

---

# FLUTTER COMPLETE REFERENCE 2.0

---

The ultimate reference for Dart and Flutter.



ALBERTO MIOLA



Dedicated to Giorgia, my friends and my family.

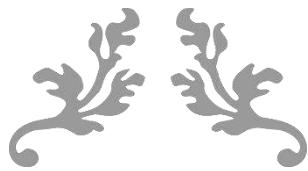
## SPECIAL THANKS TO

Jimmy Walker for the graphics

-----

The Superformula company  
(<https://www.superformula.com>)





# Table of contents

---

<b>1 – Introduction.....</b>	<b>16</b>
1.1 Welcome .....	16
1.1.1 Who is this book for .....	17
1.1.2 About the author and the team.....	17
1.2 The Dart programming language.....	18
1.2.1 Platforms support.....	20
1.2.2 Creating a “Hello world” project.....	22
1.3 The Flutter framework.....	24
1.3.1 Why Flutter uses Dart.....	29
1.3.2 Creating a “Hello world” project.....	30
<b>Part 1: The Dart programming language .....</b>	<b>32</b>
<b>2 – Variables and types.....</b>	<b>33</b>
2.1 Variables.....	33
2.1.1 final and const .....	34
2.1.2 Nullability and non-nullability .....	36
2.2 Data types.....	41
2.2.1 Numbers .....	41
2.2.2 Strings .....	43
2.2.2.1 Good practices.....	45
2.2.2.2 Runes.....	47

2.2.3 Booleans .....	48
2.2.4 Enumerated types .....	49
2.2.5 Lists .....	51
2.2.6 Dart's environment variables .....	52
2.3 Data type operators .....	53
2.3.1 Arithmetic operators.....	53
2.3.2 Relational operators .....	54
2.3.3 Type test operators.....	54
2.3.4 Logical operators .....	54
2.3.5 Bitwise and shift operators .....	55
2.3.6 Compound assignment operators.....	56
Deep dive: Null safety in Dart.....	57
Top and bottom types.....	62
Deep dive: Language specification details.....	64
<b>3 – Control flow and functions .....</b>	<b>67</b>
3.1 If statement .....	67
3.2 Switch statement .....	70
3.3 For and while loops .....	73
3.4 Functions .....	75
3.4.1 The Function type .....	78
3.4.2 Anonymous functions .....	79
3.4.3 Optional parameters.....	81
3.4.3.1 Optional named parameters .....	81
3.4.3.2 Optional positional parameters .....	84
3.4.4 Nested functions .....	85
3.4.5 Assertions.....	86
3.4.6 Good practices.....	87
3.6 Type aliases.....	88

Deep dive: Language specification details .....	90
<b>4 – Classes .....</b>	<b>93</b>
4.1 Introduction .....	93
4.1.1 Libraries and visibility .....	96
4.1.2 Encapsulation .....	98
4.1.2.1 part and part of .....	100
4.2 Constructors .....	101
4.2.1 Generative constructors .....	101
4.2.2 Factory constructors and static members .....	105
4.2.3 Constant constructors .....	107
4.2.4 Named and redirecting constructors .....	110
4.2.5 Good practices .....	112
4.3 Getters and setters .....	114
4.4 Operators overload .....	117
4.4.1 The call() method .....	119
Deep dive: Cloning objects .....	121
Deep dive: Annotations and mirrors .....	125
Deep dive: Language specification details .....	132
<b>5 – Inheritance, core classes and exceptions .....</b>	<b>136</b>
5.1 Inheritance .....	136
5.1.1 Constructors .....	138
5.1.2 Abstract classes .....	140
5.1.3 Interfaces .....	141
5.1.4 Good practices: extends vs implements .....	143
5.1.5 Mixins .....	146
5.1.6 Extension methods .....	149
5.1.7 The covariant keyword .....	150

5.1.8 Good practices.....	153
5.1.8.1 Smart casts .....	155
5.2 The Object class .....	156
5.2.1 Comparable<T> .....	163
5.2.2 Records .....	164
5.3 Exceptions and errors.....	167
5.3.1 try, on and catch.....	169
5.3.2 finally.....	170
5.3.3 rethrow .....	171
5.3.4 Exceptions, errors and good practices.....	172
Deep dive: Language specification details.....	174
 6 – Generics and collections .....	178
6.1 Introduction .....	178
6.1.1 Usage.....	180
6.2 Collections .....	182
6.2.1 List<T>.....	182
6.2.2 Set<T>.....	186
6.2.3 Map<K, V>.....	188
6.3 Advanced topics .....	192
6.3.1 Alternative implementations .....	194
6.3.2 Iterable<T> and iterators.....	198
6.3.3 from and of constructors.....	202
Deep dive: Collection methods .....	204
Deep dive: Covariance and contravariance in generics.....	206
 7 – Advanced language features .....	208
7.1 Class modifiers .....	208
7.1.1 base .....	210

7.1.1.1 Technical overview .....	212
7.1.2 interface .....	214
7.1.2.1 Technical overview .....	215
7.1.3 final. ....	216
7.1.3.1 Technical overview .....	219
7.1.4 sealed .....	220
7.1.5 mixin class.....	221
7.1.6 Considerations.....	222
7.2 Patterns .....	224
7.2.1 Detailed overview .....	229
7.2.2 Exhaustiveness checking .....	234
Deep dive: Language specification details.....	237
<b>8 – Futures, Streams and Isolates.....</b>	<b>239</b>
8.1 Introduction .....	239
8.1.1 Event loop and queues.....	241
8.2 Futures.....	245
8.2.1 The Future<T> API .....	249
8.2.2 async and await.....	254
8.2.3 Completers .....	255
8.3 Streams .....	256
8.3.1 The yield* keyword.....	261
8.3.2 The Stream<T> API.....	262
8.3.3 Good practices.....	264
8.3.4 Synchronous generators .....	266
8.4 Isolates .....	267
8.4.1 Creating long-running isolates .....	270
8.4.2 Good practices.....	276
Deep dive: Zones.....	277

Deep dive: Language specification details .....	281
<b>9 – API overview and tests .....</b>	<b>284</b>
9.1 Dart core libraries .....	284
9.1.1 The conversion library .....	284
9.1.1.1 Converting from JSON strings.....	285
9.1.1.2 Converting to JSON strings.....	287
9.1.2 The I/O library .....	289
9.1.2.1 Reading and writing from the console .....	291
9.1.2.2 The Process class .....	292
9.1.2.3 Socket and ServerSocket.....	293
9.1.3 Date and time.....	293
9.1.3.1 Representing a span of time with Duration .....	295
9.1.3.2 Measuring timespans with Stopwatch.....	296
9.1.4 Native interoperability with FFI .....	297
9.2 Testing Dart code .....	299
9.2.1 Matchers.....	302
9.2.2 Testing futures and streams.....	305
9.2.3 Tests setup.....	308
9.2.4 Code coverage .....	310
<b>Part 2: The Flutter framework .....</b>	<b>313</b>
<b>10 – Flutter, widgets and trees .....</b>	<b>314</b>
10.1 Flutter overview .....	314
10.1.1 Build modes .....	316
10.1.2 Tree shaking .....	316
10.2 Widgets .....	319
10.2.1 Stateless and Stateful widgets.....	321
10.2.2 The lifecycle of the State<T> object .....	324

10.2.3 Keys .....	328
10.2.4 Good practices .....	330
10.3 Widget, Element and RenderObject trees .....	332
10.3.1 Rebuilds and trees updates .....	335
10.3.2 Performance considerations .....	338
10.3.3 Considerations on BuildContext .....	344
Deep dive: The framework internals.....	345
<b>11 – Material, Cupertino and CustomPaint .....</b>	<b>353</b>
11.1 Basic widgets.....	353
11.1.1 Text.....	355
11.1.2 Container .....	357
11.1.3 Row, Column and Align.....	358
11.1.4 Stack.....	363
11.2 Material library .....	364
11.2.1 Scaffold.....	365
11.2.2 Buttons.....	371
11.2.3 Dialogs .....	372
11.2.4 Themes .....	378
11.3 Cupertino library .....	380
11.3.1 Cupertino scaffolds .....	381
11.3.2 Buttons.....	383
11.4 CustomPaint and CustomPainter .....	385
11.4.1 Effects .....	391
11.4.1.1 Opacity .....	391
11.4.1.2 Clip widgets.....	392
Deep dive: Asynchronous widgets.....	394
<b>12 – State management .....</b>	<b>400</b>

12.1 Introduction.....	400
12.1.1 Ephemeral and application state .....	401
12.1.2 Good practices .....	402
12.2 Using setState .....	403
12.2.1 Performance considerations .....	406
12.3 Using InheritedWidget.....	408
12.3.1 App state management using listeners.....	415
12.3.2 Performance considerations .....	420
12.3.3 ValueNotifier and ValueListenableBuilder.....	424
12.3.4 Alternatives .....	426
Deep dive: State restoration.....	426
Using RestorationMixin and restorable values .....	428
Considerations and debugging .....	431
<b>13 – Routes and navigation .....</b>	<b>433</b>
13.1 Introduction.....	433
13.1 Imperative and declarative navigation .....	434
13.2 Imperative navigation using Navigator.....	435
13.2.1 Good practices .....	438
13.2.2 Navigating between routes .....	440
13.2.3 Passing data between routes .....	443
13.2.3.1 Passing data using InheritedWidget .....	445
13.2.4 Deep linking.....	447
13.3 Declarative navigation using Navigator and Router.....	450
13.3.1 Navigating and passing data between pages .....	455
13.3.2 Deep linking.....	457
13.3.2.1 The information parser class.....	458
13.3.2.2 The router delegate .....	460
12.3.2.3 Considerations .....	462

13.4 The go_router package .....	463
13.4.1 Understanding navigation .....	467
13.4.2 Passing data between pages and using query parameters .....	469
13.5 Good practices.....	472
13.5.1 Changing the URL path strategy.....	474
13.5.2 State restoration and navigation.....	475
Deep dive: Navigation and overlays.....	476
<b>14 – Layouts and responsiveness.....</b>	<b>481</b>
14.1 Layouts in Flutter .....	481
14.1.1 Laying out widgets .....	482
14.1.2 Understanding constraints .....	487
14.1.3 Box constraints .....	491
14.2 Building responsive layouts .....	493
14.2.1 The MediaQuery type .....	495
14.2.2 Good practices .....	496
14.3 Scrollable widgets.....	499
14.3.1 Using a ScrollController .....	502
14.3.2 Other scroll-related widgets .....	504
14.3.2.1 Scrollbar.....	504
14.3.2.2 NotificationListener<T> .....	506
14.3.2.3 RefreshIndicator.....	507
14.3.2.4 ReorderableListView .....	509
14.3.2.5 GridView.....	510
14.3.2.6 ScrollPhysics .....	511
14.3.3 Slivers.....	513
14.3.3.1 Nesting scrollable widget .....	516
14.3.3.2 Using SliverAppBar for dynamic and sticky headers .....	518
Deep dive: Understanding RenderObjects.....	520

Deep dive: Scrolling and overlays .....	525
<b>15 – Internationalization and accessibility .....</b>	<b>529</b>
15.1 Internationalization .....	529
15.1.1 Internationalizing a Flutter application .....	532
15.1.1.1 Dynamically changing locale.....	535
15.1.1.2 Manual localization.....	538
15.1.2 Using the intl package.....	540
14.1.3.1 Numbers .....	540
15.1.2.2 Date and time .....	541
15.2 Accessibility .....	542
15.2.1 The Semantic widget .....	543
15.2.2 The semantic tree.....	546
15.2.3 The semantics debugger.....	548
Deep dive: Advanced localization techniques .....	550
<b>16 – Assets and images .....</b>	<b>556</b>
16.1 Defining assets .....	556
16.1.1 Image assets and variants.....	558
16.1.2 Font assets.....	559
16.1.3 Platform assets .....	561
16.2 Images in Flutter .....	562
16.2.1 Good practices .....	566
16.2.2 Vectorial images .....	569
Deep dive: Loading images in CustomPainter .....	571
<b>17 – Animations .....</b>	<b>574</b>
17.1 Implicit animations .....	574
17.1.1 Hero animations.....	580
17.2 Explicit animations.....	583

17.2.1 Tweens.....	588
17.2.1.1 Using a TweenAnimationBuilder.....	591
17.2.2 Widget transitions.....	592
17.2.3 Routes transitions.....	595
17.3 Good practices.....	597
17.3.1 Combining animations.....	599
17.3.2 Using AnimatedWidget .....	603
Deep dive: Tickers and animations.....	605
<b>18 – Forms and gestures .....</b>	<b>608</b>
18.1 Form input widgets.....	608
18.1.1 Using a TextEditingController .....	610
18.1.2 Other form input widgets.....	612
18.2 Handling forms.....	618
18.2.1 Focus overview.....	621
18.2.2 The Focus widget.....	624
18.2.3 Controlling focus traversal.....	625
18.3 Gestures.....	628
18.3.1 Drag and drop.....	630
18.3.2 The MouseRegion widget.....	633
Deep dive: Actions, intents, and shortcuts .....	635
The FocusableActionDetector widget.....	638
<b>19 – Testing Flutter applications .....</b>	<b>641</b>
19.1 Introduction to widget testing.....	641
19.1.1 Using finders .....	644
19.1.2 Pumping methods and rebuilds.....	647
19.2 Widget testing strategies.....	649
19.2.1 Accessing widgets and state instances .....	650

19.2.2 Testing forms and gestures.....	652
19.2.3 Golden tests .....	657
19.2.3.1 Fonts in golden tests.....	660
19.2.4 Good practices .....	663
19.3 Integration tests .....	664
Deep dive: Performance profiling with integration tests .....	667
<b>Part 3: Dart and Flutter ecosystems .....</b>	<b>671</b>
<b>20 – Creating and maintaining a package.....</b>	<b>672</b>
20.1 Package creation.....	672
20.1.1 Package organization .....	673
20.1.2 Documentation.....	676
20.1.3 Static analysis and linter rules .....	680
20.2 Package maintenance.....	684
20.2.1 GitHub repository setup.....	688
20.2.2 Dependencies and pub package manager .....	691
Deep dive: Publishing packages at pub.dev.....	694
Verified publishers .....	697
Flutter favorites .....	698
<b>21 – Creating and maintaining a Flutter app.....</b>	<b>700</b>
21.1 Flutter project creation.....	700
21.1.1 Project organization .....	701
21.1.2 Good practices and adaptive apps.....	703
21.2 Maintaining a Flutter project.....	707
21.2.1 Tests and code coverage .....	707
21.2.1 GitHub repository setup.....	709
21.3 Creating release builds.....	711
21.3.1 Android.....	711

21.3.2 macOS and iOS .....	715
21.3.3 Windows and Linux .....	718
21.3.4 Web.....	719
21.3.4.1 Web renderers .....	722
21.3.4.2 Image limitations on the web .....	723
Deep dive: Embedding Flutter in HTML pages .....	724
<b>22 – HTTP servers and low-level HTML.....</b>	<b>730</b>
22.1 Creating HTTP servers with dart:io.....	730
22.1.1 Handling requests and responses.....	733
22.1.2 Unit testing HttpServer using HttpClient .....	736
22.2 Creating HTTP servers with the shelf package .....	739
22.2.1 Handlers and middleware .....	740
22.2.2 The shelf_router and shelf_static packages .....	743
22.2.3 Good practices .....	746
22.3 Creating and maintaining an HTTP server.....	748
22.3.1 Setup and deployment.....	751
22.3.2 Unit tests .....	752
22.4 Low-level HTML manipulation with dart:html.....	755
22.4.1 DOM elements interactions .....	758
22.4.2 Deferred imports .....	761
Deep dive: The http package.....	763
<b>23 – Platform interactions .....</b>	<b>769</b>
23.1 Platform-specific features .....	769
23.1.1 Platform-specific packages for Flutter.....	772
23.2 Writing platform-specific code.....	776
23.2.1 Platform channels.....	778
23.2.2 Android implementation .....	781

23.2.3 iOS implementation .....	782
23.2.4 Windows implementation .....	783
23.2.5 Linux implementation .....	783
23.2.6 macOS implementation .....	785
23.3 Creating Flutter plugins and FFI plugin packages .....	786
23.3.1 Developing a federated plugin package .....	787
23.3.1.1 The app-facing package.....	788
23.3.1.2 The platform-interface package .....	790
23.3.1.3 The platform-specific packages.....	791
23.3.2 Flutter FFI plugin package.....	793
Deep dive: Type-safe native communication with pigeon.....	795
<b>Appendix – Performance and profiling.....</b>	<b>800</b>
A.1 Working with DevTools .....	800
A.1.1 Flutter inspector .....	801
A.1.2 Performance view .....	804
A.1.3 CPU profiler .....	806
A.1.4 Debugger .....	807
A.1.5 Network and logging views.....	808
A.1.6 Memory view.....	809
A.2 Performance best practices.....	811
A.2.1 Minimize widgets rebuilds.....	811
A.2.2 Good practices for scrollable widgets .....	813
A.2.3 Widgets usage recommendations.....	815
A.2.3.1 Use RepaintBoundary judiciously .....	816

# 1 – Introduction

---

## 1.1 Welcome

Thank you for having put your faith in this book. Given the constantly increasing popularity that Dart and Flutter are getting over the years, we felt it was essential to write a book to cover both the language and the framework. Both Flutter and Dart have been shaped through community engagement to create a robust set of tools for writing great code. This approach has resulted in an explosive growth for developers and companies seeking them.

### Note

This book is for Dart 3.0 and Flutter 3.10 (or later versions).

In this book you will find a lot of technical content, which will always be supported by examples and good practice advice based on our professional experience. We encourage you to play and interact with the code examples we've built and shared in our GitHub repository<sup>1</sup>. We have divided the book into three parts:

1. The first part focuses entirely on Dart, the programming language in which Flutter is written.
2. The second part covers the Flutter framework: an open-source, cross-platform, and natively compiled framework.
3. The third part is about the Flutter and Dart ecosystems.

You will often find some “Good practices” paragraphs or sections because we firmly believe that theoretical knowledge is never enough. We want to share what we've learned during our years of professional experience with Dart and Flutter clearly and concisely. This is the value we think this book has: our depth of practical experience and knowledge produces a truly unique source of information. Any chapter containing an example application has an informative box right after the title with the link to the online example code. For example:

---

<sup>1</sup> [https://github.com/albertodev01/flutter\\_complete\\_reference\\_2](https://github.com/albertodev01/flutter_complete_reference_2)



[https://github.com/albertodev01/flutter\\_book\\_examples/chapter\\_XX/](https://github.com/albertodev01/flutter_book_examples/chapter_XX/)

This box indicates that the chapter has an associated example application and you can find the Dart source code at the given link. The source code is guaranteed to work with Flutter 3.10, but it might not be compatible with lower versions.

### 1.1.1 Who is this book for

To get the most out of this book, we expect you to be familiar with object-oriented programming and preferably at least a language such as Java, C#, or C++. Our goal is to try to make the contents of this book understandable for the broadest range of developers. Nevertheless, you should already have some previous experience.

If you already know about classes, inheritance, and types, the first part of this book will be a walk in the park for you. Regardless of your expertise level, we will talk about Dart and Flutter from the ground up to ensure that you learn about everything we feel is valuable for a professional.

### 1.1.2 About the author and the team

Alberto is an Italian software engineer that started working with Delphi (Object Pascal) for desktop and mobile applications. After a few years, he moved to Java to work with Android and back-end software. He fell in love with Flutter in 2018, when it still was in preview, and since then, he's always been using it. Alberto graduated in computer science at the University of Padova with a thesis on Flutter and cross-platform frameworks.

This book owes a lot to some people the author needs to mention here to express his gratitude for their contribution. They have provided technical content reviews, comments, and critiques that improved the quality of the book:

- Matthew Solle;
- Enzo Lizama;
- Carlo Lucera;
- Jonathan Sande (@Suragch).

Throughout the chapters, you'll always see the “we” pronoun because all the people we mentioned above actively contributed to the creation of this book. A special acknowledgment also goes to the entire Superformula company that supported me in this book's writing and reviewing process. Without further say, let's start introducing Dart and Flutter!

## 1.2 The Dart programming language

Dart is a client-optimized, garbage-collected, OOP programming language for creating fast apps that run on any platform. Its syntax and the various constructs are very similar to those found in the most popular languages, such as Java and C#.



Figure 1.1: The Dart logo.

The Dart language is particularly suited for client development, especially regarding high-quality production quality across multiple web, desktop, and mobile platforms. Nevertheless, it's also very efficient and production-ready for server-side projects. The two most significant strengths of the language are:

- **Type safety.** Dart uses static type checking to ensure that the type of a variable always matches the variable's static type. The language supports type inference so the compiler can assign the type to a variable according to its value.
- **Sound null safety.** By default, a variable in Dart cannot be null unless you explicitly say it can be. A “sound” type system ensures that you can never get into a state where an expression evaluates to something different from its static type. For example, a `String` can never be null because the type system is sound. A `String?` Instead (with the question mark at the end of the name) could either be a `String` or `null` at runtime.

A sound type system protects you from null-related exceptions, catches type errors at compile-time, and allows the compiler to generate smaller and faster code. Along with the strong typing approach, there is an impressive set of core libraries in Dart:

- `dart:core` library: core functionalities for any Dart program, such as built-in types and collections;
- `dart:convert` library: encoders and decoders for JSON, UTF-8, and much more;
- `dart:io` library: filesystem management, HTTP and I/O handling for Desktop, mobile, and embedded platforms;

- `dart:html` library: for web-based applications that have to interact with the browser and the DOM;
- `dart:isolate` library: allows the execution of concurrent tasks;
- `dart:async` library: adds support to asynchronous and reactive programming;
- `dart:math` library: standard mathematical functions and constants;
- `dart:collection` library: provides implementations of hash maps, linked lists, queues, and much more generic containers;
- `dart:mirrors` library: essential reflection support for Dart;
- `dart_ffl` library: allows Dart code to load and use native C APIs.

You can find any other library not part of the SDK at <https://pub.dev>, the official package manager for Dart. It contains libraries from the Dart team, the Flutter team, and a myriad of other developers around the world. In *chapter 20 – “Creating and maintaining a package”* we will see a complete example of how to publish a package at *pub.dev*. Here is the most straightforward Dart program we can think of:

```
void main() {
  print('Hello world!');
}
```

Once you've downloaded the latest Dart SDK<sup>2</sup>, you can save the code in a file whose extension is `.dart` and run the program calling `dart run` from the console. While quick and intuitive, this process is not the best for large projects. When you need to work on business projects and large applications, we recommend choosing one of the two most popular IDEs in the Dart and Flutter world:

- Android Studio<sup>3</sup>, with the official Dart plugin.
- Visual Studio code<sup>4</sup>, with the official Dart extension.

It's worth mentioning that the Dart team created DartPad, an online editor you can use to run Dart code snippets on your browser. It has a minimal set of functionalities because it's not meant to be a fully-fledged IDE: it simply is a sort of “playground” code editor for Dart and Flutter where you can quickly run small pieces of code. DartPad also integrates with GitHub gists<sup>5</sup> to publicly share

<sup>2</sup> <https://dart.dev/get-dart>

<sup>3</sup> <https://developer.android.com/studio>

<sup>4</sup> <https://code.visualstudio.com/>

<sup>5</sup> <https://github.com/dart-lang/dart-pad/wiki/Sharing-Guide>

your Dart code. On a side note, if you check the DartPad source code<sup>6</sup> you will see that it is written with Dart. Here's a screenshot:

A screenshot of the DartPad web application. The interface has a dark theme. At the top left is the DartPad logo and a 'Reset' button. To the right are 'Samples' and a three-dot menu icon. The main area is divided into two sections: 'Console' on the right and a code editor on the left. The code editor contains the following Dart code:

```
1 void main() {  
2     for (int i = 0; i < 3; i++) {  
3         print('hello ${i + 1}');  
4     }  
5 }
```

A blue 'Run' button is positioned above the code editor. In the 'Console' section, the output is:

```
hello 1  
hello 2  
hello 3
```

Figure 1.2: A simple Dart program running at <https://dartpad.dartlang.org>.

A Dart program can be written on any general-purpose text editor and manually compiled with command line tools. However, this approach may be less productive than using a fully-fledged IDE. We recommend using either Android Studio or Visual Studio Code.

### 1.2.1 Platforms support

When your Dart source code needs to run on mobile or desktop platforms, the compiler bundles in the final executable a copy of the Dart VM<sup>7</sup> runtime (with Just-In-Time compilation, if required). In release mode, it also includes AOT-compiled native code. To be more precise, here's what is being shipped when the application is compiled:

- While developing, you're working in debug mode. The compiler bundles together the Dart VM with a JIT compiler, the debugging services, and metrics collection tools. A JIT compiler is essential to enable the *hot-reload* feature, which allows for real-time code recompilation.
- When deploying your application, you're working in release mode. The JIT compiler and all the other debugging services/tools are removed. A stripped-down version of the Dart VM is included along with the AOT-compiled (ARM or x64) native machine code.

---

<sup>6</sup> <https://github.com/dart-lang/dart-pad>

<sup>7</sup> VM = Virtual Machine

The term “AOT compilation” is an abbreviation for “Ahead-of-Time compilation”, which indicates the act of translating a high-level programming language, like Dart, into native machine code. To summarize, the mobile or desktop binaries produced by Dart have these contents:

Debug mode	Release mode
Dart VM runtime (with a JIT compiler)	Dart VM runtime (with native machine code)
Debugging services	
Metrics collection tools	

As you can notice, the Dart runtime is always required because it executes three essential tasks:

1. **Managing isolates.** An *isolate* is the place where all Dart code is run. We will cover isolates in detail in *chapter 8 – “Futures, Streams and Isolates”*.
2. **Managing memory.** Dart has an efficient garbage collector, automatically handled by the runtime, that manages the program’s memory.
3. **Managing types.** Albeit the majority of type checks are static (meaning that they happen at compile-time), some are dynamic. For example, the runtime enforces dynamic checks and casts using special operators (`is` and `as`).

Dart applications can be translated by the compiler into a fast, compact, and deployable JavaScript file. In other words, the Dart code is compiled into JavaScript code which can be run in a browser.

### Note

Dart is a general-purpose language used to create console applications or web servers that can run on any platform. If you want to make a cross-platform application with a UI and fancy animations, use Flutter instead (which is a framework written in Dart).

From the console, you can use `dart run` to execute your program in debug mode or `dart compile exe` to AOT-compile the code and create a standalone executable file. Use the `dart compile js` command to run applications on a browser. In *chapter 22 – Section 4 “Low-level HTML manipulation with dart:html”* we will see an example of how to connect Dart and HTML on a web application.

## 1.2.2 Creating a “Hello world” project

For a simple “hello world” application, a single Dart file is enough. For larger applications that you can also use in a production environment instead, you need a proper project structure with files and folders organized in the best way possible. Even if you’re free to set up the project as you wish, we encourage you to follow Dart’s guidelines. There are two recommended ways to create a new Dart project:

1. Use the console and choose among one of these commands:
  1. `dart create`: an alias of `dart create -t console`;
  2. `dart create -t console`: creates the skeleton of a Dart command-line app (this is the default option);
  3. `dart create -t package`: creates the structure for the creation of a Dart package (more on this in *chapter 20 – “Creating and maintaining a package”*);
  4. `dart create -t server-shelf`: creates the skeleton of a server application that uses `shelf`<sup>8</sup> (more on this in *chapter 22 – “HTTP servers and low-level HTML”*);
  5. `dart create -t web`: creates a web application with an `index.html` page that you can run in a web browser (more on this in *chapter 22 – Section 4 “Low-level HTML manipulation with dart:html”*).

In this section, we’re going to create a hello world project with `dart create -t console-full hello_world`. This command generates a Dart project with the recommended files and folder structure.

2. Use an IDE. As we’ve already mentioned, the two leading IDEs are Android Studio (with the Dart plugin) and Visual Studio Code (with the Dart extension).

Under the hood, the IDE invokes the console commands for you anyway, so the final result in both cases is the same. We will use Android Studio but you can choose the IDE or the code editor of your preference. In any kind of project, you will always find the following content:

1. the `pubspec.yaml` file is probably the most important of all. It defines the project name, the version, which SDK versions are supported, which packages the application depends on, and much more;

---

<sup>8</sup> <https://pub.dev/packages/shelf>

2. the `CHANGELOG.md` file should constantly be updated to the latest project version with a brief summary of the changes or features introduced over time;
3. the `README.md` file generally contains a description of your package, usage information, and minor code examples;
4. the `analysis_options.yaml` file is used to define additional static analysis rules to improve the code quality;
5. the `test/` folder, as the name suggests, contains Dart tests to ensure that the application works as expected;
6. the `lib/` folder contains the public Dart code that can also be shared by other packages. All of your `.dart` files go here;
7. the `bin/` folder contains public tools, scripts, and executables of a project. On command line applications, this file (by convention) includes the `main()` entry point.

In *chapter 20 – “Creating and maintaining a package”* we will cover all the above contents. For now, let's just change the contents of `lib/hello_world.dart` with the actual “Hello world!” message we want to display:

```
void greetings() {
  print('Hello world!'); // This is a single line comment!
}
```

Go back to the root and replace the content of `bin/hello_world.dart` with the following code:

```
/*
 * This is a
 * multi-line
 * comment!
 */
void main(List<String> arguments) {
  hello_world.greetings();
}
```

Now you simply need to hit *Run* in the IDE or call the `dart run` command in case you are using the console. To create an executable file, use `dart compile exe bin/hello_world.dart` Now that we've got some familiarity with Dart, let's talk about Flutter!

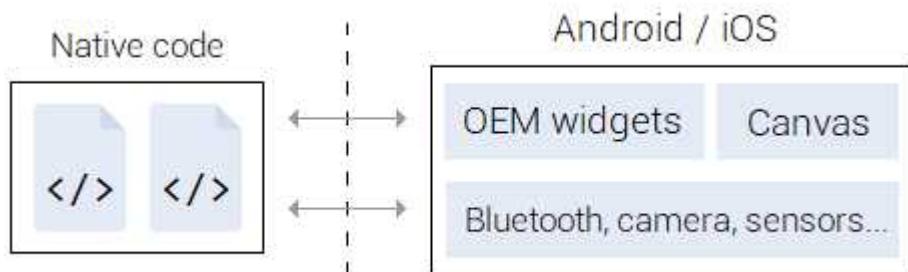
## 1.3 The Flutter framework

Flutter is an open-source framework made by Google for building natively compiled applications for mobile, desktop, web, and embedded systems with a single Dart codebase. Using widgets, you can take control over each pixel on the screen to build beautiful and performant user interfaces.



**Figure 1.3:** The Flutter logo.

To get an overview of how Flutter works and why it is different from all the other cross-platform frameworks, we're going to make some comparisons between Android and iOS. Since we want to keep this introduction simple, we're only looking at mobile platforms but for other platforms (such as Windows, macOS, or Linux) the logic is the same. Let's start by looking at how native applications generally work:



**Figure 1.4:** The native approach for mobile development.

Native UI components, also known as OEM widgets (*Figure 1.4*), are standard system components used by applications to paint contents on the screen. Each platform defines its own OEM widgets and expects them to be called by specific languages, such as Kotlin for Android or Swift for iOS. There is no way to use Kotlin on iOS, for example, because the operating system doesn't support that language and its libraries.

Albeit very productive and efficient, it is impossible with this mechanism to use a single language for multiple platforms. Some popular frameworks solve this problem by adding a “translator” between the framework and the application.

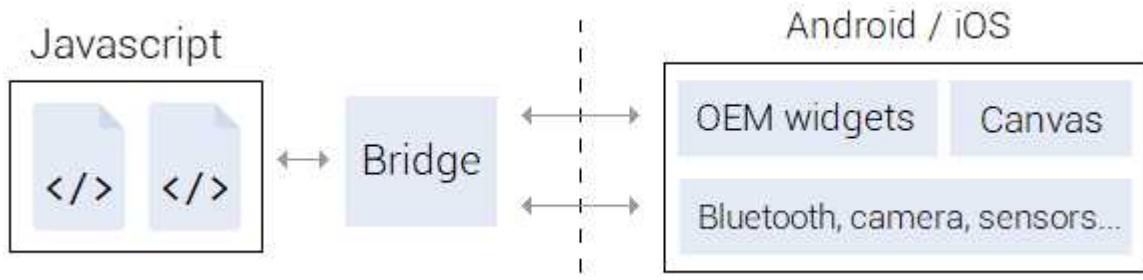


Figure 1.5: A “bridged” approach for cross-platform development.

A bridge is an abstraction layer between the application and the underlying operating system. With reference to *Figure 1.5*:

- The JavaScript code only interfaces with the bridge and knows nothing about the operating system. The bridge exposes APIs to communicate with the native platform and acts as an intermediary.
- The operating system only interfaces with the bridge to send and receive calls. As such, the bridge must be implemented in various programming languages. In *Figure 1.5*, for example, there must be an Android-specific and an iOS-specific bridge to run on different platforms. These implementation details are hidden by the framework, and you don’t care about them because you’ll always write JavaScript code.
- Since the bridge is built ad-hoc for a specific platform, it will also use platform-specific OEM widgets. In practice, even if the JavaScript code is the same for both Android and iOS, you won’t get an identical, pixel-perfect UI on both platforms because the bridge calls different (specific) OEM widgets.

You can see that a bridge is a sort of “real-time translator” between JavaScript and the underlying platform. A high volume of method calls and serialization/deserialization processes could put a lot of “pressure” on the bridge and cause a bottleneck, which negatively impacts the overall application performance.

There indeed are strategies to avoid too much “traffic” on the bridge and keep your application responsive, but that’s still something the developer has to deal with. The good news is that Flutter adopts an entirely different approach where no bridges or intermediary layers are needed. Flutter is compiled into native code, meaning that the application “speaks” the exact language of the host platform. Check this diagram:

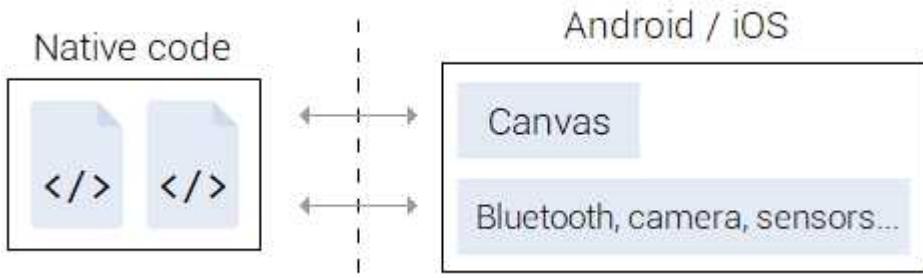


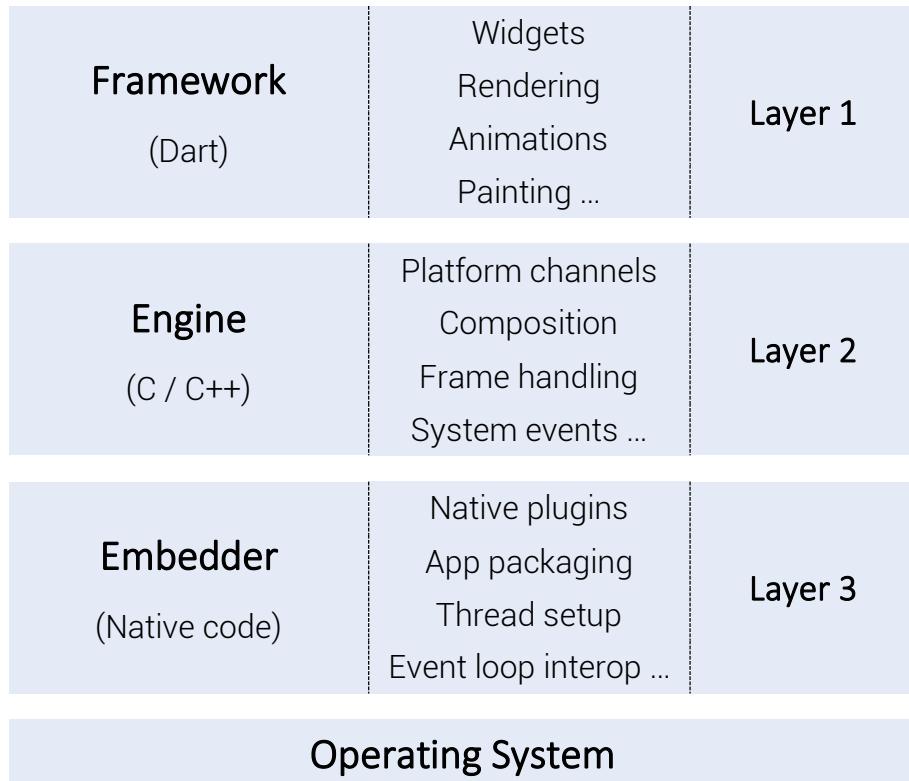
Figure 1.6: The Flutter approach for cross-platform development.

Since the Dart compiler can produce native ARM code, there is no need for code translation. The Flutter application directly communicates with the platform's native APIs. In addition, Flutter uses its own very efficient rendering engine called *Impeller*, which controls each pixel of the screen. This leads to a few significant consequences:

- As you can see in *Figure 1.6*, OEM widgets are not required because Flutter uses *Impeller* to paint pixels on the screen. Thanks to this architectural choice, Flutter is able to render anything it wants in a consistent way on any platform. For example, the same application made with Flutter looks identical (pixel-perfect) on Android and iOS (and all the other platforms).
- No bridge is required because the Dart code is compiled into native machine code. The platform can directly “understand” Dart instructions without interpreters. Flutter ships with a copy of its small and efficient C++ engine (which includes *Impeller*). Without the need for a bridge, the runtime performance of the applications is incredibly high (up to 120 fps on high-end devices).
- Flutter can still perform native API calls to the camera, sensors, Bluetooth, and much more. The engine provides hooks to the framework to make native calls in the underlying platform. We will cover all of this in *chapter 23 – “Platform interactions”*.

Flutter has a layered architecture that glues together the Dart source code, the C++ engine, and the platform-specific implementation. Layers are independent, have no access to the level below, and are replaceable. From the operating system point of view, a Flutter application is packaged in the same way as any other native application.

For mobile, desktop, and embedded operating systems, there are two levels between Dart and the host platform:



The operating system treats a Flutter application as if it was a native application thanks to the embedder, which is written in a platform-specific language. It's the glue that connects the Flutter engine to the underlying operating system. The currently available embedders are:

- Android: written in C++ and Java;
- iOS and macOS: written in Objective C++;
- Windows and Linux: written in C++.

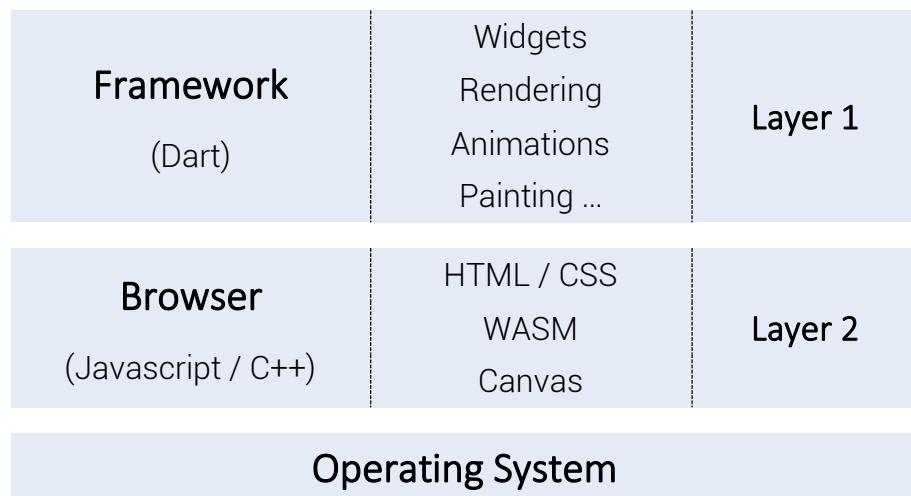
If you wanted a Flutter application to run in any other platform not listed above (like a PlayStation or a Nintendo Switch), you would need to create a new embedder for the target platform. This may seem the same idea as a bridge, but here everything is AOT compiled into native machine code so nothing is “translated” or “serialized” at runtime.

### Note

Imagine that a Flutter application was a plug, and you wanted to insert it in different kinds of socket walls. In this case, the embedder would be an adapter with the same shape for the plug but different configurations for the wall socket.

The Flutter C++ engine can be accessed in Dart using the `dart:ui` library, which basically contains C++ code wrapped in Dart classes. This isn't often the case because Flutter libraries provide a more developer-friendly API to create UIs.

You may be wondering why we haven't spoken about web support yet. The reason is that there are some special considerations to make. There are different layers for a Flutter web application:



The Flutter engine, written in C++, is designed to interact with an operating system rather than a web browser. A different approach is therefore required to run Flutter applications on the web.

Thanks to the Dart compiler's capabilities, Flutter applications are compiled into JavaScript code so the C++ engine is not required anymore. This means that Flutter generates a reimplementation of the engine on top of browser APIs with two different ways of rendering the contents:

1. **HTML mode:** The Flutter web application is rendered using HTML, CSS, a canvas, and SVG (vectorial images).
2. **CanvasKit mode:** The Flutter web application is rendered using CanvasKit, which is built using the Web Assembly (WASM) instruction format.

The HTML build mode produces smaller code sizes, the CanvasKit mode is faster and the graphic is more consistent with native platforms.

When building a Flutter web application for production, the Dart compiler bundles together the Dart code and the Flutter framework into a minified `.js` file. It also performs various optimization steps, such as tree shaking, which is a dead-code elimination technique.

### 1.3.1 Why Flutter uses Dart

Prior to the massive wave of success that Flutter was (and currently is) surfing, Dart was not a very popular language. Google still decided to choose it, although it didn't have much attention, and it's been a perfect guess for a few reasons:

1. The object-oriented programming style. The vast majority of developers are familiar with OOP languages, so the learning curve of Dart will not be steep. Its syntax is also very similar to popular programming languages (it's nothing revolutionary).
2. Dart is a predictable and high-performance language that can guarantee the best runtime quality and the lowest chance of frame dropping. The compiler produces native code for both ARM and x64 platforms which is a huge advantage.
3. Both Dart and Flutter are maintained by Google so the two teams can best cooperate to fulfill each other's needs, evaluate solutions together and better solve issues. The two can also evolve at their own pace and set (if needed) common goals.

As you already know, Dart is a strongly typed language so the compiler is very severe about types. This is an important layer of safety that enhances the developer's experience and reduces the risk of runtime errors. The language also has much more to offer:

- Tree shaking optimization, which basically is a dead code elimination strategy performed at compile-time;
- Thanks to Dart's JIT compiler, Flutter has a blazing-fast *hot reload* feature for applications;
- A package manager<sup>9</sup> and the possibility to either play with Dart or Flutter code snippets using an online playground tool (DartPad);
- DevTools is a rich suite of performance profiling tools with a powerful debugger. It is covered in *Appendix – Section 1 “Working with DevTools”*;

---

<sup>9</sup> <https://pub.dev>

Both Dart and Flutter are open-source projects, so anyone with a GitHub account can contribute for free. Throughout the book, we will cover as many Dart and Flutter feature as possible to help you become a true guru.

### 1.3.2 Creating a “Hello world” project

To properly create a new Flutter application, you can either:

- Use the console and choose among one of these commands:
  1. `flutter create`: creates a new Flutter project with a simple counter application;
  2. `flutter create --template=skeleton`: creates a new Flutter project with a more complex starter application having a theme changer and navigation;

This is the same advice we gave for new Dart projects. Even if you could use any project structure you want, prefer following the recommended structure and adhere to the well-established standards for Flutter project creations.

- Use an IDE. The leading IDEs we recommend using are Android Studio (with the Flutter plugin) and Visual Studio Code (with the Flutter extension). If you prefer going this way, create a new Flutter project and make sure to name it `hello_world`.

A Flutter project is nothing more than a traditional Dart project with some additional files and folders. The `lib/` folder contains the source code, the `test/` folder is for unit and widget testing, and the `pubspec.yaml` file is for SDK constraints, dependencies, and project-wide settings. Some Flutter-exclusive contents you won’t find in a Dart project are:

1. the `110n.yaml` file, which is used by the framework to internationalize your application;
2. the `assets/` folder, conventionally named this way, which contains static files to be bundled in the application such as images, audio, or video files;
3. the `android/, ios/, web/, macos/, windows/` and `linux/` folders contain platform-specific configuration required by Flutter when building for the desired platform.

In *chapter 21 – “Creating and maintaining a Flutter app”* we will work with these files and folders. The `main()` entry point is located in the `main.dart` file, along with some other Flutter-specific initialization calls:

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

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

The `runApp()` function prepares the execution environment and ensures that it can be rendered on the screen. In *chapter 10 – “Flutter, widgets and trees”* we will learn that widgets are the building blocks of any Flutter application and they can be composed to build beautiful UIs. For example, when we run the application on Windows, the `MyApp` widget produces this result:

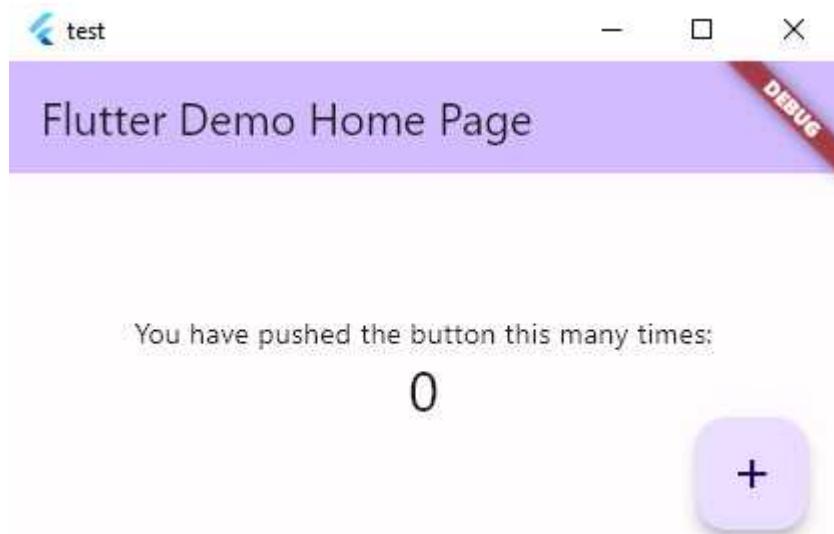
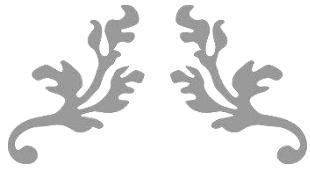


Figure 1.7: The starter Flutter app running on Windows.

Running the application on Android, iOS, web or any other platform will produce the same pixel-perfect result. There is no way to cross-compile for any platform (for example, you can't build a macOS executable from Windows).

You've made it to the end of the introduction! Now that we've introduced Dart and Flutter, it's time to go deeper and explore them in detail.



## Part 1: The Dart programming language

---

2 – Variables and types

3 – Control flow and functions

4 – Classes

5 – Inheritance, core classes and exceptions

6 – Generics and collections

7 – Advanced language features

8 – Futures, Streams and Isolates

9 – API overview and tests



# 2 – Variables and types

---

## 2.1 Variables

All variables in Dart have a type. You can either decide to explicitly declare the type or just let the compiler determine it. The syntax is very similar to other high-level programming languages:

```
var name = 'Alberto';
var age = 25;
```

In the above example, the `name` variable stores a reference to a `String` object whose value is `Alberto`. In this case, the type is inferred because the compiler is able to automatically determine the type by looking at the value. Any variable declared with `var` can later be re-assigned with a different value of the same type.

```
String name = 'Alberto';
int age = 25;
```

This is equivalent to the previous example, with the only difference being that we have explicitly declared the types rather than letting the compiler figure them out. All types in Dart descend from the `Object` supertype so we could have also declared the variables in this way:

```
Object name = 'Alberto';
Object age = 18;
```

There also is a fourth way to initialize a variable, but it's the least recommended way since it has no type safety:

```
dynamic name = 'Alberto';
dynamic age = 18;
```

The `dynamic` keyword is a sort of “wild card” for all types: its evaluation happens at runtime when the variable is created, so static analysis tools can't do any checks. This also means that the compiler and the IDE cannot detect type errors because they will only be discovered at runtime. According to the official Dart guidelines<sup>10</sup> and our experience, we recommend the following:

- prefer initializing variables using `var` or `final` (or `const`, which we're covering in the next section);

---

<sup>10</sup> <https://dart.dev/guides/language/effective-dart/design#types>

- whenever you see that the type of a variable might not be trivial to guess, avoid type inference and write down the type itself;
- avoid using `Object` and `dynamic` whenever possible.

There are few cases where the `Object` type is worth using, and you will see an example in *chapter 9 – Section 1.1 “The conversion library”*. Regardless, you will hardly ever work with it directly and the same also goes for `dynamic`.

### 2.1.1 final and const

When you know that the value of a variable is not going to change over time, you should use `final` instead of `var`. If the value of the variable will never change and it can be determined at compile-time, prefer the `const` keyword. Let's see a few examples:

- When using `var`, the value of the variable can be changed:

```
var age = 18;
if (isBirthday) {
    age = 19; // Ok, the new value is now 19
    age = '19'; // Error, the type is different
}
```

- When using `final`, the value of the variable cannot change:

```
final age = 18;
if (isBirthday) {
    age = 19; // Error, cannot re-assign a 'final' variable
    age = '19'; // Error, cannot re-assign a 'final' variable
}
```

- When using `const`, the value of the variable cannot change and it must be a compile-time constant value (such as a number, a string, or an `enum`):

```
const age = 18;
final newAge = 19;

if (isBirthday) {
    age = 19; // Error, cannot re-assign a 'const' variable
    age = newAge; // Cannot re-assign AND 'newAge' is not constant
}
```

In any case, thanks to Dart's strong typing system, after you assign the type to a variable it will never change. A `const` variable is implicitly `final` too. Most of the time, you will be using either `var` or `final`. The static analysis tools (or the IDE) will produce warnings to remind you to prefer one over the other (if it's the case).

### Note

Note that `final` doesn't provide any performance benefit when compared to `var`. They do the same thing, with the only difference that the former disallows re-assigning the variable.

You can also explicitly write down the type of a variable when using `const`, `final`, or `var`. This is useful when you want to "override" the default type-inference rule of the compiler. For example:

```
void main() {
    const integer = 18;
    const double notInteger = 18;

    print(integer.runtimeType.toString()); // int
    print(notInteger.runtimeType.toString()); // double
}
```

The compiler is smart enough to figure out that `18` is a whole number so the `int` type fits perfectly. The `const double` syntax instead disables type inference and tells the compiler the exact type we want to use. The `runtimeType` property (since it's inherited from `Object`) is available for any type and gives information about the runtime type of a variable. We can obtain the same output with a small trick:

```
void main() {
    const integer = 18;
    const notInteger = 18.0; // notice the '.0' at the end
    print(integer.runtimeType.toString()); // int
    print(notInteger.runtimeType.toString()); // double
}
```

The compiler treats `18` and `18.0` differently because, in the second case, there is a decimal digit (the zero) and thus the value doesn't fit into an `int`. Since `18` is a literal and also a compile-time constant, we used `const` but `final` would have also worked:

```

void main() {
    final integer = 18;
    final double notInteger = 18;

    print(integer.runtimeType.toString()); // int
    print(notInteger.runtimeType.toString()); // double
}

```

Since `const` values are also `final`, constant values can be assigned to `final` variables. However, a `final` value cannot be assigned to a `const` variable. For example:

```

// 'const' values can be assigned to 'final' variables
const constVal = 1;
final finalVal = constVal; // OK

// 'final' values CANNOT be assigned to 'const' variables
final finalVal = 1;
const constVal = finalVal; // Error

```

In other words, `const` is a stricter version of `final`. From all of the information we've covered up to now, we want to underline the most important parts to remember:

- If the value is never going to change, prefer initializing variables using `const` (for compile-time values) or `final` (for non-compile-time values). When possible, use `const` because it's more efficient than `final` (we'll see why in *chapter 4 – Section 2.3 “Constant constructors”*);
- If the value may change after its initialization, use `var`;
- When using `var`, `final` and `const` you can still manually define the type to “disable” the type-inference machinery and assign the type you want.

Let's move on and talk about variable initialization and null safety.

