file, you'd expose private details of the class. We've seen that private members exist only if classes are put in separated libraries (files).

Try to have one file per class or at maximum a few classes that are closely related (they should have the same purpose). When you write a Dart program, in general you have this folder structure:

```
root
 | -- lib
 |      | -- main.dart
 |      | -- routes
 |      | -- other_folders
 | -- tests
 | -- tools
```

You'll learn throughout the chapters how to create a robust folder hierarchy. The minimal Dart/Flutter project is made up of a `lib/` folder and a `main.dart` entry point file.

4.2 Constructors

The constructor is a special function with the same name of the class and doesn't carry a return type. To invoke it, the syntax is the most common one you can imagine:

```
final myObject = new MyClass();
```

Starting from Dart 2, the keyword `new` can be omitted. It's something you're always going to do, especially while writing Flutter apps.

```
final myObject = MyClass();
```

As example, in this chapter we are creating a library to work with fractions and rational numbers. To get started, there's the need to create a file called `fraction.dart` with the following contents:

```
class Fraction {
    int? _numerator;
    int? _denominator;
    Fraction(int numerator, int denominator) {
        _numerator = numerator;
        _denominator = denominator;
    }
}
```

In this case variables must be nullables because they are initialized, by default, to `null`. The body of the constructor is called **after** the variables initialization. If you want to get rid of nullables there are two options:

1. Declare variables as `late final` to tell the compiler to not emit an error. They are going to be initialized later but anyway before being accessed for the first time ever.

    ```
    class Fraction {
        late final int _numerator;
        late final int _denominator;

        Fraction(int numerator, int denominator) {
            _numerator = numerator;
            _denominator = denominator;
        }
    }
    ```

    Because of the `final` modifier, variables cannot be changed anymore after their initialization. You could have only used `late` but variables would be mutable then:

    ```
    class Fraction {
        late int _numerator;
        late int _denominator;

        Fraction(int numerator, int denominator) {
            _numerator = numerator;
            _denominator = denominator;
        }
    }
    ```

    In both cases, members are **NOT** initialized immediately because the body of the constructor is executed after the variable initialization phase.

2. The Dart team recommends going for the "initializing formal" [1] approach as it's more readable and it initializes the variables immediately.

    ```
    class Fraction {
        int _numerator;
        int _denominator;
    ```

[1] https://dart.dev/guides/language/effective-dart/usage#do-use-initializing-formals-when-possible

```
        Fraction(this._numerator, this._denominator);
    }
```

It's just syntactic sugar to immediately assign values to members. In this case, variables initialization is executed first so no need to use nullable types or `late`. This kind of initialization happens **before** the execution of the constructor's body.

Keep in mind that constructor bodies are executed **after** the variable initialization phase. The second approach is very common and you should get used to it, even if your class only declares `final` fields.

```
class Fraction {
    final int _numerator;
    final int _denominator;

    Fraction(this._numerator, this._denominator);
}
```

If your class doesn't define a constructor, the compiler automatically adds a *default* constructor with no parameters and an empty body. With the "initializing formal" you can still declare a body to perform additional setup for the class.

```
class Fraction {
    final int _numerator;
    final int _denominator;
    late final double _rational;

    Fraction(this._numerator, this._denominator) {
      _rational = _numerator / _denominator;
      doSomethingElse();
    }
};
```

You could only write `Fraction(this._numerator)` to initialize exclusively `_numerator` but then `_denominator` would be set to `null` by the compiler. Keep in mind that you cannot have **named** optional parameters starting with an underscore.

```
// Doesn't compile
Fraction({this._numerator, this._denominator});
```

However, you can have **positional** parameters starting with an underscore:

```
// Ok but pay attention to non-nullability
Fraction([this._numerator, this._denominator]);
```

4.2.1 Initializer list

When using the initializing formal approach, the names of the variables must match the ones declared in the constructor. This could lead to an undesired exposure of some internals of the class:

```
class Test {
    int _secret;
    double _superSecret;

    Test(this._secret, this._superSecret);
}
```

What if you wanted to keep `int _secret` (private) but with a different name in the constructor? Use an initializer list! It's executed before the body and thus variables are immediately initialized. No need for nullable types or `late`.

```
class Test {
    int _secret;
    double _superSecret;

    Test(int age, double wallet)
        : _secret = age,
          _superSecret = wallet;
}
```

In this way the constructor is asking you for `age` and `wallet` but the user has no idea that internally they're treated as `_secret` and `_superSecret`. It's basically a way to "rename" internal private properties you don't want to expose.

4.2.2 Named constructors

Named constructors are generally used to implement a default behavior the user expects from your class. They are the only alternative to have multiple constructors since Dart has no method overload.

```
class Fraction {
    int _numerator;
    int _denominator;

    Fraction(this._numerator, this._denominator);

    // denominator cannot be 0 because 0/0 is not defined!
    Fraction.zero() :
      _numerator = 0,
      _denominator = 1;
}
```

At this point you can use the named constructor in your code like if it were a static method call.

```
void main() {
    // "Traditional" initialization
    final fraction1 = Fraction(0, 1);

    // Same thing but with a named constructor
    final fraction2 = Fraction.zero();
}
```

In general constructors aren't inherited by a subclass so, if they are needed across the hierarchy, every subclass must implement its own named constructor. If we had written a named constructor with a body...

```
class Fraction {
    int? _numerator;
    int? _denominator;

    Fraction.zero() {
        _numerator = 0;
        _denominator = 1
    }
}
```

... we would have had to use nullable instance variables as constructors' bodies are always executed **after** variables' initialization.

4.2.3 Redirecting constructors

Sometimes you might have a constructor that does almost the same thing already implemented by another one. It may be the case to use redirecting constructors in order to avoid code duplication:

```
Fraction(this._numerator, this._denominator);
// Represents '1/2'
Fraction.oneHalf() : this(1, 2);
// Represents integers, like '3' which is '3/1'
Fraction.whole(int val) : this(val, 1);
```

Where `Fraction.oneHalf()` is just another way to call `Fraction(1, 2)` but you've avoided code repetition. This feature is very powerful when mixed with named constructors.

4.2.4 Factory constructors

The `factory` keyword returns an instance of the given class that's not necessarily a new one. It can be useful when:

- You want to return an instance of a subclass instead of the class itself,

- You want to implement a singleton (the Singleton pattern),
- You want to return an instance from a cache.

Factory constructors are like static methods and so they don't have access to `this`. There cannot be together a factory and a "normal" constructor with the same name.

```
class Test {
    static final _objects = List<BigObject>();

    factory Test(BigObject obj) {
        if (!objects.contains(obj))
            objects.add(obj);

        return Test._default();
    }

    // This is a private named constructor and thus it can't be called
    // from the outside
    Test._default() {
        //do something...
    }
}
```

In the example, since `BigObject` requires a lot of memory and the list is very very long, we've declared *objects* as `static`. This technique is often used to save memory and reuse the same object across multiple objects of the same type.

> If the list weren't `static` (just a normal instance variable), it would be created **every** time that a Test object is instantiated. It'd be a waste of memory and a performance problem; in this way we're guaranteed that there's an unique list created only once.

In this case the factory constructor is essential because it takes care of updating the `_objects` cache. Factories are called "normally" like if they were a regular constructor:

```
// Calls the factory constructor
final a = Test();
```

4.2.5 Instance variables initialization

As we've already seen, "normal" non-nullable variables have to be initialized either via initializing formal or initializer list. In any other case, they're set to `null` because constructor bodies run after the instance initialization phase.

```
// Initializing formal
class Example {
    int _a;  // Ok - 'a' initialized by the constructor
    Example(this._a);
}

class Example {
    int _a;  // Error - 'a' not initialized
    Example(int a) {
        _a = a;
    }
}
```

If variables aren't initialized immediately and you want them to be non-nullanle, you can use the `late` modifier. It works like a "lazy initialization" because with this keyword you allow a non-nullable to be initialized later (but anyway before it gets accessed for the first time ever).

```
// For Dart 2.10 and earlier versions, 'late' does not exist so just
// remove it. Not initialized variables will be set to 'null' by default.
class Example {
    late int a;

    void printExample() {
        a = 5;
        print("$a");
        a = 2;
        print("$a");
    }
}
```

If you tried to use the variable before it gets assigned for the first time, you would get a compilation error. Always be sure to have the initialization done before using it.

```
class Example {
    late int a;

    void printExample() {
        // Compilation error
        print("$a");
        a = 5;
    }
}
```

Using `a` like above causes an error because it's accessed before being initialized. There's also the possibility to declare a `late final` variable which behaves in the same way but with the only exception that it can be assigned only once.

```
class Example {
    late final int a;

    void printExample() {
        a = 5;
        print("$a");
        //a = 6; <-- This would be an error
    }
}
```

Once `a` is set with `a = 5;` you can't re-assign it anymore because of the `final` modifier. Instead, if it were a simple `late int a;`, you could have re-assigned it multiple times.

```
late final double a = takesLongTime();
```

Thanks to the usage of the `late final` combination you can lazily initialize a variable that is going to hold a value computed from a function. You can assign it immediately or, as we've just seen, in a second moment. The function `takesLongTime()` will only be called once `a` is accessed.

4.2.6 Good practices

Following what the official documentation [2] suggests, here's some tips you should consider while writing constructors for your classes:

1. Prefer using the "initializing formal" approach rather than initializing variables directly in the body of the constructor. Doing so, you'll avoid the usage of nullable types or `late`.

2. When you use the initializing formal, the types of the variables are deduced automatically to reduce the verbosity. Omit the types because they're useless.

   ```
   // Ok but useless
   Constructor(String this.a, double this.b) {}
   // OK
   Constructor(this.a, this.b);
   ```

3. When you have an empty constructor with no body, use the semicolon instead of the empty brackets.

   ```
   // Bad
   Constructor(this.a, this.b) {}
   // Good
   Constructor(this.a, this.b);
   ```

4. Do not use the `new` keyword when creating new instances of objects. In modern Dart and Flutter code, `new` never appears.

[2] https://dart.dev/guides/language/effective-dart/usage#constructors

5. We recommend using redirecting constructors to avoid code duplication. It makes maintenance easier.
6. Try to not use the late keyword because it could lead to hard maintenance (you'd have to **manually** keep an eye on the initialization of variables). Whenever possible, initialize variables as soon as they are declared.
7. In the second part of the book we'll show a case where late or late final are required. They'll be used to initialize values inside the state of a Widget.

```
class Example extends State {
    late int value;

    @override
    void initState() {
        super.initState();
        value = 0;
    }
}
```

   In short, subclasses of State cannot define a constructor and you have to perform the initialization of the variables in the initState() method. In order do to this, late is essential.

The syntax for private constructors in Dart might seem a bit weird at first. Generally, a private constructor is used in conjunction with a factory that returns a subtype or an instance of the actual object with certain criteria.

```
class Example {
    final a;
    // Private constructor
    Example._(this.a);

    factory Example(int value) {
        final c = value * 3;
        return Example._(c);
    }
}
```

A private constructor is declared using the ._() notation. In the example, the class can still be instantiated but only because we've defined a factory. In this case...

```
final ex = Example(10);
```

... we're not calling the "normal" constructor (because it's package private) but instead the factory one.

4.3 const keyword

The const keyword can be used when you have to deal with compile-time constant values such as strings or numbers. It can automatically deduce the type.

```
// type of 'number' is int
const number = 5
// explicitly write the type
const String name = 'Alberto';

const sum = 5.6 + 7.34;
```

It's true that final and const are very similar at first glance and they also share the same syntax style. A very intuitive way to determine which one you can choose is ask yourself: is this value already well defined? is it known at compile time? Let's find out why.

- final. Use it when the value is not known at compile time because it will be evaluated/obtained at runtime. Common usages are I/O from the disk or HTTP requests. For example, this is how you read a text file from the disk (more on this in A.1):

  ```
  final contents = File('myFile.txt').readAsString();

  // const contents = File('myFile.txt').readAsString();
  // ^ does not compile!
  ```

  The compiler doesn't know in advance which is the content of myFile.txt because it will be read only when the program will be running (so *after* the compilation). For this reason, you can only use final.

- const. Use it when the value is computed at compile time, for example with integers, doubles, Strings or classes with a *constant constructor* (more on it in the next section).

  ```
  const a = 1;
  // final a = 1 -> it works as well
  ```

  If it works with const, it works also with final because anything that is const is also final.

Instance variables can only be declared as final while const can be applied in combination with the static keyword.

```
class Example {
    // OK
    final double a = 0.0;
    // NO, instance variables can only be 'final'
    const double b = 0.0;

    // OK
    static const double PI = 3.14;
```

```
        // OK but without type annotation
        static const PI = 3.14;
    }
```

The real power of `const` comes when combined with constructors and, in Flutter, it can lead to an important performance boost.

Variables and methods marked with the `static` modifier are available on the class itself and not on instances. In practice it means that you can use them without having to create an object.

```
class Example {
    static const name = "Flutter";
    static String test() => "Hello, I am $name!";
}

void main() {
    final name = Example.name;
    final text = Example.test();
}
```

Both variables are strings and they've been retrieved without creating an instance of `Example`; `static` members belong to the "class scope" and you cannot use `this`.

```
class Example {
    int a = 0;
    static void test() {
        // Doesn't compile
        final version = this.a;
        print("$version");
    };
}
```

Since you don't have to create objects to call static methods, `this` cannot work because it refers to the current instance that gets never created.

4.3.1 const constructors

In Dart you can append the `const` keyword in front of a constructor only if you're going to initialize a series of `final` (immutable) variables.

```
// Compiles
class Compiles {
    final int a;
    final int b;
```

```
        const Compiles(this.a, this.b);
    }

    // Does not compile because a is mutable (not final)
    class DoesNot {
        int a;
        final int b;
        const DoesNot(this.a, this.b);
    }
```

If your class only has final variables it's said to be an "immutable class" and you should really instantiate it with a const constructor. The compiler can perform some optimizations.

```
final example1 = const Compiles(); // (1) constant object
final example2 = Compiles();       // (2) not a constant object!
```

In example *(1)* we're calling the constant constructor but in *(2)* we're **not**. Even if your class only has constant constructors, objects can be instantiated as constants only with the const keyword. When put in front of a collection, such as a list, everything inside that container will automatically be const (if it is allowed to be constant).

```
class Test {
    // constant constructor
    const Test();
}

const List<Test> listConst = [Test(), Test()];  // (1)
final List<Test> listConst2 = [Test(), Test()];  // (2)
```

In *(1)* everything inside the list is automatically "converted" into a const value while in *(2)* it doesn't happen, meaning that the contents of `listConst2` aren't constant. The final keyword in front of a list just makes it impossible to change the reference assigned to `listConst2` but does **NOT** call the constant constructor (while example *(1)* does).

```
// Bad
const List<Test> list1 = [const Test(), const Test()]; // (1)
// Good
const List<Test> list2 = [Test(), Test()];  // (2)
```

You should avoid version *(1)* because calling `const Test()` is not necessary. Any constant collection initializes its children calling their const constructor (if any, otherwise a compilation error occurs).

4.3.2 Good practices and annotations

If you know that variables in your class will never change, you really should make them final and use a const constructor. As we've already said, constant constructors play a very important role in

Flutter because they allow "caching" on instances.

> Don't get "obsessed" by immutable classes and `const` constructors because you simply can't use them in every situation. If your instance variables cannot be `final` that's perfectly fine; the environment and the compiler are very powerful and your final product won't suffer of speed degradations due to the lack of immutability.

Any class with a constant constructor can be used as **annotation**: they're are generally put before the name of a class or method. An annotation is preceded by the "at" sign (@). In the next section, we will see that overriding methods is usually done in the following way:

```
class MySubclass extends SuperClass {
    @override
    void defineMethod() {}
}
```

The `@override` annotation does nothing in practice: it just tells the developer that `defineMethod` has been overridden. If you looked at how the override annotation is declared in the Dart SDK, you'll find simply the following:

```
// This class has a constant constructor and so it can be used as annotation
class _Override {
    const _Override();
}

// the actual "@override" annotation
const Object override = _Override();
```

The class has been made private (`_Override`) because its instantiation is useless as it does nothing. However, thanks to the `const Object override` variable being public, there's an "alias" of the `_Override` class which can be used as annotation. They're generally used for:

- reminding the developer about something, such as in the case of `@override`;
- before Dart 2.10, the `@required` annotation was used by the IDE to bring the developer's attention to the fact that a named optional parameter is required;
- some packages, such as *json_serializable* we're going to cover in chapter 15, rely on annotations to add additional information about a class, a method or a member. Annotations can be used to pass data to code generation tools.

Annotations can also have parameters but in this case you aren't doing the above "trick" of declaring a global variable exposing a private class. Just create a normal class, with a `const` constructor and use it as follows:

```
// Use this as annotation but it takes a param
```

```
class Something {
    final int value;
    const Something(this.value);
}

@Something(10)
class Test {}
```

4.4 Getters and setters

When a public variable is declared, anyone can freely manipulate it but it may not a good idea because the class partially loses control of its members. If we had written this...

```
class Fraction {
    int numerator;
    int denominator;
    Fraction(this.numerator, this.denominator);
}
```

... at a certain point someone could have changed the numerator and the denominator without any control introducing unexpected behaviors due to a wrong internal state.

```
void main() {
    final frac = Fraction(1, 7);

    frac.numerator = 0;
    frac.denominator = 0;
}
```

Having set both numerator and denominator to 0 there will be problems at runtime due to an invalid division operation. We can fix this problem using a *getter*, which makes the variables read-only.

```
class Fraction {
    int _numerator;
    int _denominator;
    Fraction(this._numerator, this._denominator);

    // Getters are read-only
    int get numerator => _numerator;
    int get denominator {
        return _denominator;
    }
}
```

The getter `numerator` returns an `int`, `_numerator`; the getter `denominator` does the same thing with an equivalent syntax. Like it happens with methods, when there is an one-liner expression or value to return, the => (arrow) syntax can be used.

```
void main() {
    final frac = Fraction(1, 7);

    // Compilation error, numerator is read-only
    frac.numerator = 0;
    // No problems here, we can read its value
    final num = frac.numerator;
}
```

The code is now safe because we can expose both numerator and denominator but it's guaranteed that they cannot be freely modified. Internally, `_numerator` and `_denominator` are "safe" because they aren't visible from the outside. Just as example, we're going to see how to write a **setter** for the denominator. So far it's read-only but with a setter it becomes editable:

```
class Fraction {
    int _numerator;
    int _denominator;
    Fraction(this._numerator, this._denominator);

    // getters
    int get numerator => _numerator;
    int get denominator => _denominator;

    // setter
    set denominator(int value) {
        if (value == 0) {
            // Or better, throw an exception...
            _denominator = 1;
        } else {
            _denominator = value;
        }
    }
}
```

There can be the same name for a setter and a getter so that a property can be read/written using the same identifier. Setters should be used to make "safe edits" on variables; they often contain a validation logic which makes sure that the internal state of the class doesn't get corrupted.

```
void main() {
    final frac = Fraction(1, 7);

    var den1 = frac.denominator; // den1 = 7
    frac.denominator = 0; // the setter changes it to 1
    den1 = frac.denominator      // den1 = 1
}
```

To sum it up, getters and setters are used to control the reading/writing on variables. They are methods under the hood but with a "special" syntax that uses the `get` and `set` keywords.