## 2.1.2 Nullability and non-nullability

As you may guess from the definition, a nullable variable can either be `null` or have a value of its type. On the other side, a non-nullable variable can never hold the `null` value. For example:

```

int? age = 25; // a 'nullable' variable (because it COULD be 'null')
age = null;

```

```

String? name; // 'name' is automatically initialized with 'null'
String? name = null; // ok but the 'null' initialization is redundant

```

The question mark after the type name (`int?`) denotes a nullable type and so `age` could either be an integer or `null`. If we didn't add the question mark, then the variable would have been non-nullable and thus never `null`:

```
int age = 25; // a 'non-nullable' variable (because it will NEVER be 'null')
age = null; // Error! 'age' is 'int' (not 'int?') so it cannot be 'null'
```

From Dart 2.12 onwards, the language enables non-nullability by default. A local non-nullable variable doesn't need to be initialized immediately, but you must assign it a value before it's used. For example, the following example is valid because we're only using `cartItems` after it's been initialized with a value:

```
void main() {
  int cartItems;

  if (cartIsEmpty) {
    cartItems = 0;
  } else {
    cartItems = getCartSize();
  }

  print(cartItems); // OK because 'cartItems' was initialized
}
```

On the other side, it is a compile error trying to access a non-nullable before it's initialized:

```
int cartItems;
print(cartItems); // Error, 'cartItems' wasn't initialized before being used

if (cartIsEmpty) {
  cartItems = 0;
} else {
  cartItems = getCartSize();
}
```

This is very safe because you're getting the error at compile-time and not at runtime. In addition, you're guaranteed that a value will never be `null` so there is no need to make manual nullability checks on the code:

```
int value;
value = 10;

if (value != null) { // Useless because 'value' will NEVER be null
  print(value);
}
```

Note that this initialization rule is only valid for local variables. For top-level and class variables instead require the `late` modifier, which is used in two cases:

- to declare a non-nullable (top-level or class) variable that it initialized after its declaration;
- lazily evaluating functions (more on this in *chapter 3 – “Control flow and functions”*);

The `late` keyword has a lot of interesting cases where it can be used, and we will see all of them in the next chapters. For now, let's see how it can be used to initialize variables:

- We have just seen that, in the case of local variables, no `late` keyword is required. The Dart analyzer can automatically detect whether the variable was already initialized or not:

```
void main() {
    int a;
    a = 10;
    print(a);
}
```

- In the case of top-level variables, we need the `late` keyword to initialize them lazily:

```
late int a;

void main() {
    a = 10;
    print(a);
}
```

In this case, the analyzer cannot determine whether the variable was already initialized or not. As such, we need to use `late` and manually ensure (as we did) that it's initialized before being used. We paid attention to call `a = 10;` before printing the value. If we didn't, we would have had a runtime error:

```
late int a;

void main() {
    print(a); // Runtime exception
    a = 10;
}
```

The runtime exception is thrown because a `late` non-nullable variable is accessed before being set. In other words, `late` disables some compile-time type checks and gives you the responsibility of correctly initializing variables. This isn't a bad thing but you indeed have to pay more attention to the code you write.

- You can add the `final` modifier in front of `late` for one-time initialization of the variable.

Our recommendation is always to initialize a variable immediately whenever possible and avoid top-level variables, which really remind global variables. We also recommend using `late` only when needed.

When you declare a nullable variable, it is `null` by default so you can always use it even without assigning a value:

```
int? value; // initialized to 'null'
print(value); // prints 'null'

// This null-check ensures that 'value' is not 'null'
if (value != null) {
    doSomething(value);
}
```

Keep in mind that there absolutely is **nothing wrong with `null`**. It is extremely useful to represent the absence of a value. The problem is not `null` itself but rather the fact that it might lead to certain situations that you don't expect. Nullable variables may be harder to maintain, but sometimes they're just the right choice. In general, our recommendation is:

- Try to use non-nullable variables as much as possible. For example, if you want to indicate that the cart of your shopping app may be empty, prefer `int cartCount = 0;` rather than `int? cartCount;`.
- When you need to represent the “absence” of a value and using a default one doesn’t make sense, then use a nullable variable.

Before moving on, we need to spend some words on the conversions you can make between nullable and non-nullable. Any non-nullable variable can be assigned to a nullable variable but the contrary is not allowed. For example:

```
// This is a function that returns a nullable integer and accepts no values
int? value() {
    return 5;
}

void main() {
    int? nullable = value(); // OK
    int notNullable = value(); // Compiler error
}
```

This example doesn't compile because the function `value()` returns a nullable integer, which cannot be assigned to a non-nullable value. On the other side, if the function returned an `int`, then it could be given to both variables. For example:

```
// Notice that it returns 'int' rather than 'int?'
int value() {
    return 5;
}

void main() {
    int? nullable = value(); // OK
    int notNullable = value(); // OK
}
```

You can manually use the bang operator to convert a non-nullable variable into a nullable variable. For example:

```
int? value() {
    return 5;
}

void main() {
    int? nullable = value(); // OK
    int notNullable = value()!; // OK because of the '!' operator
}
```

The function's return type is `int?`, a nullable value, but we know it is returning a non-nullable value so we can use `value()!` (with the exclamation mark at the end) to tell the compiler: "I know what I'm doing. Convert the result to a non-nullable value". If it returned `null`...

```
int? value() {
    return null;
}

void main() {
    int? nullable = value(); // OK
    int notNullable = value()!; // Runtime error
}
```

... we would get a runtime error because we used the bang operator on `null`, which is forbidden. Notice that the bang operator (!) throws runtime errors because it makes a runtime cast. As such, the compiler won't be able to help you: you will need to manually check if the cast is safe or not.

Now that you have a complete overview of variables initialization and null safety, we can proceed to discover Dart's data types.

## 2.2 Data types

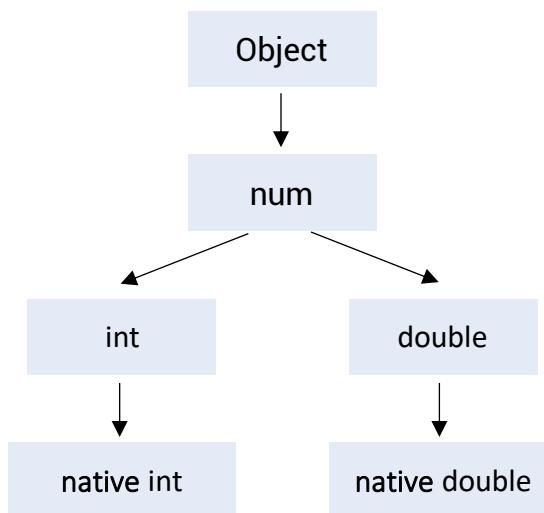
In most cases, Dart types can be initialized with literals, such as `true` (a boolean literal), `19` (an integer literal), or `'Hello world'` (a literal for a String). In *chapter 6 – “Generics and collections”* we will cover generic data types used for containers like lists and maps. In *chapter 5 – section 2.2 “Records”* we will discuss records (particular data types that map to classes).

### 2.2.1 Numbers

In Dart, there only are two types that represent numbers:

- `int`. On the web, integers are represented using JavaScript so values range from  $-2^{53}$  to  $2^{53}$ .
  1. On native platforms instead, values can range from  $-2^{63}$  to  $2^{63}-1$ .
- `double`. Double precision, 64-bit floating point numbers that follow the popular IEEE 754 standard.

Both `int` and `double` are classes that descend from `num`, an abstract supertype that includes various operators and basic math functions such as `abs()`, `floor()`, `ceil()`, `round()` and much more. Note that `int` and `double` are abstract types too, because the compiler provides their concrete implementation according to the platform for which the application was built.



For native targets (mobile and desktop), `int` is a 64-bit int and `double` is a 64-bit IEEE floating point number. On the web, Dart interoperates with JavaScript and so in both cases the 64-bit IEEE format is used. `num`, `double`, and `int` cannot be extended or implemented by regular Dart classes.

## Note

These implementation details are handled internally by the compiler so you shouldn't worry about them. Just remember that integers and floating-point values are platform specific for performance and size reasons.

As we've already seen, integers are inferred from literals without decimal points:

```
final a = 1; // 1
final b = 0xA5; // 165
final c = -3e5; // -300000
```

The hexadecimal representation is prefixed with `0x`. Whenever a number includes a decimal, then it's treated as a double:

```
final a = 7.63; // 7.63
final b = 1.5e-1; // 0.15
```

You can also declare variables using `num` and, since it's the supertype of integers and double, it can hold both values:

```
num a = 12.534;
print(a.runtimeType); // double

a = 6;
print(a.runtimeType); // int
```

We recommend sticking with `int` and `double` or, even better, using type inference with `final`/`var` to let the compiler resolve types. Literals are compile-time constants, and they produce constant expressions as well:

```
const a = 10.5;
const b = 12;
const c = a * b; // 'c' is const too

const x = 10.5;
final y = 12;
const z = x * y; // Compiler error
```

In the second code block, since `y` isn't a constant, the result of the operation cannot be determined at compile-time and so `z` cannot be `const`. The type inference machinery never infers `num`: it always assigns either `int` or `double`.

## 2.2.2 Strings

Strings are represented by the `String` object, which is a sequence of UTF-16 code units. They can be initialized with either single or double quotes:

```
const name = 'Alberto';
const surname = "Miola";
```

The general rule is that you should always use single quotes unless the string itself doesn't already contain one<sup>11</sup>, in which case a double quote is required. For example:

```
print("O'Neil"); // Correct
print('O'Neil'); // Error
```

Since `O'Neil` already has a single quote, we must use double quotes to build the string correctly. The second example confuses the compiler because the apostrophe conflicts with the single quote sign. Dart has no `char` type to hold single characters: they are represented by the `String` class too:

```
const hello = 'Hi';

final String h = hello[0]; // H
final String i = hello[1]; // i

print(hello[2]); // Runtime error
```

Identifiers and expressions can be interpolated in a string using the dollar sign:

```
const single = 'Single quote';
const double = "Double quote";

// Interpolate an identifier into a string
const age = 25;
print('I am $age years old');

// Expressions need '{' and '}' after '$'
const actualAge = 25.5;
print('I am ${actualAge.round()} years old');

// '$' already calls 'toString()' so this is not required
print('I am ${actualAge.toString()} years old');
```

---

<sup>11</sup> [https://dart-lang.github.io/linter/lints/prefer\\_single\\_quotes.html](https://dart-lang.github.io/linter/lints/prefer_single_quotes.html)

The  `${expr}`  syntax already calls the `toString()` method so you don't need to call it explicitly. Dart also allows defining multiline strings, which are initialized with triple quotes, and still follow the interpolation rules we've just seen. For example:

```
const singleQuotes = '''
SELECT name, surname, age
FROM people
WHERE name = $name
''';

const doubleQuotes = """
SELECT name, surname, age
FROM people
WHERE name = $name
""";
```

In this case, single or double quotes make no difference. String concatenation also happens with the `+ operator`, which is optional on new lines, but interpolation with `$` should be preferred<sup>12</sup>:

```
// Preferred way
print('I am $age years old');

// Good, but '$age' should be preferred
print('I am ' + age.toString() + ' years old');
```

To go to a new line or add a tabulation use the `\n` or `\t` symbols, respectively. They can be escaped in raw strings, which are prefixed by an `r` in front of the quotes:

```
// I am
// 25 year old
print('I am\n25 years old');

// I am\n25 years old
print(r'I am\n25 years old');

// I am      25 years old
print('I am\t25 years old');
```

The `r` in front of the string basically evaluates the `\` as a regular character, so no special formatting is applied. It also disables string interpolation:

---

<sup>12</sup><https://dart.dev/guides/language/effective-dart/usage#prefer-using-interpolation-to-compose-strings-and-values>

```

const age = 25;

// Prints: I am 25 years old
print('I am $age years old');

// Prints: I am $age years old
print(r'I am $age years old');

```

Raw strings also are handy when working with regular expressions.

#### 2.2.2.1 Good practices

`String` is an immutable object so whenever the content needs to be changed, a new instance is always created. For this reason, using the `+` operator to concatenate a lot of strings might be very inefficient because a lot of instances are re-created each time. For example:

```

var value = '';

for(var i = 0; i < 9000000; ++i) {
  value += '$i, \n';
}
print(value);

```

Each time the `+` operator is called, `value` is assigned with a new instance containing the old value plus the newly appended number. In other words, the above code creates 9000000 new `String` objects, one per iteration! A more efficient approach is the following:

```

final value = StringBuffer();

for(var i = 0; i < 9000000; ++i) {
  value.writeln('$i, ');
}

// The string is built for the first time here
print(value.toString());

```

A `StringBuffer` is used to efficiently concatenates strings. It incrementally builds strings without creating new instances on each concatenation and thus it is recommended by the Dart<sup>13</sup> team. The `String` object is only created when the buffer is converted using `toString` or interpolation.

---

<sup>13</sup> [https://dart-lang.github.io/linter/lints/use\\_string\\_buffers.html](https://dart-lang.github.io/linter/lints/use_string_buffers.html)

The language also has built-in support for number-to-string and string-to-number conversions. For example, the static method `parse` is used to convert a string into a number:

```
const value = '12';

// Using 'tryParse' for String-to-number conversions
final a = int.tryParse(value); // 12
final b = int.tryParse(value, radix: 5); // 7
final c = double.tryParse(value); // 12.0

// Invalid conversions return 'null'
final d = int.tryParse(''); // null
final e = double.tryParse(''); // null
```

The `tryParse` method converts the given `String` into an `int` or a `double` and returns `null` if the conversion fails. The optional `radix` parameter converts the given integer from a specific base: in the example, we're parsing 12 in base 5 (which is 7). Alternatively, you could also use the `parse` method, which returns a non-nullable value, but it throws an exception if the conversion fails:

```
final a = int.parse('12'); // 12
final b = double.parse('-5.76'); // -5.76

// Runtime errors
final c = int.parse('');
final d = double.parse('');
```

On the other hand, the string-to-number conversion is also straightforward. There are multiple variants of the `toString` methods:

```
final a = 100.toString(); // 100
final b = 3.44.toString(); // 3.44
final c = (-3.44).toString(); // -3.44

// Conversion rounds or adds trailing zeroes
final d = 5.126.toStringAsFixed(2); // 5.13
final e = 5.126.toStringAsFixed(5); // 5.12600
final f = 5.126.toStringAsFixed(7); // 5.1260000

// Conversion only considers the given significant digits
final g = 5.126.toStringAsPrecision(2); // 5.1
final h = 5.126.toStringAsPrecision(5); // 5.1260
final i = 5.126.toStringAsPrecision(7); // 5.126000

// Exponential notation
final j = 5.126.toStringAsExponential(2); // 5.13e+0
```

```
// Converting from base 10 to other bases.  
final k = 53.toRadixString(2); // 110101  
final l = 53.toRadixString(8); // 65  
final m = 53.toRadixString(12); // 45
```

Other recommendations we have are:

- Prefer interpolating strings with `$` or  `${expr}` ;
- Avoid concatenating strings using the `+` operator;
- Prefer using `StringBuffer` for composing long strings.

The API of the `String` object is huge: it has the `length` getter to retrieve the string length, trimming methods, replace methods and much more. Consider visiting the official documentation<sup>14</sup> for an exhaustive overview of the `String` API.

#### 2.2.2.2 Runes

Unicode is an encoding system that associates an integer value (called code unit) to each character, digit, and symbol that humans use in all the world's writing systems. The Unicode standard can be implemented using various transformation formats<sup>15</sup> such as UTF-8, UTF-16, and UTF-23. For example, in UTF-16:

- The letter `a` is associated with `97` (61 in hexadecimal) code unit;
- The number `5` is associated with `53` (35 in hexadecimal) code unit;
- The Japanese symbol `本` is associated with the '26412' (672C in hexadecimal) code unit.

The `a5` string, for example, in UTF-16 is represented as `\u0061\u0035` because the letter `a` is associated with the `0x0061` code unit, and the digit `5` is associated with `0x0035`.

A `String` object in Dart is a sequence of Unicode UTF-16 code units that are called runes. Each string has the `codeUnitAt(int index)` function to get the 16-bit UTF-16 code unit at the given index. For example:

```
const string = 'Hi';  
print(string.codeUnitAt(0)); // 72  
print(string.codeUnitAt(1)); // 105
```

---

<sup>14</sup> <https://api.dart.dev/stable/dart-core/String-class.html>

<sup>15</sup> UTF stands for “Unicode Transformation Format”, which is a format describing how Unicode is implemented.

Considering the above example, we can also build the `Hi` string by only using the `fromCharCode` constructor, which asks for a code unit value:

```
final h = String.fromCharCode(72);
final i = String.fromCharCode(105);

print('$h$i'); // Hi
```

From these examples, we can understand that `72` is the integer UTF-16 code unit for the letter `H` and `105` instead is associated with the letter `i`. If we converted these integers into their hexadecimal form, we could still use code points to build the `Hi` string but using the `\u` escape value:

```
// 72 in base 10 is 48 in base 16
final h = '\u{0048}';

// 105 in base 10 is 69 in base 16
final i = '\u{0069}';
print('$h$i'); // Hi
```

You can always use the `runes` property to return all of the runes (code units) as base ten integer values. They can directly be used in the `fromCharCode` constructor to build the string, but if you want to use the `\u` escape character, then they must be converted into base 16 values:

```
// Code points for the 'Dart' string
print('Dart'.runes.toList()); // [68, 97, 114, 116]

// Printing 'Dart' from its integer, base 10 code points
print(String.fromCharCode(68)); // D
print(String.fromCharCode(97)); // a
print(String.fromCharCode(114)); // r
print(String.fromCharCode(116)); // t

// Printing 'Dart' from the hexadecimal code points
print('\u{0044}'); // 68 in base 10 --> 44 in base 16 --> \u{0044} --> D
print('\u{0061}'); // 97 in base 10 --> 61 in base 16 --> \u{0061} --> a
print('\u{0072}'); // 114 in base 10 --> 72 in base 16 -> \u{0072} --> r
print('\u{0074}'); // 116 in base 10 --> 74 in base 16 -> \u{0074} --> t
```

Emojis can also be expressed using UTF-16! For example, the laughing one is `\u{1f606}`. Runes are for low-level string manipulations and you'll hardly ever need to work with them.

## 2.2.3 Booleans

The `bool` type represents boolean values, which can only hold the `true` or `false` literals. Both are compile-time constants:

```
const test = 5 == 0; // false
const notTest = !test; // true

const isDartNice = true;
const isDartBad = false;
```

Unlike C++, in Dart you cannot assign `0` and `1` to a `bool` type: for example, `bool test = 0` is not valid in Dart because only `bool test = false` is. The literal `0` doesn't convert to `false`.

## 2.2.4 Enumerated types

Also known as *enums*, enumerated types are containers for constant values declared with the `enum` keyword. Here's a simple example:

```
enum Fruit {
  apple,
  blueberry,
  strawberry,
  lemon,
  melon,
}

void main() {
  const blueberry = Fruit.blueberry;

  print('$blueberry'); // Fruit.blueberry
  print('$blueberry.name'); //blueberry

  print('${blueberry.index}'); // 2
}
```

Each item in the enumeration has an `index`, which corresponds to the zero-based position of the value in the declaration list. In the above example, `Fruit.blueberry.index` returns `2` because `blueberry` is the third item in the 0-indexed list. You can also build an enumeration from a string but an exception is thrown if the name doesn't match a value:

```
const blueberry = 'blueberry';
const watermelon = 'watermelon';

print('${Fruit.values.byName(blueberry)}'); // Fruit.blueberry
print('${Fruit.values.byName('strawberry')}); // Fruit.strawberry

print('${Fruit.values.byName(watermelon)}'); //Runtime error
print('${Fruit.values.byName(')}'); //Runtime error
```

An `enum` in Dart can also have variables and methods to enhance the capabilities of the values. For example, we could decide to add a `color` property to the `Fruit` enumeration of the previous example to associate a color with each value:

```
enum Fruit {  
    // Elements always go first  
    apple('green'),  
    blueberry('blue'),  
    strawberry('red');  
  
    // Enum fields must always be 'final'  
    final String color;  
    // Only one constructor is allowed and it MUST be constant  
    const Fruit(this.color);  
}  
  
void main() {  
    print('${Fruit.apple.color}'); // green  
}
```

As you can see, they may remind you of a class but they're not the same thing. Enumerations cannot be abstract, inheritance doesn't work on them, they can only have a single `const` constructor, and enumeration values always need to be declared first.

## Note

The `const` keyword can also be used in front of a `class` constructor declaration. We will cover this in detail in *chapter 4 – Section 2 “Constructors”*.

Without member variables in the `enum`, we would need to manually write the conversion logic. For example:

```
String enumValueColor(Fruit fruit) {  
    switch (fruit) {  
        case Fruit.apple:  
            return 'green';  
        case Fruit.blueberry:  
            return 'blue';  
        case Fruit.strawberry:  
            return 'red';  
    }  
}
```

Rather than creating helper functions to convert enumeration values, using member variables is less verbose and more maintainable. Enumerations can define `static` methods and properties:

```
enum Test {  
    someValue;  
    static const int number = 0;  
    static void hello() => print('Hello world');  
}
```

Dart also allows defining non-static methods, getters, and variables but they're useless since an `enum` cannot be instantiated as if it was a class.

## 2.2.5 Lists

Any kind of container in Dart is represented by a generic type, which we will cover in detail in *chapter 6 – “Generics and collections”*. What you may know under the name of “array” in Dart it is called “list,” and it’s represented by the generic type `List<T>`.

### Note

The `T` you see in the `List<T>` declaration is a generic placeholder for any type. For example, you could replace `T` with `String` to create a `List<String>` type.

Lists are 0-indexed collections of items that can be accessed using the `[]` operator. A list can be declared using square brackets:

```
const integerList = [1, 2, 3];  
const stringList = ['a', 'b', 'c'];  
  
print('$integerList'); // [1, 2, 3]  
print('${integerList[2]}'); // 3  
  
print('$stringList'); // [a, b, c]  
print('${stringList[5]}'); // Runtime error
```

A runtime error is thrown if the index value is not within the array bounds. Generic collections, including lists, have a lot of interesting properties we will explore later. For now, it’s enough for you to be aware that “arrays” in Dart are in reality `List<T>` objects and they can be declared using square brackets.

## 2.2.6 Dart's environment variables

Any Dart application can initialize strings, numbers, or boolean values from variables being declared from “outside” the program itself. The `--define` flag can be added to the `dart run` command to create environment variables. For example:

```
dart run --define=api-key=abcdefg
```

Using this command, we can read the content of the `api-key` variable from our Dart program using a special constructor called `String.fromEnvironment`:

```
void main() {
    final apiToken = String.fromEnvironment('api-key');
    print(apiToken); // abcdefgh
}
```

In the same way, environment variables can also be loaded from numbers and booleans. You can even define a default value in case the key wasn't associated to a value when running the code:

```
void main() {
    // dart run --define=myage=10
    final intValue1 = int.fromEnvironment('age', defaultValue: -1); // -1

    // dart run --define=age=10
    final intValue2 = int.fromEnvironment('age', defaultValue: -1); // 10

    // dart run --define=status=true
    final boolValue = bool.fromEnvironment('status'); // false
}
```

To define multiple Dart environment variables at once, you just need to repeat the `--define` command with the same key-value format:

```
dart run --define=age=10 --define=status=false
```

Hardcoding the strings into the app is no different and no less secure than environment variables. There is a Dart linter rule that suggests avoiding environment variables<sup>16</sup> since they create a hidden global state, which leads to less readability and more challenging maintenance of the project. We recommend reading command-line arguments or reading data from a configuration file inside the application itself.

---

<sup>16</sup> [https://dart-lang.github.io/linter/lints/do\\_not\\_use\\_environment.html](https://dart-lang.github.io/linter/lints/do_not_use_environment.html)

## 2.3 Data type operators

Dart builds expressions using operators, such as + and -, on primitive data types. The language also supports operator overloading but we will cover it in *chapter 4 – “Classes”*.

### 2.3.1 Arithmetic operators

Arithmetic operators are commonly used on `int` or `double` to build numerical expressions. They may also be used somewhere else, as it happens with operator `+` to concatenate strings.

Symbol	Meaning	Example
<code>+</code>	Add two values	<code>2 + 3 // 5</code>
<code>-</code>	Subtract two values	<code>3 - 5 // -2</code>
<code>*</code>	Multiply two values	<code>6 * 3 // 18</code>
<code>/</code>	Divide two values	<code>9 / 2 // 4.5</code>
<code>~/</code>	Integer division of two values	<code>9 ~/ 2 // 4</code>
<code>%</code>	Remainder (modulo) of an int division	<code>5 % 2 // 1</code>

Prefix and postfix increment or decrement operators work as you’re used to see in the most popular programming languages:

```
var a = 10;  
++a; // a = 11  
a++; // a = 12  
  
var b = 5;  
--b; // b = 4;  
b--; // b = 3;  
  
var c = 6;  
c += 6; // c = 12  
  
a++++; // invalid, postfix operators cannot be chained  
++++a; // invalid, prefix operators cannot be chained
```

The prefix operator (`++a`) first increments the value and then returns it, but the postfix (`a++`) operator returns the value immediately and increments it afterward.

### 2.3.2 Relational operators

Equality and relational operators are used in boolean expressions, which are generally found in `if`, `for`, or `while` statements, which will be covered in the next chapter.

Symbol	Meaning	Example
<code>==</code>	Equality test	<code>2 == 6 // false</code>
<code>!=</code>	Inequality test	<code>2 != 6 // true</code>
<code>&gt;</code>	Greater than	<code>2 &gt; 6 // false</code>
<code>&lt;</code>	Less than	<code>2 &lt; 6 // true</code>
<code>&gt;=</code>	Greater than or equal to	<code>2 &gt;= 6 // false</code>
<code>&lt;=</code>	Smaller than or equal to	<code>2 &lt;= 6 // true</code>

The operator `==` logic only works with objects of the same type. A warning is emitted when you try to compare two objects with unrelated types.

### 2.3.3 Type test operators

These operators are used to make runtime checks on object instances.

Symbol	Meaning	Example
<code>as</code>	Cast to another type	<code>obj as String</code>
<code>is</code>	True if the object is of a specific type	<code>obj is String</code>
<code>is!</code>	True if the object isn't of a specific type	<code>obj is! String</code>

The `as` operator performs a cast, so it returns a new type, but `is` and `is!` make a check, so they return a boolean value.

### 2.3.4 Logical operators

Logical operators are commonly used inside if statements to create boolean expressions.

Symbol	Meaning	Example
!	Changes true to false and vice versa	<code>!true // false</code>
&&	Returns true if both sides are true	<code>true &amp;&amp; true // false</code>
	Returns true if at least one is true	<code>true    false // true</code>

Keep in mind that the `!` operator could be used in two completely different cases:

- When used in front of a boolean variable, it returns the negation:

```
const isEven = 20 % 2 == 0; // true
const isOdd = !isEven; // false
```

- When used after a nullable variable, it tries to cast it (at runtime) to a non-nullable type:

```
int? value = 1; // 'value' is int?
int newValue = value!; // 'newValue' is int
```

Even if you confused them, the analyzer would help you reminding the difference.

### 2.3.5 Bitwise and shift operators

Unless you need to work with bit manipulation or low-level code, you'll hardly ever need to use these operators. Bit manipulations should always be documented because long-series of low-level operations are hard to read and understand.

Symbol	Meaning	Example
a & b	AND	<code>0xF &amp; 0xA // 10</code>
a   b	OR	<code>0xF   0xA // 15</code>
a ^ b	XOR	<code>0xF ^ 0xA // 5</code>
~	NOT	<code>~0xF // -16</code>
a << b	Shift left (SHL)	<code>0xAC &lt;&lt; 2 // 2B0</code>
a >> b	Shift right (SHR)	<code>0xAC &gt;&gt; 2 // 2B</code>
a >>> b	Triple SHR	<code>42 &gt;&gt;&gt; 2 // A</code>

## 2.3.6 Compound assignment operators

Compound assignment operators are used to perform an action and return a result at the same time. For example, the \*= operator multiplies the left and right values storing the result on the left variable.

Symbol	Meaning	Example
+=	Sum and assign	<code>var a = 2; a += 2; // a = 4</code>
-=	Subtract and assign	<code>var a = 2; a -= 2; // a = 0</code>
*=	Multiply and assign	<code>var a = 2; a *= 2; // a = 4</code>
/=	Divide and assign	<code>var a = 5.0; a /= 2; // a = 2.5</code>
~/=	Integer divide and assign	<code>var a = 5.0; a ~/= 2; // a = 2</code>
%=	Remainder and assign	<code>var a = 5; a %= 2; // a = 1.0</code>
??=	Assign only when null	<code>int? a; a ??= 10; // a = 10</code>

In addition to the operators above, there also are bitwise compound assignment operators that work in the same way:

```
var a = 0x3;

// Double shift operators
a <<= 0xEA;
a >>= 0xEA;

// Triple shift operator
a >>>= 0x1;

// Logical AND, OR and XOR operators
a &= 0x2;
a |= 0x2;
a ^= 0x1;
```

These operators cannot be used on `const` and `final` variables.

## Deep dive: Null safety in Dart

Null safety is a major change (introduced in Dart 2.12) that replaced the old unsound optional type system with a sound static type system. Before diving into the details, let's make an introduction to understanding the problem that had to be solved. For example, look at this Dart 1.24 code which has no null safety:

```
bool isEven(int value) {  
    return value % 2 == 0;  
}  
  
void main() {  
    isEven(null);  
}
```

This program compiles successfully and throws a runtime exception when trying to execute the modulo operation. The reason is that `null` is an instance of the `Null` class, which doesn't define operator `%`. The main issue with this code is that `value` could either be `int` or `null` and there is no way to determine it at compile-time. You're forced to make a runtime check:

```
bool isEven(int value) {  
    if (value == null) {  
        return false;  
    } else {  
        return value % 2 == 0;  
    }  
}
```

As you can imagine, writing `if` statements everywhere to check for the nullability of a variable is not reasonable and also adds a ton of boilerplate code. The Dart team didn't want to remove `null`: instead, they tried to make its usage `safe`. Note that there is nothing wrong with `null` because it is sometimes required to represent the absence of a value.

### Note

The problem is that without a sound null-safe system (like the one in Dart 1.x), `null` can go *where you don't expect* and cause problems. On the contrary, a sound null-safe system gives you the power to control where `null` can flow through your program and avoid unexpected scenarios.

From Dart 2.12 onwards, the Dart language has a *sound null safe* system. In practical terms, it means that variables can never be `null` by default unless you manually say otherwise. For example, an `int` can never be `null` unless you append a question mark at the end. The `int?` type is nullable because it can either be `int` or `null` at runtime.

### Note

The biggest difference here is that you can precisely control where `null` can go. For example, if a variable is of `String` type, it will never be `null`. If a variable is of `String?` type, you are aware that it could be `null` and thus you act accordingly.

This system dramatically reduces the chances of encountering unexpected `null` values because you decide where `null` can go.

Let's dig deeper to compare the old, unsafe system with the new null safe one. In both type systems, `null` is an instance of the particular `Null` class (which cannot be extended, implemented, mixed, or instantiated). This a simplified diagram that describes how types are structured along the hierarchy without null safety:

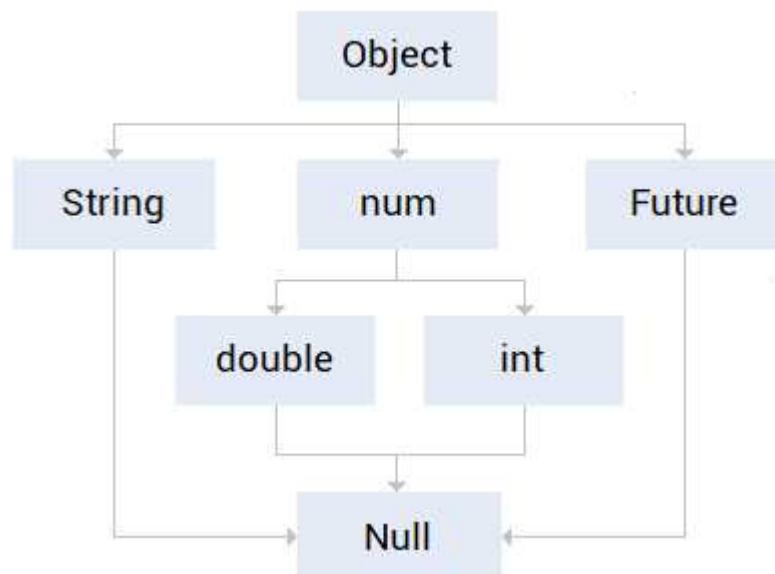


Figure 2.1: A simplified diagram of Dart's type hierarchy without null safety.

As you can see, `Null` is a “bottom type” (meaning that it’s a “subtype of all types”). This is why you can replace any type with `null` and the compiler doesn’t give errors. Runtime errors come from the

fact that the `Null` class only defines the `toString()` and comparison methods (`hashCode` and `operator==`). For example, you can call the `round()` method on a `double` but if it was `null`, you would get a runtime error because the `Null` object doesn't define `round()`. Null safety solves this problem by changing the type hierarchy:

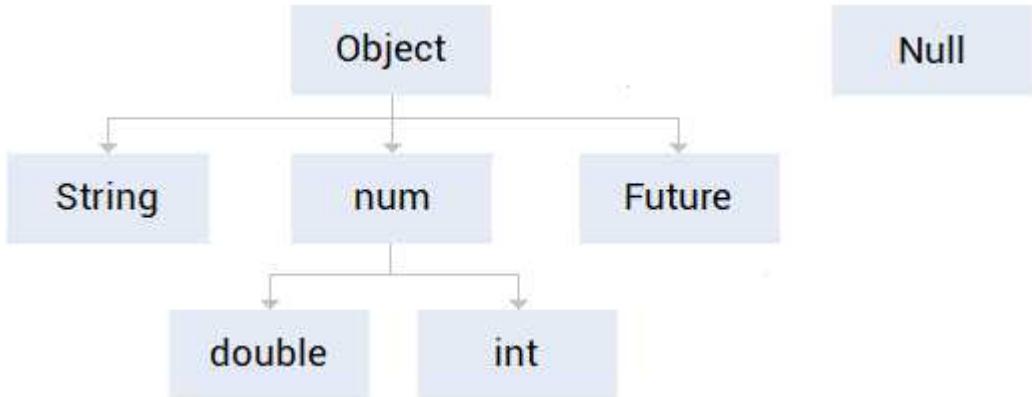


Figure 2.2: A simplified diagram of Dart's type hierarchy with null safety.

Since `Null` no longer is a subtype of all types, no type except `Null` is compatible with `null`. In other words, `Null` still exists but it stays outside of the “regular” `Object` hierarchy. If you had a variable whose type is `String`, it would never be null because `Null` is not a bottom type anymore. This change fixes all null-reference errors.

Even if this change moved `Null` outside of the `Object` hierarchy, it's still there. The difference is that `null` is treated differently. Null-safe Dart code handles nullable types by appending a question mark (?) at the end of the type name:

```
bool isEven(int? value) {
  if (value == null) {
    return false;
  } else {
    return value % 2 == 0;
  }
}

void main() {
  isEven(null);
}
```

The example is correct because the compiler understands that we're using an `if` statement to make sure that we're not referencing `null`. In fact, the body of the `else` branch is a safe place because

within that scope `value` cannot be `null`. Since the system is “nullsafe”, for “safe-ty” you can’t do much with nullable types: you can only call `toString()` and make equality checks but you cannot directly access its properties. For example:

```
double? pi() => 3.1415;

void main() {
    final value = pi(); // inferred as 'double?'
    print(value.toString()); // OK
    print(value.round()); // Compiler error
}
```

The compile-time error here is caused by the fact that you’re using `value` as if it could never be `null`, which is not the case. The `Null` object only defines `operator==`, `hashCode`, and `toString` so those are the only methods you can use. The compiler is very smart: it notices that you haven’t checked whether `value` is `null` or not and thus blocks you immediately with an error. To fix the error of the example, there are a few possibilities:

1. Use an `if` statement to protect the code from accessing `null`. The compiler is smart and understands when the scope is safe:

```
if (value != null) {
    print(value.round());
}
```

In this case, you (and the compiler) see that within the `if` scope `value` can never be `null` because we have just evaluated it. As such, you’re free to use `value.round()` as if `value` was an `int` and not an `int?`.

2. Use the null aware operator (`?.`) that safely reads the right-hand side only if it’s not `null`. Consider this example, where `pi()` has a nullable return type but actually never returns `null`:

```
double? pi() => 3.1415;

void main() {
    final value = pi();
    print(value?.round()); // prints '3'
}
```

If we used `value.round()`, we would have got a compiler error because we tried to read a nullable type without any check. However, the `value?.round` syntax automatically makes a nullability check and calls `round()` only if `value` is not `null`. Here’s another case:

```

double? pi() => null;

void main() {
    final value = pi();
    print(value?.round()); // prints 'null'
}

```

This example is valid too because the `?.` operator notices that `value` is `null` and thus it does not call `round()`. Instead, it just returns `null`. Using the null aware operator is the same as doing this:

```

if (value != null) {
    print(value.round());
} else {
    print(null);
}

```

Instead of doing this kind of `if` statement, prefer using `?.` which is an equivalent version but less verbose.

3. Use the bang (`!`) operator to convert a nullable type into a non-nullable one at runtime. Note that this operator throws a runtime exception if the conversion failed so make sure to use it judiciously. In our case, it would work as follows:

```

double? pi() => 3.1415;

void main() {
    final value = pi();
    print(value!.round()); // prints '3'
}

```

When the compiler sees `!.` it treats the variable as if it was non-nullable. If you used `!` on a `null` value, a runtime exception is thrown.

4. Use the null-check operator to give a default value before the variable is used. This is a good strategy to give default values to variables in case they were `null`. For example:

```

double? pi() => null;

void main() {
    final value = pi();
    print((value ?? 3.14).round()); // prints '3'
}

```

The `a ?? b` syntax returns `a` if it's not `null` and `b` otherwise. In our case, since `value` is `null`, the `??` operator returns the “default” (or “fallback”) value on the right-hand side.

It's always safe to pass a non-nullable type to a nullable type. For example, `int?` can either hold `int` or `null`. The type system is designed so that every nullable type is a supertype of `Null` and its underlying type. For example:

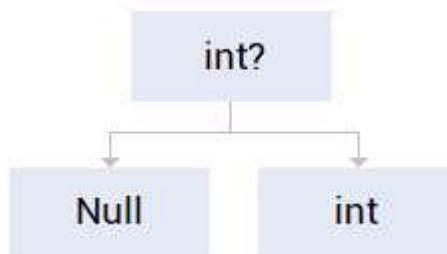


Figure 2.3: How nullable types in Dart are treated.

Since `int?` is a supertype, you can assign it either `Null` or `int`. Going in the other direction, which is giving an `int` and `int?` value, is not possible because of the relationship in the hierarchy.

In a larger view of the Dart type system, you can identify two worlds:

- one with non-nullable variables (`Object, int, bool...`);
- one with their nullable counterparts (`Object?, int?, bool?...`);

Non-nullable values let you access all of their members without restrictions. On the other side, for safety reasons, you can't do much with nullable themselves. You have to manually check their `null` status before safely accessing the underlying non-nullable type without exceptions.

## Top and bottom types

We have previously seen in *Figure 2.2* that the Dart team moved `Null` outside the `Object` hierarchy to make the whole type system “null safe”. However, `Null` is still associated to the type hierarchy otherwise you wouldn't be able to use it. A **top type** is the “highest” supertype in the hierarchy while a **bottom type** is the “lowest” subtype in the hierarchy.

It is beneficial for a type system to have top and bottoms type because they constrain the hierarchy to a closed set where all entities are somehow linked together. As a consequence, it is possible to implement an intelligent type inference system. Here is the simplified diagram of the null safe type system hierarchy:

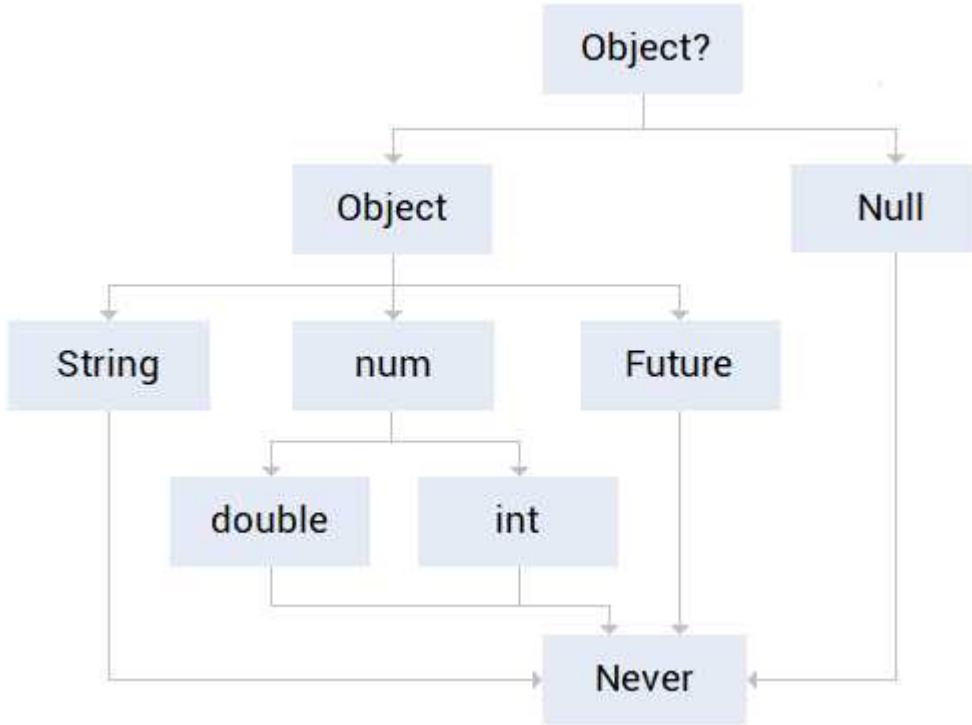


Figure 2.4: A simplified diagram of Dart’s type hierarchy with null safety.

With null safety (Figure 2.4), `Object?` is the top type, meaning that anything in Dart is an `Object?`. The bottom type is called `Never`. Notice that `Null` isn’t an `Object` subtype anymore so there is no way by design that, for example, a `bool` or an `int` could be `null`. A few more things:

- If you want to indicate that you want to allow a value of any type, use `Object?` instead of `Object`. Let’s see an example:

```

Object? value1 = null; // OK
Object? value2 = 5; // OK

Object value3 = null; // Compiler error
Object value4 = 5; // OK

```

As you can see, `Object?` could be an `int` or `null` but `Object` can be anything except for `null`. In other words, `Object?` means “anything” while `Object` means “anything that is non-nullable”.

- In the rare case that you need a bottom type, use `Never`. It is a particular type, like `void` or `dynamic`. The main reason why `Never` was created is to give a bottom type to the Dart type

system. In *chapter 3 – Section 4 “Functions”* we will see a possible use-case for `Never` but aside from that, there aren’t many other relevant cases where you really need it. It’s handy in the type system, for static analysis tools and other compiler-related tasks. For the developer, it’s not much use in practice.

Here is a final diagram that summarizes the main difference between the old (non-null safe) and the new (null safe) type system in Dart:

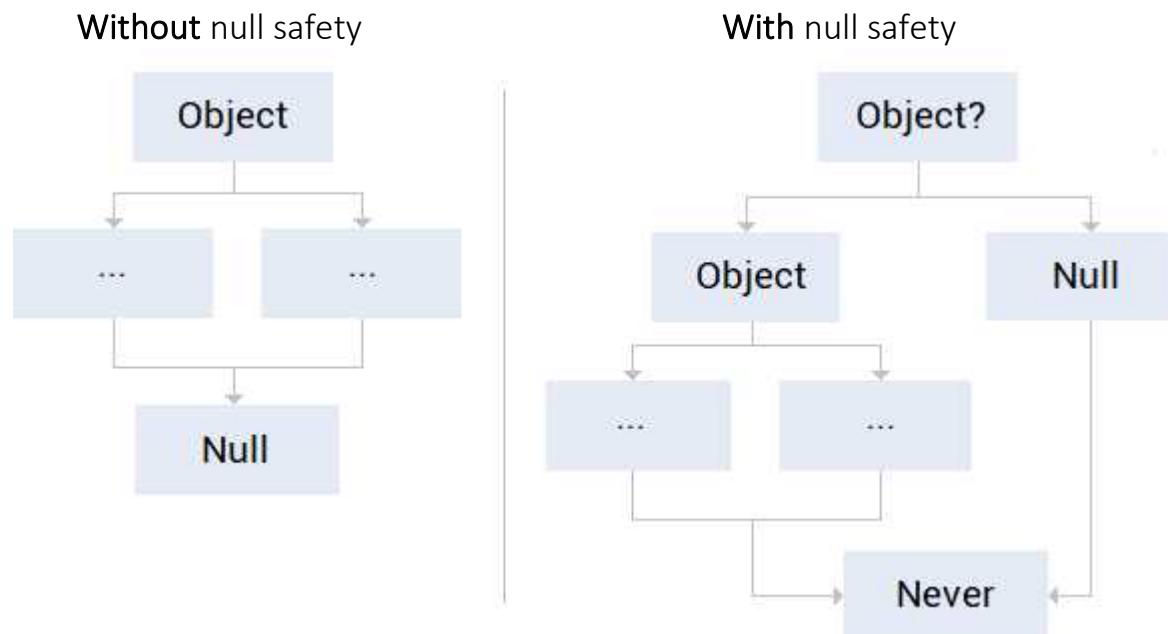


Figure 2.5: a comparison between the old (on the left) and the new (on the right) type system.

The difference is even more visible in *Figure 2.5*. In the null safe type system, `Null` stays outside of the `Object` hierarchy so it’s impossible by design that `null` is assigned to an `Object` subtype.

## Deep dive: Language specification details

This section contains some interesting language features that are covered in the Dart specification document<sup>17</sup>. The contents of the document are very rigorous and formal but here we’ve extracted and rephrased some curiosities about variables and type assignments:

---

<sup>17</sup> <https://dart.dev/guides/language/spec>

- You can initialize one or more variables in the same declaration site. In this case, they can either hold a value or not. For example:

```
void main() {
    var x, y = 1;

    print(x.runtimeType); // Null
    print(y.runtimeType); // int
}
```

Since `x` isn't initialized with a value, the compiler initializes it as `Null`. It is a compile-time error if you use `final` or `const` to declare a variable without initializing it. For example:

```
void main() {
    final a = 10, b = true;
    final x, y = 1;

    print(a.runtimeType); // int
    print(b.runtimeType); // bool

    print(x.runtimeType); // compiler-error
}
```

The outcome would have been the same if we had used `const` instead of `final`.

- In the same declaration site, you can initialize two or more variables with a defined type without using the compiler's inference. For example:

```
double x = 1.34, y = 10;
```

- The runtime type of every object is represented as an instance of the `Type` class, which can be obtained using the `runtimeType` method. For example:

```
void main() {
    Type type = true.runtimeType;

    // same as 'print(type.toString())';
    print(type); // bool
}
```

The `Type` class only defines `operator==`, `hashCode`, and `toString()` but it doesn't have any particular purpose. It's only used to represent the type of an object and converts the type name into a `String` object.

- You can use `Type` in regular expressions, like `final Type boolType = bool`.
- A variable declaration can include the `covariant` modifier, which we will describe later in *chapter 5 – Section 1.7 “The covariant keyword”*.

# 3 – Control flow and functions

---

## 3.1 If statement

This is probably the most famous statement of any programming language, and it exactly works as you would expect in Dart. It evaluates a boolean value or expression and splits the execution flow two or more branches using the `if` and the (optional) `else` keywords:

```
void main() {
  final value = int.fromEnvironment('something');

  // Using both branches
  if (value >= 0 && value != null) {
    print('Good');
  } else {
    print('Bad');
  }

  // Using a single branch, without 'else'
  if (value == 0) {
    print('Zero!');
  }
}
```

While in C++ the `if (0)` construct is valid because `0` evaluates to `false`, in Dart this is not valid because only boolean values are allowed. You could also avoid using curly braces but, as the official Dart guidelines suggest<sup>18</sup>, we recommend to not do it. You can chain `if` statements as much as you want with this syntax:

```
if (value == 'Dart') {
  doSomethingWithDart();
} else if (value == 'Java') {
  doSomethingWithJava();
} else if (value == 'Go') {
  doSomethingWithGo();
} else {
  print('Error!');
}
```

---

<sup>18</sup> [https://dart-lang.github.io/linter/lints/curly\\_braces\\_in\\_flow\\_control\\_structures.html](https://dart-lang.github.io/linter/lints/curly_braces_in_flow_control_structures.html)

Conditional expressions are used to select one of two values based on a boolean condition. They let you concisely evaluate expressions that might otherwise require `if-else` statements. In Dart, there are two operators for conditional expressions:

- `valueA ?? valueB`: evaluates the first value (or expression) and returns the second one if it resolves to `null`. To better understand when the `??` operator can be helpful, look at this example:

```
// 'tryParse' returns 'null' if the conversion fails
final int? value = int.tryParse(number);

int year;
if (value != null) {
    year = value;
} else {
    year = 0;
}
```

In the example, we want to convert a `String` into an `int` and use `0` as the default value. An equivalent version, but less verbose, can be written using the `??` operator:

```
final int? value = int.tryParse(number);
final year = value ?? 0;
```

The `??` operator provides a default value when the left-hand operator is `null`. Note that in this example, `year` is `0` only if `value` is `null`. Furthermore, `year` does not need to be nullable because it's guaranteed to either be the parsed value or zero.

- `condition ? valueA : valueB`: this expression (also known as *ternary expression*) is just syntactic sugar that replaces the `if` with a question mark and the `else` with the colon. Prefer using this with one-liner statements to preserve readability:

```
// This is verbose
String sign1;
if (someNumber >= 0) {
    sign1 = '+';
} else {
    sign1 = '-';
}

// This is the same, but shorter
final sign2 = someNumber >= 0 ? '+' : '-';
```

Ternary operators can be used in sequence but the longer the chain, the lower the code readability. For example::

```
void main() {
    final id = getId();
    final isAdmin = getAdminStatus();

    final canAccess = isAdmin ? true : id < 0 ? true : false;
}
```

At first glance, you can't tell how `canAccess` is being built because nested ternary operators aren't very expressive. In these cases, you should prefer the regular `if-else` statements

The `if` statement is also useful to create a “safe scope” for nullable variables. When a null-check succeeds, for example, the compiler understands it gives for granted that the variable can never be `null` within the branch scope. For example:

```
void doSomething(int value) {
    print(value);
}

void main() {
    final int? value = int.tryParse('');

    if (value != null) {
        doSomething(value);
    }
}
```

Even if `value` has a nullable integer type, inside the scope of the `if` statement it cannot be `null` and the compiler understands it. As such, the above example compiles because `value` is treated as an `int` inside the if statement scope. This code instead raises a compiler error:

```
void doSomething(int value) => print(value);

void main() {
    final int? value = int.tryParse('');
    doSomething(value); // compiler error
}
```

Since `value` is not “protected” by an `if` guard anymore, the compiler cannot assume that the variable will never be `null`.

## 3.2 Switch statement

The `switch` statement compares a value against a series of options and executes the first match in declaration order. It's generally used when choosing from a defined set of possibilities. In particular, a `switch` statement can compare:

- compile-time constants;
- enumerated values;
- numbers;
- strings;
- classes.

The `case` keyword identifies the various options to be matched. At the end of the options list, the optional `default` keyword may be used as a fallback if none of the previous cases matched the item being compared. For example:

```
enum Status {  
    ready,  
    paused,  
    finished,  
}  
  
void main() {  
    const status = Status.paused;  
  
    switch (status) {  
        case Status.ready:  
            print('ready');  
        case Status.paused:  
            print('paused');  
        case Status.finished:  
            print('finished');  
        default:  
            print('undefined');  
    }  
}
```

You could see the `switch` statement as a more readable version of an `if` when it has too many `else` branches to evaluate. In the example, the `switch` first evaluates `status` and then looks for the first `case` clause that matches. Here are a few more examples:

- This code is equivalent to the previous one because the `break` statement at the end of each `case` block is optional:

```

switch (status) {
  case Status.ready:
    print('ready');
    break; // 'break' allowed but optional
  case Status.paused:
    print('paused');
    break; // 'break' allowed but optional
  case Status.finished:
    print('finished');
    break; // 'break' allowed but optional
}

```

We don't recommend using `break` when not needed. It was required in Dart 2.19 and lower versions but from 3.0 onwards, it is optional.

- When a `case` doesn't have a body, it's associated with the next `case`:

```

switch (status) {
  case Status.paused:
    print('paused');
  case Status.ready:
  case Status.finished:
    print('finished');
}

```

In this example, the `ready` and `finished` statuses will print "finished". This is an equivalent, but more verbose, version of the same code:

```

switch (status) {
  case Status.paused:
    print('paused');
    break;
  case Status.ready:
    print('finished');
    break;
  case Status.finished:
    print('finished');
}

```

- There is a case where the `break` statement is needed. For example, consider this code where all `case` blocks continue to fall through the next one:

```

case Status.paused:
case Status.ready:
case Status.finished:
    print('finished');

```

In this example, any value of `status` will always print “finished” because the first two `case` statements have no body. If you added a `break` (and thus a body to a case statement), the flow wouldn’t fall through anymore. For example:

```

case Status.paused:
    break; // the 'switch' exits here
case Status.ready:
case Status.finished:
    print('finished');

```

Because of the `break`, the `paused` status will not print “finished” anymore.

- The `default` case must always be the last statement otherwise a compiler error is thrown. For example, this code will not compile:

```

switch (status) {
    default: // compiler error: 'default' not allowed here
        print('none');
    case Status.paused:
        print('paused');
}

```

As a guideline, we recommend to use `switch` statements when your program flow can split into more than two branches. The `if` statement should preferably be used to distinguish between two options, not multiple ones. For example:

Good	Not so good
<pre> switch (exitCode) {     case Status.paused:         return 2;     case Status.ready:         return 1;     case Status.finished:         return 0;     default:         return -1; } </pre>	<pre> if (exitCode == Status.paused) {     return 2; } else if (exitCode == Status.ready) {     return 1; } else if (exitCode == Status.finished) {     return 0; } else {     return -1; } </pre>

Note that it's just a matter of readability. Both versions are equivalent, but we think that a `switch` is more readable in that example. In *chapter 7 – Section 2 “Patterns”* you will see more advanced ways to use the `switch` statement.