4.4.1 Good practices

Our recommendation is to keep the body of getters and setters as short as possible in benefit of code readability. You shouldn't put any loop that might slow down the assignment/retrieval of a value. The official documentation [3] also has something to say:

- When a variable has to be both public and read-only, just mark it as `final` without associating a getter to it.

  ```
  // Bad
  class Example {
      final _address = "https://fluttercompletereference.com";
      String get address => _address;
  }

  // Good
  class Example {
      final address = "https://fluttercompletereference.com";
  }
  ```

  If it's `final`, it's already a read-only variable because nothing can change its content. In this case a getter is simply useless.

- Avoid wrapping public variables with getters and setters if there's no validation logic.

  ```
  class Example {
      var _address = "https://fluttercompletereference.com";

      String get address => _address;
      set address(String value) => _address = value;
  }
  ```

  It compiles but there's no point in doing that: both getter and setter don't perform any particular logic as they just serve the variable as it is. Prefer doing this:

  ```
  class Example {
      var address = "https://fluttercompletereference.com";
  }
  ```

In general, use getters when you want to expose a variable but in "read-only mode" and setters when you want to filter/check the value that is going to be assigned.

[3] https://dart.dev/guides/language/language-tour#getters-and-setters

4.5 Operators overload

When you deal with primitive types you use operators very often: `5 + 8` is a sum between two `int` types happening with the + operator. We are going to do the same with `Fraction` class.

```
class Fraction {
    Fraction operator+(Fraction other) =>
        Fraction(
          _numerator * other._denominator +
          _denominator * other._numerator,
          _denominator * other._denominator
        );

    Fraction operator-(Fraction other) => ...

    Fraction operator*(Fraction other) => ...

    Fraction operator/(Fraction other) => ...
}
```

Operator overloading gives the possibility to customize the usage of operators in your classes. We have overloaded the + operator so that we can easily sum two fractions instead of having to create an `add(Fraction value)` method, like it happens with Java.

```
void main() {
    // 2/5
    final frac1 = Fraction(2, 5);
    // 1/3
    final frac2 = Fraction(1, 3);

    // 2/5 + 1/3 = 11/15
    final sum = frac1 + frac2
}
```

They work like normal methods with the only exception that the name must be in the form *operator{sign}* where *sign* is a supported Dart operator:

- **Arithmetic** operators like +, -, *, or /.
- **Relational** operators such as >=, <=, > or <.
- **Equality** operators like != and ==

And many more. There are no restrictions on the types you can handle with the operators, meaning that we could also sum fractions with integers: `operator+(int other)`. You cannot overload the same operator more than once in the same class.

4.5.1 callable classes

There is a special `call()` method which is very closely related to an operator overload because it allows classes to be called like if they were functions with the `()` operator.

ℹ You can give `call()` as many parameters as you want as there are no restrictions on their types. The function can return something or it can simply be `void`.

Let's give a look at this example.

```
class Example {
    double call(double a, double b) => a + b;
}

void main() {
    final ex = Example();         // 1.
    final value = ex(1.3, -2.2); // 2.
    print("$value");
}
```

1. Classic creation of an instance of the class
2. The object `ex` can act like if it were a function. The `call()` method allows an object to be treated like a function.

Any class that implements `call()` is said to be a **callable class**. In Dart, everything is an object and you've seen in 3.6.1 that even functions are objects. You can now understand why with the following example:

```
void test(String something) {
    print(something);
}
```

This is a typical `void` function asking for a single parameter. Actually, the above code can be converted into a callable class that overrides `call()` returning nothing and asking for a string.

```
// Create this inside 'my_test.dart' for example
class _Test {
    const _Test();

    void call(String something) {
        print(something);
    }
}

const test = _Test();
```

```
// Somewhere else, for example in main.dart
import 'package:myapp/my_test.dart';

void main() {
    test("Hello");
}
```

The function is nothing more than a package private class that overrides `call()` with a certain signature. Thanks to `const test = _Test();` in the last line we're "hiding" the class and exposing a callable object to be used as function.

4.6 Cloning objects

Even if it's not mentioned in the official Dart documentation, there is a standard "*pattern*" to follow when it comes to cloning objects. Unlike Java, there is no `clone()` method to override but still you might need to create deep copies of objects:

```
class Person {
    final String name;
    final int age;
    const Person({
        required this.name,
        required this.age,
    });
}

void main() {
    const me = const Person(
        name: "Alberto",
        age: 25
    );
    const anotherMe = me;
}
```

As you already know, the variable `anotherMe` just holds a reference to `me` and thus they point to the same object. Changes applied to `me` will also reflect on `anotherMe`. If you want to make deep copies in Dart (cloning objects and making them independent), this is the way:

```
class Person {
    final String name;
    final int age;
    const Person({
        required this.name,
        required this.age,
    });

    Person copyWith({
        String? name,
        int? age,
    }) => Person(
        name: name ?? this.name,
        age: age ?? this.age
    );

    @override
    String toString() => "$name, $age";
}
```

This method is called `copyWith()` by convention and it takes the same number (and name) of parameters required by the constructor. It creates a **new**, independent copy of the object (a clone) with the possibility to change some parameters:

```
    const me = const Person(
        name: "Alberto",
        age: 25
    );

    // Create a deep copy of 'me'.
    final anotherMe = me.copyWith();

    // Create a deep copy of 'me' with a different age.
    final futureMe = const.copyWith(age: 35);

    print("$me");         // Alberto, 25
    print("$anotherMe"); // Alberto, 25
    print("$futureMe");  // Alberto, 35
```

Both `anotherMe` and `futureMe` have **no** side effects on `me` because the reference is not the same. In fact, `copyWith()` returns a fresh new instance by copying internal data. Let's take a look at this line:

```
    name: name ?? this.name,
```

Thanks to the ?? operator, if `name` is `null` then initialize the clone with value of `this.name` taken from the instance. In other words, if you don't pass a custom `name` to `copyWith()`, by default a copy of `this.name` is made. Pay attention to generic containers and objects in general:

```
    class Skills {...}

    class Person {
        final List<Skills> skills;
        const Person({
            required this.skills
        });

        Person copyWith({
            List<Skills>? skills,
        }) => Person(
            skills: skills ?? this.skills
        );
    }
```

This code doesn't do what you'd expect because `List<T>`, like any other generic container, is an object and not a primitive type. With the above code you're just copying **references** and not making copies. The correct solution is the following:

```
    Person(
        skills: skills ?? this.skills.map((p) => p.copyWith()).toList();
    );
```

In this way you're making a copy of the entire list rather than passing a reference. The above code is just an one-liner way to iterate on each element of the source, making deep copies using `copyWith` and returning a new list. However, when the list is made up of primitive types, you could use a shortcut:

```
    class Person {
        final List<int> values;
        const Person({
            required this.values
        });

        Person copyWith({
            List<int>? values,
        }) => Person(
            values: values ?? []..addAll(this.values)
        );
    }
```

Primitive types are automatically copied so instead of using `map()` (which would be perfectly fine as well) we can use `addAll()` for a shorter syntax. There is no difference however because it still iterates on every element of the source list. The same example also applies to `Map<K, V>` and `Set<K>`. To sum it up, what you have to keep in mind is:

- Deep copies in Dart are made using the `copyWith()` method. You can give it any other name but you'd better follow the conventions.
- When making copies, be sure that classes (like generic containers) are deep copied using the

convenient `map((x) => x.copyWith())` strategy.

- If you have a list of primitive types (like `double`s or `int`) you can use the `[]..addAll()` shortcut. Do this **only** with primitive types.

5 | Inheritance and Exceptions

5.1 Inheritance

In any OOP language you can create hierarchies of classes and Dart is no exception. Here's the most basic example we can imagine.

```dart
class A {}
class B extends A {}
```

As you might expect, *A* is called superclass while *B* is the subclass (or "child class"). Methods can be overridden in subclasses because they all are "virtual" by default.

ℹ The term *virtual* indicates the possibility to redefine the behavior of a method in the subclasses. It's a very common OOP concept.

Actually Dart doesn't have the `virtual` keyword but it's "implicit" because any method can be overridden along the hierarchy. The annotation `@override` is optional but you'd better always use it. Starting from Dart 2.9 onwards, implicit downcasts are **not** allowed:

```dart
class A {
    double test(double a) => a * 0.5;
}

class B extends A {
    @override
    double test(double a) => a * 1.5;
}

void main() {
    A obj1 = A();
    A obj2 = B(); // Upcast
    B obj3 = obj1; // Downcast - ERROR from Dart 2.9

    print("${obj1.test(1)}"); // Prints 0.5
```

```
        print("${obj2.test(1)}"); // Prints 1.5
    }
```

This behaves in the classic OOP way; `obj1` calls the method in `class A` while `obj2` calls the overridden version in `class B`. When overriding, you can reference the original method definition in the superclass:

```
    class B extends A {
        @override
        double test(double a) {
            final original = super.test(a);
            return original * 1.5;
        };
    }
```

The special keyword `super` holds a reference to the super class. The usage of `super.test(a)` calls the `test()` method defined in the superclass. In Java for example, you get the same behavior.

> You cannot block inheritance since every class can have the `extends` modifier. In Java for example you can write `final class A {}` to say "hey, you can't inherit from me" but in Dart there is no equivalent.

You're allowed to also override setters (`set`) and getters (`get`) other than regular methods. Dart doesn't support multiple inheritance, like C++ does for example, meaning that `extends` works only with a single class:

```
    // Ok
    class A {}
    class B extends A {}

    // Doesn't work
    class A {}
    class B {}
    class C extends A, B {}
```

When overriding a parameter's type with a subtype, the compiler emits an error. Thanks to the `covariant` keyword, you can turn off static analysis for this kind of error to tell the compiler you know you're doing this intentionally. Here's an example:

```
    abstract class Fruit {}
    class Apple extends Fruit {}
    class Grape extends Fruit {}
    class Banana extends Fruit {}

    abstract class Mammal {
```

```
    void eat(Fruit f);
}
class Human extends Mammal {
    // Ok
    void eat(Fruit f) => print("Fruit");
}
class Monkey extends Mammal {
    // Error
    void eat(Banana f) => print("Banana");
}
```

Instead of `eat(Banana f)` we should have written `eat(Fruit f)` inside `Monkey` because the superclass method is asking for `Fruit`. However, we can allow the definition of a subtype in an overridden method with `covariant`:

```
class Monkey extends Mammal {
    void eat(covariant Banana f) => print("Banana");
}
```

Now the code compiles. Usually the superclass method is the best place where you could use `covariant` because it removes the "subtype constrain" along the entire hierarchy.

```
abstract class Mammal {
    void eat(covariant Fruit f);
}
class Human extends Mammal {
    // Ok
    void eat(Fruit f) => print("Fruit");
}
class Monkey extends Mammal {
    // Ok
    void eat(Banana f) => print("Banana");
}
```

Thanks to `covariant Fruit` at the top, any subclass is allowed to override `eat()` with `Fruit` or a subtype. This keyword can also be applied to setters and fields.

5.1.1 super and constructors

Every subclass in Dart automatically tries to call the default constructor of the superclass. If there isn't one, you **must** call the superclass constructor manually in the initializer list.

```
class Example {
    int a;
    Example(this.a);
}
```

```
class SubExample extends Example {
    int b;
    // If you don't call 'super(b)' the compilation will fail
    // because the father class has NO default constructor
    SubExample(this.b) : super(b);
}
```

The superclass constructor must be called in `SubExample` because the compiler has to somehow initialize the variable `a` of `Example`. However `super` is not needed when the class has a default constructor:

```
// The compiler automatically generates the default constructor
// which is just 'Example();'
class Example {
    int a = 0;
}

class SubExample extends Example {
    int b;
    // No need to call super
    SubExample(this.b);
}
```

Which is equivalent to...

```
class Example {
    int a = 0;
    Example();
}

class SubExample extends Example {
    int b;
    SubExample(this.b) : super();
}
```

... but you shouldn't do this because the compiler adds the missing pieces for you. You should call the superclass constructor only when there are parameters to pass. The call to `super()`, in case of an initializer list, always goes last:

```
// Compiles
MyClass(int a) : _a = a, super(a*a);

// Doesn't compile
MyClass(int a) : super(a*a), _a = a;
```

5.1.2 Abstract classes

The `abstract` keyword defines a class that cannot be directly instantiated: only its derived classes can. An abstract class can define one (or more) constructors as usual.

```
abstract class Example {
    // This is an abstract method
    void method();
}
```

Usually abstract classes contain abstract methods which can be defined putting a semicolon (;) instead of the body. You cannot define an abstract method in a class that's not been marked with the `abstract` modifier.

```
// Wrong: doesn't compile
class Example {
    // This method is abstract (no body) BUT the
    // class doesn't have the 'abstract' modifier
    void method();
}

// Correct: it compiles
abstract class Example {
    // Good job, this abstract method is in an
    // abstract class
    void method();
}
```

If your class contains **at least** one method with no body, then it must be `abstract` and the children must provide an implementation. You could also do this:

```
abstract class Example {
    void method();
}

class ExampleTwo extends Example {}
```

`ExampleTwo` is an abstract class too because it doesn't contain an implementation of `method();` yet. A class is concrete (and thus not abstract) when every method has a body.

```
abstract class Example {
    void method();
}

class ExampleTwo extends Example {
    @override
    void method() {
        print("I'm not abstract!");
    }
}
```

Now `ExampleTwo` is a concrete class. As we've already said, getters and setters can also be abstract and they're declared in the same way as methods (without a body).

```
    abstract class Example {
        final int _a;
        const Example(this._a);

        // Abstract getter
        int get calculate;

        // Abstract method
        int doSomething();
    }
```

5.1.3 Interfaces

In contrast to other programming languages, Dart does not have an `interface` keyword and you have to use classes to create interfaces. Your class can implement more than a single interface.

```
    abstract class MyInterface {
        void methodOne();
        void methodTwo();
    }

    class Example implements MyInterface {
        @override
        void methodOne() {}

        @override
        void methodTwo() {}
    }
```

The keyword is `implements` and, differently from a regular subclass, here you must override **every** method defined by the class/interface. The official documentation states the following:

Every class implicitly defines an interface containing all the instance members of the class and of any interfaces it implements. If you want to create a class A that supports class B's API without inheriting B's implementation, class A should implement the B interface.

In Dart when you use the term *interface* you are referring to a class that is going to be used by others along with implements because it only provides method signatures. The concept is the same you can find in Java, Delphi or C# with the only difference that Dart doesn't have a dedicated keyword.

```
// Instead of the 'interface' keyword, 'abstract class' is used
abstract class OneInterface {
    void one();
}

abstract class TwoInterface {
    void two();
}

class Example implements OneInterface, TwoInterface {
    @override
    void one() {}

    @override
    void two() {}
}
```

While extends can be used with only one class, implements works with one or more classes, which you should treat as interfaces (methods with no body). You could also do the following:

```
class OneInterface {
    void one() {}
}

class TwoInterface {
    void two() {}
}

class Example implements OneInterface, TwoInterface {
    @override
    void one() {}

    @override
    void two() {}
}
```

Classes and abstract classes can both be treated as interfaces with the difference that in concrete classes there must be at least an empty body. For this reason, in the above example we've put an empty body.

> ℹ We recommend to use abstract classes as interfaces so that you can write only the name of the method, like `void test();`. If you used a regular class you'd have to write `void test() {}` which is identical but it has an unnecessary empty body.

5.1.3.1 extends vs implements

You've just seen that `extends` is for subclasses and `implements` is for classes treated like if they were interfaces. It would be very fair if you were puzzled about the situation. Let's start with the technical difference:

- When you use `class B extends A {}` you are **NOT** forced to override every method of class *A*. Inheritance takes place and you can override as many methods as you want.
- When you use `class B implements A {}` you must override **every** method of class *A*. Inheritance does **NOT** take place because methods just provide an API, a "skeleton" that the subclass must concretize.

In practical terms instead:

- `extends`. This is the typical OOP inheritance that can be used when you want to add some missing features in a subclass. In chapter 8 we will see that it's a good practice deriving only abstract classes and not concrete classes.
- `implements`. Interfaces are useful when you don't want to provide an implementation of the functions but just the API. It's like if the interface was wall socket and the class was the plug that adapts to the holes.

While multiple inheritance is not allowed, you can `extends` a class and `implements` more than one. This is also how Java and C# behave, for example.

```
// Error
class A extends B, C, D {}

// Valid
class A extends B implements C, D {}
```

It might be useful if we made two final examples so that you can visualize the difference.

1. When you have a common behavior for **every** children of your hierarchy, create an abstract class and use `extends`. Say that you want to read many popular document files such as *.pdf*, *.docx* and *.txt*. Inheritance might be a very good idea:[1]

    ```
    abstract class Reader {
        bool fileExists(String path) {
            // code...
        }

        double size(File file) {
            // code...
    ```

[1]See appendix A for more info on the `File` class

```
        }

        String readContents();
    }
```

Deciding whether a file exists or not and getting its size is something common we always want to be able to call. Reading contents instead is specific to the implementation, it cannot be shared along the hierarchy, so it has to be abstract.

```
class PDFReader extends Reader {
    @override
    String readContents() {
        // code...
    }
}

class DocxReader extends Reader {
    @override
    String readContents() {
        // code...
    }
}

class TxtReader extends Reader {
    @override
    String readContents() {
        // code...
    }
}
```

It's been the right choice because `fileExists()` and `size()` have been defined only once, so no code duplication, and they're available in every subclass "for free". If we used `implements` we'd have had to define all of methods in the subclasses, including the "shared" ones leading to code duplication.