### 3.3 For and while loops

The syntax of the `for` and `while` loops is so traditional that if you didn't know this book is about Dart, you couldn't guess the language. Curly braces are optional but again, we recommend always to use them for readability:

```
// Good and readable
for(var i = 0; i < 10; ++i) {
    print('Index: $i');
}

// Good but potentially not very readable
for(var i = 0; i < 10; ++i)
    print('Index: $i');
```

We can get the same result using a `while` loop, which can be used in two flavors. The `do-while` variant always executes at least one iteration because the condition is evaluated at the end:

```
var i = 0;

// Traditional 'while' Loop
while (i <= 10) {
    print('Index: $i');
    ++i;
}

i = 0;

// 'do-while' Loop
do {
    print('Index: $i');
    ++i;
} while (i <= 10);
```

A `while` loop immediately evaluates the condition so it could never execute its body. A `do-while` loop instead evaluates the condition at the end so at least one iteration always happens. You can control the loop flow by breaking or skipping some cycles:

- **break**. It immediately stops and exits the loop. In the case of nested loops, only the one whose scope contains the **break** is stopped. For example:

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

In this case, only loop 2 is terminated when `j == 2` but the “outer” one (loop 1) executes normally. In other words, **break** only “breaks” a single loop.

- **continue**. It skips to the next iteration and, in case of nested loops, it behaves in the same way as **break**:

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

Again, only loop 2 skips one iteration but the “outer” one (loop 1) is regularly executed.

Of course, **break** and **continue** can also be used with **while** and **do-while** loops. In general, **for** loops are used when you need to increment an index and **while** loops are used when a boolean condition has to be evaluated constantly.

In some cases, you may want to completely traverse a container without updating an index. For example:

```
const friends = ['A', 'B', 'C'];
for(var i = 0; i < friends.length; ++i) {
  print(friends[i]);
}
```

The `i` variable is only used to access the `i`-th element of the list. In these cases, where you only need to iterate over a container without needing an index, prefer using **for-in** loops:

```
for(final friend in friends) {
  print(friend);
}
```

This version is just less verbose and probably easier to read. For each iteration, `friend` contains the next value in the `friends` list. In *chapter 6 – Section 3.2 “Iterable<T> and iterators”* we will understand how the for-in loop works under the hood and how it can be used in custom classes.

## 3.4 Functions

Functions in Dart are defined with the return type, the name, an (optional) list of parameters and a body. More formally, a function consists of a signature and a body with a scope. Here's an example:

```
bool hasRemainder(int dividend, int divisor) { // function signature
    return dividend % divisor > 0; // function body
}
```

When the function body contains a single line, you can use the arrow syntax (which only works with expressions, not with statements). In the following example, there is a compiler error in the second function because the arrow syntax is used with a statement and not with an expression:

```
// OK - the arrow '=>' replaces the 'return' statement
bool hasRemainder(int dividend, int divisor) => dividend % divisor > 0;

// ERROR - cannot use statements with the arrow syntax
bool hasRemainders(int dividend, int divisor) => if (...) ...;
```

Because of its type-safe nature, in Dart any function must always have a return type. If you don't declare it, the compiler automatically assigns `dynamic`:

```
// GOOD - Use 'void' when no return values are needed
void helloWorld() {
    print('Hello world!');
}

// BAD - The return type here is 'dynamic' because we haven't provided one
helloWorld() {
    print('Hello world!');
}
```

We recommend always to specify the return type and to use `void` when you don't need to return a value. Interestingly, you can also use the arrow syntax in one-liner `void` functions too, but the returned value (if any) is ignored:

```
void helloWorld() => print('Hello world!'); // Hello world!

// Since the return type is 'void', the string is not returned
void returnHelloWorld() => 'Hello world!';
```

As you can see, the second example looks a bit weird so it'd be better to use the arrow syntax in `void` functions that call other `void` functions (as in the first example). In *chapter 2 – “Top and bottom types”* we introduced the particular `Never` type saying that it's rarely used. One of the few use-cases for that type is when a function does not return normally. Here is how you should work with functions and return types:

- When a function has to return a value to the caller, use the `return` keyword. For example:

```
String snakeCase(String value) {  
    if (value.contains(' ')) {  
        return value.toLowerCase().replaceAll(' ', '_');  
    }  
    return value;  
}
```

- When a function does not have a value to return to the caller, use the `void` return type. For example:

```
void snakeCase(String value) {  
    if (value.contains(' ')) {  
        print(value.toLowerCase().replaceAll(' ', '_'));  
    } else {  
        print(value);  
    }  
}
```

Note that `return` can also be used with `void` but without a value. This is generally used to “exit” the function before the program flow arrives at the end. For example, we could have re-written the previous example in this equivalent form:

```
void snakeCase(String value) {  
    if (value.contains(' ')) {  
        print(value.toLowerCase().replaceAll(' ', '_'));  
        return;  
    }  
    print(value);  
}
```

The `return` is used to exit the function earlier to avoid executing the other instructions. It is a compile-time error trying to return a value from a `void` function.

- When a function never normally returns, use the `Never` return type. For example:

```
Never testFunction(String value) {  
    throw FormatException(value);  
}
```

The `return` keyword cannot be used when the return type is `Never`. Since `testfunction` always throws an exception, it never returns a value and it never entirely executes until the end. You could have written an equivalent version using `void` instead:

```
void testFunction(String value) {  
    throw FormatException(value);  
}
```

There is no practical difference between the two versions. From a formal point of view, the `Never` version would logically be a bit more correct because `void` implies that the function always successfully terminates. This is a sort of edge case so feel free to not give it too much weight.

Always return either a type or `void`. The official Dart documentation says that `Never` is “most often used for functions that always throw an exception”<sup>19</sup>. We recommend avoiding `Never`, unless you know it is needed.

### Note

If you’re a C or C++ developer, you may be wondering if Dart function parameters can be passed by reference, by value or which is the default “passing” strategy. The answer is: “none”! Since the language does not allow you to work with pointers or references to variables, the question doesn’t make sense.

Function parameters could have the `final` modifier, but you should avoid using it because it can be either dangerous or useless:

```
// 1. Both 'first' and 'second' have the 'dynamic' type  
bool areEqual1(final first, final second) => first == second;  
  
// 2. OK... but 'final' does nothing useful here  
bool areEqual2(final int first, final int second) => first == second;
```

---

<sup>19</sup> <https://dart.dev/guides/language/language-tour#built-in-types>

In the `areEqual1` function, the compiler has no clue about the parameters' types so it just assigns `dynamic` to defer the evaluation at runtime. As you may have guessed, this is not type safe and also isn't the recommended way of declaring function parameters<sup>20</sup>. The `areEqual2` function is fine but the `final` does nothing so you can remove it. Functions can always have a comma at the end of the parameter list (this is mostly for style reasons when using the `dart format` tool):

```
bool areEqual1(int first, int second,) => first == second;  
//                                     ^ this is ok!
```

In *chapter 5 – Section 2.2 “Records”* we will see that records are often used to return multiple values from a function. In *chapter 8 – “Futures, Streams and Isolates”* you will learn that a function could also have the `async` modifier in its signature.

### 3.4.1 The Function type

In Dart everything is an object, and functions are no exception. The `Function` class is the supertype of all kinds of functions. Focus on the `someFunction` variable of this example:

```
bool hasRemainder(int dividend, int divisor) => dividend % divisor > 0;  
bool areEqual(int first, int second) => first == second;  
  
void main() {  
    // The 'someFunction' variable has the 'bool Function(int, int)' type  
    bool Function(int, int) someFunction;  
  
    // 'someFunction' is first used to get the remainder  
    someFunction = hasRemainder;  
    print(someFunction(10, 5)); // false  
  
    // 'someFunction' then used to check for numbers equality  
    someFunction = areEqual;  
    print(areEqual(5, 5)); // true  
}
```

It's nothing different from the usual: we are declaring a type (`bool Function(int, int)`) and its name (`someFunction`). The syntax is particular because the complete function signature is required but it is nothing new. For example, compare these declaration:

- `int year = 2022`: The type is `int` and the value is `2022`;

---

<sup>20</sup> [https://dart.dev/tools/linter-rules#type\\_annotation\\_public\\_apis](https://dart.dev/tools/linter-rules#type_annotation_public_apis)

- `String name = 'Alberto'.` The type is `String` and the value is '`Alberto`';
- `bool Function(int, int) fn = areEqual.` The type is `bool Function(int, int)` and the value is a function with a matching signature.

When you write a function, Dart automatically converts it into a `Function` object. You can check this by calling `runtimeType` on a function and see that it returns the signature. For example:

```
void printName(String name) {
  name.toUpperCase();
}

void main() {
  print(printName.runtimeType); // prints '(String) => void'
}
```

We always recommend defining the function's return type because, if absent, the compiler assigns `dynamic` by default. Prefer using `void` rather than omitting the return value. For example:

```
void printValue1(int? value) => print(value); // OK
printValue2(int? value) => print(value); // Bad

void main() {
  void Function(int?) good = printValue1;
  Function(int?) bad = printValue2;

  print(good.runtimeType); // Prints '(int?) => bool'
  print(bad.runtimeType); // Prints '(int?) => dynamic'
}
```

In *chapter 4 – Section 4.1 “The call() method”* we will see how classes can behave like functions.

### 3.4.2 Anonymous functions

We have only seen named functions so far because each `Function` object always had a name. Dart also allows you to create nameless functions, also known as anonymous functions or “closures”. An anonymous function looks similar to a named function: zero or more parameters, separated by commas, and optional type annotations between parentheses. For example:

```
// this is an anonymous function
final areEqual = (int a, int b) => a == b;

print(areEqual(3, 7)); // false
print(areEqual(7, 7)); // true
```

With this syntax, you can create functions “on the fly” that are directly assigned to a variable. An equivalent version of the previous example is the following:

```
final bool Function(int, int) areEqual = (int a, int b) => a == b;

print(areEqual(3, 7)); // false
print(areEqual(7, 7)); // true
```

The arrow syntax is often used for one-liner statements. In case of multiple lines, you could make the code more readable by using curly braces. For example:

```
// This is still an anonymous function but it's defined with curly braces
final namePrinter = (int version) {
    final name = 'Flutter Complete Reference ';
    return '$name $version.0';
};

print(namePrinter(2));
```

There are many ways to define an anonymous function. For example, all of these definitions are equivalent:

```
bool Function(int, int) areEqual1 = (int a, int b) => a == b;
bool Function(int, int) areEqual2 = (final a, final b) => a == b;
bool Function(int, int) areEqual3 = (var a, var b) => a == b;
bool Function(int, int) areEqual4 = (a, b) => a == b;
```

We have already seen that using `final` (or `var`) in the parameters list is unnecessary. In some cases, the anonymous function may not be well-defined. For example, consider this case:

```
// This is bad because 'a' and 'b' are dynamic!
final areEqual = (a, b) => a == b;

// Prints '(dynamic, dynamic) => bool'
print(areEqual.runtimeType);
```

The problem is that the compiler does not know the parameters’ types and thus it assigns `dynamic` to both `a` and `b`. To solve this problem (where type safety is lost), you should always make sure to have the types defined by either:

1. Writing down the type explicitly:

```
final bool Function(int, int) areEqual = (a, b) => a == b;
```

2. Writing down the types in the function’s parameters list:

```
final areEqual = (int a, int b) => a == b;
```

The compiler has enough information to infer `int` rather than `dynamic` in both cases. The second version is equivalent to the first one but it's shorter (and generally recommended).

### 3.4.3 Optional parameters

Function parameters can be optional because if you don't give them a value, the compiler assigns a default one. The following subsections describe how optional parameters of functions are handled in Dart.

#### 3.4.3.1 Optional named parameters

"Optional named parameters" are declared inside curly braces and, as the name suggests, they are optional. When an optional parameter is not initialized with a value, the default one is returned. The order in which they are called does not matter. For example:

Declaration	Invocation example
<pre>void test({int a = 0, int b = 0}) {   print('\$a');   print('\$b'); }</pre>	<pre>void main() {   // Prints '2' and '0'   test(a: 2); }</pre>

When a default value of an optional named parameter is not given, the variable must be nullable. In this case, the compiler automatically initializes the variable with `null` if it's not assigned. For example:

Declaration	Invocation example
<pre>void test({int? a, int b?}) {   print('\$a');   print('\$b'); }</pre>	<pre>void main() {   // Prints 'null' and '2'   test(b: 2); }</pre>

Despite the “*optional*” nature, any optional named parameter can be required with the `required` keyword. It forces a parameter to always be defined and does not allow default values. For example:

Declaration	Invocation
<pre>void test({required int a, int? b}) {     print('\$a');     print('\$b'); }</pre>	<pre>void main() {     test(a: 2); // OK     test(a: 2, b: -3); // OK     test(b: -3); // Compiler error }</pre>

The compile-time error happens because `a` is `required` and we have not defined it. When an optional named parameter is required, even if it is `null`, it still has to be explicitly initialized:

Declaration	Invocation
<pre>void test({required int? a, int? b}) {     print('\$a');     print('\$b'); }</pre>	<pre>void main() {     test(a: 2, b: -3); // OK     test(a: null, b: -3); // OK     test(a: 5); // OK      test(b: -3); // Compiler error }</pre>

As you can see, even if `a` is nullable, you must always initialize it. The main reason why you should use optional named parameter is readability. Look at this example:

```
void computeValue({  
    required int value,  
    required void Function(double) onSuccess,  
    void Function(String)? onError,  
}) {  
    final result = doSomething(value);  
  
    if (result >= 0) {  
        onSuccess(result);  
    } else if (onError != null) {  
        onError('Error code: $result');  
    }  
}
```

This function performs some calculations and then uses two callbacks (`onSuccess` and `onError`) to report the results. Thanks to named parameters, it's clear which callback is triggered in case of success and which one in case of error:

```
void main() {
    computeValue(
        value: 10,
        onSuccess: (double value) => print('Result: $value'),
        onError: (String errorMsg) => print(errorMsg),
    );
}
```

This also is an example of why anonymous function can be handy. Instead of defining a separated function, we can create it “on the fly” and improve the code readability. In this case, the `onError` callback can be torn off to simplify the expression even more:

```
void main() {
    computeValue(
        value: 10,
        onSuccess: (double value) => print('Result: $value'),
        onError: print, // This is a 'tear-off'
    );
}
```

A tear-off closure takes the same parameters and return type as the function. In our example, the signature of `onError` matches the signature of `print` because they both return `void` and accept a `String` as parameter. Here's another example:

```
void successCallback(double value) => print('Result: $value');

void main() {
    computeValue(
        value: 10,
        onSuccess: successCallback,
        onError: print,
    );
}
```

Since `successCallback` has the same signature as `onSuccess`, we have used a “tear-off” to assign the function directly to the parameter. This is just syntactic sugar because you could have indeed attached an anonymous function:

```
// It works, but it's more verbose than a tear-off
onSuccess: (double value) => successCallback(value),
```

The official documentation<sup>21</sup> recommends to prefer function tear-offs when possible because they make the code more concise. You can mix “regular” unnamed and optional parameters together, but optional ones always go last in the declaration list:

```
// OK - Unnamed first, named optional last
void function1(int a, {int? b, required int c}) => print('$a $b $c');

// ERROR - Named must be Last
void function2({int? b, required int c}, int a) => print('$a $b $c');
```

In Dart 2.7 and lower versions, the `required` keyword didn't exist.

### 3.4.3.2 Optional positional parameters

Optional positional parameters are defined inside square braces and are optional. They share most of named parameters' properties. When an optional positional parameter is not initialized with a value, the default one is returned:

Declaration	Invocation
<pre>void test([int a = 0, int b = 0]) {   print('\$a');   print('\$b'); }</pre>	<pre>void main() {   // Prints '2' and '0'   test(2); }</pre>

The main difference from named parameters is that the order matters. Consider the `test` function of the above example. Since parameters' values are no longer assigned by name, you can only initialize them by giving each one (in order) a value. For example, there is no way to only initialize `b` without giving `a` a value:

```
test(4); // 'a' = 4, 'b' = 0
test(0, 4); // 'a' = 0, 'b' = 4
```

There is no way to only assign `b` because `a` is declared first and thus the initialization order does matter. When a default value is not given, the variable must be nullable:

---

<sup>21</sup> [https://dart.dev/tools/linter-rules#unnecessary\\_lambdas](https://dart.dev/tools/linter-rules#unnecessary_lambdas)

Declaration	Invocation
<pre>void test([int? a, int b?]) {     print('\$a');     print('\$b'); }</pre>	<pre>void main() {     // Prints 'null' and '2'     test(null, 2); }</pre>

There is no `required` keyword for optional positional parameters. Positional parameters can be mixed with regular ones but (as it happens with named ones) they must be defined at the end:

```
// OK - Unnamed first, positioned last
void function1(int a, [int? b, int c = 0]) {
    print('$a $b $c');
}

// ERROR - Positioned must go last
void function2([int? b, int c = 0], int a) {
    print('$a $b $c');
}
```

Optional named parameters are more commonly used than positional ones (especially in Flutter) because the order doesn't matter.

### 3.4.4 Nested functions

The language allows you to declare functions inside other functions that are visible only in the scope in which they're declared. In short, nested functions can only be used by the function that contains it. For example:

```
void main() {
    int nestedFn1(int value) { // Nested function
        int nestedFn2() => 2022 + value; // Another nested function
        return value * nestedFn2();
    }

    final test1 = nestedFn1(10); // OK
    final test2 = nestedFn2(); // Compiler error
}
```

The `nestedFn2` function can only be accessed within the scope in which it's declared. As such, only `nestedFn1` can use it. The `main` function cannot see `nestedFn2`. In the other direction, a nested function can access all enclosing variables. For example:

```

void main() {
  const a = true;

  void nestedFunction() {
    const b = true;

    void moreNestedFunction() { // can use variables from enclosing scopes
      const c = true;

      print('$a');
      print('$b');
      print('$c');
    }
  }

  print(b); // Error because 'b' is not in scope
}

```

The `moreNestedFunction` function can access its members (`c`) and the ones from the enclosing functions scopes (`b` and `a`). The last `print` call raises a compiler error because `b` is not in the `main` scope, so the variable cannot be referenced.

### 3.4.5 Assertions

Assertions are used to throw exceptions<sup>22</sup> while developing an application. An assertion throws an exception whenever its condition evaluates to `false`. For example:

```

// This method returns a json-encoded string
final String json =getJSON();

// The exception is thrown if the string is empty
assert(json.isEmpty, "String cannot be empty");

// This is only executed if the assertion doesn't throw
parseJson(json);

```

The `assert` statement takes a `bool` value and an error message, which will be logged in the console whenever the exception is thrown. It basically is a `void` function that throws an exception and logs the error to the console if the condition evaluates to `true`. Assertions only work in specific cases:

1. In Flutter, assertions only work in debug mode (which is the default mode);

<sup>22</sup> Exceptions will be discussed in detail in chapter 5 – Inheritance, core classes and exceptions.

2. The `dart run` command evaluates exceptions only with the `--enable-asserts` flag;
3. Some specific, development-only Dart tools enable them by default.

Note that assertions are ignored when the application is built in release mode.

### 3.4.6 Good practices

With respect to the official Dart guidelines, we strongly encourage you to follow these suggestions to keep consistency with what the community recognizes as good practices:

- Whenever possible, prefer using tear-offs rather than anonymous functions <sup>23</sup>. A **tear-off** is a closure that takes the same parameters as the function and invokes the underlying function when you call it.

```
// Good  
[1, 2, 3].forEach(print);  
// Bad  
[1, 2, 3].forEach((value) => print(value));
```

- Do not explicitly initialize nullable variables with `null` because the compiler already does it automatically:

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

- Always define the function return type. When the function does not return a value, use `void` rather than leaving the return type unassigned (which evaluates to `dynamic`):

```
// OK - does not return a value and uses 'void'  
void somethingGood() {}  
// Bad - does not return a value but returns 'dynamic'  
somethingBad() {}
```

---

<sup>23</sup> <https://dart.dev/guides/language/effective-dart/usage#dont-create-a-lambda-when-a-tear-off-will-do>

Lastly, just a word on nested functions. From a purely technical point of view, they're allowed by the language and don't have performance concerns. From a qualitative point of view, there might be conflicting points of view. Grouping common logic into functions may make maintenance easier but watch out for the nesting depth. Too much nesting could lead to verbose and hard-to-read code. Dart supports both functional and OOP programming, so there should always be a good balance between the two styles.

## 3.6 Type aliases

Also known as `typedefs`, type aliases give another name to an existing function or type. For example, if we wanted to give a shorter or a more meaningful name to a list of objects, we could use the `typedef` keyword to create a reusable alias:

```
typedef StringList = List<String>;  
  
void main() {  
    final List<String> noAlias = ['a', 'b', 'c'];  
    final StringList withAlias = ['a', 'b', 'c'];  
  
    print(noAlias.runtimeType); // List<String>  
    print(withAlias.runtimeType); // List<String>  
}
```

The new `StringList` type is just “*another name*” (or an alias) of `List<String>` and in fact, both `runtimeType` calls return the same type. You can alias an existing alias:

```
typedef A = int;  
typedef B = A;
```

Aliases are mainly handy in two cases:

1. When the type name is too long. In *chapter 6 – “Generics and collections”* we will see that we can combine generics to create containers for any type. In some cases, the syntax may become very verbose. For example:

```
void analyze(Map<int, List<Map<String, double>>> data) {  
    print('Data analysis');  
}  
  
void main() {  
    final Map<int, List<Map<String, double>>> data = getData();  
    analyze(data);  
}
```

We have repeated a lot of types there. In such cases, you can use a `typedef` to make the code more readable with a shorter name:

```
// This is an alias of the Map<...> type
typedef ServerData = Map<int, List<Map<String, double>>>;

void analyze(ServerData data) {
  print('Data analysis');
}

void main() {
  final ServerData data = getData(); // much more readable!
  analyze(data);
}
```

The `ServerData` alias is undoubtedly more meaningful and less verbose than the complete definition of the underlying types.

2. When creating function callback aliases. The Flutter framework often uses aliases to define reusable types for callback functions. For example:

```
typedef SuccessCallback = void Function();
typedef ErrorCallback = void Function(String);

void processData({
  required SuccessCallback onSuccess,
  required ErrorCallback onError,
}) {
  // code...
}
```

The aliases could be moved to a separate file and be re-used in other parts of the code where a success or failure callback is required.

As a good practice, we recommend to always capitalize the first letter of the `typedef` name and to use camel case for composite words<sup>24</sup>. For example:

```
typedef IntegerList = List<int>; // Good
```

```
typedef integerList = List<int>; // Bad
```

---

<sup>24</sup> <https://dart.dev/guides/language/effective-dart/style#do-name-types-using-uppercasecamelcase>

## Deep dive: Language specification details

This section contains some interesting language features that are covered in the Dart specification document<sup>25</sup>. The contents of the document are very rigorous and formal but here we've extracted and rephrased some curiosities about statements and functions:

- When a for loop doesn't define a terminating condition, it just loops infinitely. For example, this is valid Dart code:

```
for (var i = 0;; ++i) {
    print(i);
}
```

The above example produces an infinite loop because Dart converts the code into this:

```
for (var i = 0; true; ++i) {
    print(i);
}
```

It is a compile-time error if the static type of the terminating condition cannot be assigned to a boolean value.

- The `switch` statement can use labels to “jump” from a case to another, altering the regular program flow. A label is an identifier followed by a colon. For example:

```
void main() {
    const value = 1;

    switch (value) {
        world: // this is a 'label'
        case 0:
            print('world!');
            break;
        case 1:
            print('hello ');
            continue world; // 'continue' jumps to the 'world' label
        default:
            print('no message');
    }
}
```

---

<sup>25</sup> <https://dart.dev/guides/language/spec>

The `switch` normally executes all cases, in order, from top to bottom. However, when `continue world;` is evaluated, the program flow jumps back to `case 0` (where the `world` label is). We could make the code even more complicated:

```
switch (value) {  
    world:  
        case 0:  
            print('world!');  
            continue defaultLabel;  
        case 1:  
            print('hello ');  
            continue world;  
        defaultLabel:  
        default:  
            print('no message');  
}
```

This small example is hard to understand because the regular top-to-bottom flow is altered multiple times. In a `switch` statement, `break` cannot have labels.

- Labels can be used to change the regular flow of a `for` or a `while` loop and jump outside in an outer loop (if any). For example:

```
outer:  
for (var i = 0; i < 5; ++i) {  
    print('outer $i');  
  
    inner:  
    for (var j = 0; j < 5; ++j) {  
        if (j == 3) {  
            break inner; // stops and moves to the outer loop  
        }  
        if (j == 4) {  
            continue outer; // stops and moves to the outer loop  
        }  
        print('inner: $j');  
    }  
}
```

Both `break` and `continue` are allowed to jump to the same label.

- The official language specification document recommends to NEVER use labels in `switch` statements. Here is the quote:

## Statement:

*"Labels should be avoided by programmers at all costs. The motivation for including labels in the language is primarily to make Dart a better target for code generation."*

They alter the regular `switch` flow and make the code hard to understand and maintain. There always is a way to rewrite your code so that it doesn't need labels.

- An external function is a function whose body is provided separately from its declaration. It uses the `external` keyword and doesn't have a body. For example, the `toString()` method of the `Object` class is defined as follows:

```
external String toString();
```

This syntax indicates that the body of the `toString` function is located somewhere in a C++ file of the Dart SDK. The `external` keyword can also be used for getters, setters and non-redirecting constructors. Unless you're working on the Dart SDK, you'll never use `external`.

- Other examples of external functions might be foreign functions (defined in JavaScript or C for example), primitives of the implementation (as defined by the Dart run-time system), or code that was dynamically generated but whose interface is statically known.
- Legacy Dart programs running on the standalone Dart VM could call C or C++ functions in a shared library using native extensions. For example:

```
import 'dart-ext:my_library_name';
int systemRandomValue() native "SystemRandom";
```

You have to use `import` to reference a shared library (using the `dart-ext` prefix) and then use the `native` keyword to specify the function name (in the C or C++ code). Instead of native extensions, modern Dart code uses FFI (Foreign Function Interface) which is covered in *chapter 9 – Section 1.4 “Native interoperability with FFI”*.

# 4 – Classes

---

## 4.1 Introduction

Dart is an object-oriented programming language with classes, inheritance, and mixins. Every object is an instance of a class and all types (except for `Null`) descend from `Object`. The syntax is similar to the most popular languages. For example:

```
// This class implicitly is an 'Object' subtype
class Person {
  final String name;
  final String surname;
  const Person(this.name, this.surname);

  @override // this is an annotation
  String toString() => '$name $surname';

  String upperCaseName() => name.toUpperCase();
}

void main() {
  const myself = Person('Alberto', 'Miola');

  print('$myself'); // 'Alberto Miola'
}
```

We have created the `Person` class, given it a `const` constructor because all of its fields are `final` and overridden the `toString` method. The example only wants to give a first glance at how classes are defined. We will analyze everything in detail in the following sections. Dart implements classes with the following properties:

- there is NO multiple inheritance. A class cannot inherit features from more than one parent class;
- there are NO access modifiers like `public`, `protected`, or `private`. All members can either be “public” (which means being visible from outside the class) or “private” (which means being only visible within the class scope).

In Dart you can override a method (re-define its implementation in a sub-class), but there is method overload. As such, there cannot be two functions with different signatures and the same name. For example:

```

// OK
class Example {
    void test1(int a) {}
    void test2(int a, int b) {}
}

// Compiler error
class Example {
    void test(int a) {}
    void test(int a, int b) {}
}

```

The second example causes a compiler error because two methods have the same name. To solve the problem, you must use a different method name to avoid ambiguity. Before moving on, let's see what *cascades* are with an example:

```

class Example {
    void one() => print('one');
    void two() => print('two');
    void three() => print('three');
    void four() => print('four');
}

void main() {
    final example = Example();

    example.one();
    example.two();
    example.three();
    example.four();
}

```

We're calling three methods, in sequence, on the same object. The example can be rewritten in an equivalent form using the cascade notation:

```

void main() {
    final example = Example();

    // The '...' notation is known as cascade
    example
        ..one()
        ..two()
        ..three()
        ..four();
}

```

Cascades, whose usage is officially recommended<sup>26</sup>, allow you to make an ordered sequence of operations on the same object. The null-aware cascade operator has the same functionality, but it works with nullable objects. If the variable is not `null`, all methods in the cascade are executed:

```
void main() {
  final Example? example = Example();

  example
    ?.one() // Notice the '?' in the first line
    ..two()
    ..three()
}
```

Cascades invoke multiple methods, in sequence, but ignore the return value (if any). They should only be used to make assignments or to invoke functions that don't return a value. For example:

```
class Example {
  String name = '';
  int compute() => 45;
  void printAll() => print('$name, $age');

}

void main() {
  final example = Example()
    ..name = 'Alberto'
    ..printAll()
    ..compute(); // the returned value is ignored

  print(example.runtimeType); // 'Example'
}
```

We have mixed variable assignments and method calls in the same cascade. Notice that we have used the cascade right after the object creation. The `Example` object is created, then assigned to the `example` variable, and finally the cascade is executed. Cascades can be nested:

```
final person = PersonBuilder()
  ..name = 'Alberto'
  ..address = (
    AddressBuilder() // the nested cascade MUST be wrapped with parenthesis
      ..city = 'Rome'
      ..street = 'Adua'
  );


---


```

<sup>26</sup> [https://dart-lang.github.io/linter/lints/cascade\\_invocations.html](https://dart-lang.github.io/linter/lints/cascade_invocations.html)

#### 4.1.1 Libraries and visibility

A **library** is a collection of classes, functions, and variables inside a file with the `.dart` extension. A **package** is a group of libraries that are organized and published together for reuse by other projects. To access the public contents of a library, such as classes or functions, use the `import` keyword. For example:

example.dart file	my_app.dart file
<pre>class Example {     final int value;     const Example(this.value);      @override     String toString() =&gt; '\$value'; }</pre>	<pre>import 'example.dart';  void main() {     const example = Example(10);      print('\$example'); // 10 }</pre>

Here we have two different files (`example.dart` and `my_app.dart`), both of which are libraries. Thanks to `import` we can use the `Example` class (from the *example* library) in the *my\_app* library. The *public members* term indicates all members (classes, functions, variables...) that are visible to other files when using `import`. Dart's built-in libraries have a unique prefix:

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

The `dart:` prefix tells the compiler that the given library is located in the Dart SDK. Anything else outside of the SDK, could either use a `package` token or not. For example:

- You can import Dart files within your local project using a relative or an absolute path:

```
// References a file in the same directory  
import 'example.dart';  
  
// References a file in a sub-directory  
import 'sub-directory/example2.dart';  
  
// References a file Located one Level above  
import '../example3.dart';  
  
// Absolute path to a file  
import '/usr/myuser/projects/data/example.dart';
```

- When the Dart file you want to import comes from an external dependency, declared in the `pubspec.yaml` file, you must use the package prefix:

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

The `fraction`<sup>27</sup> package is declared in the `pubspec.yaml` file and it's imported using the package notation. In *chapter 20 – “Creating and maintaining a package”* you will learn what packages are and how they're created.

By default, `import` gives you access to all public members of a library. You can change this behavior with *selective imports*. To understand what they are, consider this simple library that defines three classes:

### my\_library.dart file

---

```
class Example {}
class Demo {}
class Test {}
```

The `show` and `hide` directives are used to decide which library members can be imported by other libraries and which cannot. Look at these examples:

- The `show` directive imports one or more public members and discards all the others that are not declared. For example:

```
import 'my_library.dart' show Example;

void main() {
    // OK, 'Example' is visible because it is listed after 'show'
    final example = Example();

    // ERROR, because 'Test' and 'Demo' are not listed after 'show'
    final test = Test();
    final demo = Demo();
}
```

Since `Test` and `Demo` are not declared by `show`, they are not imported (and can't be used).

---

<sup>27</sup> <https://pub.dev/packages/fraction>

- The `hide` directive imports all public members except the ones you have defined. For example:

```
import 'my_library.dart' hide Example;

void main() {
    // ERROR, because 'Example' is hidden
    final example = Example();

    // OK, 'Test' and 'Demo'
    final test = Test();
    final demo = Demo();
}
```

`Example` is declared by `hide` so it's not imported, but all the others (`Demo` and `Test`) are.

You generally use `show` and `hide` when creating library packages (more on them in *chapter 20 – “Creating and maintaining a package”*) or, less frequently, to solve naming conflicts. You can use a prefix to avoid ambiguity if two libraries have conflicting identifiers (both declare the same class name, for example). Use the `as` keyword to prefix a library:

```
// Contains the 'Demo' class
import 'library1.dart';

// Also contains the 'Demo' class
import 'library2.dart' as second;

void main() {
    // Uses 'Demo' from 'Library1'
    final one = Demo();
    // Uses 'Demo' from 'Library2'
    final one = second.Demo();
}
```

Any public member of `library2` can be accessed with the `second` prefix. This also removes the ambiguity for the `Demo` name, which is declared in both libraries. Library prefixes create something like an “alternative namespace” with the purpose of removing redundant names.

#### 4.1.2 Encapsulation

We have already said that Dart doesn't use the widespread `public`, `protected`, and `private` modifiers. Consequently, we can't make comparisons with other popular languages. There only are two encapsulation rules in the language:

1. All members in a library are *public*, meaning that they can be used by other libraries.
2. Members whose name starts with an underscore (`_`) are *library-private*, meaning that they can only be used within the library in which they're defined.

Let's make a few examples to better understand the encapsulation rules and what *library-private* means in practice. Consider these two different libraries:

example.dart	main.dart
<pre>class Example {   void run() {     print('run');     _Demo().printDemo(); // 'Hi!'   } }  class _Demo {   void printDemo() =&gt; 'Hi!'; }</pre>	<pre>import 'example.dart';  void main() {   // OK, 'Example' is imported   Example().run();    // COMPILER ERROR because   // '_Demo' is not imported   _Demo().printDemo(); }</pre>

Since `Example` doesn't start with an underscore, it's *public* and it can be imported by other libraries. The `_Demo` class instead is a *library-private* type (because its name starts with an underscore) and so it cannot be imported by other libraries. `_Demo` can only be used by members in the same library:

```
class Example {
  void run() {
    print('run');

    // It works because Example and Demo are in the same library
    _Demo().printDemo();
  }
}

class _Demo {
  void printDemo() => 'Hi!';
}
```

The `import` directive does not import members that start with an underscore so `main.dart` cannot use the `_Demo` type. The term is "*library-private*" and not just "*private*" to emphasize the fact that members are private for other libraries but not the one that declares them. The underscore can also be used on functions, variables, and any other identifier:

```
// example.dart file
class Example {
  static const _hello = 'hello';

  void run() {
    print(_hello);
    _printAgain();
  }

  void _printAgain() {
    print('again');
  }
}
```

Outside of the `example.dart` library, `_hello` and `_printAgain()` are not imported because they start with the underscore. The only special case where you can access a library-private member from another library is when you use `part` and `part of`.

#### 4.1.2.1 part and part of

We have just seen that library-private members are only accessible within the same library (which is the same as saying “within the same Dart file”). The `part` directive is used to import both public and library-private members. For example:

example.dart file	my_app.dart file
<pre>part of 'my_app.dart'  class Example {} class _HiddenExample {}</pre>	<pre>part 'example.dart';  void main() {   final ex1 = Example();   final ex2 = _HiddenExample(); }</pre>

The `part of` directive is used to allow the specified library to access its library-private members. The `part` directive is used on the file that needs to use the library-private members of the other library. These two identifiers are a “less strict” version of `import`. In practice:

- `import` is used to import public members;
- `part` (along with `part of`) is used to import public and library-private members.

In general, `part` and `part of` are only used by code generation libraries. They could also be used to split a library into multiple Dart files to better organize the code but this is not recommended by

Dart team<sup>28</sup>. Avoid splitting code using `part` and prefer creating mini-libraries instead, even if they contain a single class or function. When using libraries, prefer `import` over `part`.

## 4.2 Constructors

A constructor is a particular function that must have the same name as the class and is used to create objects. Any Dart constructor is either *generative* or *factory*, with the following difference:

- a generative constructor always creates a new instance of the class it belongs to;
- a factory constructor is a sort of “static function” whose return type can be a type or a subtype of the class. It does not always return a new instance of its class.

On the practical side, factory constructors have the `factory` keyword at the front while generative constructors don’t. Both can be prefixed, where possible, with the `const` keyword and thus become constant generative or factory constructors. The next sub-sections cover a detailed breakdown of all constructor types and variants.

### 4.2.1 Generative constructors

A generative constructor is used to create new instances of a class. It consists of a name, a formal parameters list, and an optional body. Here’s an example of a generative constructor that initializes all of its members with the recommended<sup>29</sup> initializing formal syntax:

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

Both `numerator` and `denominator` are immediately initialized, before the constructor body (which is absent here) executes. To invoke this constructor and thus create a `Fraction` object, you make a normal assignment:

```
final example1 = Fraction(2, 5); // OK  
final example2 = new Fraction(2, 5); // Also OK but 'new' is redundant
```

---

<sup>28</sup> <https://dart.dev/guides/libraries/create-library-packages#organizing-a-library-package>

<sup>29</sup> <https://dart.dev/guides/language/effective-dart/usage#do-use-initializing-formals-when-possible>

From Dart 2.0 onwards, the `new` keyword shouldn't be used<sup>30</sup> because it's implicit. This is valid for both generative and factory constructors. The initializer list syntax has the same effect as the initializing formal but does not require the constructor parameters to have the same name as the class members. For example:

```
class Fraction {  
    int _num;  
    int _den;  
  
    Fraction(int numerator, int denominator) :  
        _num = numerator,  
        _den = denominator;  
}
```

This is useful when you want to initialize some variables that shouldn't be visible outside of the class scope. In our example, `_num` and `_den` are not visible from the outside and they don't even appear in the parameters list. They are hidden, and the constructor initializes them before executing the body (if any). Regardless, initializing formals would have been fine as well:

```
class Fraction {  
    int _num;  
    int _den;  
    Fraction(this._num, this._den);  
}
```

One approach isn't better than the other: they have the same effect. It is up to you to decide which one is better. The initializing formal syntax is the most widely used, especially in Flutter.

### Note.

All classes in Dart must always have a constructor. As such, if you don't specify it, the compiler implicitly generates a default, generative, parameter-less constructor<sup>31</sup>.

Constructors always have a body, which is executed after the initialization phase. When you don't give your constructor a body, the compiler automatically adds an empty one for you. For example:

---

<sup>30</sup> <https://dart.dev/guides/language/effective-dart/usage#dont-use-new>

<sup>31</sup> <https://dart.dev/guides/language/language-tour#constructors>

```

// Constructor with an empty body
Fraction(this.numerator, this.denominator);

// Same as before but '{}' is not needed because the compiler already adds it
Fraction(this.numerator, this.denominator) {}

// Constructor with a non-empty body
Fraction(this.numerator, this.denominator) {
  checkDenominator();
  doSomethingElse();
}

```

The `this` keyword is used to get a reference to the current instance. Since `this` is always implicit, the Dart team recommends<sup>32</sup> to only use it when there is a name conflict, such as when you initialize variables in constructor bodies. This is not always possible though! Consider this example:

```

class Fraction {
  int numerator;
  int denominator;
  Fraction(int numerator, int denominator) { // Compiler error
    this.numerator = numerator;
    this.denominator;
  }
}

```

In this example, `numerator` and `denominator` are non-nullable variables so they must be initialized with a value. Since the constructor body is executed after the object initialization phase, this code tries to create non-nullable variables without setting a value (which is an error). To fix the issue, you could make the fields nullable:

```

class Fraction {
  int? numerator;
  int? denominator;
  // ^ notice that they are nullable

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

```

<sup>32</sup> <https://dart.dev/guides/language/language-tour#constructors>

The code now compiles because `numerator` and `denominator` are first initialized with `null` and then, at a later point (when the constructor body is executed), they're given the values we pass as parameters. There are two main problems here:

1. we have converted variables to nullable types, even if they do not need to be nullable for our use-case;
2. we will have to constantly access variables with the bang (!) operator because we know that they are not `null` in practice, even if their static type says otherwise.

In general, you should only work with nullable types when needed. A better solution to the problem could be making both variables `late` so that they don't need an immediate initialization:

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

Variables are not nullable anymore but this still doesn't look quite right. We will see later in *Section 2.3 “Constant constructors”* the importance of `const` constructors in classes. Since the body is only used to initialize variables and has no other logic, at this point initializing formal or an initializer list are better choices. As a general guideline:

1. Use the initializing formal syntax whenever possible. If you want to “hide” internal variable names, make them library-private and use the initializer list syntax. This is the preferred way of initializing instance variables of a class.
2. If your constructor doesn't have a body, just don't create it. The compiler will automatically create an empty one for you. Try to not initialize instance variables in the constructor body, unless they are nullable (or `late`) and require additional logic (such as extra computations or sanity checks).
3. The constructor body is executed after the variable initialization phase. It's a great place to implement “startup logic”, such as throwing exceptions if variables haven't been initialized with correct values or calling functions.

The parameters of a constructors follow the same rules we have covered in *chapter 3 – Section 4.3 “Optional parameters”* for function parameters. They can be optional named or positional:

Example 1	Example 2
<pre>class Fraction {     int numerator;     int denominator;      Fraction({         required this.numerator,         this.denominator = 0,     }); }</pre>	<pre>class Fraction {     int numerator;     int denominator;      Fraction(this.numerator, [         this.denominator = 0,     ]); }</pre>

You cannot define a named optional parameter with a library-private variable. For example, the `required this._numerator` syntax is wrong because library-private variables cannot start with an underscore.

#### 4.2.2 Factory constructors and static members

The `static` keyword declares class variables or class methods. All instances share the same static members because they are similar to global variables. For example:

```
class Fraction {
    int numerator;
    int denominator;

    Fraction(this.numerator, this.denominator);

    static const zero = Fraction(0, 1);
    static void printZero() {
        print('num = ${ZERO.numerator}, den = ${ZERO.denominator}');
    }
}

void main() {
    // 'zero' holds 'Fraction(0, 1)'
    const zero = Fraction.ZERO;

    // Prints 'num = 0, den = 1'
    Fraction.printZero();
}
```

Notice that `printZero()` is not called from an object instance: it's invoked from the class type itself. Consequently, a `static` member does not have access to `this`. An object instance cannot access static members. For example:

```
class Example {  
    // static members can be placed anywhere in the class.  
    static int value = 0;  
  
    Example();  
    void printValue() => print('$value');  
}  
  
void main() {  
    Example.value = 10;  
  
    final example1 = Example();  
    final example2 = Example();  
  
    // Both print '10'  
    example1.printValue();  
    example2.printValue();  
  
    example1.value = 0; // Error - Instances cannot access static members  
}
```

In our example, `value` can only be accessed from the type itself (`Example.value`) and not from instances (`example1.value`). However, non-static members of a class (such as `printValue`) can read `static` content.

A factory constructor is very similar to a static method and thus has no access to `this`. It's used when you need to implement a constructor that doesn't always create a new instance. For example, an excellent use-case for a factory constructor is the implementation of a singleton:

```
class Singleton {  
    static final _instance = Singleton._();  
    factory Singleton() => _instance;  
  
    Singleton._(); // Library-private constructor  
}
```

Factory constructors are prefixed with the `factory` keyword and are used as any other constructor. The `Singleton._()` syntax creates a library-private constructor (we will cover it later in *Section 2.4 – “Named and redirecting constructors”*). The syntax to call a `factory` constructor is the same you would use for any other constructor:

```
void main() {
    final singleton = Singleton();
```

Here you're calling the factory constructor of the `Singleton` class. It doesn't always create a new `Singleton` object: it only does once, and then it always returns the same instance. In *Section 2.5 – "Good practices"* we will see another case where factory constructors are useful.

#### 4.2.3 Constant constructors

When all instance class variables don't need to change, you should make them all `final` and add a constant constructor. For example:

```
class Point {
    final double x;
    final double y;

    const Point(this.x, this.y); // Constant constructors must have NO body
}
```

A Dart class is said to be **immutable** when it has a `const` constructor and all of its instance variables are `final` (see the `Point` class, for example). To create a compile-time constant object, you must use the `const` keyword before the constructor's name; otherwise you would create a normal, non-`const` object. For example:

```
void main() {
    // OK - The variable is 'const' and it automatically calls the constant
    // constructor of 'Point'
    const constVariable = Point(1, 5);

    // WARNING - The variable is NOT 'const' and the constant constructor
    // is NOT called
    final notConstVariable = Point(1, 5);
}
```

As you can see, the compiler never automatically uses a constant constructor. If you want to use it, you must explicitly write down the `const` keyword in the assignment. You don't need to repeat the keyword twice because it's automatically deduced:

```
// OK
const constVariable = Point(1, 5);

// OK but the 'const' in front of 'Point' can be removed (it is redundant)
const constVariable = const Point(1, 5);
```

You can of course use `var` a `final` and invoke a constant constructor of an object (if any), but it will not make those variables constant too. For example:

```
var notConstVariable = const Point(1, 3);
final notConstVariable = const Point(1, 3);
```

In this case, both variables hold a reference to a constant object, but the variables themselves are not constant. We can't think of a valid use-case where a non-constant variable is initialized with a `const` constructor. When a constant constructor is available, we recommend to always initialize the variable with `const`. For example:

```
void main() {
    // Bad
    var notConstVar1 = const Point(1, 3);
    final notConstVar2 = const Point(1, 3);

    // Good
    const constant = Point(1, 3);
}
```

Of course, constant constructors can only be initialized with `const` values. For example:

```
void main() {
    const x = 1.0;
    final y = 1.0;

    const Point(x, 10); // OK - x is 'const'
    const Point(x, y); // Compiler error - y is not 'const'
}
```

In general, you should always try to add a `const` constructor to your class, *if that's possible*, so that the compiler can make some optimizations. For example, consider this case where we have a class without a constant constructor:

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

void main() {
    final one = Example(1);
    final two = Example(1);

    print(one == two); // false
}
```

The comparison between `one` and `two` returns `false` because the equality operator (by default) checks if the two variables hold a reference to the same object. However, look what happens when we compare the same class but with two constant objects:

```
class Example {  
    final int value;  
    const Example(this.value);  
}  
  
void main() {  
    const one = Example(1);  
    const two = Example(1);  
    print(one == two); // true  
}
```

Constant objects are created only once and then stored in special lookup tables by the compiler for reusability. In this example, `one` and `two` reference the same object in memory, so the comparison will always return `true`. You can also check this by yourself:

```
const one = Example(1);  
const two = Example(1);  
print(identical(one, two)); // true
```

The `identical` method, part of the Dart SDK, checks if two variables reference the same object in memory. Since it returns `true`, we can see that `one` and `two` point to the same object.

Note.

Compile-time constants are canonicalized and re-used as much as possible. For example, it does not matter how often you call `const Example(1)` because that object will be created only once (and then reused). Different configurations produce different constant objects. For example:

```
const one = Example(1);  
const two = Example(1);  
const three = Example(3);
```

In this case, `one` and `two` hold a reference to the same object on the heap. `three` instead holds a reference to another constant object because `Example` was built with a different configuration (we passed `3` instead of `1`).

When you don't define a constructor, the compiler generates a default one (with no parameters) that is not constant. In this case, define a constant constructor rather than create a class without a constructor. For example:

```
// Has a default, non-constant constructor.  
class Bad {}  
  
// Has a const constructor and so it can become a compile-time constant.  
class Good {  
    const Good();  
}  
}
```

Other than a consistent equality behavior, constant objects also bring in other advantages: less memory cluttering, possibility of imposing a meaningful sorting order (generally on collections) and improvements in the efficiency of various algorithms by eliminating repeated calculations.

#### 4.2.4 Named and redirecting constructors

Since Dart has no method overload, you must use named constructors if you want to create multiple constructors for the same class. They are in the `ClassName.name(parameters?)` form. Here's an example:

```
class Fraction {  
    final int numerator;  
    final int denominator;  
  
    // Default (or "unnamed") constructor  
    const Fraction(this.numerator, this.denominator);  
  
    // Named constructors  
    const Fraction.zero() : numerator = 1, denominator = 0;  
    const Fraction.whole(this.numerator) : denominator = 1;  
}
```

Constructors can be named, such as `Fraction.zero` or `Fraction.whole`, or unnamed, such as `Fraction` (which is also known as *default constructor*). Named constructors allow classes to have as many constructors as they want. They are also used to create library-private constructors, which make a class impossible to be instantiated:

```
class Example {  
    // Note the '._.()' ' syntax  
    const Example._();  
}
```

This syntax makes the class impossible to be instantiated because it hides the default `Example()` constructor. If you declare another named constructor, you could use it instead. For example:

```
class Example {  
  const Example._();  
  const Example.named();  
}  
  
const error = Example(); // Compiler error  
const valid = Example.named(); // OK
```

Named constructors can of course have a body. Both generative and factory constructors can be named or unnamed. For example:

```
class Constants {  
  final double value;  
  final String name;  
  
  const Constants({  
    required this.value,  
    required this.name,  
  });  
  
  factory Constants.euler() => Constants(value: e, name: 'Euler constant');  
  factory Constants.pi() => Constants(value: pi, name: 'PI');  
}
```

This is another common use-case for `factory` constructors: they create instances of a class with a particular configuration. If `Constants` was a supertype, the `factory` constructor could also return an object whose type is a subtype of `Constants`.

A redirecting constructor is generally used when your class has multiple constructors that share the same internal implementation but require a different name. For example:

```
class Fraction {  
  final int numerator;  
  final int denominator;  
  const Fraction(this.numerator, this.denominator);  
  
  // Named constructor  
  const Fraction.whole(this.numerator) : this.denominator = 0;  
  
  // Redirecting constructors  
  const Fraction.oneHalf() : this(1, 2);  
  const Fraction.one() : this.whole(1);  
}
```

In this case, `Fraction.oneHalf()` is just another way of calling `Fraction(1, 2)`. They are called “*redirecting*” because their behavior is “*redirected*” to another existing constructor. The only case where a `factory` constructor can be `const` is when it’s a redirecting one. We will make an example of this use-case in *chapter 5 – section 1.8 “Good practices”*.

Redirecting constructors cannot have a body.

#### 4.2.5 Good practices

There are a lot of kinds of constructors in Dart. We’ve created a list with some good practices that we believe you should follow when creating constructors:

1. If all instance variables are `final` and the constructor doesn’t need a body, always define a `const` constructor. In other words, try to create immutable classes whenever possible. For example:

```
class Fraction {  
    final double numerator;  
    final double denominator;  
  
    const Fraction(this.numerator, this.denominator);  
}
```

Constant classes are stored at compile-time in special lookup tables to be re-used later. This is a performance improvement that is particularly relevant in Flutter (more details in *chapter 10 – Section 3.2 “Performance considerations”*).

2. If your class cannot have a constant constructor, that is **absolutely fine**. Do not think that classes without a constant constructor are wrong: some objects must mutate, which is fine. A `const` constructor is a “plus” that should be used when possible, but not always. What we do not recommend is the creation of public variables. For examples:

```
class Example {  
    double value;  
    Example(this.value);  
}  
  
void main() {  
    final example = Example(2);  
  
    example.value = 5;  
}
```

This is bad because anyone can change `value` at any time in any place, and `Example` is never aware of that. Prefer using getters and setters (covered in the next section) to control the variable state.

3. Factory constructors are very helpful to initialize immutable classes whose members need to be initialized after some computations. For example:

```
EventInfo getEventInfo(DateTime date) {  
    // code here...  
}  
  
class Event {  
    final DateTime date;  
    final EventInfo info;  
    const Event._(this.date, this.info);  
  
    factory Event(DateTime date) {  
        final info = getEventInfo(date);  
        return Event._(date, info);  
    }  
}
```

The constant library-private constructor ensures that an immutable class can be built. Since the `factory` has the same name as the default constructor, this would look like a regular constructor call to the users. The Dart documentation also recommends this<sup>33</sup>. Without a `factory`, we would have had to do something else, such as using `late`:

```
class Event {  
    final DateTime date;  
    late final EventInfo info;  
    Event(this.date) : this.info = getEventInfo(date);  
}
```

The recommendation is to avoid public `late final` fields without initializer. We could have also made the variable nullable, but then we would have lost type-safety and added more complexity.

---

<sup>33</sup> <https://dart.dev/guides/language/effective-dart/design#avoid-public-late-final-fields-without-initializers>

4. Use the *initializing formal* syntax to initialize variables of a constructor. Alternatively, use the initializer list syntax to give different names in the parameters list.
  
5. Use redirecting constructors to reuse the logic of another constructor.

In *chapter 5 – Section 1.8 “Good practices”* we will see how **factory** constructors can also be useful to return default implementations of **abstract** classes.

## 4.3 Getters and setters

When you want to expose a variable to other libraries but you don't want its value to be changed, you can make it **final**. In a similar way, you could make a variable non-**final** so that it can be read and changed at any time. For example:

```
class Example {
    final double readOnly;
    int readAndWrite;
    Example(this.readOnly) : readAndWrite = 0;
}

void main() {
    final example = Example(1);

    example.readAndWrite = 100; // OK
    example.readOnly = 10; // Compiler error
}
```

In this example, the value of **readAndWrite** can be read and modified at any time. The class cannot control how the variable value is changed from outside. This generally isn't the desired behavior. Getters and setters are particular methods that provide read-and-write rules for a member. For example:

```
class Example {
    final double readOnly;
    int _readAndWrite;

    Example(this.readOnly) : _readAndWrite = 0;

    int get readAndWrite => _readAndWrite;
    set readAndWrite(int value) {
        _readAndWrite = max<int>(0, value);
    }
}
```

A getter, implemented with the `get` keyword, provides read-only access to the value of a variable. A setter, implemented with the `set` keyword, controls how the variable value changes. The library-private `_readAndWrite` member is protected by getters and setters. They control how the variable is changed from the outside. For example:

```
void main() {
  final example = Example(10);

  print(example.readOnly);
  print(example.readAndWrite); // Here you are using the getter to print 0

  example.readAndWrite = 40; // Here you are using the setter
  print(example.readAndWrite); // Here you are using the getter to print 40

  example.readAndWrite = -24; // Here you are using the setter
  print(example.readAndWrite); // Here you are using the getter to print 0
}
```

In general, getters and setters are used to control the status of a variable. In some cases, they may make your code more verbose<sup>34</sup> or just be useless. For example:

```
// Good example
class Example {
  String value = '';
}

// Bad example - 'get' and 'set' are not helpful here
class Example {
  String _value = '';

  String get value => _value;
  set value(String value) => _value = value;
}
```

In this example, the getter and the setter are not needed because they don't add any "validation" logic or checks to the variables. A normal instance variable produces the same result and requires less maintenance. Here are a few good recommendations we have for getters and setters:

- A getter and a setter should have the same name if they refer to the same variable. This is not required by the language, but it's more intuitive for the user. For example:

---

<sup>34</sup> <https://dart.dev/guides/language/effective-dart/usage#dont-wrap-a-field-in-a-getter-and-setter-unnecessarily>

```

// Good practice
class Example {
    String _value = '';

    String get value => _value;
    set value(String value) => _value = value.isEmpty ? 'None' : value;
}

// Not a good practice
class Example {
    String _value = '';

    String get readValue => _value;
    set newValue(String value) => _value = value.isEmpty ? 'None' : value;
}

```

Reading and writing the same variable is more intuitive than working with different names.

- A getter should return a value as-it-is or make some short operations that can easily be read at glance. For example, these getters look good:

```

double get radians => _angleRadians;
double get degrees => _angleRadians * 180 / 3.14;

```

You immediately understand that the first getter returns an angle (in radians) and the other one converts radians to degrees. If the getter requires computations or, in general, time-consuming calls, prefer using a method. For example:

```

double degrees() {
    final angle = computeAngle();
    final somethingElse = somethingElse();
    return angle + somethingElse * 2;
}

```

- Rather than creating a read-only value with a getter, prefer using a `final` instance variable. It has the same effect and requires less code. For example:

```

class Example {
    final String _value;
    const Example(this._value);

    String get value => _value;
}

```

This code compiles but doesn't make much sense. A `final` variable is already read-only so you can remove the getter and make `_value` public:

```
class Example {  
    final String value;  
    const Example(this.value);  
}
```

A `final` instance variable is a sort of “implicit getter” that exposes a value without allowing modification.

Getters and setters can be prefixed with the `static` modifier.

## 4.4 Operators overload

When an operator appears in an expression involving two types, the compiler internally converts the symbol into the associated function call to evaluate the result. In simpler terms, some symbols (such as `+` and `-`) can call special functions to execute an operator. This feature is widely known as operator overload. For example:

```
class Fraction {  
    final int numerator;  
    final int denominator;  
    const Fraction(this.numerator, this.denominator);  
  
    @override  
    String toString() => '$numerator/$denominator';  
  
    // Overload of the '+' operator  
    Fraction operator+(Fraction other) {  
        return Fraction(  
            numerator * other.denominator + denominator * other.numerator,  
            denominator * other.denominator,  
        );  
    }  
  
    // Overload of the '-' operator  
    Fraction operator-(Fraction other) {  
        return Fraction(  
            numerator * other.denominator - denominator * other.numerator,  
            denominator * other.denominator,  
        );  
    }  
}
```

In our example, thanks to operator overloading, we have enhanced our class to support the `+` and `-` operators. As such, objects can be used in expressions and evaluate to other objects. For example:

```
void main() {
    final sum = Fraction(1, 2) + Fraction(2, 3); // 1/2 + 2/3
    final sub = Fraction(1, 2) - Fraction(2, 3); // 1/2 - 2/3

    print('$sum'); // 7/6
    print('$sub'); // -1/6
}
```

Operator overloads are special functions whose name must be “operator” followed by a symbol. They either require one or two non-optional parameter(s) and have no restrictions on the return type. For example, this is how we could implement subtraction and negation for the `Fraction` class:

```
// This is used to subtract one fraction to the other (expects 2 operands)
Fraction operator-(Fraction other) {
    return Fraction(
        numerator * other.denominator - denominator * other.numerator,
        denominator * other.denominator,
    );
}

// This is used to NEGATE the value of a fraction (expects 1 operand)
Fraction operator-() => Fraction(-numerator, denominator);
```

The second overload is used when a type requires the negation sign at the front. For example, `a - b` invokes `operator-(other)` while `-a` invokes `operator-()`. We could have returned different types from various operators. For example:

```
Fraction operator +(int other) {
    return Fraction(
        numerator + other,
        denominator + other,
    );
}

double operator -(Fraction other) {
    return (numerator/denominator) - (other.numerator/other.denominator);
}
```

Operators cannot be `static` and take a predefined number of parameters. Here is a list of all supported operators:

- Operators that take 2 parameters: `[ ]=`.
- Operators that take 1 parameter: `< > <= >= == - + / ~/ * | ^ & << >> >>`, `%` and `[ ]`.
- Operators that take 0 or 1 parameters: `-` (the negation operator).
- Operators that take 0 parameters: `~` (unary bitwise complement, which swaps 0s and 1s).

The `operator [ ]=(index, value)` operator is used to set the value at the given `index` in the list to `value`. The `operator==` is used to test for object equality but its implementation is not trivial and requires some considerations, which we will make in *chapter 5 – Section 2 “The Object class”*.

#### 4.4.1 The `call()` method

Any Dart class you create can be called as if it was a function when it implements the `call` method. Very intuitively, a callable class is a class that implements the `call` method. This method supports the same functionalities of regular functions, such as return types and parameters. For example:

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

  void call(int other) {
    print('${value} + $other = ${value + other}');
  }
}

void main() {
  final demo = const Demo(10);

  // Use it as a normal function call...
  demo.call(5); // prints '10 + 5 = 15'

  // ... or invoke 'call' directly on the instance itself!
  demo(5); // prints '10 + 5 = 15'
}
```

The `demo.call(5)` syntax is an equivalent of `demo(5)` because they both invoke the `call` method with the given argument. This is useful, for example, when you need to invoke a nullable function without writing too much checks. Consider this code:

```

typedef SuccessCallback = void Function(int code);

void someFunction(SuccessCallback? onSuccess) {
  if (onSuccess != null) {
    onSuccess(1);
  }
}

```

Since `onSuccess` is nullable, a null check is required to use the variable. You can rewrite the code in this equivalent, but shorter, form:

```

void someFunction(SuccessCallback? onSuccess) {
  onSuccess?.call(1);
}

```

The `Function` type is a callable class and thus you can always invoke the `call` method on a function. Here is another example that shows how you can “hide” a class behind a function. Consider these two libraries:

age_printer.dart	main.dart
<pre> void printAge(int age) {   print('Age: \$age'); } </pre>	<pre> void main() {   printAge(25); } </pre>

This is how you normally create a function (`printAge` in the example). With callable classes, we can keep the same syntax inside `main()` while reworking the definition of the `printAge` function itself. The “trick” is to hide a callable class behind a variable. For example:

age_printer.dart	main.dart
<pre> class _AgePrinter {   const _AgePrinter();   void call(int age) {     print('Age: \$age');   } }  const printAge = _AgePrinter(); </pre>	<pre> void main() {   printAge(25); } </pre>

We've made the callable `_AgePrinter` class library-private (so that nobody can see it) and created a top-level `printAge` constant. The `main.dart` file did not change: we have only reworked the function implementation, but the public API is still consistent.