2. When you don't have methods implementations to share/reuse in the subclasses and you just need to give the signature, use `implements`. Say that you want to create a few sorting algorithms by yourself: using interfaces is the way to go (this is known as "*Strategy pattern*").

```
// This is going to be the "interface"
abstract class Sorter {
    void sort();
    String averageComplexity();
}

class MergeSort implements Sorter {
    @override
    void sort() {...}

    @override
    String averageComplexity() => "n*log(n)";
}

class InsertionSort implements Sorter {
    @override
    void sort() {...}

    @override
    String averageComplexity() => "n^2";
}
```

Any sorting algorithm has its own implementation and a particular name, so there's no common behavior. You just know that every concrete implementation needs a sorting method (`void sort()`) and the average time complexity (`String averageComplexity()`). Thanks to `implements` subclasses have to override every method and adopt a common API.

In summary, if you have common methods along the hierarchy go for subclasses otherwise use classes as interfaces. The usage of `implements` forces you to override every method; if this is not the behavior you're looking for, use `extends`.

5.1.4 Mixins

There's another important concept in Dart, which is the "cousin" of abstract classes and interfaces. A `mixin` is simply a class with no constructor that can be "attached" to other classes to reuse the code without inheritance.

```
mixin Swimming {
    void swim() => print("Swimming");
    bool likesWater() => true;
}

mixin Walking {
    void walk() => print("Walking");
}
```

If you use the special `with` keyword on a class, it acquires any method defined in the mixin. You could see mixins like a "copy/paste" tool to reuse methods.

```
class Human with Walking {
    final String _name;
    final String _surname;
    Human(this._name, this._surname);
    void printName() => "$_name $_surname";
}

void main() {
    final me = Human("Alberto", "Miola");

    // prints "Alberto Miola"; method is defined in the class
    me.printName();
    // prints "Walking"; method is not defined in the class
    // but it's "copied" and "pasted" from the mixin.
    me.walk();
}
```

The class `Human` doesn't have a `walk()` method inside but it's automatically imported from the `Walking` mixin. Since mixins are just classes without a constructor, you could also successfully compile the following code:

```
// use 'class' instead of 'mixin'
class Walking {
    void walk() => print("Walking");
}

class Human with Walking {}
```

In general it'd better if you used the `mixin` keyword since it helps both yourself, to remind what you want to do, and the IDE, which can give you useful hints. Furthermore, using the `mixin` keyword is less confusing than `class`.

Note that your class must have no constructors declared in order to be used as mixin.

```
class Walking {
    Walking();
    void walk() => print("Walking");
}
// Does not compile
class Human with Walking {}
```

Even if you've declared the empty non-argument constructor (`Walking()`) this class is not treated as a mixin. You should prefer `mixin` over `class` when you intend to accomplish this purpose.

Some other important features offered by mixins are:

1. A class can have more than a single mixin associated to it and its subclasses inherit the imported

methods.

```
mixin Walking {
    void walk() {}
}
mixin Breathing {
    void breath() {}
}
mixin Coding {
    void code() {}
}

// Human only has walk()
class Human with Walking {}
// Developer has walk() inherited from Human and also
// breath() and code() from the two mixins
class Developer extends Human with Breathing, Coding {}
```

2. There's the possibility to constrain the usage of a `mixin` to subclasses of a certain type. In this way you can make a sort of "restrictive reusability":

```
// Constrain 'Coding' so that it can be attached only to
// subtypes of 'Human'
mixin Coding on Human {
    void code() {}
}

// All good
class Human {}
class Developer extends Human with Coding {}

// NO, 'Coding' can be used only on subclasses
class Human with Coding {}

// NO, 'Fish' is not a subclass of 'Human' so
// you cannot attach the 'Coding' mixin
class Fish with Coding {}
```

Mixins are different from abstract classes and interfaces because they do not involve inheritance at all: they don't create hierarchies. They are just a way to reuse code without having to deal with superclasses, overrides and so on.

ℹ There is no rule of thumb to follow when it comes to the question "when should I prefer mixins over interfaces/abstract classes?" because the answer would be "It depends!". However, here's a general idea that you can follow to take your decision.

In general your first choices should be abstract classes or interfaces, with all their implementations in the various subclasses. Mixins are very handy when different pieces of architecture have the same identical code which cannot be shared due to a lack of inheritance. Let's say you have these classes in a folder:

```
import 'dart:math';

abstract class FootballTeam {
    String name();
    void playsWith() => print("Ball");
    double ballVolume(double radius) {
        const values = 4/3 * 3.14;
        return values * pow(radius, 3);
    }
}

class RealMadrid extends FootballTeam { ... }
class LiverpoolFC extends FootballTeam { ... }
```

And then, still in the same project, you have in another folder this hierarchy:

```
abstract class VolleyballTeam {
    String nameAndAbbreviation();
    void playsWith() => print("Ball");
    double ballVolume(double radius) {
        const values = 4/3 * 3.14;
        return values * pow(radius, 3);
    }
}

class TeamA extends VolleyballTeam { ... }
class TeamB extends VolleyballTeam { ... }
```

You see two separated hierarchies (which is good) with a few identical methods. You could create a superclass for `VolleyballTeam` and `FootballTeam` but it couldn't be possible due to the structure itself of your architecture. A good solution is the usage of a `mixin` which provides code sharing with no inheritance.

```
mixin BallSports {
    void playsWith() => print("Ball");

    double ballVolume(double radius) {
        const values = 4/3 * 3.14;
        return values * pow(radius, 3);
        }
    }
```

At this point you can get rid of the redundant methods and use the `mixin` to share the implementation (across the hierarchy as well). In this way you've removed code duplication, which is always a problem in terms of maintenance.

```
// Somewhere in your project...
abstract class FootballTeam with BallSports {
    String name();
}

// Somewhere else in your project...
abstract class VolleyballTeam with BallSports {
    String nameAndAbbreviation();
}
```

No more code duplication because the `mixin` solved the problem. Both hierarchies are separated because they are two different things but they have something in common. A `mixin` makes methods reusable without having to rely on inheritance.

5.1.5 Good practices

Generally, `factory` constructors are used to return "default" implementations of a certain class. For example, if you had a series of encryption algorithms, you could define a default one in this way:

```
abstract class EncryptionAlgo {
    // 'AESEncryption' is the default encryption algorithm
    factory EncryptionAlgo() {
        return AESEncryption();
    }

    void decrypt(String filePath);
}

class AESEncryption extends EncryptionAlgo { ... }
class RSAEncryption extends EncryptionAlgo { ... }
class BlowfishEncryption extends EncryptionAlgo { ... }
```

With this technique, you can use `EncryptionAlgo` to return an instance of `AESEncryption`:

```
// 'encrypt' is actually 'AESEncryption' because you're calling the
// factory constructor
final encrypt = EncryptionAlgo();
```

You are **NOT** creating an instance of an abstract class (which is impossible): you're just calling the factory constructor of `EncryptionAlgo` which returns a `AESEncryption`. This technique is used to return "default" instances of a certain hierarchy. You could use an even shorter syntax:

```
abstract class EncryptionAlgo {
    factory EncryptionAlgo() = AESEncryption;

    void decrypt(String filePath);
}
```

This kind of *constructor assignment* is just a shorter syntax to redirect a constructor to another. However, they must have the same number and type of arguments otherwise it won't compile. Basically you're telling the compiler to call the constructor of `AESEncryption` (along with the params, if any) when the `factory` is invoked.

5.2 Extension methods

Starting from Dart 2.7 the team has added *extension methods*, a way to add functionalities to a library knowing nothing about its internal implementation. Let's take a look at the `String` class, which has a lot of methods:

```
// A string
var name = "Alberto";

// Some methods of the string class
name.toUpperCase();
name.toLowerCase();
name.trimLeft()
```

Earlier we worked hard to create a `Fraction` class and it would be really cool if we were able to integrate it with strings. Something like this:

```
final Fraction value = "1/3".toFraction();

value.reduce();
value.negate();
```

Given a string, we could add the `toFraction()` method that returns a `Fraction` whenever possible. We don't have access to the `String` class and we can **not** edit the file in which it's been declared, of course. Extension methods come to the rescue:

```
extension FractionExt on String {
  bool isFraction() => ...

  // Converts a string into a fraction
  Fraction toFraction() => Fraction.fromString(this);
}
```

The this identifier refers to the object on which you are calling the method. What's written as this.contains("/") is evaluated to "2/5".contains("/") is the example:

```
extension FractionExt on String {
    // code...
}

void main() {
    var str = "2/5";

    if (str.isFraction()) {
        final frac = str.toFraction();
    }
}
```

With extension methods we've just *extended* the String class with new functionalities without changing the definition of the class. Methods have been added "from the outside".

ℹ Abstract classes, interfaces and mixins are for "internal use" in the sense that with them you touch the class directly from the inside. Extensions are for "external use" in the sense that you add methods from the outside without changing the internal part of the class.

With extensions you can also define getters, setters, operators, static fields and/or methods. If you need to use some utility private functions inside the extension, just append an underscore (_) in front of the name. It's like with classes, where void _method() {} is private to its library.

ℹ Extension methods also work with generic types. You could for example write extension Test<T> o
where T is statically determined by the compiler. Generics and collections will be discussed in the next chapter.

5.2.1 Good practices

Generally extension methods should be put in a dedicated file. For example, the previous extension of fraction could go on fraction_ext.dart so that it can be imported as a library.

```
// === file fraction_ext.dart === //
extension FractionExt on String {
    // code...
}

// === file main.dart === //
import 'package:fraction_ext.dart';
```

```
void main() {
    final check = "hello".isFraction();
}
```

Short and trivial operations should be implemented with extensions in order to improve the usability, the readability. Don't you agree that...

```
final f = "1/2".isFraction();
```

... looks better than a static method call?

```
final f = Fraction.isFraction("1/2");
```

Keep in mind that extension methods cannot be used if you're treating a variable with `dynamic`. They support type inference and then the compiler must always exactly know the types.

```
// It compiles but gives a runtime exception
dynamic f = "1/2";
f.isFraction();
```

Before moving on, let's try to make a simple summary of all the features we've seen up to now in this chapter.

- **Internal use**. These techniques are meant for "internal use" because they can work directly inside the definition of the class.
  - `extends`. It's the typical OOP inheritance: you don't need to override every method, getter or setter. Use it when there is a "common" behavior to share along the hierarchy.
  - `implements`. The "interface" OOP concept: you need to override every method, getter or setter. Use it when you want to define an API for many types (they have no common behaviors to share).
  - `mixin`. A way to "copy/paste" methods into classes: they reduce code duplication and centralize the code. The imported methods are also shared along the hierarchy.
- **External use**. These techniques are meant for "external use" because they cannot modify the class from the inside.
  - `extension`. They add functionalities to a class without changing its internal definition. They're generally used to "improve" a class without touching its internals.

5.3 The Object class

Even if it's not explicitly written in the code, any Dart class descends from `class Object {}` which is at the root of any hierarchy. This is the same structure that some popular languages such as Java, C# and Delphi adopt. Every class can, and should, override the methods declared in `Object`.

- `String toString()`. This method is very important and you should always override it because a string representation of an object is quite handy, especially while debugging. If you have it defined you can better handle the string conversion:

```
final f = Fraction(1, 2);
final s = "My fraction is $f";
```

String interpolation automatically calls `toString()` which makes the code shorter and more readable.

- `bool operator ==(SomeClass other)`. When you want to compare two objects you have to use the == operator. By default it doesn't really work as you'd expect; it returns true only if two variables point to the same object.

```
class Example {
    int a;
    Example(this.a);
}

void main() {
    final ex1 = Example(2);
    final ex2 = ex1;
    print(ex1 == ex2); //true
}
```

The console outputs true because `ex2` points to the same reference held by `ex1`; both variables point to the same object so they're equal. By default the == operator compares references and doesn't look at the object itself; it only cares about what's being pointed to.

```
void main() {
    final ex1 = Example(2);
    final ex2 = Example(2);
    print(ex1 == ex2); //false
}
```

Logically we'd think they were equal but the compiler tells us another story. `ex1` and `ex2` point to two different objects that are logically equal but practically different because each variable holds a different **reference**. You have to override the equality operator to get the desired behavior:

```
class Example {
    int a;
    Example(this.a);

    @override
    bool operator ==(Object other) {
        // 1.
        if (identical(this, other))
            return true;
        // 2.
        if (other is Example) {
            final example = other;

            // 3.
            return runtimeType == example.runtimeType &&
                a == example.a;
        } else {
            return false;
        }
    }

    // 4.
    @override
    int get hashCode => a.hashCode;
}
```

That's a lot of boilerplate code but you don't have any other choices. For sure if you come from the Java world you might have dealt with this exact same thing thousands of times.

1. The function `identical()` is provided by the Dart code API and checks if two objects have the same reference.
2. If the type of the compared object is equal to the type of the current object, a smart cast happens in the body to proceed with the comparison. Otherwise, `false` is returned.
3. Check if the runtime types are the same and then make a one by one comparison for every instance variable.
4. See the next point.

Those two overrides allow a proper object comparison as you'd expect. Running this example would output `true` because we've taught `operator==` to look at the object itself and not only at the references.

```
void main() {
    final ex1 = Example(2);
    final ex2 = Example(2);
    print(ex1 == ex2); // true -> that's what we wanted!
}
```

- `int get hashCode`. When you override the equality operator, you must always remember to also override `hashCode`. The hash code is useful when you call the `identical()` method and also when you use "hash maps" we're going to analyze in the next chapter.

  If you know what hash tables are, you are aware that the keys must not clash in order to keep good performances. Dart's implementation of hash tables guarantees O(1) in insertion if hash codes of different objects return different values (no collisions).

5.3.1 Comparable<T>

Other than overriding `operator==` and `hashCode`, if you wanted to do a precise and complete comparison setup of a class, you should also implement `Comparable<T>`. This is how a complete comparison logic for a `Fraction` class would look like:

```
class Fraction implements Comparable<Fraction> {
    final int num;
    final int den;
    const Fraction(this.num, this.den):

    double toDouble() => num / den;

    @override
    int compareTo(Fraction other) {
        if (toDouble() < other.toDouble()) return -1;
        if (toDouble() > other.toDouble()) return 1;
        return 0;
    }

    @override
    bool operator==(Object other) { ... };

    @override
    int get hashCode { ... };
}
```

`Comparable<T>` exposes the `compareTo()` method which can be used for ordering and sorting. The returned `int` value must follow this convention:

- if this instance is naturally $<$ than the other, return a negative number;
- if this instance is naturally $>$ than the other, return a positive number;
- if the two instances are equal, return 0.

Returning `1` and `-1` is just a convention; it can be any positive or negative number. In our case, we have done the following:

- a fraction is naturally $<$ than another if its double representation is smaller than the other. For this reason, we return `-1` in case the *num* / *den* ratio of the current instance were $<$ than the other.

  ```
  if (toDouble() < other.toDouble()) return -1;
  ```

- a fraction is naturally $>$ than another if its double representation is greater than the other. For this reason, we return `1` in case the *num* / *den* ratio of the current instance were $>$ than the other.

```
if (toDouble() > other.toDouble()) return 1;
```

- two fractions are equal if the rational number they represent is also equal. For this reason, we return `0` if none of the above statements passed.

```
return 0;
```

In general, comparing floating point numbers using <, <=, > and >= is safe. Comparing floating point values with `operator==` is very error prone; two values that should be equal may not be due to arithmetic rounding errors. This is the reason why we use `return 0` as "fallback" rather than checking for equality of double values.

5.4 Exceptions

Dart code can throw exceptions to signal that an unexpected or erroneous behavior has happened during the execution. When you throw an exception you should really catch it otherwise your program will forcefully terminate with an error code.

```
class Fraction {
    int _numerator;
    int _denominator;

    Fraction(this._numerator, this._denominator) {
        if (_denominator == 0) {
            throw IntegerDivisionByZeroException();
        }
    }

}
```

The denominator of a fraction cannot be zero and in our example the constructor is the best place to throw an exception. It signals that something's gone wrong, in this case an invalid parameter. You can only throw **objects**.

The `IntegerDivisionByZeroException()` class is provided by Dart and it should be used when... a division by zero happens! Despite you can throw any object, classes with meaningful names should be preferred.

```
class RandomClass() {}

void main() {
    // It compiles. It's perfectly fine
    throw RandomClass();
}
```

You could also throw a single string like `throw "Whoops";` because `String` is a Dart class.

There are a few interchangeable classes that can be thrown in exceptions. For sure, if you have a format error you're not going to throw a `TimeoutException`.

- DeferredLoadException
- FormatException
- IntegerDivisionByZeroException
- IOException
- IsolateSpawnException
- TimeoutException

You can throw everything except for `null`.

5.4.1 on and catch

If you don't catch an exception, your program forcefully terminates but you never want to be in such situation. It's always a good idea handling exceptions by catching them.

```
void main() {
    try {
        final f = Fraction(1, 0);
    } on IntegerDivisionByZeroException {
        print("Ouch! Division by zero!");
    }
}
```

This code is safe because the exception raised by the constructor of `Fraction` is caught by a `try` block. If you want to get an instance of the thrown exception, just add a `catch` statement:

```
void main() {
    try {
        final f = Fraction(1, 0);
    } on IntegerDivisionByZeroException catch (exc) {
        // use the exc object
        doSomething(exc);

        print("Ouch! Division by zero!");
    }
}
```

`exc` is an instance of the thrown object that can only be used within the scope of `on`. This approach can be useful when there's the need to print a stack trace or any useful debugging info about the exception.

```
void main() {
    try {
        final f = Fraction(1, 0);
    } on IntegerDivisionByZeroException {
        print("Division by zero!");
    } on FormatException {
        print("Invalid format!");
    }
}
```

You can handle code that could throw more than a single exception by specifying multiple catch clauses. The first clause that matches the thrown object will be picked to handle the error propagation. What if the above code threw an `IOException` or another object you haven't handled? The program would crash because there would be a not handled exception.

```
void main() {
    try {
        final f = Fraction(1, 0);
    } on IntegerDivisionByZeroException {
        print("Division by zero!");
    } on FormatException {
        print("Invalid format!");
    } catch (e) {
        // You arrive here if the thrown exception is neither
        // IntegerDivisionByZeroException or FormatException
        print("General error: $e");
    }
}
```

The final `catch` is a fallback that captures everything else that didn't match the above types. You're guaranteed that this code is safe because, worst case, the two `on` will be skipped and the execution will fall inside the final `catch`.