Note that `call` can return any type and take zero or more parameters. The only "special" thing about this function is that it can be called in various ways (`object.call()` and `object()`).

## Deep dive: Cloning objects

Even if not explicitly mentioned by the official documentation, there is a standard pattern to follow when it comes to cloning objects. When you want to create a clone (or a "copy") of an object, you can create a **shallow** or a **deep** copy.

### Note

The difference between a shallow and deep copy is only relevant for compound objects (objects that contain other objects, like lists or class instances). This isn't a Dart-specific feature: all programming languages need to clone objects. Some have built-in ways to do it, others don't.

A shallow copy creates a new object that shares the same memory with the original object. If the original object contains any references to other objects, the new object will also reference the same objects as the original one. Let's make an example to clarify the idea:

```
class Person {  
    final Name name;  
    final int age;  
    const Person(this.name, this.age);  
}  
  
class Name {  
    String name;  
    Name(this.name);  
}  
  
void main() {  
    final person = Person(Name('Alberto'), 28);  
  
    // This is a shallow copy  
    final copy = Person(person.name, person.age);  
}
```

The `copy` variable holds a reference to a new `Person` object that is initialized with an existing `Name` object (`person.name`). In practice, we are creating a new `Person` object by reusing some existing ones, such as `person.name`. This causes side-effects because the two objects are not independent. For example:

```
void main() {
    final person = Person(Name('Alberto'), 28);
    final copy = Person(person.name, person.age);

    copy.name.name = 'Alberto (changed)';
    print(person.name.name); // Alberto (changed)
    print(copy.name.name); // Alberto (changed)
}
```

We changed the `name` property on the `copy` object and the original object (`person`) also changed. This is a typical side-effect of shallow copies because they only create a clone of the “outer” object and leave internal references to the “old” object.

A deep copy constructs a new object and also copies all internal objects with new instances. In other words, all properties and fields of the original objects are copied into the new one; no references are moved. Deep copies in Dart are created with the `copyWith` method. Let’s see how to add deep copy support to the `Person` class:

```
class Person {
    final Name name;
    final int age;
    const Person(this.name, this.age);

    Person copyWith({Name? name, int? age}) {
        return Person(
            name ?? Name(this.name.name),
            age ?? this.age,
        );
    }
}
```

The `copyWith` method (whose name is a popular convention, not a requirement) is used to create deep copies of Dart objects. It can either create an *exact* (deep) or a *modified* (deep) copy of the object. In the example, we’ve created a standard implementation of the `copyWith` method, which:

- has the same parameters as the constructor, but makes all of them nullable, and
- returns a new object with either the given values or creates new ones.

Let’s see a few examples to see how this method works:

```

void main() {
    final person = Person(Name('Alberto'), 28);
    final copy = person.copyWith();

    copy.name.name = 'Alberto (changed)';

    print(person.name.name); // Alberto
    print(copy.name.name); // Alberto (changed)
}

```

The `copyWith` method makes deep copies because it creates new clones of the object itself and all of its internals. In the example, changing the `name` value in `copy` does not change the `name` value of the original `person` object. This is because `person` and `copy` are two different objects that do not share references. Another example:

```

void main() {
    final person = Person(Name('Alberto'), 28);
    final copy = person.copyWith(age: 29);

    copy.name.name = 'Older Alberto';

    print('${person.name.name}, ${person.age}'); // Alberto, 28
    print('${copy.name.name}, ${copy.age}'); // Older Alberto, 29
}

```

In this example, `copyWith` still creates a deep copy of the `person` object but uses a different value for the age. When you have to clone a collection (*chapter 6 – Section 2 “Collections”*), such as a list, you have to pay attention to clone everything correctly. For example:

```

class People {
    final List<Person> personList;
    const People(this.personList);

    People copyWith({List<Person>? personList}) {
        if (personList != null) {
            return People(personList);
        }

        // Deep copy logic
        final deepList = <Person>[];
        for (final person in this.personList) {
            deepList.add(person.copyWith());
        }
        return People(deepList);
    }
}

```

If the user provides a list from the outside, we pass it to the new object. Otherwise, we create a new list to make a deep copy and then we iterate over all the existing elements to call `copyWith`. In this way, we're guaranteed that everything is new and no references are shared. This version is equivalent but a bit more efficient (and concise):

```
People copyWith({List<Person>? personList}) {
    final list = personList ??
        List<Person>.generate(
            this.personList.length,
            (index) => this.personList[index].copyWith(),
            growable: false,
        );
    return People(list);
}
```

In *chapter 6 – Section 2 “Collections”* you will see why the `generate` constructor, in this case, is a better choice. Regardless, the point is that you must iterate over all items to make a deep copy and either call `copyWith` or manually make a clone.

When you create your own classes, consider adding a `copyWith` method for deep copying. In case it wasn't available, you have two options:

1. Manually create a clone every time you need to deep copy an object. This isn't ideal in terms of maintenance because it increases code duplication and testing efforts.
2. If you had no access to the class's source code, consider adding `copyWith` using extension methods (*chapter 5 – Section 1.6 “Extension methods”*).

When you clone collections (such as lists), traverse all elements and deep copy them. If the other object had a `copyWith` method, you could recursively use it. For example:

```
class Example {
    final int a;
    final Other other;
    const Example(this.a, this.other);

    Example copyWith({int? a, Other? example}) {
        return Example(
            a ?? this.a,
            example ?? this.other.copyWith(),
        );
    }
}
```

Assume that the `Other` class had a `copyWith` method. In this case, you could directly use it rather than manually cloning the object. This is the same thing you'd do when making a deep copy of an object in a collection.

## Deep dive: Annotations and mirrors

In Dart, annotations are metadata that give additional information about a piece of code. Any class with a `const` constructor can be used as an annotation anywhere in the code. For example:

```
class SomeAnnotation {
  const SomeAnnotation();
}

@SomeAnnotation()
class SomeOtherAnnotation {
  @SomeAnnotation()
  final int numValue;

  @AnotherAnnotation('Test', true)
  const SomeOtherAnnotation(this.numValue);
}

@YetAnotherAnnotation(1)
void main() {
  print('Hello world!');
}
```

Annotations can be used before the following keywords: `library`, `class`, `typedef`, `import`, and `export`. They can also appear before constructors, functions, function type parameters and variable declarations. When you need to create an annotation, put the class in a dedicated file and make it library-private. Then, expose a constant variable with an instance of that class. For example:

```
// This is inside the 'annotations.dart' file in the Dart SDK
class _Override {
  const _Override();
}

const override = _Override();
```

This is how the Dart SDK defines the `@override` annotation, which is used to annotate overridden members. The class itself is library-private but there is a public, constant variable that can be used anywhere in the code. Annotations themselves do nothing. They are generally used to give more information about a class, a function, or a member to other Dart code (or external tools).

## Example.

The analyzer tool for example, (invoked by the `dart analyze` command) is a program that parses your Dartcode. Among many other things, it can recognize annotations and emit warnings or alerts under certain circumstances.

If you want to write a tool that analyzes Dart code, you need the `dart:mirrors` library (which is part of the core Dart SDK). A mirror is an object that reflects another object. Thanks to mirrors, you can write Dart code that inspects itself. For example:

```
import 'dart:mirrors';

// This class can be used as annotation because it has a 'const' constructor
class MyAnnotation {
    final int number;
    final String string;
    const MyAnnotation(this.number, this.string);

    @override
    String toString() => 'number = $number | string = $string';
}

@MyAnnotation(10, 'hello!')
class Example {
    final double exampleValue;
    const Example({this.exampleValue = 0});
}

void main() {
    final classTypeData = reflectClass(Example);

    print(classTypeData.isAbstract); // false
    print(classTypeData.isEnum); // false
    print(classTypeData.isTopLevel); // true

    // Here we read all annotations (if any) associated with the 'Example' class
    for(final annotation in classTypeData.metadata) {
        print('${annotation.reflectee}'); // number = 10 | string = hello!
    }
}
```

Reflection in Dart is made with mirrors. The `reflectClass` method returns a `ClassMirror` object with much information about the `Example` class. In particular, the `metadata` getter returns a list with all the annotations (if any) associated with the `Example` object. In this example, we get a

reference to `MyAnnotation` and all of its members using `annotation.reflectee`. Reflection can extract more data about the class being inspected. For example, we could tell how many instances and static members the `Example` class had (even if it's not annotated):

```
class Example {  
    final double exampleValue;  
    const Example({this.exampleValue = 0});  
}  
  
void main() {  
    final classTypeData = reflectClass(Example);  
  
    // Prints all instance members of the 'Example' class  
    for(final instanceMembers in classTypeData.instanceMembers.entries) {  
        // Name: Symbol("==")  
        // Name: Symbol("hashCode")  
        // Name: Symbol("toString")  
        // Name: Symbol("noSuchMethod")  
        // Name: Symbol("runtimeType")  
        // Name: Symbol("exampleValue")  
        print('Name: ${instanceMembers.key}');  
    }  
  
    // Prints all static members of the 'Example' class  
    for(final instanceMembers in classTypeData.staticMembers.entries) {  
        // Empty list (no static members)  
        print('Name: ${instanceMembers.key}');  
    }  
}
```

Even if `Example` only declares a single variable (`exampleValue`), it also inherits other members from the `Object` supertype. We can also easily verify that `Object` is a supertype of `Example`:

```
// Returns the superClass type  
if (classTypeData.superclass != null) {  
    print('Super type: ${classTypeData.superclass!.reflectedType}'); // Object  
}  
  
// An alternative way to tell if A is a subtype of B  
final isSubtype = classTypeData.isSubtypeOf(reflectClass(Object));  
print('"Example" a subtype of "Object": $isSubtype'); // true
```

To make a concrete example, we could use reflection and annotations to create a code generation tool that builds web servers. Let's start by creating some annotations that will be used to describe the HTTP server configuration:

```

// Defines the path and the static HTML contents of the website
class Route {
  final String path;
  final String contents;
  const Route(this.path, this.contents);
}

// HTTP server configurations
class Https {
  final String certPath;
  final bool autoRedirect;
  const Https({
    required this.certPath,
    required this.autoRedirect,
  });
}

// Compression configurations for our server
class Compression {
  final bool useGzip;
  const Compression(this.useGzip);
}

```

Since we want these classes to be used as annotations, we must give them a constant constructor. Our tool uses the `dart:mirrors` library to inspect a class (`HttpServer`) that is decorated with the above annotations. For example:

```

@Compression(true)
@Https(certPath: '/path/to/cert/', autoRedirect: true)
abstract class HttpServer {
  @Route('/', '<html><body>Home page</body></html>')
  void home();

  @Route('/contents', '<html><body>Contents page</body></html>')
  void contents();

  @Route('/about', '<html><body>About page</body></html>')
  void about();
}

```

Since the `HttpServer` class is decorated with various annotations, we can write a Dart function (or create a separate program) that uses `dart:mirrors` to read annotations and generates other Dart code. Note that we will print the generated code to the console to simplify the example (rather than saving everything on a `.dart` file in the local filesystem). Here's a possible implementation of the code generation tool:

```

void generate() {
    final classTypeData = reflectClass(HttpServer);
    final buffer = StringBuffer();

    // Iterates over all members in the 'HttpServer' class
    for (final getters in classTypeData.declarations.entries) {
        // The 'metadata' getter returns a list of all annotations
        final metadata = getters.value.metadata;
        final route = metadata.isNotEmpty ? metadata.first.reflectee : null;

        // Generates a 'RouteEntry' by reading the 'Route' annotation data
        if (route != null && route is Route) {
            buffer.writeln(
                "    RouterEntry('${route.path}', '${route.contents}'),",
            );
        }
    }
}

Compression? compression;
Https? https;
for (final annotation in classTypeData.metadata) {
    if (annotation.reflectee is Compression) {
        compression = annotation.reflectee;
    }
    if (annotation.reflectee is Https) {
        https = annotation.reflectee;
    }
}

// The generated Dart code
print('''
Future<void> main() async {
    await SomeServer(
        hasHttps: ${https != null},
        compression: ${compression?.useGzip ?? 'true'},
        routes: [
$buffer    ],
    ).start();
}
''');
}

```

Our `HttpServer` class only has abstract methods (functions with no body) that are decorated with an annotation. The `declaration` getter returns a list with all members of a class. We iterate over them and use the `metadata` getter to read the associated annotation. This is what is printed to the console (we have formatted and highlighted the code for readability):

```

Future<void> main() async {
  await SomeServer(
    hasHttps: true,
    compression: true,
    routes: [
      RouterEntry('/', '<html><body>Home page</body></html>'),
      RouterEntry('/contents', '<html><body>Contents page</body></html>'),
      RouterEntry('/about', '<html><body>About page</body></html>'),
    ],
  ).start();
}

```

We started from an annotated class and we ended up with the generation of new Dart code, using the `dart:mirror` library. Another example of how annotation can be useful comes directly from Flutter. For example:

```

@mustCallSuper
void initState();

```

The Dart analyzer tool inspects your Dart code and since it finds the `mustCallSuper` annotation, it emits a warning if you didn't call `super.initState()` when overriding `initState`. In this example, annotations are used by an external tool (the Dart analyzer) to inspect your code and emit warnings or errors. In summary:

- Annotations are used to “decorate” Dart code and add more information. Any class with a `const` constructor can be used as an annotation.
- Annotations are often used with the `dart:mirror` library to either generate other Dart code or retrieve metadata about classes to inspect them (as the Dart analyzer does, for example).

The minification process compresses identifiers on the program to reduce the download size. This practice causes issues when using the mirrors library with strings to refer to pieces of code. What it means is that, for example, you could use a `String` to refer to a class that the minification process will remove. To solve this problem, Dart uses `symbols`. For example:

```

// Create a symbol using the 'Symbol' class
const symbol = Symbol('symbolName');
// Create the same symbol using a literal
const symbol2 = #symbolName;

print(symbol == symbol2); // true
print(identical(symbol, symbol2)); // true

```

A symbol can be created either explicitly, with the `Symbol` default constructor, or implicitly, with a literal that is prefixed by a `#` (hashtag). In the example, both variables hold a reference to the same symbol because the string in the constructor matches the literal name.

## Example.

All instances of `Symbol` are guaranteed to be stable with respect to minification. This means that symbols won't be "removed" in the minification process, as might happen with strings. Note that symbols are not meant to be used as "unique" keys.

In general, symbols are useful when using the `dart:mirrors` library or when working with dynamic invocations. For example, symbols are used by the `Function.apply` method to represent all named parameters in the invocation:

```
void sumValues(  
    bool isSum, {  
        required int a,  
        required int b,  
    }) => isSum ? print(a + b) : print(a - b);  
  
void main() {  
    // Invoke the function in the normal way  
    sumValues(true, a: 10, b: -6);  
  
    // Invoke the function using the Low-Level 'apply' methods  
    Function.apply(sumValues, [true], {#a: 10, #b: -6});  
}
```

Dart internally uses the `apply` static method of `Function` to make function calls. It uses symbols rather than strings to represent named parameters to avoid the minification issue we've described before. Note that the symbol literal names match the names of the optional parameters. If we didn't pass a literal that matched the parameter name, we would have got an error:

```
// Throws a runtime exception because '#b' is missing  
Function.apply(sumValues, [true], {#a: 10, #c: -6});
```

Another case where symbols are used is the `Invocation` class, which is passed to `noSuchMethod` when an object doesn't support the member invocation that was attempted on it. As you can see, unless you're working with `dart:mirrors` or low-level Dart features, you won't use symbols in your projects.

## Deep dive: Language specification details

This section contains some interesting language features that are covered in the Dart specification document<sup>35</sup>. The contents of the document are very rigorous and formal but here we've extracted and rephrased some curiosities about classes and its members:

- In Dart, classes cannot be nested inside other classes. Both examples raise a compile-time error:

```
class A {  
    class B {} // Compiler error  
}  
  
void myFunction() {  
    class InnerClass {} // Compiler error  
}
```

Classes can only be top-level members. As such, you can't even define a class inside the body of a getter, a setter or a function for example.

- We have seen that annotations, also known as metadata, can be retrieved at runtime using the `dart:mirror` library. However, they could also be statically retrieved by parsing the program and evaluating the constants using a suitable interpreter. In fact, many (if not most) uses of metadata are entirely static.
- The Dart core library (`dart:core`) is implicitly imported by all libraries other than itself. Any import of the core library, even if restricted with `show`, `hide` or `as`, preempts the automatic import.
- A script is a library that includes a top-level function named `main` with either zero, one, or two required arguments. In other words, a script is a library that contains the application's entry-point function (`main`). It can be of three types:

```
void main(List<String> args, Object? value) {}  
void main(List<String> args) {}  
void main() {}
```

---

<sup>35</sup> <https://dart.dev/guides/language/spec>

If `main` requires more than two parameters (so if it's not in any of the form we have listed above), the library is not considered a script and thus it can't serve as entry-point.

- When compiling a Dart application for the web (using `dart compile js`), you can lazily load libraries. This is mostly useful to reduce the application startup time and avoid loading code that might never be used. Deferred loading (or “lazy loading”) has the following syntax:

```
import 'my_library.dart' deferred as my_lib;
```

To lazily load a library, you must use `deferred as` followed by a name, as it happens with library aliased. When your code needs to use the library, you must make sure it was loaded with `loadLibrary()`. For example:

```
Future<void> doSomething() async {
  await my_lib.loadLibrary(); // <--- Call this
  my_lib.function1();
  my_lib.function2();
}
```

In this example, `await` pauses the method execution until the library is loaded. You can call `loadLibrary()` multiple times without problems because it will only load the library once.

The `async` and `await` keywords are covered in *chapter 8 – Section 2.2 “async and await”*. An example of a Dart web application that uses deferred loading is in *chapter 22 – Section 4.2 “Deferred imports”*.

- Setters, getters, and operators can never have optional parameters of any kind.
- All Dart classes have a `noSuchMethod` method that is implicitly invoked when one or more member lookups fail. For example:

```
class A {}

void main() {
  final safe = A();
  safe.test(); // Compiler error
}
```

This code is safe because the compiler sees that there is no `test` method in class `A`. Look at this example instead:

```

void main() {
    dynamic unsafe = A();
    unsafe.test(); // Runtime error
}

```

Since we're using `dynamic`, it's not possible for the compiler to determine whether `test` is a member of `A` or not. When you run the program, the `noSuchMethod` method is called when you try to access `test` from `A` because the member doesn't exist. You can see `noSuchMethod` as a "backup method" that kicks in when a runtime invocation on a member is attempted but it fails.

- The constructor of a class can be torn off using the `new` keyword after the type name. For example:

```

class MyClass {
    final double value;
    const MyClass(this.value);
}

void transform(MyClass Function(double) fn) {
    // do something
}

void main() {
    // OK - classic anonymous function
    transform((value) => MyClass(value));

    // OK - equivalent to above (uses a tear-off)
    transform(MyClass.new);
}

```

The `MyClass.new` syntax replaces the anonymous function that forwards the parameter to the constructor. Both versions are equivalent, but the constructor tear-off is less verbose.

- When you have to initialize a variable with a pre-defined value, you can do it immediately in the declaration site rather than using a constructor. For example:

```

class Example {
    String _value = 'initial';
    Example();
}

```

Here we have initialized `value` directly in the declaration site. We could have also initialized it in the constructor, but it would have been more verbose:

```
class Example {  
    String _value;  
    Example() : this._value = 'initial';  
}
```

Both examples are equivalent, but the former is shorter and, in our opinion, more readable.

# 5 – Inheritance, core classes and exceptions

---

## 5.1 Inheritance

Being Dart an object-oriented programming language, any class can inherit properties from another class. This bond creates a dependency where the “parent” class is called **superclass** and the “child” is called **subclass**. For example:

```
// Superclass of 'B'  
class A {}  
  
// SubClass of 'A' and superclass of 'C'  
class B extends A {}  
  
// SubClass of 'C' (and, transitively, subclass of 'A')  
class C extends B {}
```

In this example, the **extends** keyword indicates that **B** is a subclass of **A** and **C** is a subclass of **B**. Dart does not allow multiple inheritance. For example, this is a compile-time error:

```
class A {}  
class B {}  
  
class C extends A, B {} // Compiler error
```

A class can have one or more superclasses along the hierarchy, but only one can be a direct parent. Aside from constructors, all members of a class can be overridden. It is a good practice to decorate the overridden member in a subclass with the **@override** annotation. For example:

```
class A {  
    double test(double a) => a * 0.5;  
}  
  
class B extends A {  
    @override  
    double test(double a) => a * 1.5;  
}
```

Method overriding occurs when a subclass has the same method as the superclass. In this example, we’re overriding the **test** method in class **B** to give it a different implementation. While **this** gets a reference to the current class, **super** gets a reference to the superclass. For example, it could be used to call the superclass implementation of a method you’re overriding:

```

class A {
  double test(double a) => a / 2;
}

class B extends A {
  @override
  double test(double a) {
    final original = super.test(a); // Calls A.test(int)
    return original * 2;
  }
}

void main() {
  final value = B().test(5);
  print('value = $value'); // 5.0
}

```

In this example, the `super.test(a)` call invokes the superclass `test(int)` method and returns its value. In general, `super` is used to access any member of the superclass. Both `this` and `super` have no access to static members.

## Note

From Dart 2.9 onwards implicit up-casts are always allowed, but implicit down-casts are forbidden. For example:

```

// OK from Dart 2.8 and older versions
void main() {
  A a = A();
  B b = a;
}

// Invalid from Dart 2.9 and newer versions
void main() {
  A a = A();
  B b = a; // compiler error here
}

```

Up-casting is a “safe” operation that casts a subtype into a supertype. Down-casting is “unsafe” because it’s not guaranteed to always succeed and thus it requires type checks.

Now that you've got a general overview of how Dart supports inheritance let's dig deeper.

### 5.1.1 Constructors

We already know that, when a Dart class doesn't define a constructor, the compiler automatically generates a default one with no parameters. The same thing also happens along a class hierarchy. Remember that the generated constructor is NOT a constant one:

What you write	What the compiler creates
<pre>class A {}  class B extends A {}</pre>	<pre>class A {   A(); }  class B extends A {   B() : super(); }</pre>

The compiler automatically adds constructors to the classes along the hierarchy because we didn't define them. However, when the superclass does have a constructor, you must reference it using `super` in the initializing formal:

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

class SubExample extends Example {
  SubExample(int value) : super(value); // You must call 'super' here
}
```

Note that `super` calls always go last in the initializer list order. If you had assignments or assertions, they must be declared before `super` otherwise the compiler will throw an error:

```
class SubExample extends Example {
  int _doubleValue;

  SubExample(int value) :
    _doubleValue = value * 2,
    assert(value >= 0, 'Value must be positive'),
    super(value); // 'super' call must go last
}
```

If the superclass defined a lot of instance members, there would be a lot of repetitions. The same variable would have to be declared in the constructor and the initializer list. For example:

```

class Example {
    final int a;
    final double b;
    final String c;
    const Example(this.a, this.b, [this.c = '']);
}

class SubExample extends Example {
    // A lot of repetitions for types and variable names
    const SubExample(int a, double b, String c) : super(a, b, c);
}

```

Notice that we've repeated `a`, `b`, and `c` a lot of times. To solve this issue and make the code shorter, you can directly use `super` in the parameters list. For example, we could rewrite the code in the previous example in this equivalent form:

```

class SubExample extends Example {
    const SubExample(super.a, super.b, super.c);
}

```

This syntax doesn't have performance implications, it's just syntactic sugar. A subclass can define a constant constructor only if the superclass already has one. The *constness* of a constructor is not inherited along the hierarchy. For example:

```

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

class SubExample extends Example {
    SubExample(super.value);
}

void main() {
    // OK
    const example = Example(10);

    // ERROR because 'SubExample' doesn't define a const constructor
    const subExample = SubExample(10);
}

```

The compile-time error is caused by the fact that `SubExample` doesn't define a `const` constructor. To make the example work, we would need to add `const` in the subclass:

```
const SubExample(super.value);
```

If a superclass does not have a constant constructor, then the subclass is not allowed to define a constant constructor. Since constructor bodies are always executed after the initialization phase, you cannot call the super-constructor from there. For example:

```
// OK
SubExample(super.a);
SubExample(int a) : super(a);

// Error
SubExample(int a) {
  super(a);
}
```

If Dart allowed calling super-constructors from bodies, it would have no way to fully initialize the object when created. Of course, you can invoke named or redirecting constructors of a superclass using the same notation. For example:

```
class Example {
  final int value;
  const Example.zero() : value = 0;
}

class SubExample extends Example {
  const SubExample() : super.zero();
}
```

The only exception is that you cannot call a [factory](#) constructor using `super`.

### 5.1.2 Abstract classes

Abstract classes, declared with the `abstract` keyword, can only be instantiated by a subclass. They can contain abstract methods, getters, or setters. For example:

```
abstract class Example {
  void processValue(); // This is an abstract method
  int get value; // This is an abstract getter
}

class SubExample extends Example {
  @override
  void processValue() => value * 2;

  @override
  int get value => 10;
}
```

The `abstract` keyword can only be used in the class definition; abstract methods and getters have no body. In this example, `SubExample` is said to be a **concrete** class because it overrides all of the superclass' abstract methods. If you have an `abstract` superclass and you don't override all of its abstract members, your subclass must be `abstract` too:

```
abstract class Example {  
  void processValue();  
  int get value;  
}  
  
abstract class SubExample extends Example {  
  @override  
  void processValue() => value * 2;  
}
```

In this case, `SubExample` must be an abstract type because it doesn't override all abstract methods of the superclass. If we continued to extend the hierarchy, we would have to override the remaining abstract members to create a concrete class. For example:

```
abstract class Example {  
  void processValue();  
  int get value;  
}  
  
abstract class SubExample extends Example {  
  @override  
  void processValue() => value * 2;  
}  
  
class ConcreteExample extends SubExample {  
  @override  
  int get value => 10;  
}
```

The `ConcreteExample` type is a **concrete** class because it inherits the `processValue()` definition from its superclass and it also overrides the `value` getter. Constructors cannot be abstract.

### 5.1.3 Interfaces

By default, any Dart class can be used as an interface. This may seem a bit weird at first, but you'll quickly get used to it. In *chapter 7 – section 1 “Class modifiers”* we will see how to restrict a class to be only used as an interface. As a good practice, we recommend using abstract classes to create interfaces. For example:

```

// This abstract class is used as an interface
abstract class MyInterface {
    void method();
    int get getter;
}

// To treat a class as if it was an interface, use the 'implements' keyword
class Example implements MyInterface {
    @override
    void method() {}

    @override
    int get getter => 10;
}

```

The `implements` keyword, differently from `extends`, forces `Example` to override all members of `MyInterface` (abstract and non-abstract ones). Furthermore, a class is allowed to implement one or more interfaces. You could have obtained the same result with a regular class, but it's generally not recommended. For example:

```

class MyInterface { // This isn't abstract so members need a body
    void method() {}
    int get getter => 0;
}

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

    @override
    int get getter => 10;
}

```

Since `MyInterface` is a concrete class, we need to give `method()` an empty body and the getter must return something too. Interfaces should only provide the method signature and don't care about the implementation. This example compiles but it feels wrong because the class we're using as interface has members with a body, which goes against the nature of an interface.

## Good practice

Interfaces are sort of “templates” used to define how the structure of a class should be. For this reason, we think that an interface shouldn't be instantiated and thus you should prefer abstract classes over regular classes.

It's also very important to keep in mind that `extends` only allows a single superclass, since Dart doesn't support multiple inheritance. When using `implements` you have no such limitation and you can implement as many classes you want. For example:

```
abstract class Interface1 {  
    void method1();  
}  
  
abstract class Interface2 {  
    void method2();  
}  
  
class Example implements Interface1, Interface2 {  
    @override  
    void method1() {}  
  
    @override  
    void method2() {}  
}
```

A class can use both `extends` and `implements` together, but the order in which they are declared does matter. For example:

```
// OK  
class Example extends Interface1 implements Interface2 {  
    @override  
    void method() {}  
}  
  
// ERROR - 'extends' always goes before 'implements'  
class Example implements Interface2 extends Interface1 {  
    @override  
    void method() {}  
}
```

When using `implements` on a class, constructors do not matter because you need to override all public members except constructors.

#### 5.1.4 Good practices: `extends` vs `implements`

We have just seen that `extends` creates subclasses and `implements` is for interface-like behaviors. While they may seem similar, there are differences in how they work and their use cases. To get started, let's make a 1:1 comparison:

- When you use `class B extends A {}` you are not forced to override every method of the superclass `A`. Inheritance takes place, so you can override as many methods as you want.
- When you use `class B implements A {}` you are forced to override every method of class `A`. Inheritance does not occur because all methods implementations must be re-defined.

Here are the differences between `extends` and `implements` from a practical point of view:

- `extends`. Use it to get the typical object-oriented behavior (where public members can be overridden, if needed) and you wish to share common behaviors along the hierarchy. If two classes share the same library, you can also override library-private members, but that is not considered a good practice.
- `implements`. Use it when you just need to create the “contract” of a type without defining its concrete implementation. If two classes share the same library, you will be forced to also override library-private members, which is generally not a good practice. It would be better if classes treated as interface lived in their own file.

Abstract classes and interfaces adhere to object-oriented programming paradigms. Before moving on, we want to share two simple use cases to show how `extends` and `implements` can be used. In general:

- When you have one or more common behaviors for all the children along the hierarchy, use `extends` to take advantage of inheritance. For example, imagine you had to create a utility to read the contents of various file formats:

```
abstract class FileReader {
    final File file;
    const FileReader(this.file);

    String get path => file.path;
    bool get isPathAbsolute => file.isAbsolute;

    String readContents();
}
```

Regardless of the file format, in this example we always want to be able to retrieve the file path and whether it's absolute or not. To avoid repetitions, we can place members in this superclass so that any subclass will inherit both `path` and `isPathAbsolute`. Subclasses will only have to take care of the file parsing logic:

```

class TextReader extends FileReader {
  const TextReader(super.file);

  @override
  String readContents() => 'contents of a .txt file';
}

class DocxReader extends FileReader {
  const DocxReader(super.file);

  @override
  String readContents() => 'contents of a .docx file';
}

```

If we used `implements`, we would have needed to copy/paste the getters implementation across all types. Using inheritance is the right choice because it allows us to reuse existing code across the hierarchy and avoid duplications.

- When you don't have implementations to share across the hierarchy but you just need to define the structure, use `implements`. For example, we would treat a class as an interface if we had to create some sorting algorithms for collections of data:

```

abstract class Sorter {
  void sort();
  String get name;
}

class MergeSort implements Sorter {
  @override
  void sort() { /* merge sort implementation... */ }

  @override
  String get name => 'Merge sort';
}

class QuickSort implements Sorter {
  @override
  void sort() { /* quick sort implementation... */ }

  @override
  String get name => 'Quick sort';
}

```

In this example, we expect that all algorithms have a name and n implementation. They all share the same “structure” (a method and a getter), but each algorithm has its own name and logic. For this reason, using `implements` is a good idea because we don’t want to reuse code. For those of you that are familiar with design patterns, this example implements the “*Strategy pattern*”.

In general, use inheritance when you need to share reusable methods along the hierarchy. Prefer interfaces when there is no code to share and all types must have the same structure. In other words, abstract classes are meant to share behaviors while interfaces don’t.

### 5.1.5 Mixins

If abstract classes and interface were “brothers”, mixins would be their “cousin”. A `mixin` is a class that can be “merged” with other classes to reuse methods without using inheritance. For example, imagine you had these two classes in two separate hierarchies:

volleyball_team.dart	planet.dart
<pre>abstract class VolleyballTeam {   Country get country;    List&lt;Player&gt; lineUp();    double sphereVolume(double radius) {     const constants = 4/3 * 3.14;     return constants * pow(radius, 3);   }    String lowercase(String name) {     return name.toLowerCase();   } }</pre>	<pre>abstract class Planet {   bool get inSolarSystem;    List&lt;Satellite&gt; getSatellites();   PlanetInfo getPlanetDetails();    double sphereVolume(double radius) {     const constants = 4/3 * 3.14;     return constants * pow(radius, 3);   }    String lowercase(String name) {     return name.toLowerCase();   } }</pre>

As you can see, they share the same `sphereVolume` and `lowercase` methods. However, a volleyball team and a planet have nothing in common so creating a superclass wouldn’t make much sense. At the same time, we would also like to avoid code repetition.

In such cases, where you want to reuse methods without using inheritance, a `mixin` is the way to go. It basically is a class, without constructor, that can be “merged” with any other class to share its members. For example:

```

mixin SphereUtils {
  double sphereVolume(double radius) {
    const constants = 4/3 * 3.14;
    return constants * pow(radius, 3);
  }

  String lowercase(String name) {
    return name.toLowerCase();
  }
}

```

We have collected “common” methods inside `SphereUtils` and we can now mix it with other types to automatically “import” all of its members. We can rework our classes to use the `with` keyword and associate the mixin:

volleyball_team.dart	planet.dart
<pre> abstract class VolleyballTeam   with SphereUtils {   Country get country;    List&lt;Player&gt; lineUp(); } </pre>	<pre> abstract class Planet with SphereUtils {   bool get inSolarSystem;    List&lt;Satellite&gt; getSatellites();   PlanetInfo getPlanetDetails(); } </pre>

Even if you can’t see them, `VolleyballTeam` and `Planet` can now use both `sphereVolume` and `lowercase` as if they were regular class members. With this approach, we have avoided inheritance and reused existing code. For example, we could use `VolleyballTeam` in this way:

```

abstract class VolleyballTeam with SphereUtils {
  Country get country;
  List<Player> lineup();
}

class SomeVolleyballTeam extends VolleyballTeam {
  @override
  Country get country => const Italy();

  @override
  List<Player> lineup() => const [];
}

void main() => SomeVolleyballTeam().sphereVolume(10);

```

As you can see, even if the superclass doesn't explicitly define `sphereVolume(double)`, the method is automatically "imported" from the mixin. Note that regular classes cannot be used as mixin. You can use `with` only on a type that is declared with the `mixin` modifier. For example:

```
// 1. OK because 'MathUtils' is declared with 'mixin'.
mixin MathUtils {}
class Equation with MathUtils {}

// 2. Compiler error because 'MathUtils' is a 'class' and not a 'mixin'
class MathUtils {}
class Equation with MathUtils {} // Error.
```

In the example, notice that it is a compile-time error if you try to use `with` on a `class`. You can only mix types whose declaration contains the `mixin` modifier, as it happens in the first example. Since mixins shouldn't be instantiated, it is a compile-time error if you create a constructor in a mixin. More information about the `mixin` modifier is in *chapter 7 – Section 1.5 “mixin class”*. A class can mix with more than a single mixin. For example:

```
mixin Walking {
    void walk() {}
}

mixin Breathing {
    void breath() {}
}

mixin Coding {
    void code() {}
}

// 'Human' has 'walk' and 'breath'
abstract class Human with Walking, Breathing {}

// 'Developer' has 'walk', 'breath' and 'code'
class Developer extends Human with Coding {}

// 'Student' has 'walk', 'breath' and 'code'
class Student extends Developer {}
```

You can also constrain the usage of a `mixin` to a specific subtype with the `on` keyword. For example, we could force `Coding` to be only mixed with `Developer` subclasses:

```
mixin Coding on Developer {
    void code() {}
}
```

With this change, only those classes that subclass `Developer` can be mixed with `Coding`. You would get a compiler error if you tried to mix `Coding` with `Developer` itself or any other class that doesn't descend from it. For example:

```
// ERROR: 'Coding' can only be mixed with subclasses of 'Developer'  
class Developer extends Human with Coding {}  
  
// OK: 'Student' is a 'Developer' subclass so 'Coding' can be mixed  
class Student extends Developer with Coding {}
```

The `on` modifier can specify one or more types.

### 5.1.6 Extension methods

Extension methods provide a way to add functionalities to a library without changing its internal implementation. For example, imagine you created a `Fraction` class to work with rational numbers such as `1/3`. This is a possible implementation:

```
class Fraction {  
    final int num;  
    final int den;  
    const Fraction(this.num, this.den);  
  
    factory Fraction.fromString(String value) {  
        // Parses a string and returns, if possible, a 'Fraction' object...  
    }  
}  
  
void main() {  
    final fraction = Fraction.fromString('1/3');  
    print(fraction.num); // 1  
    print(fraction.den); // 3  
}
```

As we have seen in *chapter 4 – Section 2.5 “Good practices”*, this is a good approach because we use a `factory` to make some computations and then build an immutable object. For ease of use, you would also like to be able to initialize a `Fraction` object directly from a string:

```
final fraction = '1/3'.toFraction();
```

The problem is that `String` is located inside the Dart SDK, and we cannot add or remove methods from it. However, you can use the `extension` keyword to add methods from the outside, without changing the internal class definition. For example:

```

extension FractionExtension on String {
    Fraction toFraction() => Fraction.fromString(this);
}

void main() {
    final Fraction fraction = '1/3'.toFraction();
}

```

As you can see, it looks like the `String` class contains a `toFraction()` method. In reality, we have added a sort of “patch” from the outside using an extension method. In general, extension methods are used to add functionalities to libraries whose code can’t be modified by you.

### Note

---

`this` refers to the object on which the method is called. In our example, `this` is holding the '`1/3`' string object.

Extension methods can have more than a single method. They can also define variables, getters, setters, operator overloads, and static members.

#### 5.1.7 The covariant keyword

In some rare cases, you may need to override a class member (method, getter, or setter) and replace a type with a subtype. To better understand the situation, let’s use this simple hierarchy:

```

abstract class DomesticAnimal {
    const DomesticAnimal();
    void hunts(DomesticAnimal mammal);
}

class Rabbit extends DomesticAnimal {
    const Rabbit();

    @override
    void hunts(DomesticAnimal mammal) {}
}

class Dog extends DomesticAnimal {
    const Dog();

    @override
    void hunts(DomesticAnimal mammal) {}
}

```

We expect a dog to chase a rabbit. With the current hierarchy, we're allowed to pass `Rabbit` to the `Dog.hunt` method. However, we can also pass a `Dog` object to that same method and have a dog that chases another dog. For example:

```
void main() {
    const dog = Dog();
    const rabbit = Rabbit();

    // OK! It compiles and it's what we want
    dog.hunts(rabbit);

    // Hmm... It compiles but we want dogs to only hunt rabbits!
    dog.hunts(dog);
}
```

We may try to change the method signature in the `Dog` class, but that is not allowed. In other words, we cannot do this:

```
class Dog extends DomesticAnimal {
    const Dog();

    @override
    void hunts(Rabbit mammal) {} // Compiler error here!
}
```

To correctly compile the code, we have to use the `covariant` keyword before the type name. For example:

```
class Dog extends DomesticAnimal {
    const Dog();

    @override
    void hunts(covariant Rabbit mammal) {} // Now it works!
}
```

Using `covariant`, you can override a method and replace the parameter type with a subtype. With this change, a `Dog` object can only chase `Rabbit` and its subclasses (if any). If you try to pass a `Dog` object to `hunts` you will get a compile-time error:

```
void main() {
    const dog = Dog();
    const rabbit = Rabbit();

    dog.hunts(rabbit); // OK
    dog.hunts(dog); // Compiler error because 'hunts' has a covariant type
}
```

We could have achieved a similar result using a runtime check, but it would have been less efficient. While `covariant` works at compile-time, casts and type checks are executed at runtime. This is how you could have handled this use-case without `covariant`:

```
@override  
void hunts(DomesticAnimal mammal) {  
  if (mammal is! Rabbit) {  
    throw Exception('message');  
  }  
  // Execute your code...  
}
```

Even if the final result is similar, the behavior is entirely different because the method here throws an exception.

If you placed the `covariant` keyword in the superclass, then it would be inherited by all subclasses along the hierarchy. For example:

```
abstract class DomesticAnimal {  
  const DomesticAnimal();  
  
  // Since 'covariant' is here, all subclasses will inherit it  
  void hunts(covariant DomesticAnimal mammal);  
}  
  
class Rabbit extends DomesticAnimal {  
  const Rabbit();  
  
  @override  
  void hunts(SomeDomesticAnimalSubclass mammal) {} // Inherited covariance!  
}  
  
class Dog extends DomesticAnimal {  
  const Dog();  
  
  @override  
  void hunts(Rabbit mammal) {} // Inherited covariance!  
}
```

The superclass usually is the best place to define `covariant` parameters<sup>36</sup> because all subclasses inherit the behavior.

---

<sup>36</sup> <https://dart.dev/guides/language/sound-problems#the-covariant-keyword>

## 5.1.8 Good practices

Since this section was dense with information, you may find this summary helpful:

- Use inheritance with `extends` when you have common behaviors to share across multiple classes along the hierarchy.
- Use interfaces with `implements` when you don't have common behaviors to share, but you need to define the "*contract*" of a type (implementation details do not matter).
- When one or more identical methods can be re-used, do not copy-paste them. Create a new `mixin` to share code in various classes without using inheritance.
- When you want to add a method to a type you don't have access to (or you cannot change its internals), use an extension method.

Abstract classes, interfaces, and mixins are for "internal use" (use them when you have direct access to the code). Extension methods instead are for "external use" (use them when you can't access the code but you still want to add functionalities).

We also want to show some cases where `factory` constructors are helpful. For example, imagine you created a hierarchy of password encryption classes. For the sake of simplicity, we're keeping the method bodies short and straightforward:

```
abstract class StringEncryption {  
  const StringEncryption();  
  String encrypt(String password);  
}  
  
class RSAEncryption extends StringEncryption {  
  const RSAEncryption();  
  String encrypt(String password) => 'RSA: $password';  
}  
  
class AESEncryption extends StringEncryption {  
  const AESEncryption();  
  String encrypt(String password) => 'AES: $password';  
}
```

If you added a `factory` constructor in the `StringEncryption` base abstract class, you could return a default implementation of an encryption algorithm. For example:

```

abstract class StringEncryption {
  const StringEncryption();

  // Forwarding this factory to the 'AESEncryption' constructor
  const factory StringEncryption.aes() = AESEncryption;

  String encrypt(String password);
}

class RSAEncryption extends StringEncryption {
  const RSAEncryption();
  String encrypt(String password) => 'RSA: $password';
}

class AESEncryption extends StringEncryption {
  const AESEncryption();
  String encrypt(String password) => 'AES: $password';
}

void main() {
  const aes = StringEncryption.aes();

  print(aes.runtimeType); // AESEncryption
  print(aes.encrypt('abc')); // AES: abc
}

```

As you know, abstract classes cannot be instantiated. However, since `factory` constructors always return an object, we can use it to return a concrete subclass. In this example, we're using a `factory` to return a default implementation of `StringEncryption`. Look at this special syntax:

```
const factory StringEncryption.aes() = AESEncryption;
```

It redirects a `factory` constructor to another subclass constructor whose signature matches. This kind of constructor is called “*redirecting factory constructor*” and it's an equivalent of this:

```
factory StringEncryption.aes() => const AESEncryption();
```

The most significant difference is that a redirecting factory constructor can be `const` while a regular one cannot. For example:

```
// Can be 'const'
const factory StringEncryption.aes() = AESEncryption;
```

```
// Compiler error: 'const' is only allowed in redirecting factories
const factory StringEncryption.aes() => AESEncryption();
```

### 5.1.8.1 Smart casts

To explicitly cast a type into another, you need the `as` operator (which is evaluated at runtime). For this reason, an exception is thrown if the cast fails. For example:

```
void main() {
    final Object number = 10;

    final isEven = (number as int).isEven; // OK but dangerous
    final isNaN = (number as double).isNaN; // Ok but still dangerous
}
```

Even if the code compiles, it's very unsafe. For example, if `number` was a `bool` or a `String` the cast would fail at runtime. As a good practice, make sure to always protect your casts with a static type check. For example:

```
void main() {
    final Object number = 10;

    if (number is int) {
        final isEven = (number as int).isEven; // OK - safe operation
    }
}
```

The Dart compiler is smart enough to recognize that inside the `if` scope the cast is safe because you've tested types compatibility using `is`. As such, the variable is automatically promoted to the other type, and you don't need to make a manual cast. For example:

```
void main() {
    final Object number = 10;

    if (number is int) {
        // This is a 'smart cast'
        final isEven = number.isEven;
    }
}
```

Thanks to the smart cast, you can avoid `(number as int)` and just use `number` itself because, inside the scope of the `if` branch, `number` is automatically casted to `int`. Note that this is the preferred way of checking types because the `is` operator is evaluated at compile-time. Don't do this instead:

```
if (number.runtimeType.toString() == 'int') {
    // No smart cast here because the check is made at runtime
    final isEven = (number as int).isEven;
    print(isEven);
}
```

This is a runtime check, which is less efficient than a compile-time check. Use smart casts as much as possible. Also remember the `is!` operator, which works in the opposite way: it checks whether an object isn't of the given type.

## 5.2 The Object class

In Dart, all objects (except `null`) inherit from the `Object` class. We have covered the type system in *chapter 2 – Deep dive: Null safety in Dart*. For example:

What you write	What it actually becomes
<pre>class Fraction {     final int num;     final int den;     const Fraction(this.num, this.den); }</pre>	<pre>class Fraction extends Object {     final int num;     final int den;     const Fraction(this.num, this.den)         : super(); }</pre>

The choice of having `Object` as a superclass of all Dart objects comes from the fact that any classes must define a few important methods:

- `toString()`. This method is called when converting an object into a string. For example, the string interpolation syntax implicitly calls `toString()`. If you don't override this method, the default `Object` implementation is used:

```
class Example {  
    final int a;  
    const Example(this.a);  
  
}  
  
void main() {  
    print('${const Example(1)}'); // Prints "Instance of 'Example'"  
}
```

The default behavior is not very useful. The official `Object` documentation<sup>37</sup> says that you should consider overriding this method to return a meaningful string that describes the object. For example:

---

<sup>37</sup> <https://api.dart.dev/stable/2.18.4/dart-core/Object-class.html>

```

class Example {
    final int a;
    const Example(this.a);