5.4.2 finally

The `try` - `catch` block protects your code against runtime exceptions that alter the normal execution flow. As you know, if no exceptions occur nothing happens because the `catch` block doesn't execute.

```
void main() {
    try {
        final f = Fraction(2, 3);
    } catch (e) {
        print("Error");
    }
```

```
        print("Finish");
    }
```

This simple code only outputs "Finish" because no exceptions have been thrown. However, there is a way to force the execution of a part of the try statement.

```
    void main() {
        try {
            final f = Fraction(2, 3);
        } catch (e) {
            print("Error");
        } finally {
            print("Always here");
        }

        print("Finish");
    }
```

The body of a finally block is **always** executed, no matter if an exception occurs or not. The above code prints "Always here" and then "Finish" on a new line.

- The code inside on or catch is executed only if an exception is thrown and a valid handler for the object is provided. If no exceptions are thrown, nothing happens.
- The code inside a finally block is always executed regardless the fact that an exception object is thrown or not.

5.4.3 Good practices

The Dart API provides the Exception interface which is implemented by all core library exceptions. Production-quality code usually throws subtypes implementing one of the following classes.

- **Exception**. Implement this interface to create errors that will be caught when something goes wrong. For example, doing this is a good idea:

  ```
  class FractionDivisionByZero implements Exception {
      final String message;
      const FractionDivisionByZero(this.message);

      @override
      String toString() => message;
  }
  ```

  We can throw the class FractionDivisionByZero specific object rather than a general purpose IntegerDivisionByZeroException. When you have to throw an object, implement Exception and choose a proper name that describes what's gone wrong.

> ℹ The official documentation [2] suggests to not use `throw Exception("...");`. Yes, you could, but it's bad because it doesn't give a precise type that can be caught with the `on` keyword.

- **Error**. You should subclass this type for all those programmer-relates errors such as going out from the bounds of a list or having an assertion that evaluates to false. `Error` is thrown in case of an unexpected program flow and you should **NOT** catch it.

  ```
  class Example {
      int x;
      Example(this.x) : assert(x != 0);
  }
  ```

  In debug mode, if x is 0 an `AssertionError` is thrown. You shouldn't catch it and that's perfectly fine because you, the developer, **must** see that kind of error: it signals you've coded something in the wrong way. The IDE will produce an alert when an object of type `Error` is being caught.

  > ℹ In practice you're allowed to catch `Error`s but you shouldn't. This kind of exceptions signal that you, the programmer, failed something and you must see it because a fix is required!

  `Error` is an alert for the developer telling him that something really bad happened, such as an index gone out of the bounds or a critical I/O issue.

The usage difference lies on the logical concept.

- You almost always want to `throw` an object that implements `Exception` because you are the owner of the code and you decide what's wrong. Anyone else going to use your code will catch your exception object in order to know that something went wrong.
- You want to `extend` (and not `implement`) `Error` if you're dealing with particular code that might cause I/O issues, lists out of bounds or other "particular" error that a programmer can do.

Put in other terms, exceptions are "logical errors" that can be fixed while errors are "language specific" mistakes that cause core problems in your code. Other recommendations taken from the official guidelines [3] are:

1. In a `try` block you should use `on` to catch specific types and decide what to do according with the related error.

[2] https://api.dart.dev/stable/2.7.0/dart-core/Exception-class.html
[3] https://dart.dev/guides/language/effective-dart/usage#error-handling

```
try {
    // code...
} catch (e) {
    // handle...
}
```

This code is really bad because it catches everything, including `Error` (or its subtypes) which should not be caught. What you've just seen is the "*Pokémon exception handling*" and it's not a joke! Even the official Dart doc mentions it.

> It actually makes sense: the Pokémon motto is "Gotta catch 'em all" because the goal of the game is capturing any Pokémon encountered during the journey. You don't want to be a Pokémon programmer who bravely catches any exception with no regards, don't you?

If you really need to "eat" as many exceptions as possible, use this version:

```
try {
    // code...
} on Exception catch (e) {
    // handle...
}
```

It's still not good but at least way better than before because `Error` is not captured and it's still free to propagate.

2. Use `on SomeException catch (e)` to implement a logic for any type of thrown object.
3. Don't "eat" exceptions. If you really need to do nothing when an exception occurs, print a message to the console or show a popup to notify your users that something's gone wrong.

```
try {
    // Exception is 'eaten' because when caught, nothing happens
} on Exception catch (e) {}
```

4. If you implement `Exception`, append it at the end of your custom class name such as *Fraction**Exception*** or *NumeratorNull**Exception***. The same concept applies when you deal with `Error`.
5. Do not catch exceptions objects whose type is `Error`.
6. If you want to rethrow an exception, you should use `rethrow` rather than throwing the same object again with `throw` [4].

[4]https://dart.dev/guides/language/effective-dart/usage#do-use-rethrow-to-rethrow-a-caught-exception

```
try {
    try {
        throw FormatException();
    } on Exception catch (e) {
        print("$e");
        rethrow;
    }
} catch (e2) {
    print("$e2");
}
```

```
try {
    try {
        throw FormatException();
    } on Exception catch (e) {
        print("$e");
        throw e;
    }
} catch (e2) {
    print("$e2");
}
```

Both ways of rethrowing an exception are valid but you should prefer the usage of `rethrow` (on the left) because is preserves the original stack trace of the exception. `throw` instead resets the stack trace to the last thrown position.

6 | Generics and Collections

6.1 Generic types

Nowadays a very common feature of languages is the support for parameterized programming, also known as *generic programming* in the Dart, Java, C# and Delphi world. The biggest advantages brought from this approach are type-safety and code reusability.

6.1.1 Introduction

The best way to show how generics can be incredibly useful is via example. In Flutter you'll use very often caches to store data coming from your device or from the internet. Pretend you had to store a complex object: the first idea popping in your mind could be:

```dart
class ComplexObjCache {
    final _obj = ComplexObj();
    // Constructor, getters, setters, methods...
}
```

The next day you have the need for another cache, with the same structure, but it has to handle floating point numbers. Good, you already know what to do: copy-paste the previous class and change the types from `ComplexObj` to `double` using the IDE refactor tool.

```dart
class DoubleCache {
    final _obj = 0.0;
    // Constructor, getters, setters, methods...
}
```

One week later you need the same cache also for Strings, integers, and other objects but things start to get complicated. You've copy-pasted classes because they have the exact same structure and methods but the only difference is the data type.

```dart
class ComplexObjCache { }
class ComplexObj2Cache { }
class ComplexObj3Cache { }
class StringCache { }
class IntegerCache { }
class DoubleCache { }
```

Massive code duplication is guaranteed to be a maintenance nightmare. With generics you can easily solve the problem using a *placeholder* type that will be evaluated at compile time. In general a single letter is enough, such as `T` or `K`, but there are no restrictions on the length (`PIZZA` would be valid for example).

```
class Cache<T> {
    final T _obj;
    // Constructor, getters, setters, methods...
}
```

Using the <T> notation you're telling the compiler that, at compile time, the letter T has to be substituted with the type given by the actual instance. In this way you have a single class allowed to work with multiple types and thus maintenance gets way easier.

For example, in `final c = Cache<String>("");` the letter `T` is substituted with `String` at compile time. Potential errors, such as bad type casts, produce a compilation error.

6.1.2 Type safety

Generics might remind you of `dynamic`, as it works with any type, and actually they could do the same things. For example, rather than `Cache<T>` you could have created a `Cache` with `dynamic` fields:

```
class Cache {
    dynamic _obj;
    dynamic get value => _obj;

    // Constructor, getters, setters, methods...
}
```

The biggest problem is that `dynamic` has **zero** type safety as it's intended to work with runtime casts. You'd have to deal with long series of `if` statements for each `dynamic` variable because casts must be safe. The compiler and the IDE won't be able to help you with static analysis.

Manually dealing with lots of casts and type checks is one of the worst maintenance nightmares, especially if you're in a complex architecture with tons of `dynamic` types. In addition, potential problems with types are spotted at runtime while with generics they're immediately caught by the compiler.

With generics, casts are not needed because the compiler ensures a "protection" against wrong types usages. The code is said to be *safe* because type errors are known at compile time and not at runtime.

```
class Cache<T> {
        final T _obj;
        T get value => _obj;
        // Constructor, setters, methods...
    }

    void main() {
        final cache = Cache<int>(20);
        String value = cache.value; // Error!
    }
```

The compiler generates an error because you're trying to assign an integer value (`cache.value`) to a string type. You'll also get an alert from the IDE before compiling and the debugger will find this error immediately. In contrary:

```
    // using 'class Cache {}', the non-generic one
    void main() {
        final cache = Cache(20);
        String value = cache.value; // Compiles, but it's wrong!
    }
```

Here you get no compilation errors because `dynamic` is evaluated at runtime. The IDE won't help you because it cannot predict the future and imagine that at runtime an `int` will be assigned to a `String`. Debugging might not be so easy and you're going to waste time.

We've said *waste* because you could have saved that time with generics! The compiler would have been able to tell you in which line the error was but the usage of `dynamic` defers everything at runtime, so no help at all.

As you may have guessed, using generics in place of `dynamic` is almost always the best choice you can make. When you try to write parameterized code, always consider generics first!

6.1.3 Usage

There's nothing new about hierarchies of one or more generic types; what we've already covered about classes and inheritances also applies here. They work in the same way as "regular" non-generic classes with the only difference that the type goes inside the diamonds < >.

```
    abstract class Cache<T> {
      final T _obj;
      Cache(this._obj);

      T get value => _obj;
      void handle();
    }
```

```
// 1.
class LocalCache<T> extends Cache<T> {
  LocalCache(T obj) : super(obj);
  void handle() {}
}

// 2.
class CloudCache<T, K> extends Cache<T> {
  CloudCache(T obj, K obj2) : super(obj);
  void handle() {}
}
```

There's the possibility to define a infinite number of generic types per class, they just need to be separated by commas in the diamond list (< >) with the letters inside. Generics also work with `extends` and `implements`.

1. A subclass must declare **at least** the same number of parameterized types defined by its superclass.
2. You cannot write `LocalCache<A> extends Cache<T> {}` because the compiler tries to pass `A` to the superclass constructor which expects a `T`. The letters must match.
3. You can declare how many parameters you want. In the above example `T` and `K` are separately treated even if they will be used to represent the same type. The usage of `CloudCache<int, int>` gives no problems because the compiler treats `T` and `K` as two different entities and it doesn't care about the type they represent.

There's the possibility to put constraints on the types if you don't want them be "anything". For example, if we wanted our caches to be used only with numbers, we could have done the following:

```
abstract class Cache<T extends num> { }

abstract class LocalCache<T extends num> extends Cache<T> { }
abstract class CloudCache<T extends num, K> extends Cache<T> { }
```

Remember that `int` and `double` are subclasses of `num`. With these changes, the generic type is allowed to be only a subtype of `num` and nothing else.

```
void main() {
    // OK. 'int' and 'double' are subclasses of
    // 'num' so this is allowed
    final local1 = LocalCache<int>(1);
    final local2 = LocalCache<double>(2.5);

    // NO. 'String' is not a subclass of 'num'
    // so this is NOT allowed
```

```
        final local3 = LocalCache<String>(3);
    }
```

You could also use `LocalCache<num>(0);` but superclasses of `num` are not allowed. In Dart methods can return generic types and/or have it as parameter, you just have to put the diamonds after the name as you'd do with classes.

```
    // T used as return type and parameter
    T printValue<T>(T val) {
        // ...
        return val;
    }

    // T user as parameter
    void check<T>(T val) {
        // ...
        return val;
    }

    void main() {
        // Diamond syntax uses the symbols '<' and '>'
        final a = printValue<int>(1);
        final b = printValue<String>("1");
        check<double>(6.45);
    }
```

A non-generic class can have one or more generic methods inside:

```
    class Example {
        void doSomething<T>(T value) { ... }
    }
```

6.2 Collections

Dart implements the most common types of containers using generics so that you can take advantage of type safety without having to deal with specific implementations for each type. We've already seen all the benefits in the previous section so let's get started.

6.2.1 List

As you already know from chapter 2, the language doesn't have arrays but only lists. They're implemented with the `List<T>` generic container you've already seen many times up to now:

```
    // The type of 'intList' is List<int>
    var intList = [2, 5, -8, 0, 1];
```

```
// The type of 'stringList' is List<String>
var stringList = ["a", "hello"];
```

Lists are 0-indexed and have many methods such as `add()`, `length`, `clear()` and so on; check out your IDE or the documentation [1] for a complete list. Like with any other generic type, you have the "protection" of the compiler against bad types assignments:

```
final list = List<int>();
list.add("oops");
```

This is clearly a compilation error because you're trying to assign a string to a container that expects only integers. There is a very nice syntax shortcut allowing you to insert a series of values directly in the array:

```
// 'list1' contains [1, 2]
var list1 = [1, 2];
// 'list2' contains [-2, -1, 0, 1, 2]
var list2 = [-2, -1, 0, ...list1];
```

Given a list (`list1`), you can put every element inside another list using the **spread operator** (`...`). If you aren't sure about the non-nullability of the source, you can add a safety check with the ***null-aware* spread operator** (`...?`).

```
// this is null
List<int>? list1;

var list2 = [-2, -1, 0, ...list1];  // Error
var list3 = [-2, -1, 0, ...?list1]; // All good
```

There's the need to use `...?` because elements are being added only if the source list is **not** null; if we had used `...list1` we would have got an error. A list can also be initialized by specifying the type in the diamonds:

```
final list = const <int>[1, 3, 6, 7];
```

6.2.1.1 Collection statements

Starting from Dart 2.3, there's the possibility to use a very convenient syntax you might use some times in Flutter to add elements in a `ListView` (a scrollable container of UI items). You can put an `if` statement inside a list to decide whether it's the case to add or not an item.

```
final hasCoffee = true;

final jobs = [
    "Welder",
    "Race driver",
    "Journalist",
```

[1] https://api.dart.dev/stable/2.7.0/dart-core/List-class.html

```
        if (hasCoffee) "Developer"
    ];
```

This list contains for sure the first 3 elements but *"Developer"* is going to be added only if `hasCoffee` is true. Putting an `if` statement with no parenthesis in a list decides whether the element has to be added or not.

```
    const hasCoffee = true;

    final jobs = const [
        "Welder",
        "Race driver",
        "Journalist",
        if (hasCoffee) "Developer"
    ];
```

If we declared `const hasCoffee` (rather than using `final`) then we could have also created a constant list of values. This is even better because the compiler could perform some optimizations at compile time. In a very similar way, you could also use a `for` loop:

```
    final numbers = [
        0, 1, 2,
        for(var i = 3; i < 100; ++i) i
    ];
```

Like it happened with the `if`, the `for` statement adds a series of items to the list. It has no parenthesis or `return`: just write the values as they are. In this case, the list cannot be `const`.

6.2.1.2 Implementation

If you were to look at the online source code [2] you'd see that `List<E>` is defined as `abstract class` but objects can still be created in the "normal" way. You might be puzzled at this point: why can I instantiate an abstract class?

```
    final howIsItPossible = List<int>();
```

As you know, an `abstract class` cannot be instantiated and that's always true but `List<E>` defines a series of factory constructors to do the "trick". We have already seen this pattern in 5.1.5. This is the actual implementation of `List<E>` in the Dart SDK.

```
    abstract class List<E> implements EfficientLengthIterable<E> {
        // code...
        external factory List([int? length]);
        // other code...
    }
```

[2]https://github.com/dart-lang/sdk/blob/master/sdk/lib/core/list.dart

Thanks to this, when you write `howIsItPossible = List<int>();` you're actually calling the factory constructor which returns a **concrete** implementation of `List<E>`. You cannot see it because of the `external` keyword, which loads the body of the constructor from the Dart VM, but the factory returns a concrete (non-abstract) class.

ℹ The `external` keyword is used to interact with the Dart VM as it loads specific pieces of code. You'll **never** use it. It loads the body of a function directly from the internals of the SDK, which is almost always C++ code. So, instead of having a `factory` with a body...

```
factory List([int? length]) {
    // Dart code here...
}
```

... the definition is placed somewhere else. In other words, `external` is used to say "the body of this function is not here, in the Dart file, but it's located somewhere in the VM.

```
// Thanks to 'external,' the body is not directly loaded from here,
// the .dart file, but it's taken from somewhere else
external factory List([int? length]);
```

Unless you're a developer from the Dart team, this keyword is not for you.

Because of `factory List([int? length])` it **seems** you're instantiating an abstract class from your code but no, in reality you're calling a factory constructor returning a concrete instance. With this pattern, `List<T>` returns a default implementation which is generally good for most of the use cases. The subclasses returned by the factory can be:

- **Growable.** You can increase and decrease the length of the list with no restrictions. A growable list has no `length` parameter in the constructor.

  ```
  // No int param in the parenthesis -> no length specified
  // so this list is growable
  final growable = List<int>();
  growable.length = 5;
  ```

- **Fixed.** You can **not** increase and decrease the length of the list with no restrictions. A fixed list has the length parameter in the constructor.

  ```
  // int param in the parenthesis -> lenght is specified
  // so this list is fixed
  final growable = List<int>(3);
  growable.length = 5; // Error!
  ```

In general creating a growable list with `List<int>()` is good but if you don't need to change the size of the container go for `List<int>(10)`. There are other useful factory constructors from this class:

- `filled()` It creates a list of the given length and initializes each position with the elements you want.

  ```
  final example = List<int>.filled(5, 1, growable: true);
  // Now example has this content: [1, 1, 1, 1, 1]
  ```

- `unmodifiable()` It returns a copy of the given list in which you cannot call *add*, *remove* or other methods that would modify its content.

  ```
  var example = List<int>.unmodifiable([1,2,3]);
  example.add(4); // Runtime error
  ```

- `generate()` Creates a list of the given length and fills each position according with the value returned by the generator. This example fills the array with the power of each number.

  ```
  var example = List<int>.generate(5, (int i) => i*i);
  // Now example has this content: [0, 1, 4, 9, 16]
  ```

Note that `List.unmodifiable()` creates a new **copy** of the list; if you just want to make an unmodifiable list without having to copy the contents, use `UnmodifiableListView<T>`. They both create two unmodifiable lists but in the second case, no copies are made (it's just a "wrapper" around a `List<T>` object).