    @override
    String toString() => 'Example(a: $a)';
}

void main() {
    const example = Example(1);
    print('$example'); // Prints "Example(a: 1)"
}

```

This looks much better because it actually describes the object structure. You can override `toString` as you wish as long as it returns meaningful information about the object.

- `operator==`. This operator is used to test the equality of two objects. The default behavior is to return `true` if the references point to the same object. Consider this object:

```

class Example {
    final int a;
    final String b;
    const Example(this.a, this.b);
}

```

In *chapter 4 – Section 2.3 “Constant constructors”* we saw that constant objects are created only once, and their references will always point to the same object. For example:

```

void main() {
    final check = const Example(1, '2') == const Example(1, '2');
    print(check); // true
}

```

The situation changes when we make the same comparison without a constant constructor. For example:

```

void main() {
    final check = Example(1, '2') == Example(1, '2');
    print(check); // false
}

```

The comparison now evaluates to `false` because references point to different objects. Even if both objects have the same internal representation, they have different memory locations

(and then references are also different). The default `operator==` implementation, defined in the `Object` class, constantly compares references but that's not what we want here. We want to override the operator to make a deep comparison:

```
class Example {  
    final int a;  
    final String b;  
    const Example(this.a, this.b);  
  
    @override  
    bool operator ==(Object other) {  
        if (identical(this, other)) {  
            return true;  
        }  
  
        if (other is Example) {  
            return a == other.a  
                && b == other.b  
                && runtimeType == other.runtimeType  
        }  
  
        return false;  
    }  
}
```

With this override, `operator==` compares two objects based on their internal configuration. The first check uses `identical` to recognize whether we're comparing the same object. If it fails, we move on with a smart cast to compare the object configuration deeply. With this change, this code now works as expected:

```
final example = Example(1, '2');  
  
// These cases are covered by the 'if(identical(this, other))' test  
print(const Example(1, '2') == const Example(1, '2')); // true  
print(example == example); // true  
  
// These cases are handled by the 'if (other is Example)' part  
print(Example(1, '2') == Example(1, '2')); // true  
print(const Example(1, '2') == example); // true  
print(Example(1, '') == example); // false
```

We recommend to override `operator==` with the same strategy we used in the example. A “shallow comparison” only compares references (which is the default `Object` behavior), but a “deep comparison” compares the object structures (which is done with a good override).

- `hashCode`. When overriding `operator==`, you must remember to also override `hashCode`<sup>38</sup>. A good implementation of this getter is fundamental to make objects work correctly inside collections. The hash code is a sort of “unique ID” that represents an object and its state. We recommend to override the `hashCode` getter using Dart’s hashing utilities. For example:

```
class Example {
  final int a;
  final String b;
  const Example(this.a, this.b);

  @override
  bool operator ==(Object other) { /* code... */ }

  @override
  int get hashCode => Object.hash(a, b);
}
```

The static `Object.hash` method implements a strong hashing algorithm you should rely on. Note that the `hash` function accepts two or more values. When your class only has a single member, then override the getter like this:

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

  @override
  bool operator ==(Object other) { /* code... */ }

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

When you need to override a collection (such as a `List<T>` or a `Map<K, V>`), prefer using the `hashAll` method instead. If `myList` was a `List<int>` for example, you would do this:

```
@override
int get hashCode => Object.hashAll(myList);
```

Note that `Object.hash(a)` and `Object.hashAll([a])` produce different results. It is very

<sup>38</sup> <https://dart.dev/guides/language/effective-dart/design#do-override-hashcode-if-you-override->

important that `operator==` and `hashCode` are always overridden together for consistency of behaviors in various use-cases.

There are a few particular cases to consider when you compare objects that contain other objects. For example, consider this case:

```
class A {  
    final int value;  
    const A(this.value);  
}  
  
class B {  
    final A a;  
    const B(this.a);  
}
```

Notice that neither `A` or `B` override the equality operator, so the default behavior is used. If we only created a deep comparison logic for class `B`, we would introduce a bug. For example:

```
class B {  
    final A a;  
    const B(this.a);  
  
    @override  
    bool operator ==(Object other) {  
        if (identical(this, other)) {  
            return true;  
        }  
        if (other is B) {  
            return a == other.a && runtimeType == other.runtimeType; // Problem here  
        }  
  
        return false;  
    }  
  
    @override  
    int get hashCode => a.hashCode;  
}
```

The problem is that class `A` does not override `operator==` (and `hashCode`), so it inherits the default comparison logic. By consequence, even if we've correctly overridden members in class `B`, class `A` is still comparing references rather than comparing the object structure. In fact:

```
print(B(const A(5)) == B(const A(5))); // true  
print(B(A(5)) == B(A(5))); // false
```

To solve this problem, you can either:

1. Implement `operator==` and `hashCode` in class `A`, which is the best option.
2. Workaround the issue and deeply compare class `A` within class `B`. This is not a maintainable solution because whenever the structure of class `A` changes, class `B` will also need a refactor.

For example:

```
if (other is B) {  
    return a.value == other.a.value && runtimeType == other.runtimeType;  
}
```

If `value` was a compound object too, you would need an even deeper comparison here. This is why option 1 is more maintainable.

Similar considerations also apply to collections, such as sets or lists. A good override of `operator==` and `hashCode` for a class that contains a collection must iterate over all items and compare them. For example:

```
class Example {  
    final String string;  
    final List<int> list;  
    const Example(this.string, this.list);  
  
    @override  
    bool operator ==(Object other) {  
        if (identical(this, other)) {  
            return true;  
        }  
  
        if (other is Example) {  
            for(var i = 0; i < list.length; ++i) {  
                if (list[i] != other.list[i]) {  
                    return false;  
                }  
            }  
            return string == other.string;  
        }  
  
        return false;  
    }  
  
    @override  
    int get hashCode => Object.hash(string, list);  
}
```

If the list contained objects that do not override `operator==` and `hashCode`, you'd have to (deeply) compare them yourself. As you've seen from the examples, "deep comparison" is not easy to deal with and may require considerable maintenance (especially in larger classes).

## Good practice

The `collection`<sup>39</sup> package, published and maintained by the Dart team, has a series of utility functions and classes tailored for collections (such as lists or maps). For example, we could use this package and rewrite the previous override as follows:

```
@override  
bool operator ==(Object other) {  
  if (identical(this, other)) {  
    return true;  
  }  
  
  if (other is Example) {  
    // This class is from the 'collection' package  
    const listEquality = ListEquality<int>();  
    return string == other.string &&  
      runtimeType == other.runtimeType &&  
      listEquality.equals(list, other.list);  
  }  
  
  return false;  
}
```

Rather than manually iterating over all elements yourself, you can use `ListEquality`. The package also contains various algorithms, binary search functionalities and more.

Make sure to ONLY override `operator==` and `hashCode` on immutable classes (classes with all their fields `final`) because they guarantee a consistent hash code result. Don't override `operator==` and `hashCode` on mutable classes because it breaks the equality contract that is assumed by the language.

---

<sup>39</sup> <https://pub.dev/packages/collection>

### 5.2.1 Comparable<T>

Besides overriding `operator==` and `hashCode`, you could (in addition) implement the `Comparable` interface for a complete comparison configuration. For example, this is how a full comparison logic looks like for a `Fraction` class:

```
class Fraction implements Comparable<Fraction> {
    final int numerator;
    final int denominator;
    const Fraction(this.numerator, this.denominator);

    double toDouble() => numerator / denominator;

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

    @override
    bool operator==(Object other) {
        if (identical(this, other)) {
            return true;
        }
        if (other is Fraction) {
            return numerator == other.numerator &&
                denominator == other.denominator &&
                runtimeType == other.runtimeType;
        }
        return false;
    }

    @override
    int get hashCode => Object.hash(numerator, denominator);
}
```

The `Comparable<T>` interface defines `compareTo()`, which is generally used to implement natural sorting logics, and it must be implemented as follows:

- if the natural ordering of the instance is smaller than the other instance, return a negative number;
- if the natural ordering of the instance is bigger than the other instance, return a positive number;

- return `0` otherwise.

By convention, `1` is returned as a positive number and `-1` as a negative one (but you could return anything else). In the `Fraction` class case, we have followed the above guidelines:

- a fraction is naturally smaller than another when its double representation is smaller than the other. As such, we use this condition:

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

- a fraction is naturally bigger than another when its double representation is bigger than the other. As such, we use this condition:

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

- the only missing case is when the two `double` representations are equal, so we return `0`.

Comparing floating point numbers with `operator==` is very risky. Two values that should be equal may differ due to arithmetic rounding errors. For this reason, we used `operator<` and `operator>` (that safely compares floating point numbers) and left the equality case as a fallback.

## 5.2.2 Records

A record is a sort of “anonymous class” that combines several values of any type into a single object. It has a series of fields, which can be named or positional, and its syntax looks like the argument list of a function. For example:

```
void main() {
  const record = ('Flutter', isOpenSource: true, 'Dart');

  print(record.$1); // Flutter
  print(record.isOpenSource); // true
  print(record.$2); // Dart
}
```

Once the record is created, its fields are accessed using getters. Named fields expose getters with the same name and positional fields expose getters with a dollar followed by an ordered number. For example, `isOpenSource` is a named getter while `$2` is a positional getter:

```
print(record.isOpenSource); // has the same name as the named field
print(record.$2); // '2' refers to the second positional field in the list
```

Positional fields are obtained with the dollar symbol followed by the 1-based index position in the fields list. For example, `$1` refers to the first positional field in the list (if any). Records are internally represented by the `Record` class, which cannot be constructed, extended, mixed, or implemented. For example:

```
const record = (1.2, name: 'abc', true, count: 3);
```

The above record declaration is internally converted into a `Record` class whose signature looks like the following class (this is just a simplification to give you an idea):

```
class _ extends Record {  
    double get $1;  
    String get name;  
    bool get $2;  
    int get count;  
}
```

All records are subtypes of the `Record` class and are immutable. The `Record` class has no instance members, except those inherited from `Object`, exposes no public constructors, and overrides both `operator==` and `hashCode`.

### Note

This is similar to how the `Function` class is the superclass for all function types.

To avoid ambiguity with parenthesized expressions, a record with only a single positional field must have a trailing comma. For example:

```
void main() {  
    const number = (1); // The number '1' with parenthesis  
    const record = (1,); // A record with a single field  
    const emptyRecord = (); // A record with no fields  
  
    print(number.runtimeType); // int  
    print(record.runtimeType); // (int)  
    print(record.runtimeType); // ()  
}
```

The `()` expression refers to the constant empty record with no fields. A record type looks similar to a function type's parameter list. The type is surrounded by parentheses and could contain comma-separated positional fields. As such, there are three possible combinations:

- Positional fields only:

```
(int, String, bool) myRecord = (1, 'A', true);
```

- Named fields only:

```
{int a, String b, bool c} myRecord = (a: 1, b: 'A', c: true);
```

- Positional and named fields together:

```
(int, {String b, bool c}) myRecord = (1, b: 'A', c: true);
```

Named fields must always be defined after positional fields. Type annotations work in the same way as function parameters list with the only exception that `required` is not allowed. Here is a few more relevant information:

- A record cannot define the same named field more than once. Furthermore, a named field is not allowed to start with an underscore.
- Since records extend the `Object` class, they inherit all of its methods. By consequence, there cannot be named fields whose name is one of `hashCode`, `runtimeType`, `noSuchMethod`, or `toString`.
- The runtime type of a record is determined by the runtime type of its fields. For example:

```
(Object, num, bool) record = (1, 2, true);

print(record is Object, int, bool)); // true
print(record is int, int, bool)); // true
```

On the practical side, records are often used to return multiple values from functions. They can create “classes on the fly” that pack values together in a single object. For example:

```
Future<(double, double, DateTime)> computeGeoPosition() async {
  final position = await currentPosition();
  return (position.latitude, position.longitude, DateTime.now());
}
```

The function returns a single object (a record) that “wraps” together multiple values of different types. You could have done the same without using a record, but it would have been more verbose. For example:

```

class GeoPositionData {
  final double latitude;
  final double longitude;
  final DateTime date;
  const GeoPositionData(this.latitude, this.longitude, this.date);

  // Missing 'operator==', 'hashCode' and 'toString' overrides!
}

Future<GeoPositionData> computeGeoPosition() async {
  final position = await currentPosition();

  return GeoPositionData(
    position.latitude,
    position.longitude,
    DateTime.now(),
  );
}

```

To exactly match the record behavior, `GeoPositionData` should also have overridden `operator==`, `hashCode`, and `toString`. Without considering the associated tests and maintenance efforts, the code would have been even more verbose. With records instead, we let Dart create a class “on the fly” for us that conveniently wraps two or more values in an immutable object.

## 5.3 Exceptions and errors

In certain cases, your code may need to throw exceptions to signal that an unexpected or erroneous behavior happened at runtime. Making sure that all exceptions are always handled is more than good practice. Unhandled exception cause programs to abruptly interrupt, which probably is one of the worst user experiences. For example:

```

class Fraction {
  final int numerator;
  final int denominator;
  Fraction(this.numerator, this.denominator) {
    if (denominator == 0) {
      throw Exception('The denominator cannot be zero.');
    }
  }
}

```

When creating a `Fraction` object, we must ensure its internal state is valid. In this case, throwing an exception in the constructor is the best place to check for the denominator value. The `throw`

keyword is used to throw exceptions, whose type should generally be `Exception`<sup>40</sup>. A better implementation of the previous code would be the following:

```
// 'Exception' can only be implemented
class FractionException implements Exception {
  final String message;
  const FractionException(this.message);

  @override
  String toString() => 'FractionException: $message';
}

class Fraction {
  final int numerator;
  final int denominator;
  const Fraction._(this.numerator, this.denominator);

  factory Fraction(int numerator, int denominator) {
    if (denominator == 0) {
      throw const FractionException('The denominator cannot be zero.');
    }

    return Fraction._(numerator, denominator);
  }
}
```

Rather than throwing a generic `Exception` type, you should create specific exception types. In this case, we could reuse the `FractionException` object in other parts of our `Fraction` class to signal other erroneous behaviors. We have used a `factory` constructor to follow Dart's recommended guidelines for this use-case<sup>41</sup>. Regardless, in Dart you can throw any object:

```
// All of these examples are ok, but throwing an 'Exception' type is preferred
throw 'Throwing a String object as exception';
throw 5;
throw false;
throw const MyCustomObject();
```

You can use `throw` in constructors, function, getters, and setters bodies. Except for `null`, you can throw any `Object` subtype.

---

<sup>40</sup> [https://dart-lang.github.io/linter/lints/only\\_throw\\_errors.html](https://dart-lang.github.io/linter/lints/only_throw_errors.html)

<sup>41</sup> <https://dart.dev/guides/language/effective-dart/design#avoid-public-late-final-fields-without-initializers>

### 5.3.1 try, on and catch

More than a good practice, handling exceptions is basically a requirement because your programs should never crash and abruptly close. Exceptions are handled with the `try` keyword, whose scope contains the code to be “protected”, and then a combination of `on` and/or `catch` keywords, to handle exceptions (if any). For example::

```
void main() {
    try {
        final fraction = Fraction(3, 0); // this code throws an exception
    } on FractionException {
        print('Ouch! Division by zero');
    }
}
```

We can say that this code is safe because the `Fraction` constructor may throw an exception, but there is at least one place where it’s handled. In the example we only handle `FractionException`, so any other kind of exception object won’t be caught. You can use `catch` to get a reference to the object being thrown. For example:

```
void main() {
    try {
        final fraction = Fraction(3, 0); // this code throws
    } on FractionException catch (fractionException) {
        print('$fractionException'); // 'The denominator cannot be zero.'
    }
}
```

If there were multiple exceptions to handle, we could have chained `on` and `catch` statements one after the other. For example:

```
void main() {
    try {
        final fraction = Fraction(3, 0);
    } on FractionException catch (fractionException) {
        print('$fractionException');
    } on FormatException {
        print('There was a format error in the input');
    } on SomeOtherException catch (otherException) {
        print('$otherException');
    }
}
```

Pay attention to not place superclasses at the top of the chain. For example, adding `Exception` at the beginning would “swallow” all the other `on` clauses:

```

try {
  final fraction = Fraction(3, 0);
} on Exception {
  print('Exception!'); // This gets printed
} on FractionException catch (fractionException) {
  print('$fractionException'); // This will NEVER be printed
}

```

The error propagation stops at the first `on` statement that matches, in order, the exception type. Since `FractionException` is an `Exception`, you will never reach the second branch. The same will also happen if you put `Object` at the top of the chain (it would also “swallow” `Exception`). In such cases, the supertype should always go last and be used as a fallback. For example:

```

try {
  final fraction = Fraction(3, 0);
} on FractionException catch (fractionException) {
  print('$fractionException');
} on Exception catch (exception){
  print('A generic exception is handled here');
} catch (exception){
  print('This catches anything else');
}

```

This isn’t the best way of handling exceptions<sup>42</sup>, and we will see why in the last section<sup>43</sup>. You should have total control over your program flow and you should be able to know which errors might be thrown. By consequence, you should always be able to provide an exact chain of `on` clauses that specifically handle all cases.

### 5.3.2 finally

We have seen that the scope of a `try` statement is able to “protect” the program flow by handling exceptions using the `on` and `catch` keywords. If no exceptions are thrown, no exception-handling clauses are executed. For example:

```

try {
  final fraction = Fraction(3, 5);
} on FractionException {
  print('Exception!');
}

```

---

<sup>42</sup> [https://dart-lang.github.io/linter/lints/avoid\\_catches\\_without\\_on\\_clauses.html](https://dart-lang.github.io/linter/lints/avoid_catches_without_on_clauses.html)

<sup>43</sup> section 3.4 “Exceptions, errors and good practices”

Since this code doesn't throw exceptions, the `print` statement is never executed. However, if you need to always execute some code at the end of a `try` block, you can use `finally`. For example:

```
try {
    final fraction = Fraction(3, 5);
    print('Done!');
} on FractionException {
    // This is executed only if a 'FractionException' object is thrown
    print('Exception!');
} finally {
    // This is ALWAYS executed, regardless an exception was thrown or not
    print('Finally!');
}
```

The scope of a `finally` block is always executed, no matter if an exception was thrown or not. In this example, “`Finally!`” would have been printed even if `FractionException` was thrown. A `finally` statement always goes last, so there cannot be `on` or `catch` clauses after it:

```
try {
    someFunction();
} finally { // Compiler error: 'finally' always goes last
    print('Finally');
} on Exception {
    print('Exception');
}
```

There can only be a single `finally` statement on the chain.

### 5.3.3 rethrow

The `rethrow` statement is used to re-throw an exception along with its stack trace. It is a compile-time error if you use `rethrow` outside of a catch. To understand how it can be helpful, let's consider this example:

```
void myFunction() {
    try {
        doSomethingRisky();
    } on Exception catch (e) {
        if (!canManage(e)) {
            throw e;
        }
        manageException(e);
    }
}
```

In the example, we're checking the `Exception` object because there are some cases where we do not want to handle the exception. Although the code is valid, there is a better way to write it:

```
try {
  doSomething();
} on Exception catch (e) {
  if (!canManage(e)) {
    rethrow; // use 'rethrow'
  }
  manageException (e);
}
```

The Dart team recommends to use the `rethrow` statement to rethrow a caught exception<sup>44</sup>. When you are in a `catch` block, don't use `throw`: prefer `rethrow` to improve terseness and preserve the original stack trace of the exception.

### 5.3.4 Exceptions, errors and good practices

Besides `Exception`, the Dart API also provides the `Error` type, which must be extended in case of program failures. Let's briefly compare the two classes to understand their use cases:

- `Exception`. Exceptions are unexpected, but not erroneous, behaviors of the program flow that should be handled in a `try-catch` block. The `Exception` type can only be implemented, not extended or mixed. For example:

```
try {
  final int num = getUserInput();
  final int den = getUserInput();

  final fraction = Fraction(num, den); // this code may throw
} on FractionException {
  doSomething();
}
```

A `FractionException` is thrown when the denominator is zero. This is a logic error, but not a fatal one that should crash the entire program. In other words, it is expected that there may be an erroneous situation where the denominator is zero. For example, we could manage this situation by asking the user to try again with a different input value.

---

<sup>44</sup> [https://dart-lang.github.io/linter/lints/use\\_rethrow\\_when\\_possible.html](https://dart-lang.github.io/linter/lints/use_rethrow_when_possible.html)

- **Error.** Errors are program failures made by the developer that should crash the program to avoid further problems. The `Error` type can only be subclassed, not implemented or mixed. If you pass an out-of-bounds list index for example, you get a `RangeError`:

```
void main() {
  const list = ['a', 'b', 'c'];

  // Throws a 'RangeError' because there is no item at index 100
  print(list[100]);
}
```

This is a failure made by the developer because the list has three items, and it's obvious that index `100` is out of the container bounds. Errors indicate that the program flow went into a not recoverable state, and the programmer must fix it. For this reason, `Error` types should never be handled in `try` blocks<sup>45</sup>.

When you see that an exception throws an `Error` object, you should find and fix the line of code that causes the problem. A `try` block should never handle `Error` types, even if it is allowed:

```
const list = ['a', 'b', 'c'];

try {
  print(list[100]);
} on Error {
  print('NO! Do NOT handle errors!');
}
```

The example compiles, but the Dart analyzer will warn: “*DON'T explicitly catch Error or types that implement it*”. For this reason, you also should avoid writing this code:

```
try {
  // something
} catch (_) {
  // Catches everything, including errors. Bad!
}

try {
  // something
} on Error {
  // Catches errors. Bad!
}
```

---

<sup>45</sup> [https://dart-lang.github.io/linter/lints/avoid\\_catching\\_errors.html](https://dart-lang.github.io/linter/lints/avoid_catching_errors.html)

The `catch` clause in this example catches everything, including errors, which is a bad practice. The `catch (_)` does “exception swallowing” because it catches any kind of thrown object (including `Error`). When you see an error, do not handle it: find the source of the problem and fix it.

## Deep dive: Language specification details

This section contains some interesting language features that are covered in the Dart specification document<sup>46</sup>. The contents of the document are very rigorous and formal, but here we’ve extracted and rephrased some curiosities about inheritance and exceptions:

- When you use `super` parameters in the subclass constructor, you should use the same name for consistency (even if it’s not a requirement). For example:

```
class A {  
    final int value;  
    const A(this.value);  
}  
  
class B extends A {  
    const B(super.number); // Uses a different name  
}
```

Notice that `B` uses `number` rather than `value`. This is not a compiler error: it’s just something that is not recommended.

- A `mixin` cannot be declared with the `abstract` keyword, but it’s allowed to define abstract methods (and the mixed class must provide an implementation). It is a compile-time error if the implemented class doesn’t override the mixin’s abstract method(s). For example:

```
mixin MyMixin {  
    void test();  
}  
  
class Example with MyMixin {  
    @override  
    void test() {} // 'test()' must be overridden  
}
```

---

<sup>46</sup> <https://dart.dev/guides/language/spec>

Since `test` is an abstract method and `Example` is not an abstract class, we must override it. We recommend always defining mixins with concrete methods.

- All `static` members are never inherited. For example, it would be a compile-time error if you had a `static` getter `v` and an instance setter that was also called `v`.
- The `rethrow` statement rethrows an exception and preserves the original stack trace. The `throw` statement resets the stack trace to the last thrown position.
- All instance variables introduce an implicit getter that could be overridden when you use `extends`. For example:

```
class A {  
    final int value;  
    const A(this.value);  
}  
  
class B extends A {  
    const B(super.value);  
  
    @override  
    int get value => super.value + 1;  
}  
  
void main() {  
    print(A(0).value); // 0  
    print(B(0).value); // 1  
}
```

For each instance variable, the compiler implicitly generates a getter with the same name. In fact, in our example, we could override the `value` getter (even if we didn't define it in `A`) because the `value` variable of the superclass automatically has an associated getter with the same name. The same also happens with `implements`:

```
class A {  
    final int value;  
    const A(this.value);  
}  
  
class B implements A {  
    @override  
    int get value => 10;  
}
```

In this case, we get a compile-time error if we don't override the `value` getter.

- When you implement a class that is on the same library and has library-private members, they must be overridden as well. For example, consider this case where both classes are on the same library:

```
abstract class A {  
  void publicMember();  
  void _privateMember();  
}  
  
class B implements A {  
  @override  
  void _privateMember() => print('private');  
  
  @override  
  void publicMember() => print('public');  
}
```

In this example, it is a compile-time error if you don't override `_privateMember`. However, if class `A` were in a separate file, you'd only have to override public members. For example:

```
// This is located in 'class_a.dart'  
abstract class A {  
  void publicMember();  
  void _privateMember();  
}  
  
// This is located in 'class_b.dart'  
class B implements A {  
  @override  
  void publicMember() => print('public');  
}
```

In this case, you aren't forced to override library-private members. It's still possible, but it's not mandatory anymore.

- When two or more unrelated libraries create records with the same set of fields, the type system understands that those records are the same type.

- It is a compile-time error if, in a record declaration, you try to use a field name that collides with the getter name of a positional field. For example, you’re not allowed to write `($1: 1)` because the named field `$1` collides with the getter for the first positional field.
- You can use `extend` only if the superclass has at least one generative constructor. Consider these two examples:

example.dart	sub_example.dart
<pre>class Example {   const Example(); }</pre>	<pre>class SubExample extends Example {   const SubExample(); }</pre>

This code compiles because `Example` has at least one generative constructor (the default one). If no generative constructors were available, you couldn’t use `extends`. For example:

example.dart	sub_example.dart
<pre>// Library-private constr. class Example {   const Example._(); }</pre>	<pre>// This code does NOT compile class SubExample extends Example {   const SubExample(); }</pre>

Now `Example` has no more public generative constructor and thus `SubExample` cannot use `extends` anymore. If you added a factory constructor the situation wouldn’t change because there would still be no generative constructor for `SubExample` to be called.

- The above example shows how to “block” inheritance on a class, but it’s not the only way you have to do it. Make sure to check *chapter 7 – Section 1 “Class modifiers”* to learn more about blocking inheritance and other properties of a class.

# 6 – Generics and collections

---

## 6.1 Introduction

Modern languages often support *generic* (or *parameterized*) programming and Dart is no exception. Consider the following scenario: you want to develop a cache to store a JSON response from a web service. It's quite simple:

```
class JsonCache {  
  final String json;  
  final DateTime creationTime;  
  const JsonCache(this.json, this.creationTime);  
}
```

After some time, your application gets more features and you realize you need to cache more data. Since you really want to avoid code duplication, you decide to change the cache name and make it “general purpose”. As such, you replace `String` with `Object`:

```
class DataCache {  
  final Object data;  
  final DateTime creationTime;  
  const DataCache(this.data, this.creationTime);  
}
```

Although this implementation would achieve the desired result, it would not be the most suitable solution for our purpose. There is a high risk of runtime exceptions because we will need to make explicit casts whenever the object is retrieved. There also is zero type-safety. For example:

```
void main() {  
  final stringCache = DataCache('hello', DateTime.now());  
  final intCache = DataCache(true, DateTime.now());  
  
  print(stringCache.data is String); // true  
  print(intCache.data is int); // false  
  
  final int value = intCache.data as int; // Runtime error  
  print('doubled = ${value * 2}');
```

Notice that we've initialized with a `bool` value a cache that is meant to hold integer numbers. This only is a logic error but not a compile-time error because `bool` actually is an `Object` subtype. Since there is no type-safety, this approach is very error-prone and forces us to make lots of runtime casts

and checks. Here is where generic data types come to the rescue. In Dart, you can use diamonds (`<>`) to specify a parameter that will be replaced with an actual type at compile-time. We could rewrite our cache in this safer way:

```
class DataCache<T> {  
  final T data;  
  final DateTime creationTime;  
  const DataCache(this.data, this.creationTime);  
}
```

The letter `T` is a placeholder for a type that will be defined in another moment. For example:

```
void main() {  
  final stringCache = DataCache<String>('hello', DateTime.now());  
  final intCache = DataCache<int>(true, DateTime.now()); // Compiler error!  
}
```

The letter `T` is replaced with the actual type at compile-time and we gain static type safety. In fact, we get a compiler error when we try to assign `true` to the integer cache because the `data` type is deduced as `int`. The runtime type also recognizes the generic type:

```
void main() {  
  final stringCache = DataCache<String>('hello', DateTime.now());  
  final intCache = DataCache<int>(10, DateTime.now());  
  
  print(stringCache.runtimeType); // DataCache<String>  
  print(intCache.runtimeType); // DataCache<int>  
}
```

If we wanted specialized types for different kinds of caches, we could use a `typedef`. It would define different types backed by specialized `DataCache` types. For example:

```
typedef IntCache = DataCache<int>;  
typedef BoolCache = DataCache<bool>;  
  
void main() {  
  final boolCache = BoolCache(true, DateTime.now()); // DataCache<bool>  
  final intCache = IntCache(10, DateTime.now()); // DataCache<int>  
  
  print(boolCache.runtimeType); // DataCache<String>  
  print(intCache.runtimeType); // DataCache<int>  
}
```

Generics provide a type-safe way to “generalize” classes and members. They favor code-reuse and static type-safety. It’s never a good idea to use `Object` instead of a parameterized type.

### 6.1.1 Usage

Generic classes behave in the same way as non-generic classes, so there really is nothing new to learn. There are a few things to point out, especially with subtypes. Look at this example:

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

  T get value => _object;
}

class LocalCache<K> extends Cache<K> {
  const LocalCache(super.object);
}

class CloudCache<A, OTHER> extends Cache<A> {
  const CloudCache(super.object);

  @override
  A get value => super.value;
}
```

Even if the base class uses the letter `T` as generic parameter, subclasses can use any other letter like `K` or `OTHER`. A subclass must define at least the same number of parameters as the superclass. We recommend to always specify types when creating generic<sup>47</sup> objects because the actual type may not always be inferred. For example:

```
class DataController<T> {
  const DataController();

  void processData(T value) {}

void main() {
  final good = DataController<int>();
  final bad = DataController();

  print(good.runtimeType); // DataController<int>
  print(bad.runtimeType); // DataController<dynamic>
}
```

---

<sup>47</sup> <https://dart.dev/guides/language/effective-dart/design#do-write-type-arguments-on-generic-invocations-that-arent-inferred>

The same rules also apply to functions declared outside of the class scope. For example:

```
T globalFunction<T>(T value) => value;

void main() {
    T nestedFunction<T>(T value) => value;
}
```

Generic functions must define the type(s) name(s) in the diamonds right after the name. You can also specify a generic function inside a non-generic class:

```
class Example {
    void method<T>(T value) => print(value);
}

void main() {
    Example().method<String>('Hello!'); // 'Hello!'
}
```

A type parameter may be suffixed with the `extends` keyword to specify an upper bound. Otherwise, the default upper bound is `Object`. For example, imagine you had to create caches that only store numbers. This would be a possible approach:

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

class LocalCache<K extends num> extends Cache<K> {}
class CloudCache<A extends num, B> extends Cache<A> {}

void main() {
    LocalCache<int>(); // OK
    LocalCache<double>(); // OK
    CloudCache<num, String>(); // OK, 'num' itself is still valid
    LocalCache<String>(); // Error! 'String' is NOT a 'num' subtype
}
```

The `extends` syntax tells the compiler that only subclasses of `num` can be passed in the diamonds. In fact, in our example, `int` and `double` are accepted but `String` is not. This is a strong rule because subclasses must always respect the constraints, if any. For example:

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

class LocalCache<K extends num> extends Cache<K> {} // OK, same constraint
class LocalCache<K extends int> extends Cache<K> {} // OK, stricter constraint
class LocalCache<K> extends Cache<K> {} // Compiler error
```

Since the superclass constrains `T` to be a `num`, subclasses must do the same or define stricter bounds. In fact, `int` (which is a `num` subtype) is allowed because it imposes a stricter requirement. The same rule is also valid for nullable type parameters:

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

class LocalCache<K extends num?> extends Cache<K> {} // OK
class LocalCache<K extends num> extends Cache<K> {} // OK
class LocalCache<K extends int?> extends Cache<K> {} // OK
class LocalCache<K extends double> extends Cache<K> {} // OK

class LocalCache<K> extends Cache<K>; // Compiler error
class LocalCache<K extends String> extends Cache<K>; // Compiler error
```

In this example, subclasses must only define stricter types than `num?`. Mixins and extension methods are also allowed to have generic parameters and follow the same rules as classes.

## 6.2 Collections

The Dart SDK is shipped with an implementation of common data structures and methods that use generic types to offer flexibility and type safety. For example, both lists and sets are subclasses of `Iterable<E>`, which provides basic methods for collections that are sequentially accessed, such as:

- `contains`: determines whether a given element is in the collection or not;
- `join`: converts each element of the list into a string and returns a concatenation;
- `length`: a getter that returns the total list length;
- `shuffle`: a method that randomly shuffles all elements in the collection;
- `first` and `last`: two getters that, respectively, return the first and the last element;
- and much more<sup>48</sup>!

Lists and sets are very similar but maps have different structures. The next sections describe how list, sets, and maps work in Dart.

### 6.2.1 List<T>

As you already know, Dart doesn't have "arrays" because you can only work with the `List<T>` type. There are three kinds of lists in Dart:

---

<sup>48</sup> <https://api.dart.dev/stable/2.15.1/dart-core/Iterable-class.html>

- Growable lists. They are created with the special `[]` literal, which can optionally be prefixed with the diamonds to specify the type. For example:

```
void main() {
    // Creates a 'List<int>'
    const intList = const [1, 2, 3];

    // Creates a 'List<num>'
    const numList = const <num>[1, 2.0, -3];
}
```

You can use methods like `add` and `remove` to respectively increase or decrease the list size. Internally, a buffer grows when needed to ensure that subsequent additions of elements can be executed in *amortized* constant time

- Fixed length lists. Their size is defined when creating the object and cannot be changed. An exception occurs when trying to change the length. For example:

```
void main() {
    // Creates a fixed-length List with three items and initializes each
    // of them to 0.
    final intList = List<int>.filled(3, 0);
    print(intList); // [0, 0, 0]

    intList.add(1); // Exception!
}
```

Of course, the list size must be a positive number. The `filled` named constructor initializes each item of the list with the given value. Alternatively, you can use the `List.generate` named constructor initialize each position with a different value:

```
void main() {
    // Creates a fixed-length List with three items and initializes them
    // with custom values.
    final intList = List<int>.generate(3, (i) => i);
    print(intList); // [1, 2, 3]

    intList.add(4); // Exception!
}
```

To create an empty but still fixed-length list, use the `List.empty()` constructor.

- **Unmodifiable** lists. They are unmodifiable copies of other lists whose lengths cannot be changed. If your list contains custom objects, the `List.unmodifiable` constructor DOES NOT make a deep copy of the items: it just creates a new list and passes references along.

```
void main() {
    final intList = [1, 2, 3];
    final unmodifiable = List<int>.unmodifiable(intList);
    unmodifiable.add(1); // Exception!
}
```

In this case, the constructor creates a new `List<int>` object and copies each item from the source list. If you don't want to create a new object, use the `UnmodifiableListView<T>` type. For example:

```
void main() {
    final intList = [1, 2, 3];
    final unmodifiable = UnmodifiableListView<int>(intList);
}
```

This class is just a wrapper that takes a reference of an existing list object and protects it against changes. No new objects are created, and no copies are made. To learn how to deep copy a list, see *chapter 4 – Deep dive “Cloning objects”*.

We recommend to always set `growable: false` when you know that the list size will never change. It allows internal optimizations. For example:

```
void main() {
    List<int>.filled(10, -1, growable: true);
    List<int>.empty(growable: true);
    List<int>.generate(10, (i) => -i, growable: false);
}
```

Growable lists can easily be manipulated using “*collection statements*”, a special syntax that is only allowed on lists and sets. For example, you could conditionally add items to a list using this simple syntax:

```
const hasCoffee = false;
final jobs = [
    'Welder',
    'Journalist',
    if (hasCoffee) 'Developer'
];

print(jobs); // 'Welder', 'Journalist'
```

You can place `if` conditions directly inside lists but without curly braces. The `else` statement is also allowed, but you still cannot use curly braces. For example:

```
const hasCoffee = false;
const jobs = [
  'Welder',
  'Journalist',
  if (hasCoffee) 'Developer' else 'Zombie'
];

print(jobs); // 'Welder', 'Journalist', 'Zombie'
```

In a very similar way, you can also use a `for` loop inside a growable list to add series of values. This syntax is called “*collection for*”. For example:

```
void main() {
  final values = [
    0, 1,
    for(var i = 2; i <= 5; ++i) i,
    6, 7
  ];

  print(values); // 0, 1, 2, 3, 4, 5, 6, 7
}
```

If you wanted to shallow copy an entire list inside another one, you could use a collection for or, even better, the *spread operator*. It is less verbose and was designed exactly for this purpose. For example:

```
void main() {
  const nonNullableList = [-2, -1, 0];
  List<int>? nullableList;

  // Using the 'spread operator'
  final spreadList = [...nonNullableList, 1, 2];
  print(spreadList); // -2, -1, 0, 1, 2

  // Using the 'null-aware spread operator'
  final nullSpreadList = [...?nullableList, 1, 2];
  print(nullSpreadList); // 1, 2
}
```

The spread operator `...` entirely copies the content of a list inside another one; the null-aware operator `...?` copies the contents only if the source list is not `null`. In our example, `nullableList` is `null`, and so nothing is added. You can use these operators anywhere in the list:

```
final spreadList1 = [0, ...nonNullableList, 1];
final spreadList2 = [...?nullableList, ...nonNullableList];
```

The `List<T>` class has an extensive API, which includes useful methods such as `add`, `clear`, `remove`, `sort`, and much more. For a complete overview of all the possible methods you can use, make sure to visit the official Dart documentation<sup>49</sup>.

## 6.2.2 Set<T>

A `Set<T>` is a generic container that cannot contain duplicates. For each object in the set, you can consider it present or absent. While lists are initialized with square brackets, sets require curly braces instead:

```
void main() {
  const set = <int>{1, 2, 3, 3, 4, 5};
  //                                     ^ NOT included in the set (it's a duplicate)

  print(set); // {1, 2, 3, 4, 5}
}
```

As you can see, since duplicates are not allowed, the number `3` is inserted only once. Unlike lists, sets don't have a fixed length, and they can only be created as growable containers. There are three constructors for sets:

- **Default.** We have just seen that a set can be initialized using curly braces. Alternatively, you could use the default constructor and manually insert items using the cascade notation. This is not very common:

```
void main() {
  const set = Set<int>()..add(1)..add(2)..add(2);
  print(set); // {1, 2}
}
```

Prefer initializing sets with curly braces. Note that when initializing an empty set with curly braces, you must use the diamonds to avoid ambiguity with maps:

```
// Set<int>
const set = <int>{};
// Map<dynamic, dynamic>
const map = {};
```

---

<sup>49</sup> <https://api.dart.dev/stable/dart-core/List-class.html>

In the first case, `<int>{}` initializes an empty `Set<int>` as we expect, but `{}` initializes a `Map<dynamic, dynamic>`. The diamonds are required to disambiguate between the two data structures since they share similar initialization syntaxes.

- **Identity.** Creates an empty set that can later be filled with values.

```
void main() {
    final set = Set<int>.identity();
    print(set); // {}
}
```

- **Unmodifiable.** Just as we saw for lists, this constructor creates a *shallow copy* of another set. As such, remember that `Set.unmodifiable` DOES NOT make deep copies:

```
void main() {
    const set = <int>{1, 2, 3};
    final unmodifiable = Set<int>.unmodifiable(set);

    unmodifiable.clear(); // Exception!
}
```

We cannot use `clear()` to clear the collection because it is unmodifiable. If we didn't want to create a new set object, we could use the `UnmodifiableSetView<T>` wrapper:

```
void main() {
    const set = <int>{1, 2, 3};
    final unmodifiable = UnmodifiableSetView<int>(set);
}
```

This class is a wrapper that takes a reference of an existing set object and protects it against changes. No new objects are created, and no copies are made. To learn how to deep copy a set, see *chapter 4 – Deep dive “Cloning objects”*.

Sets also support the *spread operator* and the *null-aware spread operator*. They work in the same way as lists, with the only difference that duplicates are discarded. For example:

```
const set = <int>{1, 2};
Set<int>? nullSet;

final spread = {-1, 0, 1, ...set}; // Spread operator
print(spread); // {-1, 0, 1, 2}

final nullSpread = {...?nullSet, ...set}; // Null-aware spread operator
print(nullSpread); // {1, 2}
```

Unlike lists, sets cannot be accessed via index using square brackets `[ ]`. You have to use the `elementAt(int)` method. Adding existing values on a set has no effect on the container itself. For example:

```
const set = <int>{1, 2, 3};  
print(set); // {1, 2, 3}  
  
set.add(2); // Does nothing because 2 already exists  
print(set); // {1, 2, 3}
```

Sets also support *collection for* and `if` conditions statements, with the same rules we've already seen for lists:

```
void main() {  
    const someCondition = true;  
  
    final set = <int>{  
        for(var i = 0; i < 5; ++i) i,  
        if (someCondition) 10,  
    };  
  
    print(set); // {0, 1, 2, 3, 4, 10}  
}
```

In *Section 6.3 – “Advanced topics”* we will see alternative implementations of lists and sets, along with performance considerations.

### 6.2.3 Map<K, V>

Also known as *dictionary*, a `Map<K, V>` is a generic collection that stores key-value pairs and doesn't allow duplicate keys. It is declared using curly braces as it happens with sets, and the diamonds expect two types:

```
void main() {  
    const map = <int, String>{0: 'A', 1: 'B', 1: 'Z', 2: 'C';  
  
    print(map); // {0: A, 1: Z, 2: C}  
}
```

In this example, the key is an `int` and the value is a `String`. Since the same key cannot appear twice on the map, the compiler keeps the last occurrence of the key. In fact, in our example the `{1: 'Z'}` key-value pair is kept in the map while `{1: 'B'}` is discarded. Constructors are very similar to the ones provided by lists and sets:

- Default. Creates an empty map, which is the equivalent of the `<K,V>{}` literal. For example:

```
void main() {
    final map = Map<int, String>();
    const map2 = <int, String>{};
}
```

The main difference is that the `<K,V>{}` literal can be constant while the `Map` constructor cannot (because it's a [factory](#)). Whenever possible, we recommend initializing maps with a `const` constructor.

- Iterable. You can build a map from lists or sets (both are `Iterable<T>` subclasses) with the `fromIterable` named constructor. Keys and values are computed using two callbacks that must return the respective types. For example:

```
void main() {
    const values = <int>{0, 1, 2}; // Could have been a List<int> as well

    final map = Map<int, String>.fromIterable(
        values,
        key: (element) => element + 1,
        value: (element) => '${element + 1}',
    );

    print(map); // {1: 1, 2: 2, 3: 3}
}
```

In our example, keys are integers and values are strings so the `key` and `value` callbacks must respectively return `int` and `String` types. The constructor traverses the entire set (or the list) and thus `element` represents each element in the collection.

- Iterables. The `fromIterables` constructor (note the plural) differs from the previous one because it builds a map from two iterables: one for the keys and one for the values. For example:

```
void main() {
    const rings = <bool>[false, false, true];
    const planets = <String>['Earth', 'Mars', 'Jupiter'];

    final map = Map<String, bool>.fromIterables(planets, rings);
    print(map); // {Earth: false, Mars: false, Jupiter: true}
}
```

The two iterables must have the same lengths otherwise an exception is thrown.

- **Entries.** You can fill maps with `MapEntry<K,V>` objects, which are key/value wrappers. This type does nothing more than hold the entries of a map:

```
void main() {
    const entries = [
        MapEntry<String, bool>('Earth', false),
        MapEntry<String, bool>('Mars', false),
        MapEntry<String, bool>('Jupiter', true),
    ];

    final map = Map<String, bool>.fromEntries(entries);

    print(map); // {Earth: false, Mars: false, Jupiter: true}
}
```

- **Unmodifiable.** As it happens for lists and sets, this constructor creates a *shallow copy* of the other map. As such, remember that `Map.unmodifiable` DOES NOT make a deep copy of the items:

```
void main() {
    const map = <double, bool>{1: true, 0.5: false, 0: true};
    final unmodifiable = Map<bool, double>.unmodifiable(map);

    unmodifiable.clear(); // Exception!
}
```

We cannot use `clear()` to clear the map because it is unmodifiable. If you don't want to create a new map object, use the `UnmodifiableMapView<T>` wrapper:

```
void main() {
    const map = <double, bool>{1: true, 0.5: false, 0: true};
    final unmodifiable = UnmodifiableMapView<double, bool>(map);
}
```

This class is a wrapper that takes a reference of an existing map object and protects it against changes. No new objects are created, and no copies are made. To learn how to deep copy a map, see *chapter 4 – Deep dive “Cloning objects”*.

The default `<K,V>{}` literal is the most used constructor. We prefer the `fromEntries` constructor over `fromIterable` and `fromIterables` because it's easier to read and does not throw exceptions. All map constructors do not allow duplicated keys: every time you insert a duplicate key, the old

entry is replaced with the new one. This also happens when inserting new values after the map creation. For example:

```
void main() {
    const map = <String, bool>{}; // Empty map

    map['Earth'] = false; // Add {'Earth': false}
    map['Mars'] = true; // Add {'Mars': true}
    map['Mars'] = false; // Replace {'Mars': true} with {'Mars': false}

    print(map); // {Earth: false, Mars: true}
}
```

We're using square brackets to insert a new key in the map and assign it a value (if the key were already in the map, the value would be overwritten). Alternatively, we could have used the [update](#) method to update already existing values in the map and handle the case where the key was not in the map. For example:

```
void main() {
    final map = <int, String>{1: '', 2: '2'};

    map..update(1, (value) => '1', ifAbsent: () => '0')
        ..update(3, (value) => '$value', ifAbsent: () => '0');

    print(map); // {1: 1, 2: 2, 3: 0}
}
```

We think that adding and updating map values with the `[ ]=` operator is easier to read. The [update](#) method is useful when the data update requires a more complicated logic than a simple value assignment. Maps also support *collection for* and *collection if* statements:

```
void main() {
    const someCondition = true;

    final map = <int, String>{
        for(var i = 0; i < 3; ++i) i: '$i',
        if (someCondition) 10: '10',
    };

    print(map); // {0: 0, 1: 1, 2: 2, 10: 10}
}
```

In the next section, we are going to dig a bit deeper into the [Map](#) implementation and explore some good practices for this container.

## 6.3 Advanced topics

The most important rule to keep in mind when working with sets and maps is to always override both `operator==` and `hashCode` with the correct implementation (we have already covered it in *chapter 5 – Section 5.2 “The Object class”*). Since `operator==` is used to find duplicates, you must correctly override its behavior otherwise the container might not behave as expected. For example:

```
class Example {  
    final int a;  
    final String b;  
    const Example(this.a, this.b);  
  
    @override  
    bool operator ==(Object other) {  
        // Very bad implementation  
        return false;  
    }  
  
    @override  
    int get hashCode => Object.hash(a, b);  
}  
  
void main() {  
    final set = <Example>{Example(1, '1'), Example(1, '1')};  
    print(set); // {(1, '1'), (1, '1')}
```

Since we've created a bad implementation of `operator==`, equal objects are still inserted because the set does not recognize them as duplicates. This is bad. Since our override always returns `false`, Dart will never detect duplicates. This is how the equality operator should be implemented:

```
bool operator ==(Object other) {  
    if (identical(this, other)) {  
        return true;  
    }  
  
    if (other is Example) {  
        return a == other.a && b == other.b && runtimeType == other.runtimeType;  
    }  
  
    return false;  
}
```

Now that we have correctly overridden `operator==`, the set inserts `Example(1, '1')` only once because duplicates are correctly detected:

```
void main() {  
    final set = <Example>{Example(1, '1'), Example(1, '1')};  
    print(set); // {(1, '1')}
```

The same rule is also valid for maps since they rely on `operator==` to store data correctly. In the next section, we will see that some map implementations also require a correct override of the `hashCode` getter, which is covered in *chapter 5 – Section 5.2 “The Object class”*. Lists do not care about object equality since they insert data without caring about duplicates.

## Good practice

Maps and sets should be used with classes that:

1. have all of their instance fields `final` (immutable classes);
2. override both `operator==` and `hashCode`.

Hash-based collections assume that the hash code of an object **never** changes and so you should really work with immutable classes only.<sup>50</sup>

Before moving on, we want to show an example where the collection spread operator is useful. Even if the example is made with a list, the same functionality also works with maps and sets. When you need to add two or more values to a list, use the *spread operator* (`...[ ]`), which integrates a new list into the existing one:

```
const rainTomorrow = true;  
  
const mountainKit = <String>[  
  'Food',  
  'Water',  
  if (rainTomorrow) ...[ // Adding multiple options  
    'Umbrella',  
    'Boots',  
  ],  
];  
  
print(mountainKit); // [Food, Water, Umbrella, Boots]
```

<sup>50</sup> <https://dart.dev/guides/language/effective-dart/design#avoid-defining-custom-equality-for-mutable-classes>

The example shows that we need to add two items to the list in case of rain. Since the *collection if* statement only returns a single value, we return a new list of items using `...[]` which adds an umbrella and a pair of boots to the kit.

### 6.3.1 Alternative implementations

As we've seen, the `List<T>` type represents an indexable collection of objects with a length. For example, accessing items by index is very fast but objects deletion at the middle of the list is quite slow. Depending on the use case, you may decide to use different container implementations. Here are the most notable classes implemented by Dart:

- `List<T>`. A container that stores an ordered collection of objects (meaning that the order in which elements are inserted is remembered). Data is inserted in amortized constant time because, on average, `add` calls take  $O(1)$  time. Deleting or inserting items in the middle of the list is a slow operation (even worse if deleting or adding an item at the head of the list).
- `Queue<T>`. A container that stores objects in a FIFO (first-in-first-out) ordering. It means that the first added element will always be the first one to be removed. Furthermore, elements are always inserted at the back and removed from the front. `Queue<T>` type is an abstract class implemented by:
  - `ListQueue<E>`. A queue implemented with a list. As such, adding elements is an  $O(1)$  operation on average but removing the front element is an  $O(n)$  operation, where  $n$  is the number of objects in the list.
  - `DoubleLinkedQueue`. A queue implemented using a double linked list. Adding and removing elements, regardless of the position, is always an  $O(1)$  operation (much better than the previous list-based implementation). The drawback is a relatively expensive dynamic allocation overhead.

The `Queue<T>` type is abstract, but it has a factory constructor that creates a `ListQueue` object by default.

- `LinkedList<E extends LinkedListEntry<E>>`. This is an implementation of a doubly linked list so inserting and removing elements at the edges is an  $O(1)$  operation. However, searching items and accessing them by index is an  $O(n)$  operation, where  $n$  is the total number of elements in the list.