6.2.2 Set

A `Set<T>` is a generic container in which there cannot be duplicate objects. Sets can be directly initialized using braces without having to create a new instance and then add elements with a loop.

```
final keys = {1, 2, 3, 3, 4, 5};

for(var key in keys)
    print(key);
```

The console will output *1 2 3 4 5* and not *1 2 3 3 4 5* because duplicates are not added (no exceptions are thrown). Elements can be inserted by using functions like `add()` or `addAll()`; for any possible method on `Set<T>` you should check the official documentation [3].

When creating an empty set, there's the need to use the diamonds to specify the type. Otherwise, avoid using type inference and annotate the type directly:

```
// 1. Direct type annotation
Set<int> emptySet = {};
// 2. Type inference with diamonds
final emptySet = <int>{};
```

[3]https://api.dart.dev/stable/2.7.0/dart-core/Set-class.html

```
    // 3. This is a Map, not a set!!
    final emptySet = {};
```

Only in cases *1* and *2* you're creating an empty set. In *3* the compiler considers the `final emptySet = {}` like if it were a map of `dynamic` key/value pair.

`Set<T>` supports the `...` and `...?` operators we've seen earlier with lists. You can also use `if` and `for` collection statements.

- Adding items in a `Set<T>` is pretty easy with `add()` but if you want to insert a series of values, you can also pass an array.

  ```
  final example = <int>{};
  example.addAll([5, 3, 7]);
  ```

- Use `bool contains(T value);` to check whether an element is in the set or not.
- Remove elements with `remove()`.

If you use the factory `Set<T>.unmodifiable(Set<T>> other)` you get an instance of the same set with no possibility to add/remove values. There's no `UnmodifiableSetView<T>`.

6.2.2.1 Implementation

In Dart sets are `abstract class Set<E> extends EfficientLengthIterable<E>` and, like we've already seen with lists, they use factory constructors to return concrete instances. The implementation is easier to understand because there is no `external` keyword.

```
abstract class Set<E> extends EfficientLengthIterable<E> {
    // code...
    factory Set() = LinkedHashSet<E>;
    // code...
}
```

When we do something like `final s = Set<int>();` we're not instantiating the abstract class, which is impossible, but we're calling `factory Set()` which returns a concrete class. These lines are equivalent...

```
final set1 = Set<int>();
final set2 = LinkedHashSet<int>();
```

... because in Dart a `Set` is implemented by default as a `LinkedHashSet`. This class is a hash-based implementation of a `Set` and it keeps track of the order in which elements have been inserted.

6.2.3 Map

Also known as *dictionary*, a `Map<K,V>` is an unordered generic collection that stores key-value pairs. You can retrieve the object you're looking for by using the associated key.

```
final m = <int, String>{
    0: "A",
    1: "B",
    2: "C"
};
```

`Map.unmodifiable()` creates an unmodifiable **copy** of the map while `UnmodifiableMapView()` just takes a reference (no copies are made). There are two ways to insert new pairs in the map:

- Use the `putIfAbsent()` method to insert a new pair of values only if the key is not already in the list.

  ```
  final example = <int, String>{
      0: "A",
      1: "B"
  };

  // The key '0' is already present, "C" not added
  example.putIfAbsent(0, () => "C");
  // The key '6' is not present, "C" successfully added
  example.putIfAbsent(6, () => "C");
  ```

  The first parameter is the desired key while the second one is a no-param function returning the value to be inserted.

- Use brackets `[]` to add an element at the given index without checking if the key is already in the collection.

  ```
  final example = <int, String>{
      0: "A",
      1: "B"
  };

  // "A" has '0' as key and it's replaced with "C".
  // Now the map contains {0: "C", 1: "B"}
  example[0] = "C";
  // The key '6' is not present, "C" gets added
  example[6] = "C";
  ```

  Maps doesn't allow duplicate keys. When we do `example[0] = "C"` there are no errors because the element with key 0 is updated with the new value.

Retrieving a value from a map is fairly simple: `var v = example[0];`. If you pass a key that's **not** in the map, a `null` reference is returned.

```
final example = <int, String>{
    0: "A",
    1: "B"
```

```
};

final ex1 = example[0]; // ex1 = "A"
final ex1 = example[8]; // ex1 = null
```

In general it'd be better using `bool containsKey(T key);` before accessing the item as it tells whether the key is present or not in the map. The removal of an element simply happens with the *remove* method.

6.2.3.1 Implementation

You've already seen how lists and sets work and maps are no different; the core library defines an abstract class with a factory constructor that returns a concrete instance. It's the usual pattern adopted by the Dart API for collections.

```
abstract class Map<K, V> {
    // code...
    external factory Map();
    // code...
}
```

The Dart VM loads the body of the constructor from its internals thanks to the `external` keyword. If you could read the code, you'd see that the constructor returns a concrete instance of a class called `LinkedHashMap`. There are three types of maps you can use.

1. `LinkedHashMap<K, V>`. It's the default class returned by the factory constructor of `Map` so when you use `final map = <int, int>{}` you get an instance of `LinkedHashMap`. It's based on a hash-table, the insertion order is remembered and you can iterate over key/values.
2. `HashMap<K, V>`. It's based on a hash table and `null` can be a valid key. The insertion order is **not** remembered so if you iterate over key/value pairs you won't get them in the order in which they have been added.
3. `SplayTreeMap<K, V>`. It's based on a self-balancing BST and keys are compared with the `comparator` function you're asked to pass in the constructor. If the comparison function is not passed, the compiler assumes that keys implement the `class Comparable<T>` [4] interface.

The first implementation, the default one, guarantees the insertion order, the second one doesn't care about the order and the last keeps the keys sorted. There is not the best implementation because you have to choose one of them according with what you have to do; in general the default is good.

6.3 Good practices

You've seen that the language offers three important categories of containers and in general you should use them with their default implementations. A "default" map for example is generally

[4] https://api.dart.dev/stable/2.7.0/dart-core/Comparable-class.html

`final myMap = <int, int>{};` but if you need to keep the keys sorted, go for a `SplayTreeMap`. From the official documentation [5]:

- Use the literal initialization syntax when you're good with the default implementation given by the language. Instead, for specific implementations, use regular constructors or factories.

```
final example = List<int>();   // Bad
final example = <int>[];       // Good
```

- When you iterate on a container prefer doing `for(item in list) {}` instead of using the `forEach()` method which adds verbosity to the code. The loop is clearer. However, if you have a function that can be referenced such as `print`, you can use `list.forEach(print);`.
- There's a function called `cast()` we've never discussed because it's not good to use. Don't do casts with `cast<T>()` because it adds verbosity and it actually doesn't do a neat work. Ditch it.

Each container offers a series of factories, such as `unmodifiable()` to return a collection with no add/remove operations (it's a read-only container). Refer to the docs [6] for a complete list of utilities methods.

6.3.1 operator== and hashCode

You already know from *5.3* "The Object class" how to properly override the equality operator **AND** the `hashCode` property. Sets and maps heavily use comparisons and the hash code of a given object so you really want to do a good override.

> Very shortly, imagine a *hash table* as a table with two columns: the key is on the left and the value is on the right. The key is needed to search values because, if present, you'll get access to the associated object: that's why maps are key/value pairs.

Common implementations of maps and sets are based on hash tables and the hash code determines how an item should be stored in the container. You should always override the equality operator **and** `hashCode` in your classes.

- `bool operator==(Object other)`. We've already seen how to override it in 5.3.
- `int get hashCode`. There are multiple ways to correctly override this getter but what's important is that a different integer is returned for every different value.

```
class Test {
    final int a;
    final int b;
    final String c;
    Test(this.a, this.b, this.c);
```

[5] https://dart.dev/guides/language/effective-dart/usage#collections
[6] https://api.dart.dev/stable/2.7.0/dart-core/dart-core-library.html

```
            bool operator==(Object other) {...}

            int get hashCode {
                const prime = 31;
                var result = 1;

                result = prime * result + a.hashCode;
                result = prime * result + b.hashCode;
                return prime * result + c.hashCode;
            }
        }
```

In general, it's a good practice taking every instance variable of the class and perform a series of multiplications with prime numbers. In this way, every time you create an instance of the same object, such as `Test(0, 1, "a");` you'll always get the same hash code. Any other object, will return a different value.

If you had a class with a lot of member variables, there would be for sure a lot of boilerplate code. The *Equatable* [7] package by Felix Angelov overrides `operator==` and `hashCode` in your class automatically so that you won't have to deal with multiplications and comparisons.

```
class Test extends Equatable {
    final int a;
    final int b;
    final String c;
    Test(this.a, this.b, this.c);

    @override
    List<Object> get props => [a, b, c];
}
```

You just need to subclass `Equatable` and override the `props` getter passing it every `final` field of your class. The package does nothing special internally: it overrides `operator==` (with `identical()`) and `hashCode` (with a series of XORs similarly to what we did). It's much less code for you to write!

```
class Test extends SomeClass with EquatableMixin {
    final int a;
    final int b;
    final String c;
    Test(this.a, this.b, this.c);

    @override
    List<Object> get props => [a, b, c];
```

[7] https://pub.dev/packages/equatable

```
    }
```

Extending `Equatable` might not always be possible because, for example, your class might already have a superclass and Dart doesn't allow multiple inheritance. In this case, use a `mixin` which does the same work.

> ℹ If your class is **NOT** immutable, because not every instance field is `final`, do **NOT** override `operator==` and `hashCode`. Overriding `hashCode` with a mutable object could break hash-based collections. This is also written in the official Dart design guidelines [8].

If your class is mutable, do not define a custom equality logic because it could break hash-based collections; for the same reasons, do not use `Equatable` if you class is not immutable.

6.3.2 Transform methods

Collections give you a very nice way to filter data and act on them with a series of methods that can be chained. There are similarities in Java with *streams* and in C# with *LINQ* queries.

```
void main() {
    // Generate a list of 20 items using a factory
    final list = List<int>.generate(20, (i) => i);

    // Return a new list of even numbers
    final List<String> other = list
        .where((int value) => value % 2 == 0) // 1.
        .map((int value) => value.toString()) // 2.
        .toList();                            // 3.

}
```

In this example we're creating a list containing numbers from 0 to 19 using the `generate` factory constructor. The interesting part is how we've built `other` so that it contains only strings representing even numbers.

1. The `where()` method iterates across the entire collection and returns a boolean expression. Here we analyze each element of the list, represented by `int value`, and we discard it in case it's not even. This method is a "filter" that adds values only if the boolean expression returns `true`.
2. The `map()` method transforms a type into another. Since `other` must be a list of strings, we transform each filtered element (represented by `int value`) into a `String value`.
3. Now that we have a filtered list of transformed values, the terminal function returns an instance of a list.

[8] https://dart.dev/guides/language/effective-dart/design#avoid-defining-custom-equality-for-mutable-classes

Manually doing this kind of operation is actually verbose because you should create a function with temporary variables, loops and conditional statements. This syntax instead is elegant and very easy to understand.

> ℹ While you traverse a collection with these methods, do **NOT** alter the content of the container itself! If you have a list of strings, don't change each item calling, for example, `toUpper()` while you're in a `where` condition.

The example is shown with a list but these methods are also available for sets and maps. There are two kind of operations: "intermediate" operations to process data and "terminal" operations to return values. Intermediates are meant to elaborate data and can be chained while terminals are called at the end to "group" the data.

- **Intermediates**. This is a category of functions that can be chained like you've seen above to create complex expressions. Most of them accept a function whose parameter is the element of the collection being accessed.
  - `where()`: goes through the entire list and discards elements that evaluate the condition to `false`.
  - `map()`: transforms the element from a type to another.
  - `skip()`: skips the first n elements of the source collection.
  - `followedBy()`: concatenates this container with another one, passed as parameter.
- **Terminals**. This is a category of function that can only be called at the end of the chain to return a value or an object.
  - `toList()`/`toSet()`/`toMap()`: gathers the elaborated data through the "pipes" and returns an instance of a list/set/map.
  - `every()`: returns a boolean indicating if every element of the collection satisfies the given condition.
  - `contains()`: returns `true` or `false` whether the collection contains or not the object you're looking for.
  - `reduce()`: reduces a collection to a single value which can be the result of operations in the elements of the container. You cannot use `reduce()` on empty collections. For example:

    ```dart
    final list = <int>[1, 2, 3, 4, 5];
    final sum = list.reduce((int a, int b) => a + b);

    print(sum); // 15
    ```

    The variable `sum` contains the sum of the elements in the list since `reduce((a,b) => c)` takes 2 elements of the source (`a`, `b`) and performs the given action on them (in this case, it sums the values).

- `fold()`. It's very similar to `reduce()` but it asks for an initial value and the returned type doesn't have to be the same of the collection.

  ```
  final list = <int>[1, 2, 3, 4, 5];
  final sum = list.fold(0, (int a, int b) => a + b);

  print(sum); // 15
  ```

Both `reduce()` and `fold()` can do the same things but the latter is more powerful. First of all, `fold()` can define a custom initial value for the operations:

```
final list = <int>[1, 2, 3, 4, 5];

final sum1 = list.fold(0, (int a, int b) => a + b);
final sum2 = list.fold(5, (int a, int b) => a + b);

print(sum1); // 15
print(sum2); // 20
```

With `fold()` you can perform operations on different data types while with `reduce()` you cannot. In this example, we're computing the sum of the lengths of strings in a collection.

```
final list = ['hello', 'Dart', '!'];

final value = list.fold(0, (int count, String item) => count + item.length);
print(value); // 10
```

`count` has the same type of the initial value (0 in this case, which is an `int`) and `item` represents an object in the collection. The returned value of the function must match the type of the initial value. You can't do the same in the other way:

```
final list = ['hello', 'Dart', '!'];

// It doesn't compile
list.reduce((String a, String b) => a.length + b.length);
print(value);
```

This version doesn't work because `reduce()` expects the return type of the callback to be a `String`, the same type of the container. With `fold()` you don't have this constrain: it will always work. In reality, `reduce()` can be seen as a shortcut of the following:

```
final withReduce = list.reduce(someCallback);
final withFold = list.skip(1).fold(list.first, someCallback);
```

The two versions are equivalent but `withReduce` is just shorter. We strongly encourage you to use this fluent syntax when you have to work on collections rather than using temporary variables and/or conditional statements.

7 | Asynchronous programming

7.1 Introduction

Nowadays computers and mobile devices are very fast and users are well aware of this; they hate when the application "*freezes*" for a moment or if it doesn't always react immediately to inputs. There are however some situations in which the user must wait:

- database operations;
- usage of the internet connection, which might be slow and thus the entire process could take longer than expected;
- many I/O operations might slow down your app's performances due to the policies adopted by the OS.

In Flutter for example you often use an internet connection and, in the worst case, the user has to wait a few seconds. You *must* use asynchronous programming to show something like an animated progress bar while, at the same time, data are processed in the background.

> ℹ We'll talk about this in the Flutter part but the idea is that the app should never stop at a single long task. Asynchronous programming is made for executing time-consuming operations in the background so that, in the meanwhile, we can do something else.

Let's say you're working in a team and a colleague of yours sends you this function which is going to be use very often in your application.

```
int processData(int param1, double, param2) {
    var value = 0;

    for(var i = 0; i < param1; ++i) {
        for (var j = 0; j < param1*param2; j++) {
            // a lot of work here...
        }
    }
```

```
        return httpGetRequest(value);
    }
```

Suppose that the two nested loops may take up to 2 seconds to complete and the network request at the end adds other hundreds of milliseconds. For sure this function is slow as it returns the `int` after quite a lot of time (in the order of seconds).

```
    void main() {
        final data = processData(1, 2.5);
        print(data);

        print("Welcome to... Dart!");
    }
```

The `main()` function is going to be "blocked" for some seconds due to the long execution time of `processData`. The entire flow is stuck due to a bottleneck produced by a function call and the app itself looks like it's frozen.

7.2 Futures

A `Future<T>` represents a value or an error that will be available in the future. This generic class should be used whenever you're working with time-consuming functions returning a result after a notable amount of time. Here's what you can do to easily slim your execution flow:

```
    Future<int> processData(int param1, double, param2) {
        var value = 0;

        for(var i = 0; i < param1; ++i) {
            for (var j = 0; j < param1*param2; j++) {
                // a lot of work here...
            }
        }

        final res = httpGetRequest(value);
        return Future<int>.value(res);
    }
```

It's almost identical the original code: we've just changed the type from `int` to `Future<int>` and the final statement, which uses a named constructor of `Future` to return a new instance. Of course you must be sure that the `Future<T>.value()` object is built with the proper type.

The code didn't change so much but there's a huge difference in how the function is going to be called. In addition, every time you see `Future<T>` as return value, you immediately figure out the usage and thus you write your code consequently.

Since the function now returns a `Future<T>` we have to treat it differently:

```
// Types are explicit for sake of simplicity
void main() {
    Future<int> val = processData(1, 2.5);
    val.then((result) => print(result));
}
```

The `then()` callback gets called once the execution has finished, when the value is ready to be used. Thanks to a `Future<T>` you're able to execute the time-consuming task in the background and be notified of the completion via `then()`.

```
// Types are explicit for sake of simplicity
void main() {
    Future<int> val = processData(1, 2.5);
    val.then((result) => print(result))
       .catchError((e) => print(e.message));
}
```

Methods can be chained; catching potential exceptions thrown during the background execution happens via `catchError()`, which is the equivalent of a `try catch` block. Of course you can create more complex chains such as:

```
val.then((result) => anotherFunction1(result))
   .then((another) => anotherFunction2(another))
   .then((ending) => anotherFunction3(ending))
   .catchError((e) => print(e.message));
```

There are not limits, you can always append a `then()` or a `catchError()`. You might have noticed that this approach is quite verbose and it's not so easy to read when many methods are chained.

ℹ That's the reason why `async` and `await` must be your primary choice; they drastically reduce the verbosity making the code look almost identical to its synchronous counterpart.

In certain cases, you might want to wait for a series of `Future<T>`s to complete but you still don't want to block the execution. This is the perfect use case for `Future.wait<T>()`.

```
Future<int> one = exampleOne();
Future<int> two = exampleTwo();
Future<int> three = exampleThree();

Future.wait<int>([
    one,
    two,
    three
]).then(...).catchError(...);
```

The `wait()` method takes a list of `Future<T>`s, executes them and waits until everyone has finished. You can chain `then()` and/or `catchError()` because `wait()` returns a `Future<T>`. The API is very rich of useful named constructors you can use:

- `Future<T>.delayed()`

  ```
  final future = Future<int>.delayed(const Duration(seconds: 1), ()=> 1);
  ```

  Creates a `Future<T>` object that starts running after the given delay (in this case, it executes after 1 second).

- `Future<T>.error()`

  ```
  final future = Future<double>.error("Fail");
  ```

  Creates a `Future<T>` object that terminates with an error. Other than the message, you can also pass as second parameter the stack trace.

- `Future<T>.value()`

  ```
  final future = Future<String>.value("Flutter Complete Reference");
  ```

  Creates a `Future<T>` object that completes immediately returning the given value. Basically, this constructor is used to "wrap" a non-future value into a future value.

- `Future<T>.sync()`

  ```
  final future = Future<void>.sync(() => print("Called immediately"));
  ```

  Creates a `Future<T>` object that immediately calls the given callback. Generally, when calling `then()` you don't know when its body will be executed. In this case, you know that the callback is called immediately. This constructor is intended to be used when a `Future<T>` has to execute immediately but in practice, there are a very few usages (we will see one in 13.2.2 for example).

7.2.1 Comparison

In this section we're comparing a synchronous code snippet, which uses a "simple" `int`, and its asynchronous version, which uses a `Future<int>`, to emphasize the different behaviors.

1. **Non-future version** (synchronous code).

   ```
   int processData(int param1, double, param2) {
       // takes 4 or 5 seconds to execute...
   }

   void main() {
       final data = processData(31, 2.5);
       print("func result = $data");

       print("Future is bright");
   }
   ```

Nothing difficult to understand up to here, you can easily predict what the console is going to output (10 is just there as example, it doesn't matter):

```
func result = 10;
Future is bright
```

Both `print()` statements are executed in sequence (as usual) but they appear after a few seconds. There is a visible delay which temporarily "freezes" the program because `processData()` blocks the execution flow while performing calculations.

2. **Future version** (asynchronous code).

```
Future<int> processData(int param1, double param2) {
    // function that takes 4 or 5 seconds to execute...
}

void main() {
    final process = processData(1, 2.5);
    process.then((data) => print("result = $data"));

    print("Future is bright");
}
```

The output is now different:

```
Future is bright
result = 10; // <-- printed after 4 or 5 seconds
```

With the usage of a `Future<T>` object the execution flow doesn't get blocked anymore. The `then(...)` callback returns **immediately** so that other operations can take place; its body will be executed later once data are actually ready. If you had written...

```
final process = processData(1, 2.5);
process.then((data) {
    print("result = $data");
    print("Future is bright");
});
```

... then the console would have printed ...

```
result = 10;
Future is bright
```

... as in the first example. The body of `then()` executes synchronously in our example so operations are executed in sequence.

Inside a `then()` callback you can execute asynchronous code as well but it's difficult to read, due to the verbosity of the code, and hard to understand, in case there were too much asynchrony in the flow.

ℹ To put it very simply, when you use `then()` you're telling Dart: "*Continue doing your work, I don't want to wait for the operation to finish. When the result will be ready, notify me with the callback*".

If you have to deal with time-consuming operations, using a `Future<T>` is basically a **must** because blocking the execution flow is dangerous and wrong. Thanks to asynchronous code you keep your app always busy and **responsive**, which is a fundamental user experience factor.

7.2.2 async and await

The usage of `async` and `await` makes the code less verbose and consequently easier to understand. It's just syntactic sugar to avoid the usage of `then()` to write callbacks:

- **Using then**.

  ```
  void main() {
      final process = processData(1, 2.5);
      process.then((data) => print("result = $data"));
  }
  ```

- **Using async and await**.

  ```
  void main() async {
      final data = await processData(1, 2.5);
      print("result = $data")
  }
  ```

The above snippets are **equivalent** because the result is the same but the syntax is different. Let's start with three very important facts:

1. You can use `await` only in a function marked with `async`.
2. To define an asynchronous functions, put the `async` keyword before the body.
3. You're allowed to call `await` only on a `Future<T>`.

Our `main()` has the `async` modifier so that we're allowed to call `await`. Calling `await` on a `Future` moves the execution to the background and proceeds once the computation is done (which is exactly what `then()` does). To be clear, writing...

```
processData(1, 2.5).then((data) => print("result = $data"));
```

... is the same as ...

```
final data = await processData(1, 2.5);
print("result = $data");
```

... because the lines **after** the `await` keyword are executed only when the `Future<T>` completed (without blocking). In regard to the previous example, this code...

```
void main() async {
    final data = await processData(1, 2.5);
    print("result = $data");
    print("Future is bright");
}
```

... is equivalent to ...

```
void main() {
    final process = processData(1, 2.5);
    process.then((data) {
        print("result = $data");
        print("Future is bright");
    });
}
```

... but absolutely **NOT** equivalent to ...

```
void main() {
    final process = processData(1, 2.5);
    process.then((data) {
        print("result = $data");
    });
    print("Future is bright");
}
```

because **everything** after `await` is executed only when the `Future` is completed. You're guaranteed that functions are asynchronously executed and you won't block the normal execution flow. Exceptions are also easier to catch because...

```
void main() {
    processData(1, 2.5)
        .then((result) => print(result))
        .catchError((e) => print(e.message));
}
```

... gets simplified with the usage of `async` and `await`:

```
void main() async {
    try {
        final result = await processData(1, 2.5);
        print(result);
    } on Exception catch (e) {
        print(e.message);
    }
}
```

The second version is closer to what you're used to see in the traditional synchronous world. Furthermore there are no nested methods/callbacks and thus the code is way shorter and more readable.

7.2.3 Good practices

The first thing stated by the official [1] usage guidelines is "*prefer async/await over using raw futures*". Since asynchronous code can be hard to read and debug, you should prefer `async` and `await` over a chain of `then()` and `catchError()`.

```
Future<String> example() async {
    try {
        final String data = await httpGetRequest();
        final String other = await anotherRequest(data);
        return other;
    } on Something catch (e) {
        print(e.message);
        return "fail";
    }
}
```

If it weren't for `await` it would look like "normal" synchronous code with no callbacks at all. It's neater and more readable if compared to the following:

```
Future<String> example() {
    return httpGetRequest().then((data) {
        anotherRequest(data).then((otherData) {
            return otherData;
        });
    }).catchError((e) {
        print(e.message);
        return "";
    });
}
```

It's not a matter of efficiency or performances because both examples are fine. The problem is about writing code with potentially many nested callbacks and functions that become impossible to read (the so called *"callback hell"*).

ℹ Again, thanks to `await` asynchronous code can be written in the same way as synchronous code. Other than leading to less boilerplate, it's also easier to read, maintain and understand!

[1] https://dart.dev/guides/language/effective-dart/usage#asynchrony

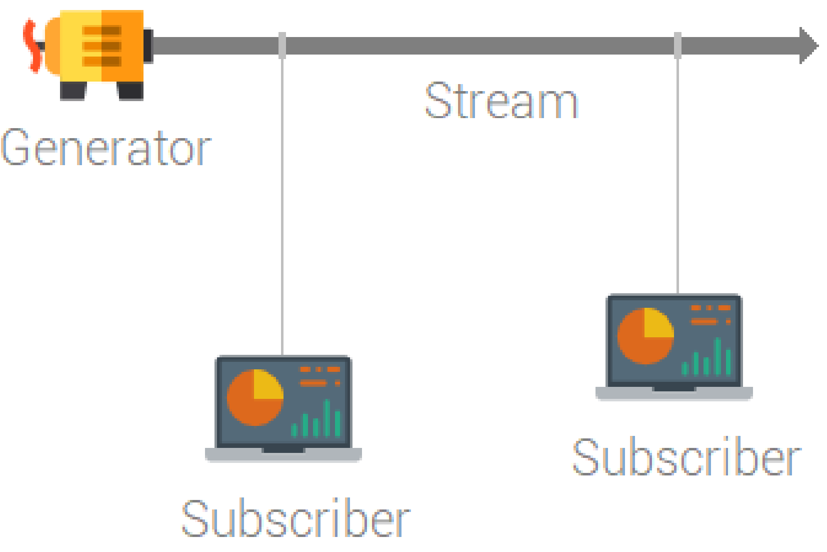
Returning a `Future<T>` from a function can be done via named constructor `Future.value()` or, more easily, by making the function `async` and returning the plain value. The compiler will make an automatic conversion.

```
// Use the named constructor
Future<int> example() => Future<int>.value(3);

// Use async and the compiler wraps the value in a Future
Future<int> example() async => 3;
```

Both ways are valid but maybe you should prefer the second approach as it's a bit less verbose.

7.3 Streams

In Dart a *stream* is a sequence of asynchronous or synchronous events we can listen to. There's no need to check for updates because the stream notifies us automatically when there's a new event available.



This picture helps you to visualize how streams are intended to be used and who are the main actors involved. A *generator* is a source of information that lazily generates new data with a certain frequency; the *stream* is the pipe in which generated data flow.

- **Generator**. Creates new data and sends them over the stream.
- **Stream**. It's the place in which the generated data flow. You can start listening to a stream so that, when the generator emits new data, you will be notified.

- **Subscribers**. A subscriber is someone interested in the data travelling in the stream. If new data are sent over the stream by the generator, everyone listening (*subscribers*) will be notified.

A generator has to emit data only into a stream and nowhere else because a stream is the only reference for listeners to subscribe. There are two types of generators:

1. Asynchronous generators: they return a `Stream<T>` object. Because of this, you have to deal with an asynchronous flow of data that has to be handled by a subscriber with `await`.
2. Synchronous generators: they return an `Iterable<T>` object. Because of this, you have to deal with a synchronous flow of data that can be handled in a loop because data are sent in a sequential order.

A Flutter developer is used to work with `Stream<T>` because the framework has many asynchronous generators. In practice, unless you're creating a package or a specific tool, you won't create generators too often but still you should at least be aware of how they generally work. Futures and streams are at the basics of Dart's asynchronous model.

7.3.1 Streams and generators

As a basic example, we're going to create an asynchronous generator producing 100 random numbers, one per second. In order to tell the compiler that this function is a generator, it has to be marked with the `async*` modifier.

```
Stream<int> randomNumbers() async* {            // 1.
  final random = Random();

  for(var i = 0; i < 100; ++i) {                // 2.
    await Future.delayed(Duration(seconds: 1)); // 3.
    yield random.nextInt(50) + 1;               // 4.
  }
}                                               // 5.
```

1. Since the function is a generator of asynchronous events (random numbers), the return type must be of type `Stream<T>`. The `async*` modifier allows the usage of `yield` to emit data.
2. The loop generates 100 random numbers.
3. The `Future.delayed(...)` named constructor creates a `Future` that returns after a certain delay given by the `Duration`[2] object. It's used to "sleep" the execution flow for a certain time without blocking.
4. The `yield` keyword pushes data on the stream. It is responsible of sending new events on the stream and it doesn't alter the loop (it continues to cycle regularly).
5. When a function has the `async*` modifier there **cannot** be a `return` statement. It would also be logically wrong because data are already sent over the stream by `yield` and thus you'd have

[2]See appendix A.3 to know more about `Duration`

nothing to return when the generator "turns off".

So generators are created with the `async*` modifier and events are emitted on the stream with the `yield` keyword. To make a comparison, here's the synchronous version of the generator which has a similar structure.

```
// contains the 'sleep' function
import 'dart:io';

Iterable<int> randomNumbers() sync* {
  final random = Random();

  for(var i = 0; i < 100; ++i) {
    sleep(Duration(seconds:1));
    yield random.nextInt(50) + 1;
  }
}
```

The `sync*` star modifier tells the compiler that this function is a synchronous generator. Due to its nature of being *synchronous* you **cannot** use futures and thus we must use `sleep()` instead of awaiting `Future.delayed()`.

Asynchronous generators are meant to be used with asynchronous code. Synchronous generators are meant to be used with synchronous code. Intuitively, if your code needs to `await` something, you're going to need an asynchronous generator.

Both kind of generators start emitting data **on demand**, meaning that values are produced when a listener starts iterating on `Iterator<T>` or starts listening to `Stream<T>`. There can be more than a single `yield` statement in the same block and it would simply push many values, in sequence, on the stream:

```
Stream<int> randomNumbers() async* {
  final random = Random();

  for(var i = 0; i < 100; ++i) {
    await Future.delayed(Duration(seconds: 1));
    yield random.nextInt(50) + 1;
    yield random.nextInt(50) + 1;
    yield random.nextInt(50) + 1;
  }
}
```

This code emits 3 random number at each iteration so the stream is going to generate 300 values before completing. Streams can also be created using a series of useful named constructors for a "quick setup":

- `Stream<T>.periodic()`

  ```
  final random = Random();

  final stream = Stream<int>.periodic(
      const Duration(seconds: 2),
      (count) => random.nextInt(10)
  );
  ```

  Creates a new a stream that repeatedly emits events at the given `Duration` interval. The argument of the anonymous function starts at 0 and then increments by 1 for each event emitted (it's an "event counter").

- `Stream<T>.value()`

  ```
  final stream = Stream<String>.value("Hello");
  ```

  Creates a new stream that emits a single event before completing.

- `Stream<T>.error()`

  ```
  Future<void> something(Stream<int> source) async {
      try {
          await for (final event in source) { ... }
      } on SomeException catch (e) {
          print("An error occurred: $e");
      }
  }

  // Pass the error object
  something(Stream<int>.error("Whoops"));
  ```

  Creates a new stream that emits a single error event before completing; the behavior is very similar to `Stream<T>.value()`.

- `Stream<T>.fromIterable()`

  ```
  final stream = Stream<double>.fromIterable(const <double>[
      1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9
  ]);
  ```

  Creates a new single-subscription stream that only emits the values in the list.

- `Stream<T>.fromFuture()`

  ```
  final stream = Stream<double>.fromFuture(
      Future<double>.value(15.10)
  );
  ```

Creates a new single-subscription stream from the given Future<T> object. In particular, when the Future<T> completes, 2 events are emitted: one with the data (or the error) and another to signal the stream has terminated (the "done" event).

- Stream<T>.empty()

  ```
  final stream = Stream<double>.empty();
  ```

  This kind of stream does nothing: it just immediately sends a "done" event in order to signal the termination.

You should really check out the Stream<T> online documentation for a complete refernece of all the methods you can call on it. The most "popular" ones for example are:

- drain(...): it discards all the events emitted by the stream but signals when it's either done or an error occurred;
- map(...): transforms the events of the current stream into events of another type;
- skip(int count): skips the first count event on the stream;

7.3.2 Subscribers

The generator is now ready to periodically emit new random numbers and thus we can subscribe to get notified for new values.

```
Stream<int> randomNumbers() async* {
    // see code above...
}

void main() async {                   // 1.
  final stream = randomNumbers();     // 2.

  await for (var value in stream) {   // 3.
    print(value);
  }

  print("Async stream!");             // 4.
}
```

1. Since we're dealing with an asynchronous stream, there's the need to mark the function with async because we're going await soon.
2. We subscribe to the stream by simply getting a reference to it. This is the moment in which the generator starts emitting data because someone has just started listening (on-demand initialization).
3. When dealing with streams, the await for loop is able to "catch" values sent over the stream by a yield in the generator. It works exactly like a regular for loop.

4. The string is printed when the loop terminates so it will appear at the end.

To be precise, the example is *almost* good because generators are generally put inside classes and exposed with a getter, which returns a `Stream<T>` instance, or with a `listen()` method which does a manual subscription. Let's give a look at the synchronous version of the generator:

```
Iterable<int> randomNumbers() sync* {
    // see code above...
}

void main() {
  final stream = randomNumbers();

  for(var value in stream) {
    print(value);
  }

  print("Sync stream!");
}
```

Apart from a plain `for` instead of an `await for` the code is identical. The `List<T>` and `Set<T>` classes are both `Iterable<T>`, exactly like most types in the collection library. Exceptions are caught in the usual way:

```
void main() {
  final stream = randomNumbers();

  try {
    await for(var value in stream) {
        print(value);
    }
  } on Something catch (e) {
    print("Whoops :(");
  }
}
```

Other than *data events* a stream can also send *error events*, which may occur because an exception has been thrown in the generator. Here's another example of a stream continuously sending data at a given interval:

```
Stream<int> counterStream([int maxCount = 10000]) async* {
    final delay = const Duration(seconds: 1);
    var count = 0;

    while (true) {
        if (count == maxCount) {
            break;
        }
        await Future.delayed(delay);
        yield ++count;
    }
}

void main() async {
    await for(var c in counterStream) {
        print(c);
    }
}
```

Each second a new number is printed to the console until `maxCount` is reached; if you wanted the loop to last forever, just remove the `if` condition. For an even more detailed coverage about streams, check out the official Flutter YouTube channel along with the documentation which is full of details and examples:

1. Stream usage: https://dart.dev/tutorials/language/streams
2. Generators: https://dart.dev/articles/libraries/creating-streams
3. Stream docs: https://api.dart.dev/stable/2.9.2/dart-async/Stream-class.html

In Flutter the `StreamBuilder<T>` object is used to subscribe to a stream and we're going to use it quite often, especially in Part III of the book.