```

class NumberEntry<T extends num> extends LinkedListEntry<NumberEntry> {
  final T value ;
  NumberEntry(this.value);

  @override
  String toString() => '$value';
}

void main() {
  final list = LinkedList<NumberEntry>()
    ..add(NumberEntry(1)) // Adds to the end of the list
    ..add(NumberEntry(2)) // Adds to the end of the list
    ..addFirst(NumberEntry(3)); // Adds to the beginning

  print(list); // (3, 1, 2)
}

```

All `LinkedList` children must be subclassed of `LinkedListEntry` because the container internally manages references to nodes in the list. There is no `const` constructor defined in the superclass.

There also are various implementations for sets. Under the hood, the `Set<T>()` constructor is a `factory` that returns a `LinkedHashSet` object. This piece of code is extracted from the Dart SDK:

```

abstract class Set<E> extends EfficientLengthIterable<E> {
  factory Set() = LinkedHashSet<E>;

  // ... plus other constructors and methods ...
}

```

The redirecting factory constructor creates a new `LinkedHashSet<T>` object, which is a `Set<T>` subclass. Here are all set implementations:

- `LinkedHashSet<T>`. A set that maintains the insertion order and detects duplicates using `operator==` and `hashCode`. To have amortized O(1) complexity in add, remove and search operations, the hash codes of the objects in the collection must be well distributed. In this regard, use the static `Object.hash` function to be safe.
- `HashSet<T>`. A set that doesn't maintain the insertion order and detects duplicates using `operator==` and `hashCode`. Add, remove, and search operations are O(1) if the hash codes are well distributed. The main difference with the previous type is that the iteration order of a `HashSet<T>` depends on the objects' hash codes. For example:

```

class Example {
    final String a;
    final bool b;
    const Example(this.a, this.b);

    @override
    bool operator ==(Object other) {
        if (identical(this, other)) {
            return true;
        }
        if (other is Example) {
            return a == other.a &&
                b == other.b &&
                runtimeType == other.runtimeType;;
        }
        return false;
    }

    @override
    int get hashCode => Object.hash(a, b);

    @override
    String toString() => '($a, $b)';
}

void main() {
    final linkedSet = LinkedHashSet<Example>()
        ..add(const Example('4', false))
        ..add(const Example('a', true))
        ..add(const Example('1', false));

    final notLinkedSet = HashSet<Example>()
        ..add(const Example('4', false))
        ..add(const Example('a', true))
        ..add(const Example('1', false));

    // {(4, false), (a, true), (1, false)}
    print(linkedSet);

    // {(1, false), (4, false), (a, true)}
    print(notLinkedSet);
}

```

The `LinkedHashSet` prints the values in insertion order. The `HashSet` instead has a different output because it traverses the set based on the `hashCode` of each object in the collection.

- `SplayTreeSet<T>`. This set is based on a self-balancing binary tree whose operations (most of them) happen in amortized logarithmic time. Objects are compared using the `compare` callback passed via constructor:

```

final set1 = SplayTreeSet<int>((a, b) => b.compareTo(a))
..add(1)
..add(2)
..add(3);

final set2 = SplayTreeSet<int>((a, b) => a - b)
..add(1)
..add(2)
..add(3);

print(set1); // {3, 2, 1}
print(set2); // {1, 2, 3}

```

The callback returns an integer that follows the same logic as the `Comparable.compareTo` method, which was covered in *chapter 5 – Section 5.2.1 “Comparable<T>”*. If you don't pass a callback in the constructor, all elements must implement the `Comparable<T>` interface.

As you may guess, maps also have more than a single implementation. The abstract `Map` class has a `factory` constructor that returns a `LinkedHashMap` object (the default implementation). Two other implementations are backed by the same structure used by sets:

- `LinkedHashMap<K,V>`. A map that maintains the insertion order and detects duplicates using `operator==` and `hashCode`. This implementation has O(1) complexity in add, remove, and search operations only if hash codes well distributed. In this regard, just use the static `Object.hash` function to be safe.
- `HashMap<K,V>`. A map that does not maintain the insertion order and detects duplicates using `operator==` and `hashCode`. Add, remove, and search operations are O(1) if the hash codes are well distributed. The main difference with a `LinkedHashMap<K, V>` is that the iteration order of this map may happen in any order, and it changes whenever the map is modified.
- `SplayTreeMap<K,V>`. This map is based on a self-balancing binary tree whose operations (most of them) happen in amortized logarithmic time. Objects are compared using the `compare` callback passed via constructor:

```

void main() {
  const entries = [
    MapEntry<int, String>(2, 's'),
    MapEntry<int, String>(1, 'a'),
    MapEntry<int, String>(5, 'none'),
  ];

  final map1 = SplayTreeMap<int, String>((a, b) => b.compareTo(a))
    ..addEntries(entries);

  final map2 = SplayTreeMap<int, String>((a, b) => a - b)
    ..addEntries(entries);

  print(map1); // {5: none, 2: s, 1: a}
  print(map2); // {1: a, 2: s, 5: none}
}

```

The callback returns an integer that follows the same logic as the `Comparable.compareTo` method, which is covered in *chapter 5 – Section 5.2.1 “Comparable<T>”*. If you don't pass a callback in the constructor, all elements must implement the `Comparable<T>` interface

### 6.3.2 Iterable<T> and iterators

The abstract `Iterable<T>` class is the superclass of both `List<T>` and `Set<T>`. Formally speaking, an iterable is a generic collection of values that can be accessed in sequence. It doesn't guarantee that reading elements by index will always be an efficient operation. In fact, there are performance differences between lists and sets for certain operations.

#### Note

`Map<K, V>` isn't an `Iterable<T>` subclass because a map is a different data structure that uses hash tables or trees to store data. The Dart API has helper methods to convert a map into a list or a set to read its contents sequentially. However, maps are not used to sequentially store and access items.

As we've already seen, an iterable object (such as a `List<T>` or a `Set<T>`) can be traversed in two ways. A `for` loop is good when you need to keep track of an index; a `for-in` loop is good when you need to traverse the entire collection without an index. The `for-in` loop syntax can only be used on iterable objects. For example:

```

void main() {
  const Iterable<int> list = [1, 2, 3];
  const Iterable<int> set = {1, 2, 3};
  const Map<int, String> map = {0: 'a', 1: 'b'};

  // OK - 'list' is an 'Iterable'
  for (final value in list) {
    print(value);
  }

  // OK - 'set' is an 'Iterable'
  for (final value in set) {
    print(value);
  }

  // Compiler error - 'map' is NOT an 'Iterable'
  for(final value in map) {
    print(value);
  }
}

```

The reason why `Iterable<T>` can be used in `for-in` loops is that it defines an `Iterator<T>`, a class (used as an interface) for getting items one at time. Under the hood, the compiler transforms the `for-in` syntax into a `while` loop that consumes the iterator. For example:

What you write	What the compiler generates
<pre> const list = &lt;int&gt;[1, 2, 3];  for (final value in list) {   print(value); } </pre>	<pre> const list = &lt;int&gt;[1, 2, 3];  final iterator = list.iterator; while (iterator.moveNext()) {   final value = iterator.current;   print(value); } </pre>

The `for-in` loop is converted into a `while` loop that uses an iterator. The `moveNext` method is used to access all items in the collection sequentially. An iterator is initially positioned before the first item of the container. To access the elements of the iterable, the iterator must advance using the `moveNext()` method, which returns `false` if no element is left.

Because of this mechanic, if you try to use the `for-in` syntax on an object that does not expose an iterator, you get a compiler error. For example, consider this code:

```

class Student {
    final String name;
    final String surname;
    const Student(this.name, this.surname);
}

class School {
    final List<Student> students;
    final String name;
    const School({required this.students, required this.name});
}

void main() {
    final school = School(
        students: const [Student('A', 'B'), Student('C', 'D')],
        name: 'Some school',
    );

    for (final student in school) { // <- Compiler error here
        print(student);
    }
}

```

This code doesn't compile because the `School` object doesn't expose an `Iterator<T>`, and so the compiler doesn't know how to read data sequentially. We could iterate over `school.students` to quickly solve the problem, but we want to show how the `School` type itself can be an iterator. To do this, we first need to extend the `Iterator<T>` class:

```

class SchoolIterator extends Iterator<Student> {
    final Iterable<Student> _iterable;
    int _index = 0;
    Student _current = const Student('', '');

    SchoolIterator(Iterable<Student> iterable) : _iterable = iterable;

    @override
    Student get current => _current;

    @override
    bool moveNext() {
        if (_index >= _iterable.length) { return false; }
        _current = _iterable.elementAt(_index);
        _index++;
        return true;
    }
}

```

The `Iterator<T>` type is an interface for getting items individually. The `current` getter returns the currently selected item in the iteration order. The `moveNext()` method updates the value to the next item and returns `false` if there are no more elements to visit in the list.

### Note

We've kept the `SchoolIterator` implementation easy to not make the example too difficult. In reality, the `moveNext()` method should ensure that the collection is not modified during the iteration. This requires some `if` guards that would make the code more complicated. To see an example of a complete implementation, check out the `ListIterable<T>` implementation in the Dart SDK<sup>51</sup>.

Now that we have created an iterator for the `School` object (`SchoolIterator`), we can “install” it by extending the `IterableBase<T>` type. For example:

```
class School extends IterableBase<Student> { // Add 'IterableBase<T>'  
  final List<Student> students;  
  final String name;  
  const School({required this.students, required this.name});  
  
  @override  
  Iterator<Student> get iterator => SchoolIterator(students); // override this  
}  
  
void main() {  
  final school = School(  
    students: const [Student('A', 'B'), Student('C', 'D')],  
    name: 'Some school',  
  );  
  
  // Now the for-in Loop syntax works on 'School' objects too  
  for(final student in school) {  
    print(student);  
  }  
}
```

The `IterableBase<T>` abstract class is used to “assign” an iterable object to a type so that it can be treated as if it was a list. You need to override the `iterator` getter and return your own iterator

---

<sup>51</sup> <https://github.com/dart-lang/sdk/blob/main/sdk/lib/internal/iterable.dart#L28>

(`SchoolIterator` in our case) to enable the `for-in` loop syntax. Overall, there is a lot of code to write and a notable degree of complexity. You generally don't need to manually create iterators because it'd be easier using existing collections. For example:

```
void main() {
    final school = School(
        students: const [Student('A', 'B'), Student('C', 'D')],
        name: 'Some school',
    );

    // This is easier: no need to create an iterator for 'School'
    for(final student in school.students) {
        print(student);
    }
}
```

This is much simpler, and it doesn't require new classes. Unless you feel like you want to create your own iterables, we recommend preferring using lists and sets.

It's also very easy to get confused with the terminology because `iter-able` and `iter-ator` have similar consonants. Working with iterables and iterators adds complexity to your code so we recommend relying on lists and sets instead.

### 6.3.3 from and of constructors

Lists, sets, and maps have two interesting constructors that might seem identical but have slightly different purposes. Both can be used to make shallow copies of the elements but `from` can also down-cast types. Here are some examples:

- `of` constructor. Creates a new list, set or map from the given source:

```
void main() {
    final list = List<int>.of([1, 2, 3]);
    final set = Set<int>.of(<int>{1, 2, 3});
    final map = Map<int, int>.of({1: 2, 3: 4});

    print(list); // [1, 2, 3]
    print(set); // {1, 2, 3}
    print(map); // {1: 2, 3: 4}
}
```

The `of` constructor makes a *shallow* copy of the source container. To make *deep* copies, do not use `of` and prefer, for example, iterating over all elements and call `copyWith`. More info on deep copying in *chapter 4 – Deep dive “Cloning objects”*.

- `from` constructor. Creates a new list, set or map from the given source and performs runtime down-casts if needed:

```
void main() {
  // Source
  final List<Object> list = <Object>[1, 2, 3];

  // Down-casting at runtime
  final List<int> otherList = List<int>.from(list);
  print(otherList.runtimeType); // List<int>
}
```

In this example, the `from` constructor makes a *shallow* copy and down-casts the source from `Object` to `int`, at runtime. If we tried to do the same using the `of` constructor, we would have got a compiler error. For example:

```
void main() {
  // Source
  final List<Object> list = <Object>[1, 2, 3];

  // Compiler error!
  final List<int> otherList = List<int>.of(list);
}
```

The same rules are also valid for sets and maps.

From the examples, we see that the `of` constructor is statically typed and the `from` constructor is dynamically typed. This doesn't necessarily mean that "`of` is better than `from`" because there could be some use cases where runtime casts are needed. We recommend following these guidelines:

- When you need to deep copy a list, don't use the `of` and `from` constructors. In *chapter 4 – Deep Dive: “Cloning objects”* we cover how to deep copy objects in Dart.
- When you need to make a shallow copy, and you also want to down-cast to another type, then use the `from` constructor.
- When you need to make a shallow copy and not make a down-cast, use the `of` constructor.

Note that calling the `of` constructor is the same as using the `[...list]` literal and the `toList()` or `toSet()` method. In general, both `of` and `from` can be replaced by list literals which are shorter and maybe cleaner.

## Deep dive: Collection methods

When you need to manipulate the data inside a collection, you can take advantage of the wide `Iterable<T>` and `Map<K,V>` APIs. For example, we can use collection methods to convert a list of integers into something else easily:

```
void main() {
    final sourceList = List<int>.generate(20, (i) => i);

    final List<String> newList = sourceList
        .where((value) => value.isEven) // Only keep even numbers
        .map((value) => '$value') // Convert int numbers into strings
        .toList(); // Build the list

    print(sourceList.runtimeType); // List<int>
    print(newList.runtimeType); // List<String>
}
```

In this example, we are creating a `List<String>` from a `List<int>`. Note that the source object (`sourceList`) is not modified at all. In particular:

- The `where` method is a sort of “filter” that only keeps those elements that satisfy the given expression. In this case, the `newList` list is only filled with even numbers.
- The `map` method is a sort of “converter” that changes the source type into another one. In this case, we’re converting `int` numbers to their `String` representation. The `map` method receives data that was filtered by `where` (so it’s only converting even numbers).
- The `toList()` method is a sort of “collector” that puts together the data processed by the previous functions and builds the final `List<String>` object.

You could have obtained the same result with loops, `if` statements, and some temporary variables. However, the code would have been more complex and more verbose. Our example is very easy to read. Here is a bit of terminology:

- The collection that needs to be manipulated is called source. It doesn’t get modified: it’s just used as source for subsequent operations.
- Methods that process data (such as `where` and `map`) are called intermediate methods. They can be chained to refine the result step by step.

There are a lot of intermediate methods, with the most relevant being:

- `where`: filters the elements keeping only those that evaluate the condition to `true`;
- `map`: converts a type into another;
- `followedBy`: lazily concatenates two iterables;
- `skip(int count)`: keeps all but the first `count` elements in the list;
- `take(int count)`: only keeps the first `count` elements on the list.

All of these functions return an `Iterable<T>`. At the end of the chain, you should use a terminator function to create the object that holds the data you've processed. The most relevant terminator functions are:

- `toList()`: creates a `List<T>` containing the result of the processed elements;
- `toSet()`: creates a `Set<T>` containing the result of the processed elements;
- `join()`: converts each element into a string and concatenates them;
- `contains(element)`: determines whether the `element` is contained in the iterable;
- `any`: determines whether any element satisfies the given condition;
- `every`: determines whether every element satisfies the given condition;
- `reduce`: reduces all the elements into a single value. For example:

```
void main() {
    // Creates a List with numbers from 0 to 9
    final source = List<int>.generate(10, (i) => i);

    // 'reduce' does 0 + 1 + 2 + 3 + ...
    final numbersSum = source.reduce((a, b) => a + b);

    print(numbersSum); // 45
}
```

This code returns the sum of all the numbers in the `source` list. Note that `reduce` can only be used in non-empty collections, and it must return the same collection type. In our example, `source` is a list of integers and so `reduce` must return an integer.

- `fold`: reduces all the elements into a single value without restrictions on types. For example:

```
void main() {  
    final source = <String>['x', 'yy', 'zzz'];  
  
    // 'fold' 1 + 2 + 3 (the sum of each string length)  
    final length = source.fold<int>(0, (int currCount, String element) {  
        return currCount + element.length;  
    });  
  
    print(length); // 6  
}
```

This code sums the lengths of all strings in the `source` list. The first parameter, `0`, is the initial value of the counter. The callback contains the current counter value (`currCount`) and the next element in the collection.

Note that `fold` can always be used while `reduce` only works in non-empty collections. Furthermore, `reduce` must return the same type of the iterable being processed. In other words, `reduce` is a more specific version of `fold`. For example, this code doesn't compile:

```
// Assume that 'source' is a 'List<String>' object  
final lengths = source.reduce((String a, String b) => a.length + b.length);
```

Both `a` and `b` are `String` types, and the expected returned value must be a `String` as well. The sum returns an integer, which doesn't match the iterable type (`String`). If you used `fold` instead, you'd be free to return any type you wanted.

## Deep dive: Covariance and contravariance in generics

The “covariant” and “contravariant” terms refer to the ability to respectively use a more derived or a less derived type than originally specified. These properties give generic types flexibility in usage and assignments. Let's make some examples to understand the difference better:

- Covariance allows you to use a more derived type than originally specified. For example, you could assign an instance of `List<int>` to a variable of type `List<num>`. This property could be seen as a “way to assign a subtype to a supertype”.

- Contravariance allows you to use a more generic type than originally specified. For example, you could assign an instance of `List<num>` to a variable of type `List<int>`. This property is the opposite of covariance and can be seen as a “way to assign a supertype to a subtype”.
- Invariance means that you can only use the type that was initially specified. For example, you cannot assign a `List<num>` to a variable of type `List<int>` or vice versa. This property could be seen as a “way to force the same type during assignment”.

Covariance and contravariance are generally referred as *variance*. In Dart, generic type parameters always are covariant, and there is no way to change this behavior. As such, you can only assign the same type or a subtype to a generic parameter. For example:

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

void main() {
  // OK - same type
  final List<A> list1 = <A>[];

  // OK - B is a subtype of A
  final List<A> list2 = <B>[];

  // Compiler error - A is NOT a subtype of B
  final List<B> list2 = <A>[];
}
```

From this example, we see that `B` is a subtype of `A` but also `List<B>` is a subtype of `List<A>` since generic parameters are covariant. In some ways, covariance looks like ordinary polymorphism, but contravariance seems counterintuitive.

Regardless, you don't have to worry about the difference because Dart only adopts the covariant behavior. You have no choice: contravariance is not supported, so you cannot get confused.

# 7 – Advanced language features

---

## 7.1 Class modifiers

The Dart language is permissive by default. In *chapter 5 – “Inheritance, core classes and exceptions”* we have already seen that any class can be constructed, extended, implemented, or mixed in. This flexibility sometimes introduces the possibility of breaking one or more invariants. For example, consider this class that represents a bank account:

```
class BankAccount {  
    double _balance = 0;  
  
    void deposit(double amount) {  
        // code ...  
    }  
    void withdraw(double amount) {  
        if (canWithdraw(amount)) {  
            // code ...  
        }  
    }  
  
    bool canWithdraw(double amount) => amount <= _balance;  
}
```

The `canWithdraw` method ensures that you can withdraw money from a bank account only if the balance has enough. In other words, the `_balance` variable should never be negative. The problem is that you can break this invariant very easily:

```
class BrokenAccount extends BankAccount {  
    @override  
    bool canWithdraw(double amount) => true; // this breaks the class logic  
}
```

Extending `BankAccount` gives you the possibility to override `canWithdraw` as you wish. In this case, we have broken the requirement of always having a positive balance. A good idea would be to prevent any user from subclassing `BankAccount` to avoid the risk of introducing unwanted bugs in the code. You will see a possible solution shortly. Here is another example:

```
/// Use this class with 'implements' to create a sorting algorithm.  
abstract class SortingAlgorithm {  
    void sort();  
    String get algorithmName;  
}
```

The inline documentation says that the `SortingAlgorithm` class should be implemented and not extended. However, thanks to Dart's permissive nature, the current configuration allows anyone to do this:

```
// OK
class MergeSort implements SortingAlgorithm {
    // code ...
}

// OK but 'SortingAlgorithm' should not be extended
class QuickSort extends SortingAlgorithm {
    // code...
}
```

This is valid Dart code, but it is logically wrong because `SortingAlgorithm` should not be extended. The bank account and the sorting algorithms examples highlight some cases where class modifiers are useful. The following list summarizes the capabilities of class modifiers:

- **base**: This modifier forbids the implementation of a class or a mixin that is located in another library. In practice, `base` forbids the usage of the `implements` keyword on a class or a mixin that is defined in a different Dart file.
- **interface**: This modifier forbids the extension of a class that is located in another library. It also forbids mixing in with a type that is located in another library. In practice, `interface` forbids the usage of `extends` on a class that is defined in a different Dart file. It also prohibits the usage of `with` on a mixin that is defined in a different Dart file.
- **final**: This modifier forbids extension, implementation, or mixing of a type that is located in another library. In practice, `final` forbids the usage of the `extends`, `implements`, and `with` on a type that is defined in a different Dart file.
- **sealed**: This modifier forbids extension, implementation, or mixing of a type that is located in another library. It also makes the type `abstract` and allows for exhaustiveness checking. In practice, it creates an abstract class that forbids the usage of the `extends`, `implements`, and `with` on a type that is defined in a different Dart file.
- **mixin class**: This declaration defines a class that can also be used as a mixin. In practice, it allows the usage of the `with` keyword on a class so that it can be used as if it was a mixin.

For example, we could use `final` for the bank account class to avoid extension or implementation:

```
final class BankAccount {  
    // code...  
}
```

For the sorting algorithm class, which should only be implemented but not extended, we could use the `interface` modifier. For example:

```
abstract interface class SortingAlgorithm {  
    // code ...  
}
```

Class modifiers do not apply to `enum`, `extension`, and `typedef` declarations.

### 7.1.1 base

The `base` modifier allows a class to be extended but not implemented. In other words, it takes away the “implicit” interface of a class so that the `implements` keyword doesn’t work anymore. This also transitively applies to all subtypes (if any). For example, imagine you created a generic HTTP client object for your application:

```
abstract class Client {  
    final int timeout;  
    const Client({  
        this.timeout = 5000,  
    }) : assert(timeout > 1000, 'Must be greater than 1 second');  
  
    String get hostName => 'website.com';  
    Future<String> httpRequest();  
}
```

This class makes two important assumptions. All subclasses will always have a connection timeout greater than a second and HTTP requests will always point to `website.com`. We expect that `Client` will always be extended to make sure that those two invariants will be inherited:

```
class Client1 extends Client<String> {  
    // code ...  
}  
  
class Client2 extends Client<String> {  
    // code ...  
}
```

However, thanks to Dart’s permissive nature, any class could use `implements` on the `Client` type and break the invariants. For example:

```
abstract class BrokenClient implements Client {
  @override
  int get timeout => 0;

  @override
  String get hostName => 'abc.com';
}
```

The `BrokenClient` class breaks the two assumptions of the `Client` type. In fact, it sets the timeout to `0` and changes the host name to a different one. To ensure that `Client` can always be extended but never implemented, we have to use the `base` keyword:

```
abstract base class Client { // Note the 'base' modifier
  final int timeout;
  const Client({
    this.timeout = 5000,
  }) : assert(timeout > 1000, 'Must be greater than 1 second');

  String get hostName => 'website.com';
  Future<String> httpRequest();
}
```

In this way, the compiler guarantees that no external libraries will implement the `Client` type. For example, the code on the right column throws a compile-time error:

http_client.dart	broken_client.dart
<pre>abstract base class Client {   final int timeout;   const Client({     this.timeout = 5000,   }) : assert(timeout &gt; 1000, '...');    String get hostName =&gt; 'website.com';   Future&lt;String&gt; httpRequest(); }</pre>	<pre>abstract class BrokenClient   implements Client { // compiler error   @override   int get timeout =&gt; 0;    @override   String get hostName =&gt; 'abc.com'; }</pre>

The `base` class modifier is useful when you want to force inheritance and block implementation. In other words, `base` should be used when subtypes must inherit the supertype's configuration and invariants. In our example, we wanted `Client` subtypes to point to the same server and have a timeout greater than a second. These “assumptions” (or “invariants”) can be broken by `implements` so we had to block it using `base`.

### 7.1.1.1 Technical overview

The `base` modifier disallows implementation outside of its own library. For example, consider these classes located in two different libraries:

person.dart	developer.dart
<pre>// Because of the base modifier, Person // can only be extended by classes that // are located in different .dart files base class Person {   final String name;   const Person(this.name);    String information(DateTime date) =&gt;     '\$name was born on \$date'; }</pre>	<pre>import 'person.dart';  // Compile-time error: cannot implement class Developer implements Person {   @override   DateTime get date =&gt; DateTime.now();    @override   String get name =&gt; 'Developer'; }</pre>

We get a compiler error if we try to use `implements` with the `Developer` type because `Person` is a base class defined in another library. In this way, you're guaranteed that the superclass constructor is always invoked. When you extend a `base` class, the subtype must have the same restrictions as the superclass. For example, this code is not valid:

person.dart	developer.dart
<pre>base class Person {   final String name;   const Person(this.name);    String information(DateTime date) =&gt;     '\$name was born on \$date'; }</pre>	<pre>import 'person.dart';  // Compile-time error too! class Developer extends Person {   const Developer(super.name);    List&lt;String&gt; get skills =&gt;     const ['Dart', 'Go']; }</pre>

This code is not valid because `Developer` breaks the base class guarantees. While `Person` can only be extended, `Developer` can be extended or implemented (which breaks the base class guarantee). In other words, the fact that a type cannot be implemented must be valid for all subclasses. To fix the above example, `Developer` must be either `base`, `final`, or `sealed`. For example:

## person.dart

```
base class Person {  
    final String name;  
    const Person(this.name);  
  
    String information(DateTime date) =>  
        '$name was born on $date';  
}
```

## developer.dart

```
import 'person.dart';  
  
// OK  
base class Developer extends Person {  
    const Developer(super.name);  
  
    List<String> get skills =>  
        const ['Dart', 'Go'];  
}
```

This code is now valid because the subclass (`Developer`) has the same or higher restrictions than its superclass (`Person`). In the next sections, you will see that the `final` and `sealed` modifiers block implementation, so they would have also worked here. The situation changes when classes are in the same library. For example, consider these two classes that are inside the `person.dart` library:

```
base class Person {  
    final String name;  
    const Person(this.name);  
  
    String information(DateTime date) => '$name was born on $date';  
}  
  
base class Developer implements Person { // OK  
    @override  
    String get name => 'Alberto!';  
  
    @override  
    String information(DateTime date) => '';  
}
```

When two types are in the same library (like in the above example), the restrictions are loosened. As you can see, even if `Person` has the `base` modifier, it can still be implemented. The same rules also apply to mixins: `base` forbids the implementation of a mixin that is located in another library:

## lib1.dart

```
base mixin Example {}
```

## lib2.dart

```
import 'person.dart';  
  
// Compile-time error!  
mixin Error implements Example {}
```

## 7.1.2 interface

The `interface` modifier allows a class to be implemented but not extended. In other words, it takes away the possibility of inheriting from a class because it forbids the usage of the `extends` keyword. For example, imagine you wanted to create some interface classes to serialize and deserialize Dart objects using popular data formats:

```
abstract class XMLConverter {  
    String toXML();  
    XMLConverter fromXML(String source);  
}  
  
abstract class JsonConverter {  
    String toJson();  
    JsonConverter fromJson(String source);  
}
```

These two types only define the API (or the “contract”) for converting objects between various data formats. For this reason, we want `JsonConverter` and `XMLConverted` to be implemented and not extended. For example:

```
class Class1 implements XMLConverter, JsonConverter {  
    // code ...  
}  
  
class Class2 implements JsonConverter {  
    // code ...  
}
```

Thanks to `implements`, a single class can use one or more converters. However, thanks to Dart’s permissive nature, any class could use `extends` on a converter type. It would be a problem in case two or more converters were needed for the same type:

```
// OK: only a single superclass  
class Class1 extends XMLConverter {  
    // code ...  
}  
  
// Compile-time error: only one superclass is allowed  
class Class1 extends XMLConverter, JsonConverter {  
    // code ...  
}
```

Dart does not support multiple inheritance, so the second example does not compile. It also doesn’t make sense to subclass a type that only defines a “contract” and not an actual implementation. To

make sure that converters can consistently be implemented but never extended, we have to add the `interface` keyword:

```
abstract interface class XMLConverter {  
    String toXML();  
    XMLConverter fromXML(String source);  
}  
  
abstract interface class JsonConverter {  
    String toJson();  
    JsonConverter fromJson(String source);  
}
```

In this way, the compiler guarantees that no external libraries will be able to extend the converters. For example, the code on the right column throws a compile-time error:

converters.dart	test_file.dart
<pre>abstract interface class XMLConverter {     String toXML();     XMLConverter fromXML(String source); }  abstract interface class JsonConverter {     String toJson();     JsonConverter fromJson(String source); }</pre>	<pre>// Error: 'extends' not allowed class Test extends JsonConverter {     // code..... }</pre>

An `abstract interface` class is very common: it is a type that cannot be constructed, extended or mixed-in (it can only be implemented). If you use the `interface` modifier alone, the class gains the possibility to be constructed.

The `interface` modifier should be used when you want to define the API (or the “contract”) of a type without giving an implementation. In our example, converter classes should be implemented because `implements` can define multiple types. Inheritance wouldn’t work if a class needed support for both XML and JSON conversion (because Dart only allows one supertype).

#### 7.1.2.1 Technical overview

The `interface` modifier disallows extension outside of its own library. For example, consider these two classes located in different libraries:

lib1.dart	lib2.dart
<pre>interface class Example {}</pre>	<pre>import 'lib1.dart';  // Compile-time error class Error extends Example {}</pre>

Since `Example` has the `interface` modifier, we get a compiler error if we try to use `extends` with the `Error` type. The situation changes when classes are in the same library. For example, you can always use `extends` when an interface class is in the same library as the other one:

```
// All classes are in the same library
interface class Example {}

class Example2 extends Example {} // OK
class Example3 implements Example2 {} // OK
base class Example4 extends Example {} // OK
interface class Example5 extends Example {} // OK
```

To sum up, `interface` blocks inheritance when types are on different libraries. If types are on the same library, there are some cases where an `interface` class can be extended. We have already seen this mechanism with the `base` keyword.

### 7.1.3 final

The `final` modifier removes extension, implementation, and mixing from a type. In other words, it creates a standalone class that forbids the usage of the `extends`, `implements`, and `with` keywords on a type. For example, imagine you wanted to create a class that converts a JSON string into a Dart object:

```
class UserParser {
  final String name;
  final int birthYear;
  const UserParser._({
    required this.name,
    required this.birthYear,
  });

  factory UserParser.fromJson(Map<String, Object?> source) => UserParser._(
    name: source['name'] as String,
    birthYear: source['birth-year'] as int,
  );
}
```

A `UserParser` object should only be created from a JSON string and so `fromJson` is the only public constructor. For example, this is how we expect users to build a `UserParser` object:

```
void main() {
    final decoded = jsonDecode('{"name": "Alberto", "birth-year": 1997}');
    final object = UserParser.fromJson(decoded);

    print('${object.name}, ${object.birthYear}'); // Alberto, 1997
}
```

We will cover the conversion library in detail in *chapter 9 – Section 1.1 “The conversion library”*. For now, it's enough to know that `jsonDecode` converts a JSON string into a Dart `Map`. The important part of this example is that we only want `UserParser` to be built using the `fromJson` constructor. However, thanks to Dart's permissive nature, anyone could break this requirement as follows:

```
class BrokenUserParser implements UserParser {
    final String name;
    final int birthYear;
    const BrokenUserParser({
        required this.name,
        required this.birthYear,
    });
}

// Allows the creation of 'UserParser' without using 'fromJson'
const UserParser object = BrokenUserParser(name: 'Alberto', birthYear: 1997);
```

Using `implements`, we can create `UserParser` objects without using a JSON string. That is not what we want because it breaks the class contract. Furthermore, `UserParser` should not have subtypes to avoid adding properties that are not even part of the JSON structure for example:

```
class EvenMoreBroken extends BrokenUserParser {
    const EvenMoreBroken() : super(name: '', birthYear: -1);
    String get surname => 'Miola';
}

// Allows the creation of 'UserParser' without using 'fromJson'
const UserParser object = EvenMoreBroken();

// 'surname' is not even defined in the JSON string!
final surname = (object as EvenMoreBroken).surname;
```

The `UserParser` type has no generative constructors, so it cannot have direct subclasses. However, this is not enough to protect it from inheritance. As you can see, we can use `extends` on a type that implements `UserParser` and add properties. To fix all these issues, we need the `final` modifier:

```

final class UserParser {
  final String name;
  final int birthYear;
  const UserParser._({
    required this.name,
    required this.birthYear,
  });

  factory UserParser.fromJson(Map<String, Object?> source) => UserParser._(
    name: source['name'] as String,
    birthYear: source['birth-year'] as int,
  );
}

```

Thanks to the `final` modifier, `UserParser` cannot be extended, implemented, or mixed-in in any way. In other words, you are guaranteed by the compiler that the type will always be unique, and `fromJson` will be the only available constructor. For example, the examples on the right column would always throw compile-time errors:

parser.dart	test_file.dart
<pre> final class UserParser {   final String name;   final int birthYear;   const UserParser._({     required this.name,     required this.birthYear,   });    factory UserParser.fromJson(...) =&gt;     UserParser._(...); } </pre>	<pre> // Error: 'extends' not allowed class Test1 extends UserParser {   // code..... }  // Error: 'implements' not allowed class Test2 implements UserParser {   // code..... }  class Test3 extends Test2 {   // code..... } </pre>

The `final` class modifier is useful when you want to block extension, implementation, and mixing. In other words, `final` should be used when you want to create a type that is unique. In our case, we wanted to make sure that `UserParser` objects could only be created by a single constructor. To “protect” the class from any inheritance or composition, we have made it final to remove any ability aside from construction.

A class with `abstract` and `final` modifiers cannot be created, extended, implemented, or mixed.

### 7.1.3.1 Technical overview

The `final` class modifier forbids extension, implementation, and mixing from a type. For example, consider these two libraries:

lib1.dart	lib2.dart
<pre>final class Example {}</pre>	<pre>import 'lib1.dart';  // Compile-time errors class Error1 extends Example {} class Error2 implements Example {} class Error3 with Example {}</pre>

Since `Example` is a `final` class, we get a compiler error if we try to extend, implement or mix it with any other type. As it happens with the `base` and `interface` modifiers, the situation changes when the types are in the same library. For example:

```
final class Vehicle {
  final int wheels;
  final String manufacturer;
  const Vehicle(this.wheels, this.manufacturer);
}

final class Car extends Vehicle { // OK
  const Car() : super(4, 'Ferrari');
}

final class Bike implements Vehicle { // Also OK
  @override
  String get manufacturer => 'Ducati'

  @override
  int get wheels => 2;
}
```

Since these classes are located in the same library, the restrictions on `Vehicle` are loosened. You can extend or implement a `final` class if the other type has the same limitations. For example, it would not be possible to do this:

```
// Compile-time error
class Bicycle extends Vehicle {
  const Bicycle() : super(2, 'Bottecchia');
}
```

The problem is that `Bicycle` has fewer restrictions than its `Vehicle` superclass. As we have already seen, the guarantees of a type must be consistent when extending or implementing. The `Bicycle` class must either be `base`, `final`, or `sealed` to not break the superclass constraints. The same also applies when you use implementation:

```
class Bicycle implements Vehicle { // This also is a compile-time error
  @override
  String get manufacturer => 'Bottecchia';

  @override
  int get wheels => 2;
}
```

Implementation is only allowed if `Bicycle` is `base`, `final` or `sealed`. This is because, outside of the library, the guarantees of the `Vehicle` final class cannot be broken.

#### 7.1.4 sealed

The `sealed` modifier is used to create an `abstract` class that cannot be extended, implemented, or mixed in outside of its library. In other words, it creates a “standalone” abstract class that forbids the usage of the `extends`, `implements`, and `with` keywords on a type located outside of its library. For example, consider these two libraries:

lib1.dart	lib2.dart
<pre>sealed class Class {}  sealed mixin Mixin {}</pre>	<pre>import 'lib1.dart';  // These all are compile-time errors Class(); class Error1 extends Class {} class Error2 implements Class {} class Error3 with Class {} class Error4 with Mixin {} class Error5 implements Mixin {}</pre>

Since `Class` is a `sealed` class, we get a compiler error if we try to construct, extend, implement, or mix it with any other type. The same restrictions also apply for sealed mixins (`Mixin`) because they cannot be implemented or mixed-in.

The `sealed` modifier is very important for exhaustiveness checking, which is covered in detail in *Section 2.2 – “Exhaustiveness checking”*. Since more context is needed to understand this modifier, we will defer examples and detailed explanations to the next section.

## 7.1.5 mixin class

In some rare cases, you may need a type to behave like a mixin or a class. The `mixin class` modifier merges the capabilities of both classes and mixins. For example, imagine having a utility class for calculating the perimeter of some polygons:

```
class PolygonUtils {  
    const PolygonUtils();  
  
    double triangleArea(double b, double h) => b * h / 2;  
    double squareArea(double l) => l * l;  
    double hexagonArea(double l) => 3 * sqrt(3) / 2 * l * l;  
}
```

Other classes can internally use this class to compute the areas of plane shapes. However, you could also need to plug these methods in an existing hierarchy to avoid copy-pasting code. For example, consider this case:

```
abstract class Shape {  
    // code...  
}  
  
class Circle extends Shape {  
    // code...  
}  
  
abstract class Polygon extends Shape {  
    // code...  
}  
  
class Pentagram extends Polygon {  
    // code ...  
}
```

It is not a good idea to make `PolygonUtils` a superclass of `Shape` because a circle is not a polygon. The utility could be a `Polygon` superclass, but extending concrete classes is not a good practice. If you use `PolygonUtils` somewhere else in your code, you cannot make it abstract. Creating a `mixin class` may be a good idea for this use case for two reasons:

1. You can construct and use `PolygonUtils` anywhere in your code:

```
void main() {  
    final utils = const PolygonUtils();  
    print('Hexagon area = ${utils.hexagonArea(6)}');  
}
```

2. You can plug-in `PolygonUtils` anywhere in a hierarchy and extended or implemented types can consume its API. In the previous example, you could fix the issue as follows:

```
abstract class Polygon extends Shape with PolygonUtils {
    // code...
}
```

This way, `Polygon` and all of its subtypes will have access to the `PolygonUtils` API. You will not have to copy-paste code or create dedicated methods for accessing the utility class.

You are allowed to use the `base` modifier in front of a `mixin class` declaration. This table highlights the main differences between a regular mixin and a class that is also a mixin:

mixin	mixin class
<ul style="list-style-type: none"> <li>• Cannot have any constructor (factory or generative).</li> <li>• Can implement but not extend other mixins.</li> <li>• Cannot be mixed with another mixin.</li> </ul>	<ul style="list-style-type: none"> <li>• Can have generative and/or factory constructors.</li> <li>• Can implement or extend other mixin classes (but not mixins).</li> <li>• Cannot be mixed with another mixin or mixin class.</li> </ul>

As you can see, a `mixin class` is more permissive than a `mixin`. A `mixin` can never be extended because its supertype must always be `Object`.

### 7.1.6 Considerations

These five class modifiers add up to the already existing `abstract` one for a total of six. The order in which modifiers are declared does matter. Note that some combinations don't make sense and thus are not allowed by the compiler, such as:

- The `base`, `interface`, and `final` modifiers all control the same two capabilities, so they are mutually exclusive. In practice, they cannot be combined together.
- Sealed classes cannot be extended, implemented, or mixed in from another library. Using `base`, `interface`, or `final` on a sealed type does not make sense and it is a compile-time error.

- Sealed classes are implicitly abstract, so you cannot use the `sealed` and `abstract` modifiers together. For example:

```
sealed class A {} // OK
sealed abstract class B {} // Error
```

- Mixins cannot be constructed, so the `mixin abstract` combination is redundant.
- A `mixin` or a `mixin class` is intended to be mixed in and so its declaration is not allowed to have the `final`, `sealed`, or `interface` modifiers.

This table summarizes all the allowed combinations of class modifiers; the capabilities refer to the cases where the types are not in the same library. The meaning of the “Exhaustive” column title is explained in detail in *Section 2.2 – “Exhaustiveness checking”*.

Modifier(s)	Construct	Extend	Implement	Mix in	Exhaustive
<code>class</code>	Yes	Yes	Yes	No	No
<code>base class</code>	Yes	Yes	No	No	No
<code>interface class</code>	Yes	No	Yes	No	No
<code>final class</code>	Yes	No	No	No	No
<code>sealed class</code>	No	No	No	No	Yes
<code>abstract class</code>	No	Yes	Yes	No	No
<code>abstract base class</code>	No	Yes	No	No	No
<code>abstract interface class</code>	No	No	Yes	No	No
<code>abstract final class</code>	No	No	No	No	No
<code>mixin class</code>	Yes	Yes	Yes	Yes	No
<code>base mixin class</code>	Yes	Yes	No	Yes	No
<code>abstract mixin class</code>	No	Yes	Yes	Yes	No
<code>abstract base mixin class</code>	No	Yes	No	Yes	No
<code>mixin</code>	No	No	Yes	Yes	No
<code>base mixin</code>	No	No	No	Yes	No

As we have already seen, modifiers have special behaviors when types share the same library. For this case, read the specific section of this chapter to see examples and considerations.

## 7.2 Patterns

Along with expressions and statements, Dart also supports patterns. You can see patterns as a group of language features that give you a compact and composable way to inspect an object or extract data from it. In practice, they are used to “manipulate objects and data using less code than usual”. Patterns can be used in various areas of the language, such as:

- Variable declarations. When you have a series of values held by a record type, you can get them back individually using the “record destructuring” pattern. For example:

```
final (a, b, c) = ("str", [1, 2], false);
//           ^^^^^^   ^^^^^^   ^^^^^^
//   destructuring      record object

print(a.runtimeType); // String
print(b.runtimeType); // List<int>
print(c.runtimeType); // bool
```

In practical terms, destructuring means assigning each object value to a new variable. In this case, we have created the `a`, `b` and `c` variables to hold the three values of the record. Note that these are two different things:

```
// This is destructuring
final (a, b) = (1, 2);
print(a.runtimeType); // int
print(b.runtimeType); // int

// This is assigning a 'Record' object to a variable
final record = (1, 2);
print(record.runtimeType); // (int, int)
```

Destructuring works when you create one variable per object or record value. The second case does not destructure the record: it just assigns the object to a variable. You can also destructure collections, such as lists and maps. For example:

```
final [a, b, c] = <int>[4, 8, 12];
print('sum = ${a + b + c}'); // sum = 24

final {'one': one, 'two': two} = <String, int>{'one': 1, 'two': 2};
print('sum = ${one + two}'); // sum = 3
```

Any other class that implements `List` or `Map` can also be destructured in this way. Here are some more destructuring examples:

```

final (a, b: b) = (1, b: 2); // a = 1 and b = 2
final Point(x: x, y: y) = Point(1, 2); // x = 1 and y = 2
for (var (i, j) = (5, 7); i > 0 && j > 1; i--) {
    j--;
    print('$i, $j');
}

```

As you can see, you can use patterns inside local variables declarations and all kinds of `for` statements.

- **Switch statements.** The `switch` statement allows patterns to be used within `case` branches. For example, you can match against a record object and destructure it immediately in a `case` declaration:

```

const record = (1, 2);

// Prints 'Match!'
switch (record) {
    case (1, 2):
        // ^^^^^^ destructuring pattern
        print('Match!');
    default:
        print('No match!');
}

```

In this example, the first `case` destructures the record object to check whether it matches or not. There also is the possibility to use an optional `when` guard clause to add a predicate after matching. For example:

```

const record = (1, 2);

// Prints 'No match!'
switch (record) {
    case (int a, int b) when a > b:
        // ^^^^^^^^^^^^^^^^^^^^^^^^^^ destructuring pattern + guard pattern
        print('Match!');
    default:
        print('No match!');
}

```

After the record is destructured, the `when` keyword is used to check if `a` is greater than `b`. In this example, `a` is smaller than `b` so the first branch is discarded. The program flow proceeds to the next clauses and it reaches the `default` one (which always matches).

- **Switch expressions.** A `switch` statement is mainly used to evaluate a conditional expression with multiple branches. It is a more readable alternative to a long series of `if-else` blocks. For example, consider this code:

```
enum Number {
    one,
    two,
    three,
}

int myFunction(Number number) {
    switch (number) {
        case Number.one:
            return 1;
        case Number.two:
            return 2;
        case Number.three:
            return 3;
    }
}
```

We could have created an `int` property inside the `enum`, but we will ignore it for the purpose of the example. The code can be rewritten in an equivalent form using a `switch expression` statement. For example:

```
int myFunction(Number number) {
    return switch (number) {
        Number.one => 1,
        Number.two => 2,
        Number.three => 3,
    };
}
```

The body has a series of cases separated by a comma. The `_` wildcard represents the default clause and, as usual, it must always appear at the bottom. For example:

```
// Assume that 'position' is an 'int'
final medal = switch (position) {
    1 => 'Gold',
    2 => 'Silver',
    3 => 'Bronze',
    _ => 'None', // The fallback 'default' clause
};
```

You can use the `when` guard inside any switch expression. For example, this code assigns the iron medal when `position` is between `4` and `10`:

```
final medal = switch (position) {
  1 => 'Gold',
  2 => 'Silver',
  3 => 'Bronze',
  _ when position > 3 && position <= 10 => 'Iron',
  _ => 'None',
};
```

If `position` is negative or greater than `10`, the `None` string is returned.

- If-case statements. Patterns can also appear inside `if` statements. Thanks to a combination of destructuring and `case` patterns, the `if` statement takes the properties of a `switch`. For example, consider this verbose code:

```
final json = {
  'skills': ['Dart', 'Flutter']
};

// Lots of code
if (json.containsKey('skills') && json.length == 1) {
  final skills = json['skills'];

  if (skills is List<String> && skills.length == 2) {
    final first = skills.first;
    final last = skills.last;
    print('1st $first | 2nd $last');
  }
}
```

This code makes various checks to ensure that there is a specific key and that the map has a list with exactly two `String` items. Thanks to pattern matching, we can reduce all of those `if` statements into an equivalent and shorter form:

```
final json = {
  'skills': ['Dart', 'Flutter']
};

if (json case {'skills': [String first, String last]}) {
  // ^^^^^^^^^^^^^^^^^^^^^^^^^^ destruct. pattern
  print('1st $first | 2nd $last');
}
```

The `case` clause destructures a record or a collection and evaluates if there is a match. In the example, the `if` statement checks whether the `json` object matches a map with a single key (named `skills`) and a list of two strings as a value. Here is another example:

```
final json = {
  'skills': ['Dart', null] // nullable element
};

if (json case {'skills': [String first, String? last]}) {
  print('1st $first | 2nd $last'); // ^ matches 'null'
}
```

If `null` is a possible element, we destructure the record or collection using the `?` operator. In the example, if we don't use the nullable `String?` type the match fails. You can use guard clauses as well. For example:

```
if (json case {'skills': [String a, String b]} when first.isNotEmpty) {
  print('1st $a | 2nd $b');
```

This is an equivalent of a `switch` statement that has a single case in its body.

You can use the `_` wildcard on a pattern if you don't need to have a variable that holds the value of a destructured type. For example:

```
const list = <int>[1, 2];

switch (list) {
  case [_]:
    print('Single item');
  case [_, _]:
    print('Two items');
  default:
    print('More than two items');
}
```

This code only makes some checks on the length of the list. Note that you can repeat the same `_` wildcard more than once. It is a compile-time error if you try to chain more than a single wildcard in a pattern. For example:

```
case [_, _]: // Error here
  print('Two items');
```

The wildcard also works in an if-case statement and has the same rules.

## 7.2.1 Detailed overview

A “pattern” is an expression that appears on the left side of an operator. Any pattern expression is only allowed to contain constant values. For example:

```
const one = 1;
final two = 2;

return switch (randomInteger) {
    0 => 'Zero', // OK
    //^ this and any other expression on the left is a 'pattern expression'
    one => 'One', // OK
    two => 'Bronze', // Error
    _ => 'Other',
};
```

All literals (like `0` or `true`) and `const` objects are compile-time constants and can be used in pattern expressions. Our example does not compile because `two` is not a compile-time constant (it is a final variable). The following list summarizes the patterns that are supported by the language. They can be used in any place that allows a pattern, such as an `if` statement or a `case` clause:

- Logical patterns. The “logical-or” pattern matches if at least one of the branches, separated by `||`, matches. For example:

```
final random = Random().nextInt(10);
final isBinary = switch (random) {
    0 || 1 => true, // it means '0' or '1'
    // ^^^ this is the 'logical-or' pattern
    _ => false, // fallback for all values that are not '0' or '1'
};
```

When `number` is either `0` or `1` the variable is set to `true`. The “logical-or” pattern can also be nested inside a destructuring pattern. For example, you could use it to match two or more values:

```
switch (list) {
    case [0.5 || -0.5, int b, var c]:
        // code...
    default:
        // code...
}
```

The “logical-and” pattern matches if all conditions, separated by `&&`, match. If one branch does not match, the remaining ones (if any) are not evaluated.

- Relational pattern. The “relational” pattern allows you to compare a value with a constant one using any of the equality or relational operators. This is often used with logical patterns. For example:

```
final number = Random().nextDouble();

return switch (number) {
    == 0 => 'Zero',
    > 0 && < 0.25 => 'Low',
    >= 0.25 && <= 0.75 => 'Mid',
    > 0.75 || < 1 => 'High',
    _ => 'Other',
};
```

Pay attention to not confuse relational operators (`>=`) with the return statement (`=>`).

- Cast pattern. The “cast” pattern checks the matched value against the given type. If the cast fails, an exception is thrown. For example:

```
(num, bool) record = (1, false);

final (number1 as int, bool condition1) = record; // OK
final (number2 as bool, bool condition2) = record; // Error
```

An exception is thrown in the second case because a `num` cannot be cast to a `bool`.

- Variable pattern. The “variable” pattern associates the matched value with a new variable. This is often used with the destructuring pattern to capture the values. For example:

```
const record = (1, 2, 3);

switch (record) {
    case (final a, var b, int c):
        print('$a, $b, $c'); // '1, 2, 3'
}
```

Note that variables end up initialized with the destructured values. You can use `final`, `var`, or write down the type explicitly. If you are interested in some destructured values, you can use the `_` wildcard to not bind a value to a name. For example:

```
case (final a, _, _):
    print('First value = $a'); // 'First value = 1'
```

There are some cases where the variable pattern is not allowed. For example, you cannot use it in a declaration context or an assignment context:

```
final [final bad] = [1]; // Error
final [good] = [1]; // OK

(final no1, var no2) = (0, 1); // Error
final (ok1, ok2) = (0, 1); // OK
```

In other words, type inference does not work with assignments and declarations.

- **Null patterns.** The “null-check” pattern matches if the value is not `null`. This may seem a bit counterintuitive, but the nullable variable is used to make sure that the type is non-nullable. For example:

```
String? value;

switch (value) {
  case var s?:
    print('$s is String');
  case var s:
    print('$s is String?');
}
```

The “null-assert” pattern is similar in that it permits non-null values to match. However, it throws in case the value is `null`. For example:

```
(int? x, int? y) record = (1, 2);

// Converts variables from 'int?' to 'int'
final (a!, b!) = record;
```

As always, an exception is thrown if you use the bang operator (`!`) on a `null` value.

- **List pattern.** The “list” pattern matches an object that implements `List` and extracts all its elements by position. For example:

```
const list = <int>[1, 2, 3];

final [a, b, c] = list;
```

In a list pattern, you might include a “rest element”. It allows matching lists of any length. For example:

```
final [a, b, ...rest, c] = [1, 2, 3, 4, 5, 6];
print('Values = $a $b $rest $c'); // 'Values = 1 2 [3, 4, 5] 6'
```

The rest element pattern uses the spread operator to “collect” in a new `List<T>` object all values that are not destructured by other variables. The rest element pattern may also have a sub-pattern.

- **Map pattern.** The “map” pattern matches an object that implements `Map` and gets values by key from it. For example:

```
const map = {'a': 1, 'b': 2};
final {'a': x, 'b': y} = map;
```

If you want to ensure that a map has a given set of keys and no others, you could check the length in a `when` guard. For example:

```
switch (map) {
  case {'a': _, 'b': _} when map.length == 2:
    print('Only 2 members');
}
```

The map pattern requires keys to match. If you used different values for the keys, the match would fail. For example:

```
const map = {'a': 1, 'b': 2};

switch (map) {
  case {'a': _, 'x': _}: // Does NOT match because 'x' is not a key
    print('Only 2 members');
}
```

- **Record pattern.** A “record” pattern matches a record object and destructures all of its fields. The match fails if the pattern structure doesn’t match the record shape. For example:

```
const record = (a: 1, 'value');

final (a: int a, String b) = record;
```

This is the standard way of destructuring a record. However, if you want to assign a value to a variable of the same name, you can use a shorter syntax:

```
final (:a, b) = record; // Equivalent, with untyped variables
final (:int a, String b) = record; // Equivalent, with typed variables
```

The `:a` syntax indicates that you want to create a new variable with the same name as the record value. You are allowed to use sub-patterns within the record pattern declaration.

- Object pattern. An “object” pattern matches the values of an object instance and extracts information from it. In other words, this pattern is used to destructure data from arbitrary objects using its internal properties. For example:

```
class Point {  
    final int x;  
    final int y;  
    const Point(this.x, this.y);  
}  
  
void main() {  
    const point = Point(1, 2);  
  
    switch (point) {  
        case Point(x: 0, y: 0): print('Origin');  
        case Point(x: final x, y: final y) when x == y: print('Diagonal');  
        default: print('${point.x} ${point.y}');  
    }  
}
```

In this example, we destructure the `Point` object using a familiar syntax. As it happens with the record pattern, you can omit the value name and automatically infer it using the variable pattern. For example:

```
case Point(:final x, :final y) when x == y:
```

Of course, this syntax is allowed on switch expressions and switch statements. Note that it is a compile-time error if two named fields have the same name.

As it happens with parenthesized expressions, parenthesis can be used with patterns to control precedence. The next section covers the “exhaustiveness checking” topic, which is closely related to patterns and the `switch` statement. You can match against a type and check for its properties. For example:

```
switch (const <int>[1, 2]) {  
    case [1, _] && [_, < 4]:  
        print('A');  
    case [int(isEven: false), var a]:  
        print('B ($a)');  
}
```

We have used the `int(...)` syntax to check whether the first element is an integer number and its `isEven` property is set to `false`.