7.3.3 Differences

Key differences for asynchronous and synchronous streams are summarized in this short table:

Asynchronous	**Synchronous**
Returns a `Stream<T>`	Returns an `Iterable<T>`
Mark function with `async*`	Mark function with `sync*`
Can use `await`	Cannot use `await`
Subscribers have to use `await for`	Subscribers have to use `for`

In both cases `yield` is used to send data on the stream and there must not be a `return` statement in the function. You might be wondering: "If I wanted to write a generator, should I make it synchronous or asynchronous?"

As we've already said, you rarely need to write a generator if you work with Flutter, unless you're doing something specific. In most of the cases you will **subscribe** to streams so you won't have to decide anything because the generator is exposed by the library.

A reasonable answer to the question would be "*it depends*" because there are cases and cases. As a general guideline we can say that a good decisional aspect is whether you have to use `Future<T>` or not.

- If you have to deal with `Future<T>`s because of network usages or I/O operations for example, you are forced to use an asynchronous generator otherwise you can't use `await`.
- When you don't have to deal with asynchrony and you have the need of sending a series of sequential data, go for a synchronous stream.

Sometimes, especially when using the *flutter_bloc* library we will cover in chapter 11, it can be useful splitting the logic of a `Stream<T>` into multiple pieces. This is the case where `yield*` is required:

```
Stream<int> numberGenerator(bool even) async* {
    if (even) {
        yield 0;
        yield* evenNumbersUpToTen();
        yield 0;
    } else {
        yield -1;
        yield* oddNumbersUpToTen();
        yield -1;
    }
}

Stream<int> evenNumbersUpToTen() async* { ... }
Stream<int> oddNumbersUpToTen() async* { ... }
```

Basically `yield*` is used to "pause" the execution and start emitting values from the other stream; once finished, the source stream is "restarted" so that it can regularly send its values again. To be more clear, here's an example of what happens when `numberGenerator(true)` is called:

- The value `0` is emitted with `yield`, the "normal" way of sending data on the stream.
- Because of `yield*`, `numberGenerator` pauses and starts emitting values generated from the other stream (`evenNumbersUpToTen`).
- Once `evenNumbersUpToTen` has completed, `numberGenerator` resumes and executes the next `yield` statement.

In practice `yield*` is used to say "stop here, emit values from the other stream and when it's completed you're free to restart your regular flow". This pattern is used when a stream should internally use another stream to split the logic in multiple functions to make the code more readable.

7.3.4 Using a controller

The examples we've shown so far aren't very useful actually. In particular, the creation of an "in place" stream hasn't much use cases a part from examples and demos. You're already familiar with this simple setup:

```
Stream<String> someStream() async* { ... }

void main() async {
    final stream = someStream();
}
```

The stream is started as soon as we do the `stream = someStream();` assignment. This is the manual way of creating streams but it doesn't scale well on larger applications. Especially in Flutter, you'll find out that `StreamController<T>` is a more convenient way to work with streams. We're going to create a more complex stream that periodically produces random numbers.

```
/// Exposes a stream that continuously generates random numbers
class RandomStream {
    /// The maximum random number to be generated
    final int maxValue;
    static final _random = Random();

    Timer? _timer;
    late int _currentCount;
    late StreamController<int> _controller;

    /// Handles a stream that continuously generates random numbers. Use
    /// [maxValue] to set the maximum random value to be generated.
    RandomStream({this.maxValue = 100}) {
        _currentCount = 0;
        _controller = StreamController<int>(
            onListen: _startStream,
            onResume: _startStream,
            onPause: _stopTimer,
            onCancel: _stopTimer
        );
    }

    /// A reference to the random number stream
    Stream<int> get stream => _controller.stream;

    // other methods coming soon...
}
```

Notice how we've used triple slashes (///) to document the code [3]. The `Timer` class comes from the `dart:async` package: it's a count-down timer that can be configured to fire once or repeatedly. It counts down from the given duration up to 0 and then triggers the callback. It has two constructors:

- `Timer(Duration duration, void callback())`
  Executes once the callback after the given duration.
- `Timer.periodic(Duration duration, void callback(Timer timer))`
  The callback is invoked repeatedly with `duration` intervals.

A timer can be stopped with `cancel()`. We're using it to push each second new random numbers in the stream, handled by `StreamController<T>`. This class is basically a wrapper around a stream with many facilities to easily send data, errors and done events.

- `onListen`: this callback is called when the stream is listened to (new subscription made).
- `onCancel`: this callback is called when the stream is canceled (subscription canceled).
- `onPause`: this callback is called when the stream is paused (subscription paused).
- `onResume`: this callback is called when the stream is resumed (subscription resumed).

A `StreamController<T>` is a very handy tool to easily manage a `Stream<T>`. Thanks to `get stream` we expose to the outside a reference to the stream so that listeners can subscribe and receive events. This is how we've defined the callbacks of the controller:

```
void _startStream() {
    _timer = Timer.periodic(const Duration(seconds: 1), _runStream);
    _currentCount = 0;
}

void _stopTimer() {
    _timer?.cancel();
    _controller.close();
}

void _runStream(Timer timer) {
    _currentCount++;
    _controller.add(_random.nextInt(maxValue));

    if (_currentCount == maxValue) {
        _stopTimer();
    }
}
```

[3]More on documenting code in 23.1.2

We have declared `Timer? _timer` as a nullable variable because we cannot immediately initialize the timer in the constructor. Doing so would be an error because events would start being emitted on the stream from the beginning, even if there are **no** listeners!

```
RandomStream({this.maxValue = 100}) {
    _currentCount = 0;
    _controller = StreamController<int>(...);

    // WRONG! In this way, the timer is started and thus events are
    // emitted on the stream immediately (even if no one is listening)
    _timer = Timer.periodic(...);
}
```

We safely use the `?.` operator to access the nullable variable. Inside `_startStream()` we actually initialize the timer so that it pushes new random values every 1 second. The actual processing is done inside `_runStream()`:

```
// New value added to the stream. Listeners will be notified
_controller.add(_random.nextInt(maxValue));

// When the maximum value is reached, we need to stop both the
// timer AND close the controller to stop the stream.
if (_currentCount == maxValue) {
    _stopTimer();
}
```

We can now play with our `RandomStream` class. In this example, we're subscribing to the stream using `listen()` and then we cancel the subscription after a certain delay. You'll see that random numbers are printed to the console only 3 times in total.

```
void main() async {
    final stream = RandomStream().stream;
    await Future.delayed(const Duration(seconds: 2));

    // The timer inside our 'RandomStream' is started
    final subscription = stream.listen((int random) {
        print(random);
    });

    await Future.delayed(const Duration(milliseconds: 3200));
    subscription.cancel();
}
```

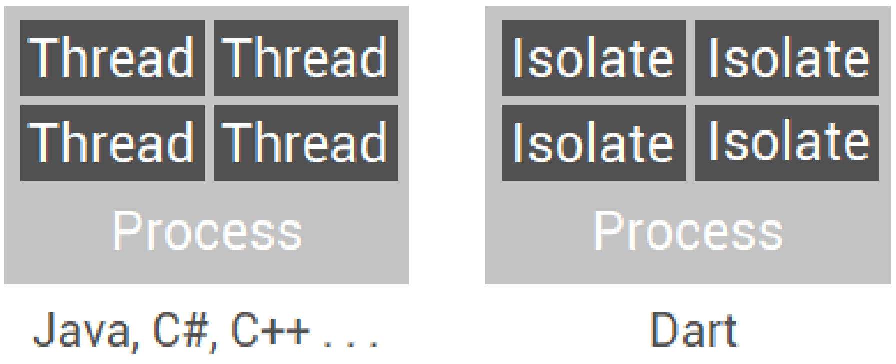
After 2 seconds, we subscribe to the stream using `listen()` but numbers are printed only three times because, 3 seconds later, we cancel the subscription. As you can see, `StreamController<T>` is more complex to use but more powerful and scalable: it's the preferred way to work with streams in Dart and Flutter.

7.4 Isolates

Many popular programming languages such as Java and C# have a very wide API to work with multiple threads and parallel computation. They can handle complex multithreading scenes thanks to the various primitives they support. Dart however has none of the following:

- there is no way to start multiple threads for heavy background computation;
- there is no equivalent, for example, of thread-safe types such as `AtomicInteger`;
- there are no mutexes, semaphores or other classes to prevent data races and all those problems arisen from multithreaded programming.

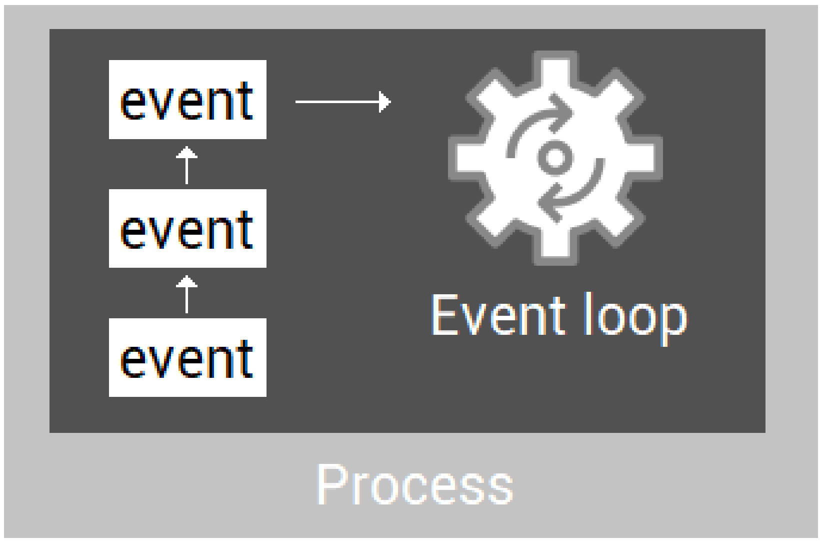
The Dart code (and thus Flutter applications) is run inside an **isolate** which has its own private area of memory and an event loop. An isolate can be seen as a special thread in which an event loop processes the instructions. If you aren't familiar with these concepts, we will break down for you what is going on:



Any program runs in a process which can be made up of one or more threads. Some programming languages (picture on the left) allow you to manually create multiple threads to execute long running tasks in the "background" not to block the UI.

> All the threads living on a process share the **same** memory. You need to be aware of this because writing the same data, at the same time, in the same memory area can lead to problematic situations known as *data races*.

In Dart, a process is made up of one or more **isolates** containing an event loop. Differently from classic threads, each isolate allocates its own memory area so there are no data sharing issues. In other words, the key difference is that threads **do** share the same memory while isolates **don't**. Thanks to this fact, Dart needs no data synchronization primitives since problems like data races can never happen by default. If we made a zoom on an isolate, it would look like this:

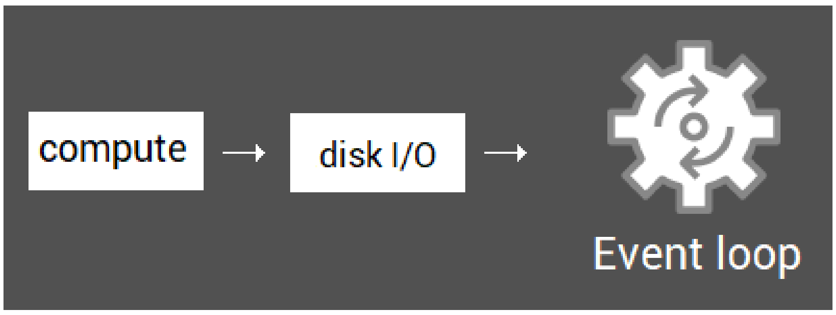


The white *event* rectangles can be anything from I/O disk operations, HTTP requests, actions triggered by a finger tap in the Flutter framework and so on. The gear on the right is the **event loop**, a sort of machinery that continuously executes events. You might be asking yourself: if there's only a single thread, how is asynchronous code executed? Let's take a look at those 2 simple examples:

1. Let's say that somewhere in our Dart program (or Flutter app) there are two methods which get called in sequence. They are synchronous, because inside they use no asynchronous code (no `Stream<T>`s or `Future<T>`s).

   ```
   // this is called first
   var json = myModel.readFromDisk();

   // and this is called after the above
   final result = computeIntegerValue();
   ```

   The event loop processes incoming events in order one by one so first it executes the I/O operation and then the computation. Here's a visual representation of the situation:



The second event is processed only when the first is finished. If there were no events available, the event loop would be in "idle" waiting for new work to do. The event loop is the "engine" that actually executes the Dart code you've written.

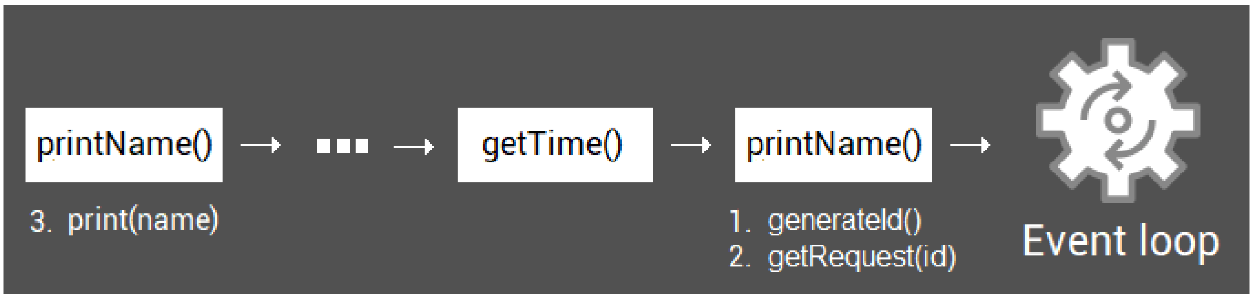
2. Let's now see another example in which a `Future<T>` is involved in order to understand how asynchronous code is processed. The same strategy is also applied when it comes to streams.

```
void printName() async {
    final int id = generateId();
    final String name = await HttpModel.getRequest(id);

    print(name);
}
```

As you already know, what comes after `await` is executed only when the `Future<T>` has terminated. In this case, the value will be printed only when the HTTP request is finished. Pretend to have this code:

```
    printName();
    final time = getTime();
```

The first event to be executed is `printName()` but since it internally calls `await`, what comes after (the `print(name)` statement) is separated and added later as a new event in the queue! This is the actual sequence that will be processed:



In practice, asynchronous calls are divided in multiple events: the synchronous part and the callbacks. What comes after an `await` is not executed immediately because there's the need to wait for the `Future<T>` to finish. In order to not waste time, the callback is divided from the event, "remembered" and added in the queue again later (when the `Future<T>` finished).

```
// This part is executed immediately; it's the rightmost rectangle on
// the image
final int id = generateId();
final String name = await HttpModel.getRequest(id);

// This callback is executed later; it's the leftmost rectangle on the
// image. This part "separated" and added later in the event loop again
// to complete the execution
print(name);
```

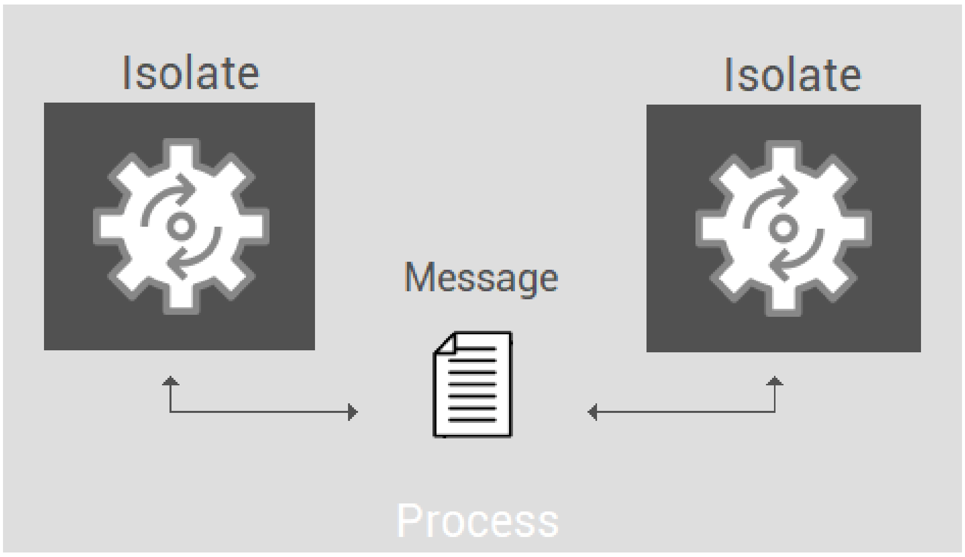
Splitting function calls is fundamental because it avoids events on the queue to wait for futures to finish. If `printName()` were executed entirely, the event loop would have been blocked until the `Future<T>` completed and other events would have to wait.

The event loop should always be busy but it shouldn't execute long-lived events otherwise others will be blocked. In addition, other than events fired by your app there are also other kind of actions to be performed such as garbage collection. To sum it up, here's a comparison with other programming languages:

- **Java** or **C#**. You can create multiple threads to run time-consuming work in the background. Threads share memory, which can be dangerous, but you have a rich API with mutexes, atomic types and so on to keep consistency in your program.
- **Dart**. There's only a single thread with its own memory. You cannot create multiple threads. The event loop processes anything sequentially as soon as possible. In order to not waste time, asynchronous calls are split so that callbacks are executed in a second moment in order to not block the loop.

7.4.1 Multiple isolates and Flutter

A single Dart application can have more than a single isolate; you can create them by using `Isolate.spawn()` from the `"dart:isolate"` library. Isolates have their own event loop and memory area, there are no dependencies or shared components at all. The only way they have to communicate is via *messages*.



Each isolate has a port from which messages enter and exit; they are respectively represented by `ReceivePort` and `SendPort`. A message is regularly processed by the event loop as any other action but this really is the only way to communicate.

> ℹ There's the possibility to also spawn new isolates in Flutter but you'd have to use `Future<T> compute(...)` rather than `Isolate.spawn`.