## 7.2.2 Exhaustiveness checking

Both switch statements and switch expressions must be **exhaustive**. From a practical point of view, a `switch` is exhaustive if its `case` clauses cover all possible combinations of values. For example, a boolean can only be either `true` or `false`: there are no other allowed values. By consequence, the following code is valid:

```
// Generates a random boolean value
final bool value = Random().nextBool();

// This 'switch' statement is exhaustive
switch (value) {
    case true: print('True');
    case false: print('False');
}
```

This `switch` is exhaustive because its `case` clauses cover all the possible values of the `bool` type. If all cases are not covered, a compile-time error happens. For example, to break the above example we could make `value` a nullable type:

```
// Note that this is a nullable 'bool?' value
final bool? value = Random().nextBool();

// This 'switch' statement is NOT exhaustive anymore
switch (value) {
    case true: print('True');
    case false: print('False');
}
```

The problem here is that a `bool?` type allows three possible values: `true`, `false`, and `null`. As such, the `switch` is not exhaustive anymore because it's not handling the `null` case. There are two ways to fix the issue:

1. Include `null` in the cases list:

```
switch (value) {
    case true: print('True');
    case false: print('False');
    case null: print('Null');
}
```

2. Add a `default` clause as fallback if all of the previous matches failed:

```
switch (value) {
  case true: print('True');
  case false: print('False');
  default: print('Null');
}
```

To make a `switch` exhaustive, you have to cover all possible cases by hand or add a `default` clause that will act as a fallback. Of course, exhaustiveness checking is not only limited to boolean values. For example, the same rules are also valid for enumerated values:

```
enum Example { a, b, c }

String function() {
  final example = Example.b;

  return switch (example) {
    Value.a => 'A',
    Value.b => 'B',
    Value.c => 'C',
  };
}
```

You are guaranteed that the values of an `enum` can only be defined within its scope. Consequently, we can create an exhaustive `switch` statement that covers all the possible cases. When you work with user-defined types, the situation changes. Consider this example, where all the code is written (for simplicity) inside the same library:

```
// sports.dart
abstract class Sport {}
class Football extends Sport {}
class Hockey extends Sport {}
class Volleyball extends Sport {}

String myFunction(Sport sport) =>
  switch (sport) { // Compiler error
    Football _ => 'Grass',
    Hockey _ => 'Ice',
    Volleyball _ => 'PVC',
  };
}
```

The `switch` is not exhaustive because the code doesn't guarantee that `Sport` will always have three subtypes. As it stands, all cases are covered but the hierarchy may be expanded at any time. If you had published this library for example, someone could define one or more subclasses in the future:

```
// another_library.dart
import 'sports.dart';

class Golf extends Sport {}
class Bowling extends Sport {}
```

At this point, the `switch` would be missing a case for `Golf` and `Bowling`. In other words, Dart is not able to consider the `switch` exhaustive because `Sport` does not define a “closed” set of subtypes. There are two possibilities to make the `switch` exhaustive:

1. As we have already seen, a `default` clause makes a `switch` exhaustive by design. In fact, it is used as a fallback in case all the previous matches fail. For example:

```
return switch (sport) {
    Football _ => 'Ball',
    Hockey _ => 'Puck',
    Volleyball _ => 'Ball',
    // This is needed to make the switch exhaustive
    _ => 'Unknown',
};
```

Any subtype of `Sport` that doesn’t fall under the previous branches will match the default case. This approach is good when you don’t need a variable that references the matched value.

2. The `sealed` modifier, covered before in *Section 1.4 – “sealed”*, serves this purpose. It creates a closed set of types that can only be expanded within the same library. No code from the outside can create new subclasses. For example:

```
// Make this 'sealed'
sealed class Sport {}

class Football extends Sport {}
class Hockey extends Sport {}
class Volleyball extends Sport {}

String function(Sport sport) {
    // OK because 'Sport' is a sealed class
    return switch (sport) {
        Football _ => 'Ball',
        Hockey _ => 'Puck',
        Volleyball _ => 'Ball',
    };
}
```

Since `Sport` is now a sealed type, it can only be extended, implemented, or mixed in within the same library. Thanks to this condition, Dart knows that the `switch` is exhaustive for sure because there is no way to add new subtypes from another library.

To sum up, exhaustiveness is about making sure that all possible values of a type are covered. There are some built-in types, such as booleans and enumerations, that are exhaustive by design. Other types can be made exhaustive using the `sealed` class modifier.

## Deep dive: Language specification details

This section contains some interesting language features that are covered in the Dart specification document<sup>52</sup>. The contents of the document are very rigorous and formal, but here we've extracted and rephrased some curiosities about variables and type assignments:

- A trivial generative constructor is a constructor that is not a redirecting one, has no body, has no parameters, has no initializer list, and has no `external` keyword. A `mixin class` is only allowed to have trivial generative constructors. Factory constructors are allowed. For example:

```
mixin class Example {
    int? x;

    // Trivial generative constructors:
    Example(); // OK
    Example.named(); // OK

    // Non-trivial generative constructors:
    Example(int x); // Error
    Example(this.x); // Error
    Example() {} // Error
    Example() : x = 0; // Error
    Example() : assert(true); // Error
    Example() : super(); // Error
    Example() : this.named(); // Error

    // Factories
    factory Example.a() => Example(); // OK
    factory Example.b() = Example; // OK
}
```

---

<sup>52</sup> <https://dart.dev/guides/language/spec>

Constant constructors are also allowed. Trivial generative constructors do not affect object construction, so they can be ignored and omitted when the `mixin class` is used as a `mixin`. However, they allow the `mixin class` declaration to also be used as a superclass, even for subclasses with constant constructors.

- The `Function` type is `final` and so it cannot be extended, implemented, or mixed in. This restriction is introduced in Dart 3.0.
- A switch expression can be used to wait for futures to complete. Of course, it has to always be exhaustive. For example:

```
final int n = 1;

await switch (n) {
  1 => Future.delayed(const Duration(seconds: 2)),
  2 => Future.delayed(const Duration(seconds: 4)),
  _ => Future.delayed(const Duration(seconds: 10)),
};
```

- It is a compile-time error if a variable pattern is used in an assignment context. Patterns in assignments can only assign to existing variables using identifier patterns, not declare new ones. For example:

```
void main() async {
  var a = 1;
  var b = 2;

  (var a, int b) = (3, 4); // Error
  (a, b) = (3, 4); // OK
}
```

- It is a compile-time error if the type of an expression in a `when` guard clause is not assignable to `bool`.

# 8 – Futures, Streams and Isolates

---

## 8.1 Introduction

Before diving into the chapter, we want to briefly review an important difference. In Dart, you can execute operations synchronously or asynchronously. Here's how these two paradigms differ:

- **Synchronous operations** are executed in order, one after the other. This paradigm is easy to understand and predict because the program flow is “linear”. Consider this example, where the second task performs an HTTP request:

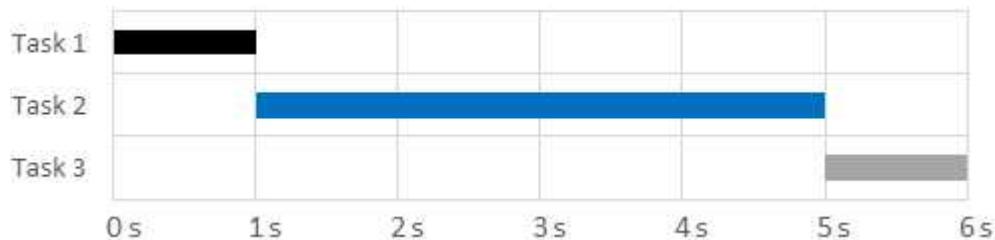


Figure 8.1: An example of synchronous operations.

There is a waste of time in *Task 2* because a slow server may take some time to respond. Since all operations are synchronous, there is no way to “do something else” while waiting for the server’s response. All tasks must execute from start to finish without interruptions.

- **Asynchronous operations** allow your program to execute other work while waiting for other processes to finish. The program flow may be split into multiple parts and thus become hard to predict. Consider this example, where the second task performs an HTTP request:

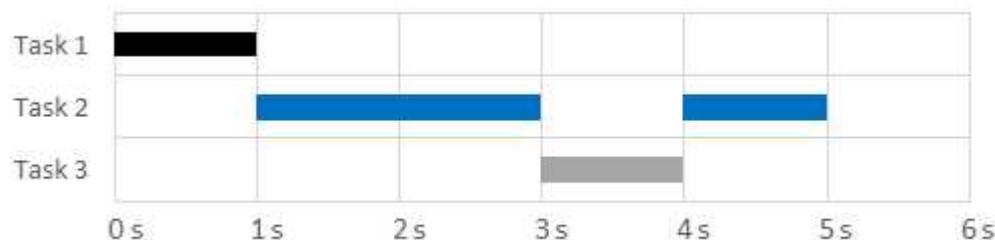


Figure 8.2: An example of asynchronous operations.

At a certain point, *Task 2* makes an HTTP request, but it doesn't know how long the server will take to respond. Since it's an asynchronous operation, *Task 2* is "paused" to wait for the server response and (in the meanwhile) *Task 3* is executed. When the HTTP response arrives, the *Task 2* execution continues.

Since asynchronous operations can be paused, there is the possibility to execute other work while waiting. The program flow doesn't have "dead times": there always is an operation being processed and thus the overall execution time is reduced. For example, asynchronous code is useful when:

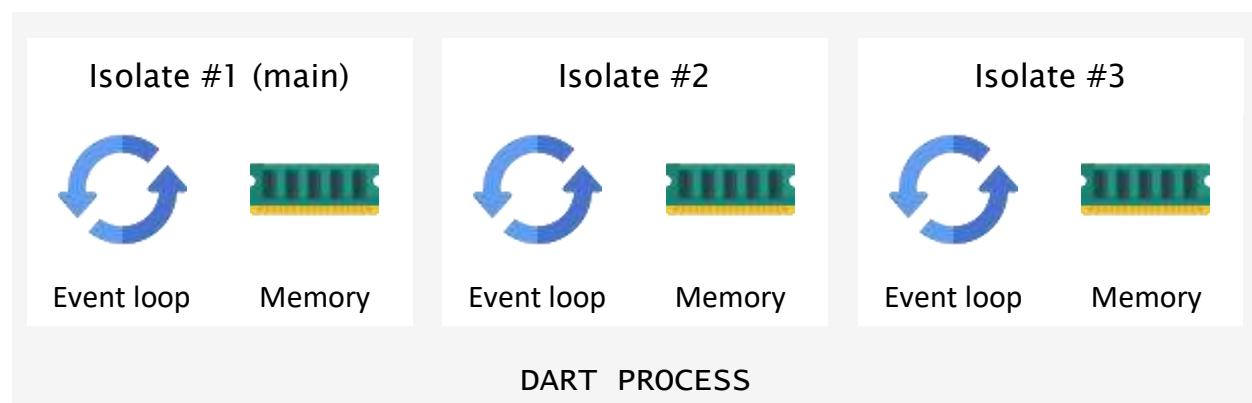
- making HTTP requests;
- interacting with a database;
- reading or writing data from a file.

Dart also supports concurrent programming with ad-hoc keywords, isolates, and futures. You can write asynchronous code using the [Future<T>](#) API and concurrent code using the [Isolate](#) API. We will cover these classes, in detail, in the following sections.

### Note

Some languages, like C# and Java, give you the possibility to spawn new threads within the same process to run concurrent operations. Since all threads also share the same memory, there is the need to use semaphores, locks, mutexes, and other strategies to avoid data races.

Instead of threads, Dart uses isolates on native platforms (and web workers when running on the web). An isolate has its own memory heap, which is not visible from the outside, and an event loop, that makes the program run. Any Dart program can have one or more isolates:



Since there is no shared memory (each isolate has its own), there cannot be data races by design. This also implies that Dart doesn't need semaphores, mutexes, locks, atomic values, or any other mechanism you would need when working in a shared memory architecture. Before diving into the details, we want to point out a few things:

- When a Dart program is executed, the runtime automatically creates an isolate (the `main isolate`) in which the `main()` method is run. All of this is done by Dart for you, so no manual setup is required.
- In general, you don't need to spawn multiple isolates. The main isolate created by the Dart runtime is often enough to make your program execute smoothly, especially if you use the techniques described in this chapter (futures and streams).
- An isolate can communicate with other isolates in a safe way using messages. We will see how this process works in *Section 8.4 – “Isolates”*.

The heart of an isolate is the `event loop`, the machinery that orchestrates the whole program flow. Being aware of why it exists and how it works is essential to understand how futures and streams work in Dart.

### 8.1.1 Event loop and queues

To understand how an isolate works, we're going to make an example of a console application that reads the user input from the console and prints the result. For now, it's enough for you to know that `stdin`<sup>53</sup> is used to read strings from console applications. Here's the example code:

```
String processText(String input) {
  final modified = input.toUpperCase().replaceAll(' ', '');
  return '> Modified: $modified';
}

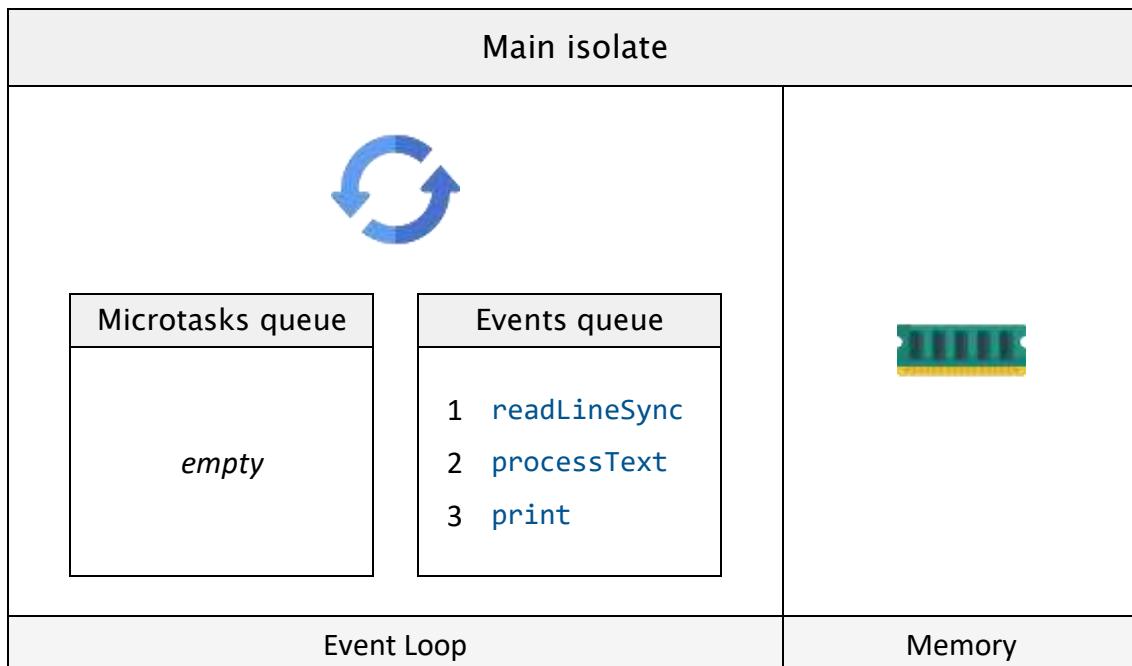
void main() {
  final input = stdin.readLineSync(); // Reads a string from the console
  final newText = processText(input);

  print(newText);
}
```

---

<sup>53</sup> The I/O library, along with `stdin` and `stdout`, is covered in *chapter 9 – section 1.2 “The I/O library”*.

When you run the program (using `dart run` for example) the Dart runtime creates the main isolate and prepares the event loop. This is a simplified view of the isolate (there may be more events in the queues, but we ignore them to not complicate the example):



The Dart event loop always processes events from either the microtask or the event queue. Once both queues are empty (and there are no pending asynchronous callbacks), the isolate is killed. If it is the main isolate, the program terminates. Here is the purpose of the two queues:

- The **microtask queue** is prioritized and it's always executed before processing entries of the event queue. This is internally handled by Dart. You can add entries here yourself with the `scheduleMicroTask` function but, unless you really know what you're doing, you shouldn't need this method.
- The **event queue** can contain I/O calls, painting requests, click events, timers, and most of the operations your program has to do. In other words, the code you write is “converted” into events that fill this queue.

The event loop first executes all microtasks in the microtask queue (if any) in FIFO order. When the microtask queue is empty, the event loop only executes the first event in the event queue. The cycle is then repeated: all microtasks are processed until the queue is empty and then the next event on

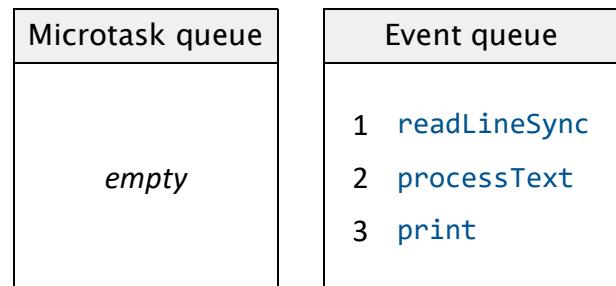
the event queue is executed. Once both queues are empty and there are no pending callbacks, the isolate is killed and the program exits. The idea is that all Dart code is converted into events (that either go in the microtask or event queues), and the event loop processes all of them, one by one.

### Note

The event loop processes only one task at time. The entire program slows down if the microtask queue or event queue contains time-consuming operations. Futures are very important because they help keep the event loop busy without spending too much time on a single task.

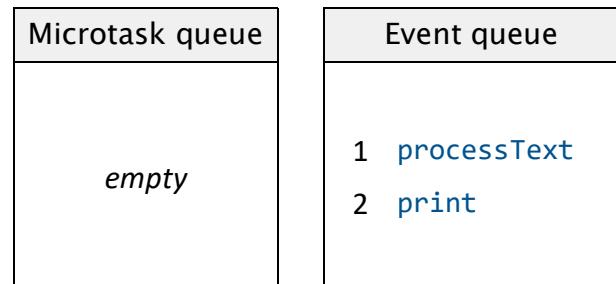
Let's see a step-by-step overview of how our program is executed within the main isolate and how the queues are handled. Remember that this is a simplified example: there may be more events in the queue (some may implicitly be added by Dart, for example). Here we're just highlighting the events generated by our code:

```
→ void main() {  
  final input = stdin.readLineSync();  
  final newText = processText(input);  
  print(newText);  
}
```



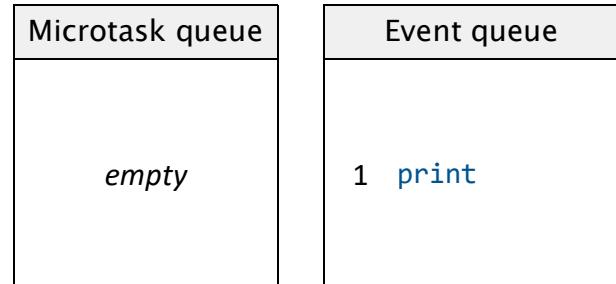
Imagine that you ran the application, and the program flow arrived at the `main` entry point (where the black arrow points). The event loop first executes all microtasks, but nothing is done because the queue is empty. At this point, the next event from the event queue is picked (`readLineSync`) and executed. Once completed, it's removed from the queue:

```
→ void main() {  
  final input = stdin.readLineSync();  
  final newText = processText(input);  
  print(newText);  
}
```



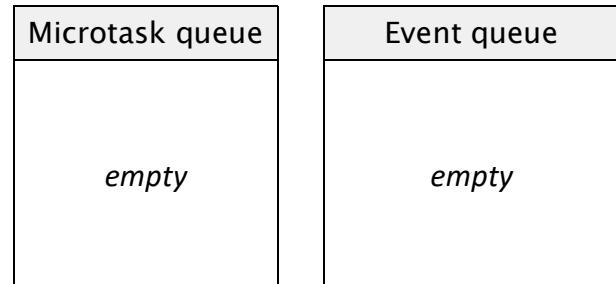
The process is repeated. The event loop first executes all microtasks, but nothing is done because the queue is empty. At this point, the next event from the event queue is picked (`processText`) and executed. Once completed, it's removed from the queue:

```
void main() {  
    final input = stdin.readLineSync();  
→  final newText = processText(input);  
    print(newText);  
}
```



The process is repeated. The event loop first executes all microtasks, but nothing is done because the queue is empty. The next event from the event queue is picked (`print`) and executed. Once completed, it's removed from the queue:

```
void main() {  
    final input = stdin.readLineSync();  
    final newText = processText(input);  
→  print(newText);  
}
```



The process is repeated. The event loop first executes all microtasks but nothing is done because the queue is empty. At this point, since the event queue is also empty, the isolate is killed because there are no more items (in both queues) to process. When the main isolate is killed, the program also terminates. Here are a few more considerations:

- You can predict the order in which the entries of the queues will be executed, but you can't tell the moment when they will be executed.
- You should worry about writing code that doesn't block the loop. For example, if you write slow functions that take much time to execute, the entire application will be unresponsive. The event loop executes one action at time: if an operation takes a lot of time, all the others will have to wait, and the program execution slows down.
- Futures and streams split large events into smaller ones that don't block the event loop.

The general idea is that you should write synchronous code for “fast” operations and asynchronous code for time-consuming operations. Let’s move on to understand what futures are and why they’re so important in Dart.

## 8.2 Futures

Time-consuming operations can block the event loop on a single task for a long time and thus make your application not responsive. Futures solve this problem by splitting large tasks into multiple and smaller ones. Before diving into the details, we are going to make an example to get a high-level overview of how futures work. Let’s start by considering this synchronous code:

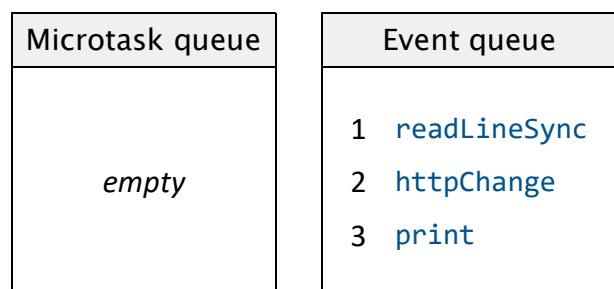
```
void httpChange(String? input) {
    // Imagine that this function makes a synchronous HTTP call and then prints
    // the result. It may take a lot to execute.
    print('Result');
}

void main() {
    final input = stdin.readLineSync();

    httpChange(input);
    print('HTTP request started...');
}
```

This code reads a string value from the terminal, then makes an HTTP request to process the text and outputs the result. Here is a simplified view of the events that are generated from our code. In reality, many other events are implicitly added by Dart, but we’re not showing them to not make the example too difficult:

```
void main() {
    final input = stdin.readLineSync();
    httpChange(input);
    print('HTTP request started...');
}
```



The biggest problem is that `httpChange` might take a lot of time to execute if the server is slow to respond. Since our code is synchronous, all events are run in order: everything that comes after the HTTP call must wait (potentially, for a long time). This is what the console prints:

```
Result
HTTP request started...
```

The result is easy to predict because the code is synchronous. Let's now improve the `httpChange` method so that it doesn't block our application anymore. To do so, we have to use futures and make an asynchronous HTTP call:

```
Future<String> httpChange(String? input) {
    // Imagine that this function makes an asynchronous HTTP call and then
    // returns the result.
    return Future<String>.delayed(const Duration(seconds: 2), () => 'Result');
}

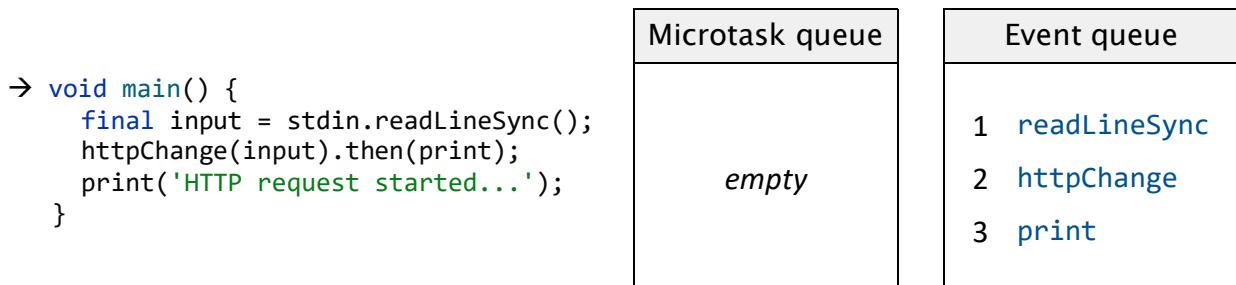
void main() {
    final input = stdin.readLineSync();

    httpChange(input).then(print);
    print('HTTP request started...');
}
```

In this example, the `Future.delayed` constructor is used to fake an asynchronous HTTP call. When `httpChange` is called, the string is returned after two seconds to simulate a slow server response. This is what the console prints:

```
HTTP request started...
Result
```

The `then` function returns immediately, but its callback will be executed when the future will have completed with a result. For this reason, the `Result` string is printed two seconds after the other `print` call. Let's give a closer look at what happens in the event loop:

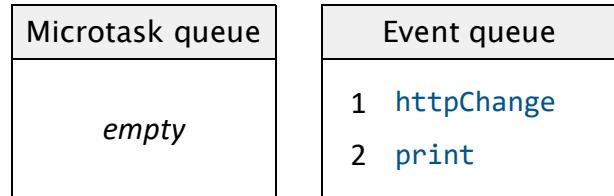


Imagine that you ran the application, and the program flow arrived at the `main` entry point (where the black arrow points). The event loop first executes all microtasks, but nothing is done because the queue is empty. At this point, the next event from the event queue is picked (`readLineSync`) and executed. Once completed, it's removed from the queue:

```

void main() {
    final input = stdin.readLineSync();
    httpChange(input).then(print);
    print('HTTP request started...');
}

```

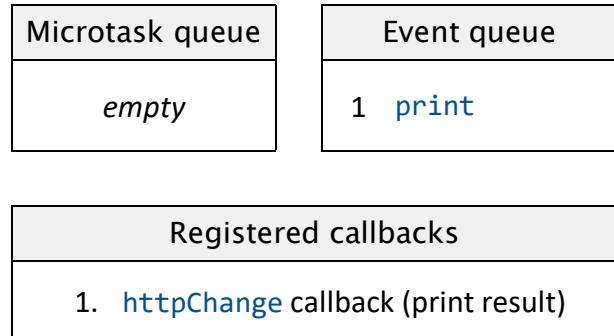


The process is repeated. The event loop first executes all microtasks, but nothing is done because the queue is empty. At this point, the next event from the event queue is picked (`httpChange`) and executed. Here happens something particular:

```

void main() {
    final input = stdin.readLineSync();
    httpChange(input).then(print);
    print('HTTP request started...');
}

```



The `then` function returns immediately and thus the event is removed from the queue. However, the callback defined by `then` is “remembered” because it will be executed later when the future terminates. In our case, the callback will be executed when the HTTP call will complete. In practice, the program flow “interrupts” and leaves space to do something else (and it will “resume” later).

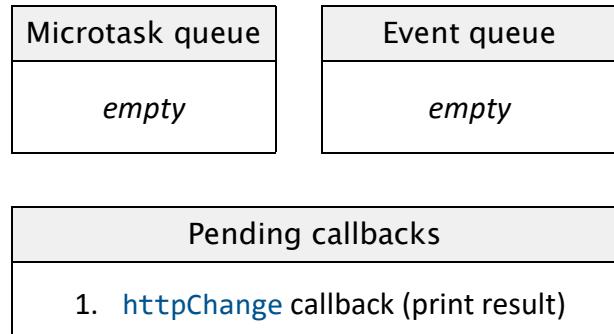
The process continues as usual. The event loop executes all microtasks, but nothing is done because the queue is empty. The next event from the event queue is picked (`print`) and executed. Once completed, it's removed from the queue:

```

void main() {
    final input = stdin.readLineSync();
    httpChange(input).then(print);
    print('HTTP request started...');

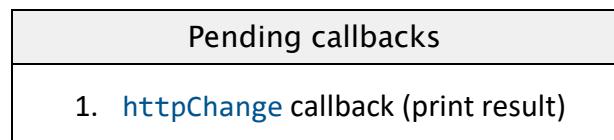
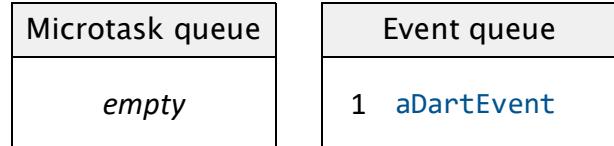
}

```



Since there still is a pending callback, the program doesn't terminate. Imagine that Dart inserted an event (that came in response to an operating system call, for example) a few moments before our future completes. The queue would look like this:

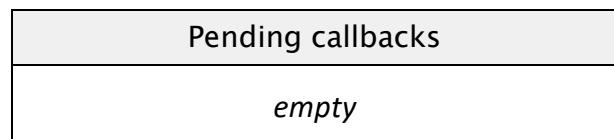
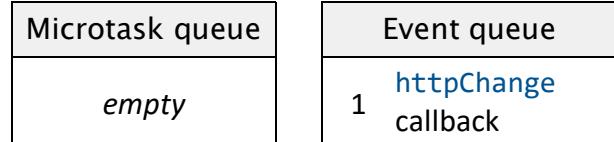
```
void main() {
  final input = stdin.readLineSync();
  httpChange(input).then(print);
  print('HTTP request started...');
→ }
```



Since the event loop is not blocked by anything, it can move on with the usual loop. It executes all microtasks, but nothing is done because the queue is empty. The next event from the event queue is picked (**aDartEvent**) and executed. Once completed, **aDartEvent** is removed from the queue.

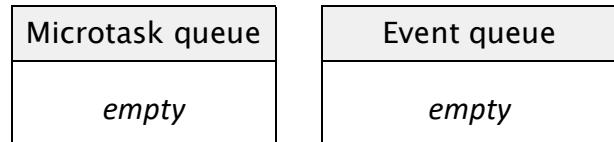
When the future completes, a “notification” is sent to the isolate and a new event is inserted in the queue to proceed with the callback evaluation:

```
void main() {
  final input = stdin.readLineSync();
→ httpChange(input).then(print);
  print('HTTP request started...');
}
```



The process continues as usual. The event loop executes all microtasks, but nothing is done because the queue is empty. The next event from the event queue is picked (**print**) and executed. Once completed, it's removed from the queue:

```
void main() {
  final input = stdin.readLineSync();
→ httpChange(input).then(print);
  print('HTTP request started...');
}
```



Now that all queues are empty and there are no pending callbacks to execute, the isolate is killed and the program exits. This detailed example is useful for understanding some key concepts:

1. Each isolate has a single thread of execution with an event loop that executes tasks one at time. A future “splits” events into smaller pieces to avoid blocking the event loop on a single task for too long.
2. Futures are used to write asynchronous code, which is still executed on a single thread (and the same event loop). In other words, futures do not execute parallel code or take advantage of multiple CPU threads.
3. To enable parallel code execution on multiple processor cores, you have to use the [Isolate API](#). We will cover concurrent programming in the next section.

Let's go through the [Future<T>](#) API and discover what the `async/await` keywords are used for.

### 8.2.1 The Future<T> API

A future is an instance of [Future<T>](#) class, which represents the status of an asynchronous task. All futures can have two statuses:

1. **Uncompleted**. When you call an asynchronous function, it returns an uncompleted future. This state indicates that a [Future<T>](#) was invoked, but it hasn't returned a value yet because some processing is still going on.
2. **Completed**. When the asynchronous operation terminates, it completes with either a value or an error. In this case, the state can be “completed with a value” or “completed with an error”.

The [Future<T>](#) named constructors ask for the [FutureOr<T>](#) type, which cannot be subclassed, implemented, or mixed. It's used as parameter type to accept either a [Future<T>](#) or [T](#). For example:

```
// Accepts both 'Future<int>' and 'int' types
void demoFunction(FutureOr<int> value) => print(value.runtimeType);

void main() {
    demoFunction(Future<int>.value(10)); // Future<int>
    demoFunction(10); // int
    demoFunction(10.0); // Compiler error! We passed 'double' instead of 'int'
}
```

The function can receive `int` or `Future<int>` thanks to the special `FutureOr` type. Most of the `Future<T>` constructors expect to receive a function with no parameters that return a `FutureOr` value. In other words, you can create a future that either returns a value or a future that returns another future:

- `Future`. The default constructor expects a callback with no parameters whose return type is `FutureOr<T>`. Alternatively, you can use the `async` modifier to avoid explicitly building the future. For example:

```
// OK, but not widely used
Future<String> withoutAsync() => Future(() => 'Dart');

// OK, widely used and less verbose
Future<String> withAsync() async => 'Dart';

void main() async {
  print(withoutAsync().runtimeType); // Future<String>
  print(withAsync().runtimeType); // Future<String>
}
```

The two `withAsync` and `withoutAsync` functions are equivalent: they both return a future holding a `String`. However, using `async` is less verbose, and it's also recommended by the Dart style guidelines<sup>54</sup>. You can also return another future from the constructor:

```
Future<String> withoutAsync() {
  return Future(
    () => Future<String>(() => 'Dart'),
  );
}
```

In this case, the “outer” future waits for the “inner” one to complete before returning. The constructor callback is the function you want to run asynchronously.

- `Future.microtask`. This constructor adds the callback in the event loop’s microtask queue. If you really need to use this, which is rarely the case, make sure to not pass time-consuming functions to avoid blocking the event loop.

```
// Creates a new entry in the microtask queue
Future<String> example() => Future.microtask(() => 'Dart');
```

---

<sup>54</sup> <https://dart.dev/guides/language/effective-dart/usage#prefer-asyncawait-over-using-raw-futures>

Internally, this constructor calls the `scheduleMicrotask` function to add the callback in the microtask queue. You could also use this constructor to return another future.

- `Future.value`. Creates a future in a *completed* state with the given value:

```
Future<String> example() => Future<String>.value('Dart');
```

Use this constructor to return a value (or a future object) you already have.

- `Future.sync`. Creates a future that immediately executes its callback:

```
Future<String> example() => Future<String>.sync(() => 'Dart');
```

The difference from the `Future.value` constructor is that `Future.sync` accepts a function rather than a value.

- `Future.delayed`. This constructor runs the callback after the given delay. The `Duration` class, which will be explored in *chapter 9 – Section 1.3 “Date and time”*, is used to define time spans:

```
Future<String> example() {
  return Future<String>.delayed(
    const Duration(seconds: 2),
    () => 'Dart',
  );
}
```

This function returns the `Dart` string after two seconds.

We have already seen that the `then` method is used to register a callback that will be invoked when the future will complete (either with a value or with an error). For example:

```
Future<String> example() {
  return Future<String>.delayed(const Duration(seconds: 2), () => 'Dart');
}

void main() {
  example().then((value) => print('Completed: $value'));
  print('Uncompleted');
}
```

This code prints `Uncompleted` first and, after two seconds, `Completed: Dart`. The function defined by `then` is registered within the isolate and executed later, when the future will complete. In the meanwhile, other events can be processed. Notice the difference:

```
void main() {
  example().then((value) {
    print('Completed: $value');
    print('Uncompleted');
  });
}

// Prints the following:
// 'Completed: Dart'
// 'Uncompleted'

void main() {
  example().then((value) {
    print('Completed: $value');
  });
  print('Uncompleted');
}

// Prints the following:
// 'Uncompleted'
// 'Completed: Dart'
```

On the left, both `print` calls are executed inside the callback, so they follow the declaration order. On the right instead, `then` returns immediately and schedules its callback for later. If the callback threw an exception, you could chain the `catchError` function:

```
// Prints 'Dart' after 2 seconds
example().then(print).catchError(errorHandlerCallback);
```

This is the asynchronous equivalent of a `catch` block. You can chain an indefinite number of `then` calls, which can eventually terminate with a `catchError` call:

```
Future<String> one() => Future<String>.value('1');
Future<String> two() => Future<String>.value('2');
Future<String> three() => Future<String>.value('3');

void main() {
  one().then((value) {
    print('$value');
    return two();
  }).then((value) {
    print('$value');
    return three();
  }).then((value) {
    print('$value');
  }).catchError((e) {
    print('Got an error: $e');
  });
}

// Prints '1 2 3'
```

There also is the `whenComplete` function, which is the asynchronous equivalent of a `finally` block. As such, the final `print` statement is always executed (regardless an exception or error was thrown or not):

```
void main() {
  one().then((value) {
    print('$value');
    return two();
  }).then((value) {
    print('$value');
    return three();
  }).then((value) {
    print('$value');
  }).catchError((e) {
    print('Got error: $e');
  }).whenComplete(() {
    print('finally');
  });

  // Prints '1 2 3 finally'
}
```

Even if very efficient, the code is also hard to read because of the verbosity of the method calls. The longer the chain, the worse the readability. Dart's official guidelines say that you should prefer using `await` instead of `then` to register callbacks<sup>55</sup>.

If you want to wait for multiple futures to complete, use the `Future.wait` static method. It returns a future that will complete once all futures have completed (either with an error or with a value):

```
void main() {
  await Future.wait<void>([
    Future<void>.delayed(const Duration(seconds: 2)),
    Future<void>(() => doSomething()),
    Future<void>(() => doSomethingElse()),
  ]);

  print('All futures completed!');
}
```

If the optional named parameter `eagerError` is `true` (it's `false` by default), the returned future completes with an error as soon as any future in the list completes with an error.

---

<sup>55</sup> <https://dart.dev/guides/language/effective-dart/usage#prefer-asyncawait-over-using-raw-futures>

## 8.2.2 async and await

Thanks to the `async` and `await` keywords, you can define asynchronous functions more concisely and use their result declaratively. There are three rules to keep in mind:

1. to define an asynchronous function, you must add `async` before the function body;
2. the `await` keyword can only be used in a function marked with the `async` keyword;
3. an `async` function must always return a `Future<T>`. The only exception is `main()`, which is allowed to return `void` rather than `Future<void>` when marked with `async` (but we don't like this because it seems misleading and not consistent with the standards).

For example, these are two equivalent versions of the same program where one uses `then` and the other `await`. There are no performance differences or tradeoffs because both are compiled in the same way:

```
Future<String> example() {
    return Future<String>.delayed(
        const Duration(seconds: 2),
        () => 'Dart',
    );
}

void main() {
    example().then((value) {
        print('Finish!');
        print('The value is $value');
    });
}
```

```
Future<String> example() {
    return Future<String>.delayed(
        const Duration(seconds: 2),
        () => 'Dart',
    );
}

Future<void> main() async {
    final value = await example();
    print('Finish!');
    print('The value is $value');
}
```

The only difference is readability. Everything that comes after the `await` keyword is treated as if it was the callback of the `then` method. To catch exceptions, you can use regular try blocks as if it was synchronous code:

```
Future<void> myMethod() async {
    try {
        final value = await example();
    } on SomeException {
        print('Exception');
    } finally {
        print('Finally');
    }
}
```

There are some cases where you cannot use `await` to precisely reproduce what the `then` method does. For example, consider this code:

```
void main() {
  example().then((value) {
    print('$value');
  });

  print('Hello');
}
```

Remember that everything after the `await` keyword is treated as if it was the callback of `then`. As such, to rewrite the above example, we have to print `Hello` before awaiting the future:

```
Future<void> main() async {
  print('Hello');

  final value = await example();
  print('$value');
}
```

In this way, `Hello` is not part of the callback. If we had kept the `print` statement at the bottom, it would have been part of the callback:

```
Future<void> main() async {
  final value = await example();

  print('Hello'); // This is printed AFTER the future completed
  print('$value');
}
```

It is always possible to interchange `then` and `await`, but you may have to rearrange some calls to keep the code consistent. The official Dart guidelines recommend to prefer using `await` instead of `then`<sup>56</sup> whenever possible.

### 8.2.3 Completers

The `Completer<T>` API is used to produce `Future` objects that will either complete with a value or an error in the future. For example:

---

<sup>56</sup> <https://dart.dev/guides/language/effective-dart/usage#prefer-asyncawait-over-using-raw-futures>

```

Future<int> computeValue() {
  final completer = Completer<int>();
  doSomething(
    onSuccess: (value) => completer.complete(value.round()),
    onError: () => completer.completeError(Exception('Whoops!')),
  );
  return completer.future;
}

// Prints:
// 'Computing...'
// 'Value = 0'
void main() {
  computeValue().then((value) => print('Value = $value'));
  print('Computing...');
}

```

A `Completer<T>` object represents a computation that will terminate at some point in the future. It has an associated `Future<T>` object (`completer.future`) that is used by other code to wait for the value to be ready. The `complete` method successfully completes the future with a value while `completeError` completes the future with an error. The general workflow is:

1. create a new `Completer` object;
2. return its `future` object so that other code can asynchronously wait for the result using the `then` function or `await`;
3. at a later point, ensure that your code somewhere calls either `complete` or `completeError` to complete the returned future with a value or an error.

Completers may be useful when you're converting a callback-based API into a future-based API, for example. Remember that futures can only be completed once. As such, if you called `complete` twice or `completeError` after `complete`, you would get a runtime error. To avoid this situation, you can use the `completer.isCompleted` getter which tells you whether the future was completed or not.

## 8.3 Streams

In Dart, a stream is a sequence of asynchronous events that can be listened. For example, you could make a Flutter application that constantly updates latitude and longitude values on the screen while

you're out for a walk. As soon as the position changes, you can push a new event into the stream, and a listener is notified. Under the hood, there are two main actors:

1. A generator, which is responsible for detecting the position changes and sending them over the stream.
2. A subscriber, which is listening on the stream populated by the generator. It's automatically notified whenever new data is available.

To see how a generator and a subscriber work together, let's create an example with a stream that produces a certain number of random values:

```
// All streams have the 'async*' modifier before the body
Stream<int> randomNumbersStream({int maxSteps = 50}) async* {
    final random = Random();
    var step = 0;

    while (step < maxSteps) {
        await Future<void>.delayed(const Duration(seconds: 1));
        yield random.nextInt(100) + 1;

        step++;
    }
}
```

This function is called *asynchronous generator* because it returns a `Stream<T>` type and it has the `async*` (with the asterisk) modifier before the body. Generators are the ones that produce data and send them to the stream once ready. There are some new, special things to point out:

- Asynchronous generators must have the `async*` modifier and must return a `Stream<T>`. It is a compile-time error if you use `async*` on a function that doesn't return `Stream<T>`.
- The `yield` keyword “sends” data into the stream. It can only be used inside asynchronous generators.
- You cannot use the `return` keyword inside a generator.

Asynchronous generators are special functions that return a type (`Stream<T>`) but cannot declare the `return` keyword. The reason is that a stream doesn't have to return data: it has to push data for its listeners (using the `yield` keyword). The stream is created when the function is called and it starts running when it's listened:

```

void main() {
    // The stream is CREATED (but not started)
    final stream = randomNumbersStream(maxSteps: 10);

    // The stream is started because a listener just registered.
    stream.listen((event) {
        print('Random number: $event');
    });
}

```

This program will call the `print` statement ten times and then the stream will close. Alternatively, we could have also listened to the stream using the special `await for` syntax:

```

Future<void> main() async {
    // The stream is CREATED (but not started)
    final stream = randomNumbersStream(maxSteps: 10);

    // The stream is started because a listener just registered.
    await for(final event in stream) {
        print('Random number: $event');
    }
}

```

Note that, in this case, you need the `async` keyword before the function body because you're using `await`. The output is the same as before, but the two approaches behave differently. Look at this comparison:

```

Future<void> main() async {
    final stream = randomNumbersStream(
        maxSteps: 2,
    );
    stream.listen((event) {
        print('Random number: $event');
    });
    print('Hello!');
}

// Hello!
// Random number: xx
// Random number: xx

```

```

Future<void> main() async {
    final stream = randomNumbersStream(
        maxSteps: 2,
    );
    await for(final event in stream) {
        print('Random number: $event');
    }
    print('Hello!');

}

// Random number: xx
// Random number: xx
// Hello!

```

The difference is that `listen` returns immediately while `await for` returns only when the stream finishes emitting events. In other words, `listen` processes events asynchronously while `await for` serializes the stream consumption. In both cases, events are asynchronously handled, and so the event loop doesn't get blocked.

## Note

Don't think that `listen` is better or more efficient than `await for`. Both are perfectly fine, but they have different use cases. Generally, you should use `await for` when you have a finite series of events, and you must wait for all of them to complete. In all the other cases, `listen` is very likely going to be the right choice.

Other than `async*` generators, you can also create streams using the `StreamController` class. It basically is a wrapper of a stream that you can manually set up and access from everywhere in the program. This is how the same random number generator stream can be made using a controller:

```
Stream<int> randomNumbersController({int maxSteps = 50}) {
    late StreamController<int> controller;
    Timer? timer;
    var counter = 0;
    final random = Random();

    void tick(_) {
        ++counter;
        controller.add(random.nextInt(100) + 1);
        if (counter >= maxSteps) {
            timer?.cancel();
            controller.close();
        }
    }

    void startTimer() {
        timer = Timer.periodic(const Duration(seconds: 1), tick);
    }

    void stopTimer() {
        timer?.cancel();
        timer = null;
    }

    controller = StreamController<int>(
        onListen: startTimer,
        onPause: stopTimer,
        onResume: startTimer,
        onCancel: stopTimer,
    );
}

return controller.stream;
}
```

This isn't a generator function anymore (because it doesn't have the `async*` modifier), but it still internally builds and returns a stream. You should always remember to call the `close()` method on a stream controller when you don't need it anymore. The implementation is more complicated because we need to use a `Timer` to periodically add new events in the stream.

## Note

`Timer` simply is a countdown timer that can fire events once or periodically. It counts from the given duration down to zero and then invokes its callback. For example:

```
Timer(const Duration(seconds: 2), () => print('Hi'));
```

This code prints `Hi` on the console after 2 seconds. The `Timer.periodic` constructor does the same but iterates forever:

```
Timer.periodic(const Duration(seconds: 2), () => print('Hi'));
```

This code keeps printing `Hi` on the console after 2 seconds until you use the `cancel()` function to stop the timer.

The advantage of a stream controller over an `async*` generator is that, being it a class, you can pass its reference and add events from anywhere in your program. The controller handles everything internally; you just need to define its callbacks:

- `onListen`: called when the stream starts emitting data because someone subscribed;
- `onPause`: called when pausing the stream;
- `onResume`: called when the stream is resumed after being paused;
- `onCancel`: called when nobody else is subscribed to the stream.

Regardless you're using a generator or a controller, when you use the `listen` method on a stream, you're getting back a `StreamSubscription<T>` type. It's commonly used to pause, resume and/or cancel the subscription:

```
void main() {
  final StreamSubscription<String> sub = someStream().listen(print);

  sub.pause();
  sub.resume();
  sub.cancel();
}
```

You can ignore the returned subscription object if you just need to listen to the stream without changing your subscription status.

### 8.3.1 The `yield*` keyword

We have seen that the `yield` keyword is (only) used inside `async*` functions to push data into the stream. In case you had to merge events coming from another stream, use the `yield*` keyword (where `yield*` is read as *yield-each*). For example:

```
Stream<int> generateZeros() async* {
    yield 0;
    yield 0;
}

Stream<int> generateNumbers() async* {
    for(var i = 0; i < 6; ++i) {
        await Future.delayed(const Duration(milliseconds: 500));
        yield i + 1;

        if (i == 2) {
            yield* generateZeros();
        }
    }
}

void main() {
    // Prints '1 2 3 0 0 4 5 6'
    generateNumbers().listen(print);
}
```

The `generateNumbers` generator regularly pushes 1, 2, and 3 into the stream using `yield` as you would expect. When the `if` condition is entered, the `yield*` pauses the current stream flow and starts pushing events from the `generateZeros` stream (hence the two zeroes in the console). Once `generateZeros` is done, the number generation stream is restarted from where it was paused and prints the remaining numbers (4, 5 and 6). In general:

- The `yield*` keyword is used to pause the current stream flow, execute another stream until the end and then come back.
- In an asynchronous generator, `yield*` can only be assigned to a `Stream<T>` type.

Generally, `yield*` is useful when you want to split your generator logic into smaller pieces or when you need to merge the events from another stream.

### 8.3.2 The Stream<T> API

Other than creating streams with generator functions and controllers, you can directly use one of the various named constructors of `Stream<T>`. They don't give you much flexibility, but they could still be useful in those cases where you don't need much setup. Here are the most relevant ones:

1. `Stream<T>.value`: creates a stream that only emits a single value before closing.
2. `Stream<T>.future`: creates a stream that waits for the future to complete, emits the result and then it closes.
3. `Stream<T>.fromFutures`: creates a stream that yields the results of multiple futures in the stream (following the order in which futures complete).
4. `Stream<T>.fromIterable`: creates a stream that iterates over the given `Iterable<T>` and emits all of the entries as events.
5. `Stream<T>.periodic`: creates a stream that repeatedly emits events with a given interval.