Working with isolates is quite low level and it's something you generally don't do on a regular basis. Using `async`/`await` is almost always enough. To make a practical example, if you had a Flutter app with a really time-expensive data computing your frame rate might drop under 60fps. This is the case for a new isolate:

```
// Very computational-heavy task
int sumOfPrimes(int limit) {...}

// Function to be called in Flutter
Future<int> heavyCalculations() {
    return compute<int, int>(sumOfPrimes, 50000);
}
```

The `compute()` method requires the function to be executed (which cannot be an anonymous function) and the parameters it needs (if any). If you had the need for multiple input parameters, simply wrap them into a model class and pass it as a dependency, like this:

```
// Model class
class PrimeParams {
    final int limit;
    final double another;
    const PrimeParams(this.limit, this.another);
}

// Use the model as parameter
int sumOfPrimes(PrimeParams data) {
    final limit = data.limit;
    final another = data.another;
    ...
}

// Function to be called in Flutter
Future<int> heavyCalculations() {
    final params = PrimeParams(50000, 10.5);
    return compute<PrimeParams, int>(sumOfPrimes, params);
}
```

In `compute<Q,R>` the parameters are defined as follows: `Q` is the type of the parameter needed by the function and `R` is the return type. In chapter 16 we will discuss when it's convenient using separated isolates in Flutter applications to optimize performances.

8 | Coding principles with Dart

This chapter is a big "good practice" section as it contains some well-known suggestions from the OOP world. We'd love to also talk about design patterns, TDD, clean code and much more but these contents go beyond the scope of this book. Many people have written books and articles on these topics, we recommend you read up on these for more in-depth details.

- Design patterns are a series reusable solutions to common, well-known problems. The original concept came by a group of four people, called *Gang of four*, but nowadays new patterns come out in parallel with the evolution of languages.
  - Look for any recent book or resource that includes the widest range of patterns. They apply to any programming language; the programming language in which they're explained is not so relevant.
- TDD, abbreviation of **T**est **D**riven **D**evelopment, is a programming style strongly centered on code testing. You have to first write the tests, cover every possible case and only after this process you can start coding.
  - https://resocoder.com/flutter-clean-architecture-tdd
- DDD, abbreviation of **D**omain **D**riven **D**esign, is a programming style which focuses on code maintainability and separation of concerns. We recommend to follow Reso Coder's DDD course which gives a step-by-step explanation about DDD using Dart and Flutter.
  - https://resocoder.com/flutter-firebase-ddd-course
- Dart has a very wide, user-friendly documentation in which you can find examples for almost any topic. It's a wide growing resource that tells you how to properly write Dart code through good practices and articles.
  - https://dart.dev/guides/language/effective-dart
  - https://dart.dev/tutorials

That said, you can of course completely skip this part since it has no core Dart or Flutter concepts but we encourage you to at least know what SOLID and DI are about. These concepts are valid regardless the programming language in which they're applied.

8.1 SOLID principles

The term *SOLID* should actually be written as *S.O.L.I.D.* because it's an acronym for 5 design principles, one for each letter, which help the programmer writing maintainable and flexible code.

8.1.1 Single Responsibility Principle

Very intuitively, this principle (abbreviated with SRP) states that a class should only have a single responsibility so that it could change for one reason and no more. In other words, you should create classes dealing with a single duty so that they're easier to maintain and harder to break.

```dart
class Shapes {
    List<String> cache = List<>();

    // Calculations
    double squareArea(double l) { /* ... */ }
    double circleArea(double r) { /* ... */ }
    double triangleArea(double b, double h) { /* ... */ }

    // Paint to the screen
    void paintSquare(Canvas c) { /* ... */ }
    void paintCircle(Canvas c) { /* ... */ }
    void paintTriangle(Canvas c) { /* ... */ }

    // GET requests
    String wikiArticle(String figure) { /* ... */ }
    void _cacheElements(String text) { /* ... */ }
}
```

This class totally destroys the SRP as it handles internet requests, painting and calculations all in one place. You'll have to make changes very often to `Shape` because it has many duties, all in one place; maintenance for this class is not going to be pleasant. What about this?

```dart
// Calculations and logic
abstract class Shape {
    double area();
}
class Square extends Shape {}
class Circle extends Shape {}
class Rectangle extends Shape {}

// UI painting
class ShapePainter {}

// Networking
```

```
class ShapesOnline {}
```

There are 3 separated classes focusing on a single task to accomplish: they are easier to read, test, maintain and understand. With this approach the attention of the developer is focused on a certain area of interest (such as mathematical calculations on `Shape`) rather than on a messy collection of methods, each with different purposes.

8.1.2 Open closed principle

The open closed principle states that in a good architecture you should be able to add new behaviors without modifying the existing source code. This concept is notoriously described with the sentence "*software entities should be open for extensions but closed for modifications*". Look at this example:

```
class Rectangle {
    final double width;
    final double height;
    Rectangle(this.width, this.height);
}

class Circle {
    final double radius;
    Rectangle(this.radius);

    double get PI => 3.1415;
}

class AreaCalculator {
    double calculate(Object shape) {
        if (shape is Rectangle) {
            // Smart cast
            return r.width * r.height;
        } else {
            final c = shape as Circle;
            return c.radius * c.radius * c.PI;
        }
    }
}
```

Both `Rectangle` and `Circle` respect the SRP as they only have a single responsibility (which is representing a single geometrical shape). The problem is inside `AreaCalculator` because if we added other shapes, we would have to edit the code to add more `if` conditions.

```
class Rectangle {...}
class Circle {...}
class Triangle {...}
class Rhombus {...}
class Trapezoid {...}

class AreaCalculator {
    double calculate(Object shape) {
        if (shape is Rectangle) {
            // code for Rectangle...
        } else if (shape is Circle) {
            // code for Circle...
        } else if (shape is Triangle) {
            // code for Triangle...
        } else if (shape is Rhombus) {
            // code for Rhombus...
        } else {
            //code for Trapezoid...
        }
    }
}
```

Having added 3 new classes, the `double calculate(...)` must be changed because it requires more `if` conditions to handle proper type casts. In general, every time that a new shape is added or removed, this method has to be maintained due to the presence of type casts. We can do better!

```
// Use it as an interface
abstract class Area {
    double computeArea();
}

// Every class calculates the area by itself
class Rectangle implements Area {}
class Circle implements Area {}
class Triangle implements Area {}
class Rhombus implements Area {}
class Trapezoid implements Area {}

class AreaCalculator {
    double calculate(Area shape) {
        return shape.computeArea();
    }
}
```

Thanks to the interface, now we have the possibility to add or remove as many classes as we want **without** changing `AreaCalculator`. For example, if we added `class Square implements Area` it would automatically be "compatible" with the `double calculate(...)` method.

> The gist of this principle is: depend on abstractions and not on implementations. Thanks to `abstract` classes you work with abstractions and not with the concrete implementations: your code doesn't rely on "predefined" entities.

8.1.3 Liskov Substitution Principle

The Liskov Substitution Principle states that subclasses should be replaceable with superclasses without altering the logical correctness of the program. In practical terms, it means that a subtype must guarantee the "usage conditions" of its supertype **plus** something more it wants to add. Look at this example:

```dart
class Rectangle {
    double width;
    double height;
    Rectangle(this.width, this.height);
}

class Square extends Rectangle {
    Square(double length): super(length, length);
}
```

We have a big logic problem here. A square must have 4 sides with the same length but the rectangle doesn't have this restriction. We're able to do this:

```dart
void main() {
    Rectangle fail = Square(3);

    fail.width = 4;
    fail.height = 8;
}
```

At this point we have a square with 2 sides of length 4 and 2 sides of length 8... which is absolutely wrong! Sides on a square must be all equal but our hierarchy is logically flawed. The LSP is broken because this architecture does **NOT** guarantee that the subclass will maintain the logic correctness of the code.

> This example also shows that inheriting from abstract classes or interfaces, rather than concrete classes, is a very good practice. Prefer composition (with interfaces) over inheritance.

To solve this problem, simply make `Rectangle` and `Square` two independent classes. Breaking LSP does not occur if you depend from interfaces: they don't provide any logic implementation as it's deferred to the actual classes.

8.1.4 Interface Segregation Principle

This principle states that a client doesn't have to be forced to implement a behavior it doesn't need. What turns out from this is: you should create small interfaces with minimal methods. Generally it's better having 8 interfaces with 1 method instead of 1 interface with 8 methods.

```dart
// Interfaces
abstract class Worker {
    void work();
    void sleep();
}

class Human implements Worker {
    void work() => print("I do a lot of work");
    void sleep() => print("I need 10 hours per night...");
}

class Robot implements Worker {
    void work() => print("I always work");
    void sleep() {} // ??
}
```

Robots don't need to sleep and thus the method is actually useless, but it still needs to be there otherwise the code won't compile. To solve this, let's just split `Worker` into multiple interfaces:

```dart
// Interfaces
abstract class Worker {
    void work();
}
abstract class Sleeper {
    void sleep();
}

class Human implements Worker, Sleeper {
    void work() => print("I do a lot of work");
    void sleep() => print("I need 10 hours per night...");
}

class Robot implements Worker {
    void work() => print("I always work");
}
```

This is definitely better because there are no useless methods and we're free to decide which behaviors should the classes implement.

8.1.5 Dependency Inversion Principle

This is very important and useful: DIP states that we should code against abstractions and not implementations. Extending an abstract class or implement an interface is good but descending from a concrete classed with no abstract methods is bad.

```dart
// Use this as interface
abstract class EncryptionAlgorithm {
    String encrypt(); // <-- abstraction
}

class AlgoAES implements EncryptionAlgorithm {}
class AlgoRSA implements EncryptionAlgorithm {}
class AlgoSHA implements EncryptionAlgorithm {}
```

Dependency injection (DI) is a very famous way to implement the DIP. Depending on abstractions gives the freedom to be independent from the implementation and we've already dealt with this topic. Look at this example:

```dart
class FileManager {
    void secureFile(EncryptionAlgorithm algo) {
        algo.encrypt();
    }
}
```

The `FileManager` class knows **nothing** about how `algo` works, it's just aware that the `encrypt()` method secures a file. This is essential for maintenance because we can call the method as we want:

```dart
final fm = FileManager(...);

fm.secureFile(AlgoAES());
fm.secureFile(AlgoRSA());
```

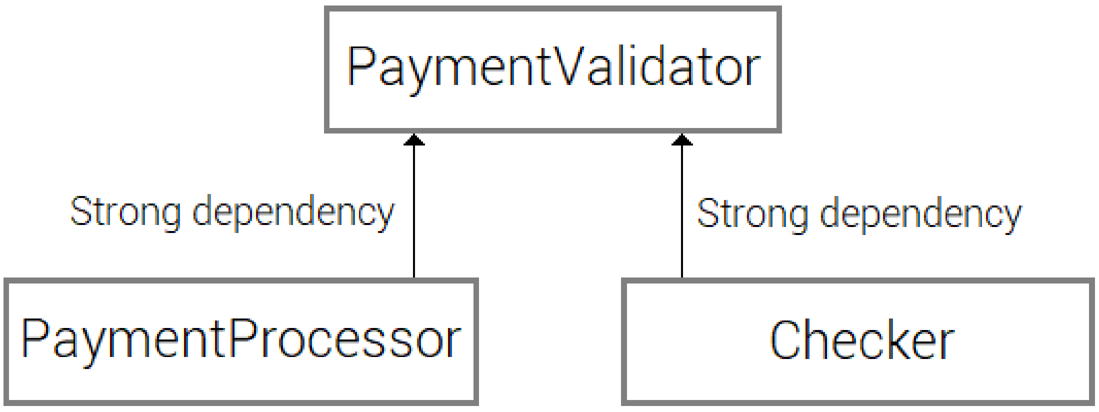
If we added another encryption algorithm, it would be automatically compatible with `secureFile` as it is a subtype of `EncryptionAlgorithm`. In this example, we're respecting the 5 SOLID principles all together.

8.2 Dependency Injection

Two classes are said to be "coupled" if at least one of them depends on the other. Class A depends on class B when you can't compile class A without the presence of class B. This can be very dangerous, let's see why.

```
class PaymentValidator {
    final Date date;
    final String cardNumber;
    const PaymentValidator(this.date, this.cardNumber);

    // Uses the MasterCard payment circuit
    void validatePayment(int amount) { ... }
}

class PaymentProcessor {
    late final _validator;
    PaymentProcessor(String cardNumber) {
      _validator = PaymentValidator(DateTime.now(), cardNumber);
    }

    Date get expiryDate => _validator.date;
    void pay(int amount) =>
        _validator.validatePayment(amount);
}

abstract class Checker {
    PaymentValidator mastercardCheck();
}
class CheckerOne extends Checker { /*... code ... */}
class CheckerTwo extends Checker { /*... code ... */}
```

Both `Checker` and `PaymentProcessor` have a strong dependency on `PaymentValidator` because it's essential in order to compile. Subclasses, of course, inherit the dependency too.



Let's say you've written this code at work. One day, your project manager tells you to ditch Mastercard and replace it with PayPal. You'll quickly get a stomach ache as soon as you realize that, from an apparently small change, the whole architecture has to be refactored.

1. Paypal just requires an email but your Mastercard implementation requires date and card number. You're forced to entirely change `PaymentValidator` but by consequence you also need to update both `PaymentProcessor` and `Checker` as they're strong dependencies.

   ```
   class PaymentValidator {
       final String _email;
       const PaymentValidator(this._email);

       void validatePayment(int amount) { ... }
   }

   class PaymentProcessor {
       late final PaymentValidator _validator;

       PaymentProcessor(String email) :
           _validator = PaymentValidator(email);

       void pay(int amount) =>
           _validator.validatePayment(amount);
   }
   ```

   There's been a "cascade" effect because changes made to a single class had consequences to other classes as well.

2. The above changes break another part of the code: the abstract class `Checker` also depends on `PaymentValidator` so there's the need to fix the code.

   ```
   abstract class Checker {
       // earlier it was called 'mastercardCheck()'
       PaymentValidator payPalCheck();
   }
   ```

   This change has consequences on `any` subclass of `Checker` which has to be updated. Dependencies on superclasses are inherited by its children and thus the coupling propagates.

3. There are no ways to solve this problem other than manually updating every single subclass of `Checker`. Your IDE will come to the rescue with a refactor tool but maintenance is a pain anyway.

As you've just seen, an apparently small and quick change on a class propagated to an entire hierarchy and other big components of our project. All of this happened because classes are strongly coupled and they depend on implementations rather than abstractions.

8.2.1 Constructor injection

Using dependency injection and abstractions rather than implementations, the above problems fade away. Dependencies passed from the outside create a *weak* coupling which is safer than a *strong* one as it relies on abstractions.

```
abstract class PaymentValidator {
    const PaymentValidator();
    void validatePayment(int amount);
}

class MasterCard implements PaymentValidator {
    // Define date, card number and the constructor
    const MasterCard();
    void validatePayment(int amount) {...}
}

class PayPal implements PaymentValidator {
    // Define an email and the constructor
    const PayPal();
    void validatePayment(int amount) {...}
}
```

The `PaymentProcessor` class is still going to have a `PaymentValidator` dependency but it's **weak** because it's just an interface. Using "constructor injection" we pass from the outside a concrete implementation, which can later be replaced with anything else.

```
class PaymentProcessor {
    final PaymentValidator _validator;
    const PaymentProcessor(this._validator);

    void pay(int amount) =>
        _validator.validatePayment(amount);
}

// And then we can freely use PayPal or MasterCard
void main() {
    final p1 = const PaymentProcessor(MasterCard());
    final p2 = const PaymentProcessor(PayPal());
}
```

In this case, we're passing an instance of a concrete class via constructor and that's **fundamental**. `PaymentProcessor` knows nothing about the implementation details of the validator object, it just knows he has to call `validatePayment(int)`. We can also use `const` constructors now!

- This code is very flexible and maintainable. If your boss told you to add support for the Visa circuit as well, you would simply have to create a new subtype of `PaymentValidator`.

```
class Visa implements PaymentValidator {
    const Visa();
    void validatePayment(int amount) {...}
}
```

No changes are required to the existing code and you at the same time you're still embracing S.O.L.I.D. principles. The architecture is robust!

- `PaymentProcessor` now doesn't care anymore about Mastercard, Paypal or whatever because they're given from the outside. Internally he weakly depends on an abstraction which just gives an abstraction: the implementation is passed via constructor.
- You have a series of classes, one per payment method, that are super easy to test. You could make a "mock" class for unit tests just like a regular validator type:

```
class TestValidator implements PaymentValidator {
    const TestValidator();
    void validatePayment(int amount) {...}
}
```

Last thing we need to refactor is the `Checker` class as it has to return an abstraction rather than an implementation.

```
abstract class Checker {
    PaymentValidator paymentCheck();
}

class CheckerOne extends Checker {...}
class CheckerTwo extends Checker {...}
```

Since `PaymentValidator` is `abstract`, any class along the hierarchy inherits a *weak* dependency which is safe.

8.2.2 Method injection

Constructor injection is used when you class really needs an external dependency to work. When you have an "optional" dependency not strictly required from your class, you can use method injection.

```
abstract class CheckProcessor {
    const CheckProcessor();
    bool isActive();
}

class MastercardCheck implements CheckProcessor {
    final MasterCardApi _api;
    const MastercardCheck(this._api);

    bool isActive() async => await _api.isOnline();
}

class PaypalCheck implements CheckProcessor {
    final PaypalApi _api;
    const PaypalCheck(this._api);

    bool isActive() async => await _api.available();
}
```

These classes connect to the internet, perform some GET requests and return `true` or `false` whether the service provider is online or not. Let's say this feature is not essential in our architecture but it's nice to have it. It might be used but it's not certain.

```
class PaymentProcessor {
    final PaymentValidator _validator;
    const PaymentProcessor(this._validator);

    void pay(int amount) => ...

    bool isProcessorActive(CheckProcessor check) =>
        return check.isActive();
}
```

In this way, if we wanted to check the availability of the service we could do this:

```
void main() {
    final api = MasterCardApi(...);
    final processor = MasterCard(api);
    final checker = MastercardCheck();

    final payment = PaymentProcessor(processor);
    final isOnline = payment.isProcessorActive(checker);
}
```

Note the difference: while the processor is fundamental, and thus it's passed via constructor, the connection checker made with `isProcessorActive` is not always required. So in general:

- constructor injection is for essential dependencies that your class is **always** going to use;
- method injection is for optional dependencies that you class **might** use.

Actually both type of injection use the same concept, which is depending on abstractions and passing implementations from the outside, but they differ in order of "importance". Dependencies passed via constructor are fundamental while the ones passed via method are just useful but not essential.

Index

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