Once you have a `Stream<T>` type (no matter how you've created it), there are lots of methods you can use to alter the stream and/or build a new one. Each one waits until someone listens on the new stream before listening to the original. For example:

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

    // The "original" stream
    final oldStream = Stream<int>.periodic(
        const Duration(seconds: 1),
        (_) => random.nextInt(100),
    );

    // The "new" stream only contains even numbers
    final newStream = oldStream.where((event) => event.isEven);

    // The stream is started here
    newStream.listen(print);
}
```

The `where` method discards all of the events whose condition evaluates to `false`. The `map` method, as you may expect, transforms a stream type into another one. All of these methods behave in the same way as their `Iterable<T>` equivalent. For example:

```
final newStream = oldStream
    .where((event) => event.isEven)
    .map((event) => '$event');
```

The `asyncMap` method has the same functionality, but it allows its callback to be an asynchronous function. In general, most of these methods that transform a stream can be replaced with an `await for` loop (but it may be hard). Do not make confusion with the `transform` method:

- `map` or `asyncMap`: these methods transform each event type into another;
- `transform`: it transforms one or more event types into another event type.

The `transform` method is generally used for I/O or data decoding, where you need to take various events to produce an output. For example, you can use a `LineSplitter` to split the stream result into multiple lines:

```
void main() {
  final stream = Stream<String>.fromIterable([
    'Hello\nworld', '\rboys!',
  ]);

  // Prints:
  // 'Hello'
  // 'world'
  // 'boys!'
  stream.transform(const LineSplitter()).listen(print);
}
```

Whenever a CR, LF, or a CR+LR is found, the `LineSplitter` produces a new line. Other very popular transformers are the so-called “converters”, which convert one data type to the other. They can be used to encode the stream data into another format. For example:

```
void main() {
  final stream = Stream<String>.fromIterable(['a', 'b']);

  // Prints:
  // '97'
  // '98'
  stream.transform(const Latin1Encoder()).listen(print);
}
```

With `Latin1Encoder` for example, you can convert a string into an integer, which represents the Latin1 respective values. Transformers can also be used as standalone classes to arbitrarily convert data. These are the currently available converters in the library:

- `AsciiDecoder / AsciiEncoder`
- `Base64Decoder / Base64Encoder`
- `JsonDecoder / JsonEncoder`
- `Latin1Decoder / Latin1Encoder`
- `Utf8Decoder / Utf8Encoder`

Most of the methods we've already seen for iterables also come back with streams. For example, you can use `take(n)` to only pick the first `n` elements of the stream, `skip(n)` to avoid emitting the first `n` events, `join` to combine multiple events into a single string, and much more<sup>57</sup>.

### 8.3.3 Good practices

In Dart, there are two kinds of streams<sup>58</sup>:

1. **Single subscription.** A “single subscription stream” only allows a single listener during the entire stream lifetime, and it doesn’t start generating events until it has a listener. It also stops sending events when the listener unsubscribes (even if the generator could still send more events). The `StreamController` default constructor and `async*` generators create single subscription streams.
2. **Broadcast.** A “broadcast stream” allows multiple listeners and it fires events when they are ready, regardless of the presence of a listener. In other words, a broadcast stream can still emit events even if there are no listeners. The `StreamController.broadcast` constructor is the primary way of creating broadcast streams. Alternatively, you can transform existing streams using the `asBroadcastStream()` method.

Single subscription streams are valuable when events have to be emitted in a specific order without missing any of them. Broadcast streams are worthwhile when events are not related and can be processed “individually” at any time. Another big difference between the two kinds of streams is the following:

- When there are zero listeners, a single subscription stream does not push events. When the subscription pauses, the stream also pauses and re-starts only if the subscriber resumes.

<sup>57</sup> <https://api.dart.dev/stable/2.16.2/dart-async/Stream-class.html>

<sup>58</sup> <https://dart.dev/tutorials/languagestreams#two-kinds-of-streams>

- When there are zero listeners, a broadcast stream doesn't care and keeps pushing events. As such, even if all subscribers paused from listening, the stream would continue anyway.

An exception is thrown if two or more listeners subscribe to a single subscription stream. It may also be helpful to make a comparison between `async*` and `StreamController` to see when one should be preferred over the other:

- Generator functions (with `async*`) are useful when you want to create a single subscription stream. You cannot add events from the outside, so the generator should also be able to always produce new events internally. You cannot create a broadcast stream using `async*`.
- `StreamController` is useful when you want to create a single subscription stream and add events from anywhere in your program. You can add events from the outside just by passing the controller reference and calling the `add` method. You can create a broadcast stream with the `StreamController.broadcast` named constructor.

A controller can do the same things of an asynchronous generator but it may require more code and complexity. The `StreamController` constructor has a `sync` property that, when `true`, creates a stream that delivers its events synchronously. For example:

```
final syncStream = StreamController(  
  onListen: () {},  
  sync: true, // By default, this is 'false'  
>);
```

A synchronous stream controller should be used when an asynchronous event triggers an event on a stream<sup>59</sup>. Instead of scheduling the event in the microtask queue, causing extra latency, the event is instead fired immediately by the synchronous stream controller. Synchronous streams controllers must be used with attention:

- Asynchronous controllers will never give the wrong behavior. However, using a synchronous controller incorrectly can cause otherwise correct programs to break. A synchronous stream controller can be used to break the contract of the `Stream` class.
- Only use the “synchronous” stream version if its use is safe and you see a good improvement in the events latency. Otherwise, just use a normal stream controller, which will always have

<sup>59</sup> <https://api.dart.dev/stable/2.18.6/dart-async/SynchronousStreamController-class.html>

the correct behavior for a `Stream`, and won't accidentally break other code. By default, the `StreamController` constructor creates asynchronous (and thus safe) streams.

- Use synchronous stream controllers only to forward (potentially transformed) events from another stream or a future.
- Adding events to a synchronous controller should only happen as the very last part of the handling of the original event. At that point, adding an event to the stream is equivalent to returning to the event loop and adding the event in the next microtask.

The synchronous broadcast stream controller also has restrictions that a normal stream controller does not<sup>60</sup>. In general, avoid using synchronous controllers unless you know what you're doing. The default `StreamController` constructor (with `sync: false`) is the safest option.

### 8.3.4 Synchronous generators

Even if not strictly related to streams, there is another kind of generator we need to cover before moving on. While an asynchronous generator allows awaiting futures and returns a `Stream<T>`, a synchronous generator doesn't allow awaiting futures, and returns an `Iterable<T>`. For example:

```
Iterable<int> randomNumbersGenerator({int maxSteps = 50}) sync* {
    final random = Random();
    var step = 0;

    while (step < maxSteps) {
        sleep(const Duration(seconds: 1)); // this is from the 'dart:io' library
        yield random.nextInt(100) + 1;
        step++;
    }
}

void main() {
    final stream = randomNumbersStream(maxSteps: 3);

    for (final value in stream) {
        print('Random value: $value');
    }
}
```

---

<sup>60</sup> <https://api.dart.dev/stable/2.18.6/dart-async/SynchronousStreamController-class.html>

The most notable difference with an asynchronous generator is the return type and the new `sync*` modifier. The synchronous `sleep` call comes from `dart:io` (which will be covered in *chapter 9 – Section 1.2 “The I/O library”*) and sleeps the program for the given time interval. Let's see other differences:

- Since we cannot use `await` inside synchronous generators, we've had to use `sleep` (which is a blocking function) rather than awaiting a delayed future.
- Since synchronous generators return an `Iterable<T>`, which is the same type of Dart lists, you can receive events using a simple `for` loop.
- The `Iterable<T>` type doesn't have the `listen` function.

A synchronous generator is useful for generating a series of values on demand rather than creating a list every time. For example:

```
Iterable<int> numberGenerator1() sync* {
  for(var i = 0; i < 10; ++i) {
    yield i;
  }
}

Iterable<int> numberGenerator2() {
  return [
    for(var i = 0; i < 10; ++i) i
  ];
}
```

Both functions produce the same output. The second function collects all the numbers, packs them into a list, and returns them all together. In the first case instead, the generator sends numbers one by one as soon as they're generated. Synchronous generators can use the `yield*` keyword.

## 8.4 Isolates

As we've already said at the beginning of this chapter, most of the time your Dart applications can smoothly run in a single isolate (the main one). If you also take advantage of futures and/or streams, your code can execute even faster. However, there are some cases where you need to use some algorithms that, even if heavily optimized, may take a lot of time to execute as the input grows. Some common use cases may be:

- parsing a huge JSON string with millions of entries;

- sorting extensive data collections;
- making time-consuming computations with prime numbers or complex data structures;
- downloading a big file from the internet.

Some operations may not be asynchronous, and then they could just block the event loop for too much time. Each isolate has a single thread of execution, so expensive operations should be moved to a separated isolate for concurrent processing. For example, consider this case:

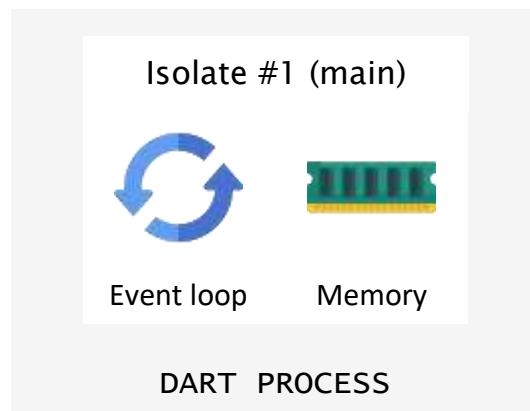
```
// The conversion library contains 'jsonDecode'
import 'dart:convert';

Future<void> main() async {
  final jsonString = await getJsonFromHTTP();

  // With a Large JSON string, 'jsonDecode' might be slow
  final jsonData = jsonDecode(jsonString) as Map<String, Object?>;

  print('There are ${jsonData.length} entries!');
}
```

In chapter 9 – Section 9.1.1 “The conversion library” we will see how the `jsonDecode` method works. For now, it’s enough to know that it transforms a JSON-encoded string into a `Map` object. For this example, assume that the JSON string is huge and so `jsonDecode` takes a few seconds to execute. This is the situation we have as soon as the program is started:



The main isolate spawns and the event loop starts running. Since the JSON string in our example is extremely big, the event loop is “blocked” on `jsonDecode` for a few seconds. This also means that other events cannot be processed while the decoding is running. Consequently, the program may become unresponsive. To solve this issue, we can spawn a new isolate and execute the work there:

```

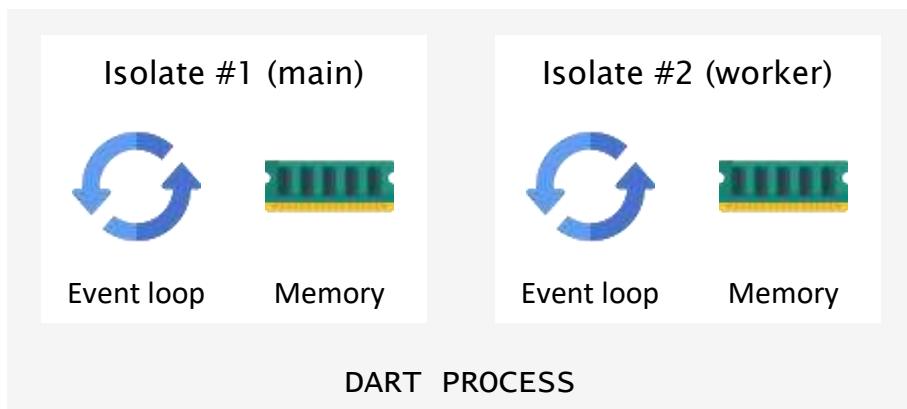
Future<void> main() async {
  final String jsonString = await getJsonFromHTTP();

  // Runs the callback in a new isolate and returns the result
  final jsonData = await Isolate.run<Map<String, Object?>>(() {
    return jsonDecode(jsonString) as Map<String, Object?>;
  });

  print('There are ${jsonData.length} entries!');
}

```

The difference from the previous example is that `jsonDecode` now runs concurrently, in another isolate. What it means is that the main isolate is “free” to do something else while the JSON is being decoded. Thanks to `async`, `await`, and the `Isolate` API, you can create new isolates for parallel code execution (and take advantage of multiple CPU cores). This is what happens under the hood when we run the decoding on a separate isolate:



To keep the program execution smooth, the main isolate should never be blocked for too long on a single task. We have created a new isolate (generally called *worker isolate*) to run `jsonDecode` “in the background” without blocking the event loop of the main isolate. Here is another example:

```

// For large values of n, this recursive algorithm is slow
int fibonacci(int n) {
  if (n <= 1) {
    return 1;
  }

  return fibonacci(n - 1) + fibonacci(n - 2);
}

```

The bigger the input value `n`, the slower the function. To concurrently execute `fibonacci` without blocking the main isolate, we can use `Isolate.run`:

```
Future<void> main() async {
  final value = await Isolate.run<int>(() => fibonacci(46));
  print('fib(125) = $value');
}
```

There is no rule that precisely determines when a function should run on a separate isolate. You should make tests, such as measuring the execution time<sup>61</sup> and/or analyze the code performance, to evaluate if concurrency might help or not.

### Note

Isolates are only implemented for native platforms. All Dart applications running on the web use web workers<sup>62</sup> for similar functionality.

The `Isolate.run` function kills the new isolate once it's finished with the computation. When you need to repeatedly execute work in the background, create a long-running isolate.

#### 8.4.1 Creating long-running isolates

When you have a single function to run in a background isolate, using `Isolate.run` is the best idea. However, in some cases, there might be tasks that run continuously throughout the entire lifespan of the application. For example, imagine you wanted to collect usage reports from your application:

```
class AnalyticsReporter {
  const AnalyticsReporter();

  Future<void> initialize() async { /* code */ }
  Future<void> reportRouteOpened(String route) async { /* code */ }
  Future<void> reportButtonPress(String buttonId) async { /* code */ }
}
```

Once initialized, our `AnalyticsReporter` class sends usage reports to an external server to track which actions users do the most. A report might make HTTP calls, log events and/or process data.

---

<sup>61</sup> chapter 9 – section 9.1.3.2. “Measuring timespans with Stopwatch”

<sup>62</sup> Web workers are used in browsers to run scripts in background threads.

These actions should execute concurrently (especially when there are a lot of reports to do) to avoid blocking the event loop of the main isolate. This is how the two isolates should interact:

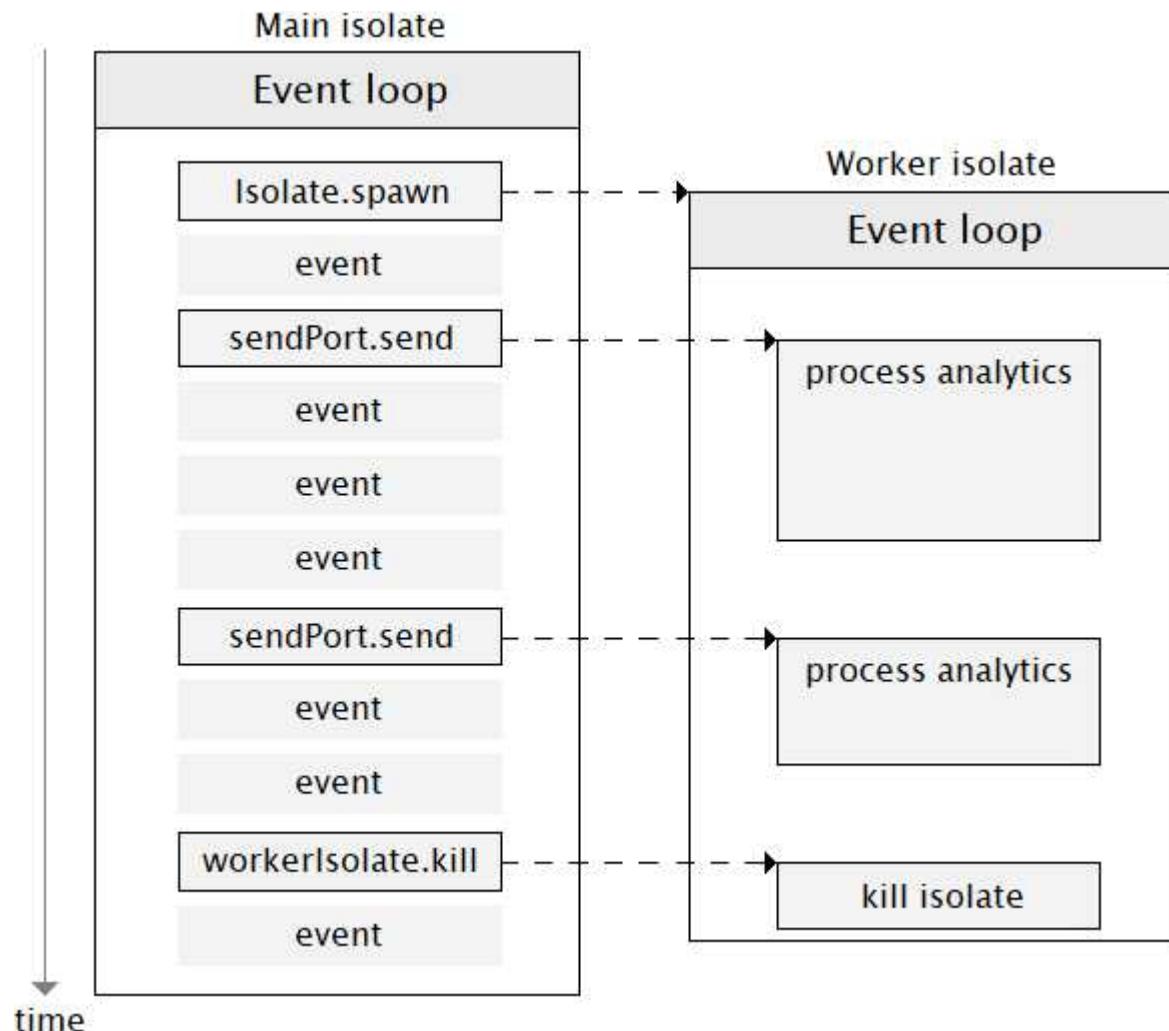


Figure 8.3: The main isolate exchanges messages with the worker isolate using ports.

Two isolates can communicate by exchanging messages. In particular, a [ReceivePort](#) object is used to listen for incoming messages and a [SendPort](#) is used to send messages. In *Figure 8.3* you see that the event loop of the main isolate is never blocked for too long:

- various events are processed by the event loop of the main isolate, but they don't take too much time to execute. This is good because it means that the program stays responsive and smoothly executes;

- we send messages to the worker isolate to execute reports concurrently. In this way, the event loop of the main isolate is “free” to move on with other tasks and doesn’t get stuck if a report takes a lot of time to complete.

To create a new isolate, use the `Isolate.spawn` static method. It takes two parameters: a function to execute in the worker isolate and a `SendPort` to exchange messages. For example:

```
Future<void> main() async {
  // Create a new isolate to run concurrent operation
  final receivePort = ReceivePort();
  await Isolate.spawn<SendPort>(setupIsolate, receivePort.sendPort);

  // Receive the port of the child isolate for two-way communication
  final sendPort = await receivePort.first as SendPort;

  // Start your application...
  await runMyNiceApp(receivePort, sendPort);
}
```

Under the hood, a `ReceivePort` is a single-subscription stream so it can be listened only once. This is enough to listen for the very first event (`receivePort.first`) to get a reference to a `SendPort` created by the worker isolate. This will allow communication between the two isolates. Here is the function that has to execute concurrently:

```
void setupIsolate(SendPort sendPort) async {
  // Send back to the main isolate a port for two-way communication. This is
  // received in the main isolate in the 'await receivePort.first' call
  final receivePort = ReceivePort();
  sendPort.send(receivePort.sendPort);

  // Initialize the analytics report
  const reporter = AnalyticsReporter();
  await reporter.initialize();

  // Listen for messages and report analytics
  receivePort.listen((Object? message) async {
    if (message is String) {
      if (message.contains('route')) {
        await reporter.reportRouteOpened(message);
      } else if (message.contains('button')) {
        await reporter.reportButtonPress(message);
      }
    }
  });
}
```

The `sendPort` parameter of the `setupIsolate` function holds the object we had passed when the worker isolate was spawned. We listen for messages in the `ReceivePort` stream and we execute analytics reporting accordingly. For example:

1. From the main isolate, since we have a reference to the `SendPort` of the worker isolate, we can send it a message in this way:

```
// This is the 'SendPort' received from the worker isolate...
final sendPort = await receivePort.first as SendPort;

// ... which is later used to send messages to the worker isolate!
sendPort.send('button-xyz');
```

For simplicity, we're using strings as messages but it could be any object. In a real use case, you should create dedicated objects rather than hard-coding strings.

2. The '`button-xyz`' message is sent to the `ReceivePort` of the worker isolate. Since it has a listener, the message is processed and the analytic event is concurrently reported:

```
receivePort.listen((Object? message) async {
  if (message is String) {
    if (message.contains('route')) {
      await reporter.reportRouteOpened(message);
    } else if (message.contains('button')) { // Goes here
      → await reporter.reportButtonPress(message);
    }
  }
});
```

If we wanted to send a message from the worker isolate to the main one, we'd have to use the `sendPort` argument of the function.

When the main isolate is killed, all worker isolates are also killed. To manually kill the worker isolate, make sure to call `close()` on the port and `kill()` on the isolate reference. For example:

```
final receivePort = ReceivePort();
final isolate = await Isolate.spawn<SendPort>(someWork, receivePort.sendPort);

receivePort.close();
isolate.kill();
```

If you don't manually close the isolate you have spawned, it will stay alive until the program shuts down. If you only call `kill()`, but you forget to call `close()`, the program won't exit because the stream is still active. To sum up, here are the steps to spawn an isolate for long-running tasks:

1. Create a new `ReceivePort` object to communicate with the worker isolate.
2. Use `Isolate.spawn` to create the new isolate. Give it a function and the `SendPort` object of the `ReceivePort` you've created in the previous step. This is fundamental because each `ReceivePort` has its own send port object to communicate.
3. From the worker isolate, if you need to listen for messages from the main isolate (like in our example), create a new `ReceivePort` and send back its `SendPort` object. This allows for two-way communication between the isolates.
4. Send messages with `sendPort.send` and listen for messages with the `ReceivePort` object.

Step number 3 is optional because there might be some cases where a worker isolate is created, but it doesn't need to listen for messages. For example:

```
void ticker(SendPort sendPort) {
    Timer.periodic(
        Duration(seconds: 1),
        (timer) => sendPort.send(timer.tick),
    );
}

Future<void> main() async {
    final receivePort = ReceivePort();
    await Isolate.spawn<SendPort>(
        ticker,
        receivePort.sendPort,
    );

    receivePort.listen((Object? message) {
        if (message is int && message == 10) {
            receivePort.close();
        }

        if (message is int) {
            print('Number: $message');
        }
    });
}
```

In this example, the worker isolate doesn't listen for incoming messages. It just sends events every second with a timer, so we don't need to set up a two-way communication system. The main isolate prints numbers and shuts down the worker isolate when the message is `10`. Since `ReceivePort` is a (single subscription) stream, you can also `await-for` messages:

```
await for (final message in receivePort) {
  if (message is int && message == 10) {
    receivePort.close();
  }

  if (message is int) {
    print('Number: $message');
  }
}
```

You can use the `Stream.asBroadcastStream` method to transform a `ReceivePort` object into a broadcast stream. In this way, it won't buffer messages until a listener is registered. For example, if you wanted to receive a `SendPort` object from the worker isolate and listen for messages, you could do something like this:

```
final receivePort = ReceivePort();
await Isolate.spawn<SendPort>(myFunction, receivePort.sendPort);

// Make it broadcast so that we can Listen on the port multiple times
final broadcast = receivePort.asBroadcastStream();

// 'SendPort' of the worker isolate, for two-way communication
final sendPort = await broadcast.first as SendPort;

// We also Listen for messages from the worker isolate
broadcast.listen((message) {
  if (message is int && message == 10) {
    receivePort.close();
  }

  if (message is int) {
    print('Number: $message');
  }
});
```

If we didn't make this a broadcast stream, we would have gotten a runtime exception because the stream was listened twice (using `first` and `listen`). Only broadcast streams can be listened more than once, so we have to transform the `ReceivePort` object.

## 8.4.2 Good practices

When you want to execute long-running computations concurrently, isolates are the key. There are two ways to spawn new isolates:

1. For short-lived background tasks, use the `Isolate.run` static function. It will automatically spawn and manage a new isolate to concurrently run the function you've passed. This is the easiest way to execute a task in the background. For example:

```
final result = await Isolate.run<int>(() => calculateValue(a, b))
```

Some examples of short-lived tasks are: sorting lists, image processing, compressing files, or data encoding/decoding. These tasks don't maintain a state so each call is independent from the others.

2. For long-lived background tasks, use the `Isolate.spawn` static function. You will have to manage the isolate lifetime and use ports to exchange data with the worker isolate. This is can become quite verbose and complicated if you have to communicate with many isolates. For example:

```
final receivePort = ReceivePort();
final isolate = await Isolate.spawn<SendPort>(
    myFunction,
    receivePort.sendPort,
);

await for(final event in receivePort) {
    // Listen...
}

// Resources cleanup (when the worker is not needed anymore)
receivePort.close();
isolate.kill();
```

Some examples of long-lived tasks are: game engines, crash reporting tools, or background services that monitor OS events or HTTP services. These tasks generally maintain a "state" which influences method calls. The game engine for example should stay alive for the entire lifetime of the program because functions call results depend on the game state.

When you use `Isolate.spawn`, you have to decide how long the worker isolate has to live. It will stay alive for the entire program's lifetime if you do nothing. If you want to manually kill the worker isolate at a certain point, there are two options:

1. Isolates are killed when there are no more events in their microtask and event queues. By consequence, you could “gracefully” end an isolate by closing the port from the main isolate. In this way, the worker isolate will be killed when there will be no more events to process:

```
receivePort.close();
```

However, this method does NOT guarantee that the worker isolate will surely close. If you forget an open connection or a running timer for example, the worker isolate will still stay alive (because there would still be events in the queues). Calling `close()` on the receive port just stops the communication channel but it doesn’t also clear pending events on the worker isolate.

2. To make sure that a worker isolate is killed and no more resources are wasted, close all ports and call `kill()`:

```
receivePort.close();
isolate.kill();
```

This is the safest option because it DOES guarantee that the isolate will shut down. Even if there still were events to be processed, the isolate would be killed anyway.

If you use the `pause()` command, the isolate stops processing tasks from the event queue. It may still add new events to the queue in response to timers or receive-port messages for example. When the isolate is resumed, it starts handling the already enqueued events. For example:

```
// Pause the isolate
final Capability id = isolate.pause();

// Restart the isolate with the SAME id
isolate.resume(id);
```

The `isolate.pause()` method returns a `Capability` object, which is used to uniquely identify the pause request across the entire program. To resume the isolate, you have to pass the same object that the pause request returned.

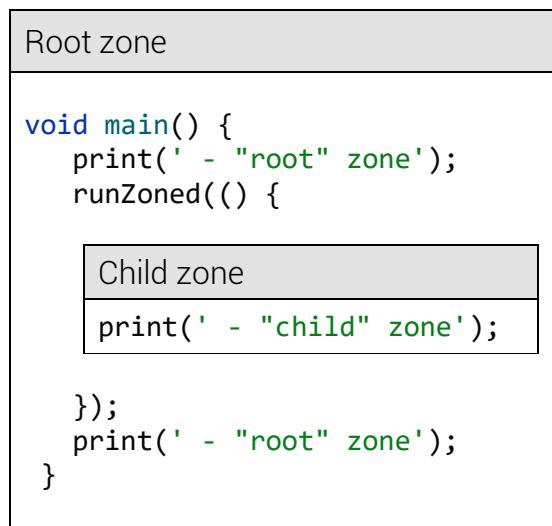
## Deep dive: Zones

Dart internally uses the Zone API to handle error propagation, asynchronous callbacks, zone-local values, microtask scheduling behavior, and more. You should use this “low-level” library judiciously and only when you really know what you’re doing. The reason is that debugging code that involves

lots of operations with zones is often hard to understand and test. Nevertheless, there are cases where you need to work with zones. Let's start with an example:

```
// This code prints:  
// - "root" zone  
// - "child" zone  
// - "root" zone  
void main() {  
  print(' - "root" zone');  
  runZoned(() {  
    print(' - "child" zone');  
  });  
  print(' - "root" zone');  
}
```

All Dart code is always executed inside a zone. The `main` function runs inside the root zone, which can be referenced using the static `Zone.root` property. To run your code in a separate zone, you can use the `runZoned` method from the `dart:async` library. This is a visual representation of how zones are created in the example:



Each zone represents an environment that is stable across asynchronous calls. In other words, you can see a zone as a “box” that executes asynchronous callbacks with the ability to handle uncaught errors within itself.

The main reason why zones are used is to handle uncaught errors, which often happen because an exception was raised using `throw` but no `catch` statement handled it. Uncaught errors also happen in `async` functions when a future completes with an error, for example. Uncaught errors are always

reported to the zone in which they were thrown, and the default behavior is to crash the program. Consider this example:

```
// Prints:  
// 'Hello!'  
void main() {  
    Future.delayed(const Duration(seconds: 2), () {  
        print('Message printed after 2 seconds');  
    });  
  
    print('Hello!');  
    throw Exception('Whoops!');  
}
```

Because of the unhandled exception, the thrown object is reported to the zone it belongs to (the root zone), and the program crashes. Consequently, the future will not print the message after two seconds. However, we can run the same code in a “protected” zone that will handle uncaught errors in a safe way. For example:

```
// Prints:  
// 'Hello'  
// ' !! > Unhandled Exception: Whoops!  
// 'Message printed after 2 seconds'  
void main() {  
    runZonedGuarded(  
        () {  
            Future.delayed(const Duration(seconds: 2), () {  
                print('Message printed after 2 seconds');  
            });  
  
            print('Hello!');  
            throw Exception('Whoops!');  
        },  
        (error, stackTrace) {  
            print(' !! > Unhandled $error');  
        },  
    );  
}
```

The `runZoneGuarded` function has a callback that is invoked when an uncaught exception is thrown. It’s the “safe” version of `runZoned` because it continues with the program execution after uncaught errors and it also processes all asynchronous callbacks scheduled (if any).

In fact, the example prints the message after two seconds (even if there is an unhandled exception caused by `throw`). A few more things to note:

- The `Isolate.run` function can handle uncaught errors, and it doesn't require you to set up a guarded zone. The reason is that the error happens in the root zone of the worker isolate and it's reported to the main isolate via message.
- A notable difference between `runZonedGuarded` and a `catch` block is that a guarded zone continues to execute after the uncaught errors occur. For example, if there were multiple asynchronous callbacks scheduled within the guarded zone, they'd still execute.
- Any zone that has an error handler (like the one created by `runZonedGuarded`) is an error zone. Errors of future chains can never go "outside" of the zone in which they were declared.
- To create new zones, always use either `runZoned` or `runZonedGuarded`. Do not manually create zones on your own with the `Zone.fork` method, which is quite complex.

A real-world use cases for zones is when you have your Dart or Flutter application running and you want to report fatal crashes (which are "uncaught errors" actually) to an external service. This is a common example of how Dart server applications are often initialized:

```
void main() {
  const service = MyCrashService();

  runZonedGuarded(
    () => runServer('127.0.0.1', 80),
    (error, stackTrace) async {
      print('LOG [Error]: $error');
      await service.reportError(error, stackTrace);
    },
  );
}
```

An HTTP server should always be online and thus the program should never crash. For this reason, servers are often executed in a guarded zone to log unexpected errors and continue the execution without crashing.

### Note

The same often happens with Flutter applications as well. Inside the `main` function, the `runZonedGuarded` function is used to protect the Flutter application and log errors to an external reporting service.

Even if zones are often used to handle uncaught errors, they are sometimes useful to redefine how the `print` method behaves. When you create a new zone, you can provide a `ZoneSpecification` object to redefine certain functions. For example, you can override the `print` method behavior:

```
void main() {
  runZoned(
    () {
      print('A'); // prints '> A :)'
      print('B'); // prints '> B :)'
    },
    zoneSpecification: ZoneSpecification(
      print: (self, parent, zone, message) {
        final newMessage = '> $message :)';
        parent.print(zone, newMessage);
      },
    ),
  );
  print('C'); // prints 'C'
}
```

Note that the override only has effect in the zone with a `ZoneSpecification` object. The root zone, which has no overrides, prints the message as usual. You can override more low-level operations, such as asynchronous callback management and microtasks handling. We are not covering these use cases since VM details and zones runtime management require low-level knowledges that go beyond the scope of this book.

## Deep dive: Language specification details

This section contains some interesting language features that are covered in the Dart specification document<sup>63</sup>. The contents of the document are very rigorous and formal, but here we've extracted and rephrased some curiosities about futures and streams:

- It is a compile-time error if the return type of a function with the `async` modifier is not a supertype of `Future<T>`. This does not include `void`. For example:

```
void ok() async {} // Valid
Future<void> stillOk() async {} // Valid
int notOk() async {} // Compile-time error
```

---

<sup>63</sup> <https://dart.dev/guides/language/spec>

Returning either `Future<void>` or just `void` from an asynchronous function does not make any difference.

- When a new isolate is created with `Isolate.run` or `Isolate.spawn`, the two isolates share the same isolate group. Because of this, Dart can make some optimizations, such as sharing code or improving message exchange efficiency.
- The `Isolate.spawnUri` method is a much slower version of `Isolate.spawn` that does not create a new isolate in the same group<sup>64</sup>. There are a few exceptional cases where `spawnUri` may be needed, but in general, you shouldn't use it.
- The `send` method of the `SendPort` class does not wait until the received has received the message. The corresponding `ReceivePort` can receive the message as soon as its isolate's event loop is ready to deliver it, independently of what the sending isolate is doing.
- The `send` method of the `SendPort` class generally creates copies of the objects that are sent to another isolate. However, in case of immutable objects (such as strings or objects created with a `const` constructor), no copies are made and the object is shared. Immutable objects cannot change, so it's safe to share them between isolates without risks.
- In *Section 8.4.2 – “Good practices”* we described two ways to kill a worker isolate. When two isolates share the same group, there actually is a third way to kill the worker isolate. Call the `Isolate.exit` method from the worker isolate to immediately kill it. For example:

```
await Isolate.spawn<SendPort>(
  sendPort) {
  sendPort..send(0)..send(1);
  Isolate.exit(sendPort, -1);
},
receivePort.sendPort,
);
```

The worker isolate first sends two messages (`0` and `1`). Then, the static `exit` method sends the final message (`-1`) through the port and then the worker isolate is immediately killed.

---

<sup>64</sup> <https://dart.dev/guides/language/concurrency>

- The `Isolate.exit` method should be used judiciously because it may be a risky operation<sup>65</sup>. It shuts down the isolate by immediately clearing the event loop queues so, for example, no `finally` blocks or asynchronous callback will ever run. As we have already recommended, prefer using `isolate.kill()` and/or close the associated port instead.
- The `Zone` class cannot be subclassed.
- Using either `runZonedGuarded` or `runZoned` is the recommended way to create new zones. Under the hood, they both use the `fork` method to create a new zone (with proper error-handling logic). You could also create a new zone in this way:

```
void main() {  
    Zone.current.fork().run(() => print('Hello from a new zone'));  
}
```

The problem is that this code doesn't handle uncaught errors and it requires you to manually create a new `ZoneSpecification` object. To avoid these problems, use `runZoneGuarded` or `runZoned`.

---

<sup>65</sup> <https://api.dart.dev/stable/dart-isolate/Isolate/exit.html>

# 9 – API overview and tests

---

## 9.1 Dart core libraries

The core SDK includes many libraries to work with I/O, dates, time, asynchronous code, and so on. However, the Dart team didn't include all the packages they made for the ecosystem. They wanted to find a good balance between what is fundamental for the SDK and what is optional. Some famous packages from the Dart<sup>66</sup> team are:

- `http`: a multiplatform, future-based API for HTTP requests;
- `path`: a path manipulation library for mobile, desktop, and web;
- `crypto`: implements SHA, MD5 and HMAC cryptographic functions;
- `collection`: enhances collections support with utility classes;
- `async`: utility functions that expand the `dart:async` library.

In this chapter, we've covered those libraries that are shipped with the SDK, leaving out packages hosted at pub. Official Dart packages hosted at `pub.dev` are regularly triaged and well-maintained.

### 9.1.1 The conversion library

The `dart:convert` library exposes a series of encoders and decoders for converting data between different representations, such as UTF-8 or JSON. This library also makes it easy to use converters into streams. Here's a first example:

```
Future<void> main() async {
  const decoder = Utf8Decoder();
  print(decoder.convert(const [72, 101, 108, 108, 111])); // Hello!

  const encoder = Utf8Encoder();
  print(encoder.convert('Hello!')); // 72 101 108 108 111
}
```

These two classes convert from `String` to `List<int>` and vice versa. There are various conversion classes you can use. All of them are subtypes of `Converter<S, T>`, which exposes the `T convert(S input)` method:

---

<sup>66</sup> <https://pub.dev/publishers/dart.dev/packages>

- `AsciiEncoder` and `AsciiDecoder`: they convert ASCII bytes to a `String` and vice versa.
- `Base64Encoder` and `Base64Decoder`: they encode lists of bytes using base64 encoding and vice versa.
- `HtmlEscape`: it escapes characters with special meaning in HTML. For example, the `&` symbol (ampersand) is replaced with `&amp;`.
- `JsonEncoder` and `JsonDecoder`: they convert JSON strings into Dart objects and vice versa;
- `Latin1Encoder` and `Latin1Decoder`: they convert strings of ISO Latin-1 characters to bytes and vice versa.

These converters can be used independently, as we've seen in the previous example, but they can also be bound to streams to convert incoming events. For example:

```
void main() {
  const encoder = Utf8Encoder();

  // Converts each tick into a string
  final stream = Stream<String>.periodic(
    const Duration(seconds: 1),
    (int tick) => '$tick',
  );

  // Binding to a stream means 'listening and encoding'
  encoder.bind(stream).listen(print);
}
```

Here we have a stream that, every second, is printing the string representation of a number. The `bind` method is used to listen on a stream and it automatically calls `convert` for each event. In this example, the stream emits '`1`', '`2`', '`3`'... but the console prints `48`, `49`, `50`... because each string is converted into its UTF-8 rune representation (which is a number).

#### 9.1.1.1 Converting from JSON strings

A very popular data exchange format is JSON, which is used by online APIs or to store configurations. For example, imagine you had to parse the following string:

```
const jsonSource = '''
{
  "name": "Dart",
  "created_in": 2011,
  "features": ["classes", "mixins", "inheritance"]
}
''';
```

The `dart:convert` library has a top-level constant variable called `json`, which is of `JsonCodec` type. The only problem is that the `decode` method returns a `Map<String, Object?>` which forces you to make casts every time you need to access an object. For example:

```
void main() {
  final jsonObject = json.decode(jsonSource) as Map<String, Object?>;

  final name = jsonObject['name']! as String;
  final creationYear = jsonObject['created_in']! as int;
  final features = (jsonObject['features']! as List<Object?>)
    .map<String>((e) => e! as String)
    .toList(growable: false);
}
```

This is very error-prone because you have to always make type casts and when it comes to lists, the code is quite verbose. Even if not explicitly mentioned in the official Dart documentation, there is a standard pattern to follow for JSON decoding. You should create a model class that internally casts the map and only exposes type-safe variables:

```
class DartModel {
  final String name;
  final int creationYear;
  final List<String> features;
  const DartModel._(
    required this.name,
    required this.creationYear,
    required this.features,
  );

  factory DartModel.fromJson(Map<String, Object?> source) {
    final list = source['features']! as List<Object?>

    return DartModel._(
      name: source['name']! as String,
      creationYear: source['created_in']! as int,
      features: list.map((e) => e! as String).toList(growable: false),
    );
  }

  void main() {
    final model = DartModel.fromJson(json.decode(jsonSource));

    print(model.name); // Dart
    print(model.features); // [classes, mixins, inheritance]
  }
}
```

The creation of a model class (`DartModel`) is a type-safe way to parse JSON strings into a Dart object. Casts and conversions happen inside the `factory` constructor, which is the only place that needs to be maintained in case of parsing errors. The same model class should also handle `null` values, if any:

```
const jsonSource = '''
{
  "name": "Dart",
  "created_in": null,
  "features": ["classes", null, "inheritance"]
}
''';
```

```
class DartModel {
  final String name;
  final int? creationYear;
  final List<String?> features;
  const DartModel._({
    required this.name,
    required this.creationYear,
    required this.features,
  });

  factory DartModel.fromJson(Map<String, Object?> source) {
    final featuresList = source['features']! as List<Object?>;

    return DartModel._(
      name: source['name']! as String,
      creationYear: source['created_in'] as int?,
      features: featuresList.map((e) => e as String?).toList(growable: false),
    );
  }
}
```

Notice that the creation year and some elements in the features list can be nullable in this new example. As such `creationYear` is now cast as to `int?`, and the `map` function of the list doesn't use the bang operator anymore. In case of nested objects, the logic is still the same: create a model class for the nested object and return it in the `factory` constructor of the enclosing object.

#### 9.1.1.2 Converting to JSON strings

To convert a Dart object into a JSON string, you need to build a `Map<String, Object?>` and then pass it to the `json.encode` method. It's generally a good idea to create a model class and define a method called (by convention) `toJson()`:

```

const jsonSource = '''
{
  "name": "Dart",
  "created_in": null,
  "features": ["classes", null]
}
''';

class DartModel {
  final String name;
  final int? creationYear;
  final List<String?> features;
  const DartModel._(
    required this.name,
    required this.creationYear,
    required this.features,
  );
}

factory DartModel.fromJson(Map<String, Object?> source) {
  final featuresList = source['features']! as List<Object?>

  return DartModel._(
    name: source['name']! as String,
    creationYear: source['created_in'] as int?,
    features: featuresList.map((e) => e as String?).toList(),
  );
}

Map<String, Object?> toJson() {
  return {
    'name': name,
    'created_in': creationYear,
    'features': features,
  };
}

void main() {
  final model = DartModel.fromJson(json.decode(jsonSource));
  final jsonString = json.encode(model.toJson());
  // Prints: {"name": "Dart", "created_in": null, "features": ["classes", null]}
  print(jsonString);
}

```

The `DartModel` class is now “complete” because it can convert from and to JSON. In particular, the `toJson()` method is meant to be used with `json.encode`, which takes care of converting the map into a JSON-encoded string.

### Note

Model classes for JSON objects should only be built from JSON-encoded strings. For this reason, the default constructor is library-private and objects can only be created using a `factory`.

This is a convention that is widely recognized in the Dart community so we recommend to follow it.

#### 9.1.2 The I/O library

The `dart:io` library gives access to files and directories in the operating system. It is not available for the web platform. For example, if we had a text file in the same location as the executable, we could easily read its contents in this way:

```
Future<void> main() async {
  // Assume that 'demo.txt' is in the same location as the executable.
  final file = File('demo.txt');

  final contents = await file.readAsString();
  final lines = await file.readAsLines();
  final bytes = await file.readAsBytes();

  print(contents.runtimeType); // String
  print(lines.runtimeType); // List<String>
  print(bytes.runtimeType); // Uint8List
}
```

The `Uint8List` type represents a fixed-length list of 8-bit unsigned integers. Very similarly, you can also write to a file (with different write-modes):

```
Future<void> main() async {
  final file = File('new_demo.txt');

  // The file is overwritten if it already exists.
  await file.writeAsString('hello!');

  // Writes at the end of the file.
  await file.writeAsString('world!', mode: FileMode.append);
}
```

In both cases, the file is created if it doesn't exist. All of these methods also have a synchronous counterpart, which cannot be awaited, such as `writeAsStringSync` or `readAsLinesSync`. In some cases, the synchronous version may be faster. For example:

```
Future<void> main() async {
  final file = File('somefile.txt');

  // OK
  if (file.existsSync()) {
    await file.delete();
  }

  // Warning: 'async exists()' is slow
  if (await file.exists()) {
    await file.delete();
  }
}
```

Both variants are fine but `exists()` is slower than `existsSync()` and there even is a warning from the `dart analyze` tool (and your IDE's analysis took). You can use a similar set of methods for the `Directory` class as well:

```
final directory = Directory('some_directory');

if (directory.existsSync()) {
  await directory.delete();
}

await directory.create();
```

The static `Directory.current` getter returns an object pointing to the current directory. You can listen for changes on directories and files as well:

```
final directory = Directory('some_directory');
final file = File('some_file');

if (directory.existsSync()) {
  directory.watch().listen(print);
}

if (file.existsSync()) {
  file.watch().listen(print);
}
```

Both `watch` methods return a `Stream<FileSystemEvent>` that emits new events when something is created, modified, deleted, or moved. You can specify the granularity of the events emitted by the `watch` stream using its named parameter:

```
file.watch(  
    events: FileSystemEvent.delete,  
) .listen(print);
```

In this example, we're only listening for events emitted when the file is deleted.

#### 9.1.2.1 Reading and writing from the console

The standard input and output streams are represented by the `stdin` and `stdout` getters. They are nothing more than streams you can use to read and write from the console. For example:

```
void main() {  
    stdout.writeln('First number:');  
    final firstNumber = stdin.readLineSync() ?? '';  
  
    stdout.writeln('Second number:');  
    final secondNumber = stdin.readLineSync() ?? '';  
  
    final a = double.tryParse(firstNumber);  
    final b = double.tryParse(secondNumber);  
  
    if (a != null && b != null) {  
        stdout.writeln('\nSum = ${a + b}');  
    } else {  
        stderr.writeln('\nError while parsing numbers!');  
    }  
}
```

This simple program reads two numbers from the console. The `readLineSync` method blocks until a full line is available (the terminator can either be CRLF or simply LF). We've used `stderr` to print the error message because it's the standard output stream for errors.

#### Note

The `Stdout` class will block until the entire output is written. In some situations, this may not be the desired behavior. In such cases, you can use the `nonBlocking` getter to make `Stdout` non-blocking.

Both `stdin` and `stdout` can handle emojis or ANSI escape codes to color the text console output.

### 9.1.2.2 The Process class

The `Process` class is used to run a process on a native machine. For example, you can use this code to spawn a new process that recursively lists all the contents of the `example` folder:

```
Future<void> main() async {
    if (Directory('example').existsSync()) {
        final process = await Process.start('ls', ['-l']);

        print('The process PID is ${process.pid}');
        process.kill();
    }
}
```

Processes can be started in two ways: using `Process.start`, which allows interacting with the running process, or using `Process.run`, which executes the process in non-interactive mode. When using `Process.start`, you need to read all data from `Process.stdout` to free all of the resources:

```
Future<void> main() async {
    final process = await Process.start('ls', ['-l']);
    stdout.addStream(process.stdout);

    print('Exit code: ${process.exitCode}');
}
```

This process runs `ls` to list all of the directories and then “links” the `stdout` of the spawned process with the `stdout` of the console app to print the output. In other words, `addStream` is used to merge the `ls` output into our program’s output stream. If the process accepts inputs, we can use its `stdin` to send commands:

```
final process = await Process.start('someprocess', []);
process.stdin.writeln('1');
```

The `exitCode` is used to indicate whether the program terminated successfully (`0`) or with an error. If you want to read the error output of a process, you can also access its `stderr` stream using the getter. For example, this is how you can easily check if the process had errors or not:

```
Future<void> main() async {
    final process = await Process.start('someprocess', []);
    final hasErrors = process.stderr.isEmpty;
}
```

The `isEmpty` getter of a stream is used to tell whether a stream contains any elements. If the `stderr` stream is empty, then the process hasn't reported errors.

### 9.1.2.3 Socket and ServerSocket

Clients and servers respectively use `Socket` and `ServerSocket` to communicate via TCP. The server creates a listening socket with the `bind` method and waits for incoming connections. For example:

```
Future<void> main() async {
    // Creates a new server
    final server = await ServerSocket.bind('127.0.0.1', 5974);
    final decoder = Utf8Decoder();

    server.listen((socket) {
        decoder.bind(socket).listen(print);
    });

    // Creates a socket that sends messages to the server
    final client = await Socket.connect('127.0.0.1', 5974);

    client.write('Hello world!');
    client.close();
}
```

The `listen` method of the server works with lists of integers, which are the raw bits received from the clients. To convert those numbers into human-readable strings, we must bind an utf8 decoder to the server stream using `decoder.bind`. This is a low-level class: in *chapter 22 – “HTTP servers and low-level HTML”* we will see the recommended ways to create servers in Dart.

### 9.1.3 Date and time

The `DateTime` class is used to indicate specific instants in the time. In particular, this class is able to represent time values that are at a distance of at most 100'000'000 days from 1970-01-01 UTC. For example, `DateTime` has many constructors:

```
void main() {
    final today = DateTime.now();
    final dartDayOne = DateTime(2011, 10, 10);
    final moonLanding = DateTime.utc(1969, 7, 20, 20, 18, 04);
    final fromString = DateTime.tryParse('2011-10-10 11:16:23Z');

    print(dartDayOne); // 2011-10-10 00:00:00.000
    dartDayOne.year = 2012; // Compiler error
}
```

The `tryParse` method is very strict about the formatting, and it returns `null` if the given string cannot be converted into a `DateTime` object. For readability and convenience, you could use named constants to build and compare dates:

```
void main() {
    // Using 'DateTime.may' instead of '5'
    final albertoBirthday = DateTime(1997, DateTime.may, 20);

    print(albertoBirthday.weekday == DateTime.friday); // true
    print(albertoBirthday.timeZoneName); // "W. Europe Daylight Time"
}
```

Days and month values begin at `1` and the week starts on Monday. By default, a `DateTime` object is created based on the local time zone unless you're using the `utc` named constructor, which is based on the UTC time zone. For example:

```
void main() {
    final italyTime = DateTime.now();
    final utcTime = DateTime.now().toUtc();

    print(italyTime.timeZoneOffset.inHours); // 2
    print(italyTime.timeZoneName); // W. Europe Daylight Time

    print(utcTime.timeZoneOffset.inHours); // 0
    print(utcTime.timeZoneName); // UTC
}
```

Since the above example was written on a platform with Italian date/time settings, the `DateTime` object was created with respect to that time zone (western Europe, DST). The second date instead was converted to UTC. The `timeZoneOffset` getter is used to calculate the difference between a certain time zone and UTC. Comparing dates is very easy:

```
void main() {
    final dartDayOne = DateTime(2011, DateTime.october, 10);
    final javaDayOne = DateTime(1995, DateTime.may, 23);

    print(dartDayOne.isAfter(javaDayOne)); // true
    print(dartDayOne.isBefore(javaDayOne)); // false
}
```

You can also use to `isAtSameMomentAs(other)` to determine if two dates occur at the same time. All of these comparisons are independent of whether the time is in UTC or in the local time zone. To add or subtract dates, you need to add `DateTime` in conjunction with `Duration`:

```

void main() {
  final dartDayOne = DateTime(2011, DateTime.october, 10);

  final dartDayTwo = dartDayOne.add(const Duration(days: 1));
  print('$dartDayTwo'); // 2011-10-11

  final dartDayMinusOne = dartDayOne.subtract(const Duration(hours: 24));
  print('$dartDayMinusOne'); // 2011-10-09

  final diff = dartDayOne.difference(dartDayTwo);
  print('${diff.inHours}'); // -24
}

```

The difference can either be positive or negative, depending on whether the starting date comes before or after the other date. The difference just subtracts the number of nanoseconds between the two points in time: it doesn't take calendar days into account. For example, the difference between two midnights in local time may be less than 24 hours times the number of days between them (if there was a DST change in between).

#### 9.1.3.1 Representing a span of time with Duration

The `Duration` class represents the difference from one point in time to another. The difference may be negative if it's computed between a later time and an earlier one. There only is a default constructor, whose parameters are all optional:

```

void main() {
  const someDuration = Duration(
    hours: 2,
    seconds: 12,
    milliseconds: 146,
  );
  print('$someDuration'); // 2:00:12.146000
}

```

Durations are context-independent. In our example, we're just creating an object representing a generic time span of 2 hours, 12 seconds, and 146 milliseconds. Under the hood, a `Duration` object uses microseconds to represent the time span, which is the sum of all the individual arguments of the constructor. For example:

```

void main() {
  const someDuration = Duration(hours: 1, minutes: 58, seconds: 13);

  print('${someDuration.inMicroseconds}'); // 7093000000
}

```

The various getters, such as `inHours` or `inMinutes`, return the number of whole minutes spanned by the object. As such, for example, the value of the minutes can be greater than 59 but you can get the remainder of 60 to get the actual minutes:

```
void main() {
    const someDuration = Duration(
        hours: 1,
        minutes: 58,
        seconds: 13,
    );

    print('${someDuration.inHours}'); // 1
    print('${someDuration.inMinutes}'); // 118
    print('${someDuration.inMinutes % 60}'); // 58

    print('${someDuration.inSeconds}'); // 7093
    print('${someDuration.inSeconds % 60}'); // 13
}
```

You can pass negative numbers to the `Duration` constructor. This class defines various operators so you can quickly make calculations with time spans. For example:

```
void main() {
    const start = Duration(minutes: 12, milliseconds: 6);
    const end = Duration(hours: 3, minutes: 68);

    print((start + end).inMinutes); // 260
    print((start + end).inMinutes % 60); // 20

    print('${start + end}'); // 4:20:00.006000 (4 hrs, 20 min)
    print('${start - end}'); // -3:55:59.994000 (-3 hrs, 55 min, 59 sec)

    print(start > end); // false
    print(start < end); // true
}
```

You can use `operator*` to multiply a `Duration` object with a `num`, which just multiplies the duration by the given factor.

#### 9.1.3.2 Measuring timespans with Stopwatch

As the name suggests, the `Stopwatch` class is used to measure time while it's running. It's a very precise tool to measure the elapsed time from the start to the end of the stopwatch. For example:

```

Future<void> main() async {
  final stopwatch = Stopwatch();

  stopwatch.start(); // Starts measuring the time
  await someFunction();
  stopwatch.stop(); // Stops the stopwatch and records the time

  print('Time elapsed: ${stopwatch.elapsed.inSeconds}');
}

```

The `elapsed` getter is of `Duration` type, so you can convert the time unit as you wish. You can stop the stopwatch with the `reset()` method, which sets the `elapsed` counter to zero.

#### 9.1.4 Native interoperability with FFI

Dart applications running on native platforms (mobile, desktop, and embedded) are able to use the `dart:ffi` library (Foreign Function Interface – FFI) to call native C APIs. For example, imagine you had the following folder structure:

demo/	demo/demo_library/
<ul style="list-style-type: none"> <li>• demo_library/</li> <li>• demo.dart</li> <li>• pubspec.yaml</li> </ul>	<ul style="list-style-type: none"> <li>• demo.h</li> <li>• demo.c</li> <li>• demo.def</li> <li>• CMakeLists.txt</li> </ul>

On the left, we have the contents of the `demo` folder, which contains a simple Dart program. On the right, we have the contents of the `demo/demo_library` folder, which contains a simple C program. We recommend using `CMake` for a smooth, cross-platform `Makefile` generation (and also to build the `.def` file, a module-definition file used when creating the library). This is the C source:

demo.h	demo.c
<pre>void demo();</pre>	<pre>int main() {   demo();   return 0; }  void demo () {   printf("Hello world!"); }</pre>

Once you have the C code ready, just call `cmake .` and then `make` to build all the necessary files. Make sure that the `demo.def` file exports the demo function we're about to load. On the Dart side, the code is a bit more complicated:

```
import 'dart:ffi' as ffi;
import 'dart:io'; // Also contains the 'Platform' class
import 'package:path/path.dart' as path;

typedef CFunction = ffi.Void Function(); // FFI signature of the C function
typedef DartFunction = void Function(); // Signature of the Dart function

void main() {
  final currentPath = Directory.current.path;
  var library = path.join(currentPath, 'demo_library', 'Debug', 'demo.dll');

  if (Platform.isMacOS) {
    library = path.join(currentPath, 'demo_library', 'libdemo.dylib');
  }

  if (Platform.isLinux) {
    library = path.join(currentPath, 'demo_library', 'libdemo.so');
  }

  final lib = ffi.DynamicLibrary.open(library);
  final DartFunction demo = lib
    .lookup<ffi.NativeFunction<CFunction>>('demo').asFunction();

  demo(); // Calling the C function, which prints "Hello world!"
}
```

We need to add a dependency to `path`, the official package from the Dart team to manipulate paths across various operating systems. In our example, we use the `join` function to join paths using a platform-specific separator. This is what we have defined in the program.

1. The two `typedefs` at the top are the signature of the C function we need to call and its Dart counterpart. You must provide the C function signature using FFI types (see `ffi.Void`) and then the Dart function signature (`void`).
2. In `main()`, we use `path.join` to join strings using a platform-specific separator. However, we still need to use a series of `ifs` to determine in which OS the program is running because libraries have different names and extensions. The `Platform` class, from `dart:io`, is used to retrieve info about the environment in which the program is running.

3. The `ffi.DynamicLibrary.open` call opens the dynamic library at the given path and reads all the functions it exports.
4. The `lookup` function gets a reference to the C function and “converts” it into a Dart type. It looks up a symbol in the `ffi.DynamicLibrary` and returns its address in memory. Inside the diamond, the `ffi.NativeFunction` expects the FFI signature of the C function, and the parameter (`demo`) must match the function name,

If the `lookup` function executes without throwing errors, you end up with a `demo` variable of type `DartFunction` which can be invoked as a regular Dart function. To make sure that the C function can be looked up, you can use the `providesSymbol(String)` function on `ffi.DynamicLibrary` to see if it evaluates to `true`.

## 9.2 Testing Dart code

Testing software is a vital development process since it helps verify that your program is working as expected. The more tests you write, the more confident you can be that known bugs have been fixed. As we have already seen in *chapter 1 – Section 1.2.2 “Creating a hello world project”*, the `dart create` command generates a few files and folders, among which we can find:

- `lib/`: the folder containing the Dart source code;
- `test/`: the folder containing the tests for the Dart code written inside `lib`;
- `pubspec.yaml`: the file that identifies a Dart project and declares its dependencies.

In this section, we’re exploring how to write tests. The `dart test` command considers any file that ends with the `_test.dart` postfix to be a test file. To write tests, you need to make sure that you’ve added the official Dart testing<sup>67</sup> package as a dependency in your `pubspec.yaml` file:

```
dev_dependencies:
  test: <latest-version>
```

The `dependency` section defines packages you want to include when deploying your program. The `dev_dependencies` section instead defines packages that are only used during development, and thus they aren’t included in the final executable. Tests are only executed during development so the `test` package should be included as a `dev_dependency`. The testing package is not included in

<sup>67</sup> <https://pub.dev/packages/test>

the Dart SDK, so you always need to add it as external dependency. To make an example, imagine we had to test the following class:

```
class Calculator {  
    const Calculator();  
  
    double sum(double a, double b) => a + b;  
    double subtract(double a, double b) => a - b;  
    double multiply(double a, double b) => a * b;  
    double divide(double a, double b) => a / b;  
  
    @override  
    String toString() => 'Calculator class';  
}
```

First of all, we have to create a new file inside the `test/` folder, which must have the `_test` postfix in the name. For example, we could call it `calculator_test.dart`. Then, we can write the test code using `expect` and `matchers`:

```
// Contents of the 'test/calculator_test.dart' file  
void main() {  
    const calculator = Calculator();  
  
    test('Making sure that numbers can be added correctly', () {  
        expect(calculator.sum(2, 3), equals(5));  
    });  
  
    test('Making sure that numbers can be subtracted correctly', () {  
        expect(calculator.subtract(2, 3), equals(-1));  
    });  
  
    test('Making sure that numbers can be multiplied correctly', () {  
        expect(calculator.multiply(2, 3), equals(6));  
    });  
  
    test('Making sure that numbers can be divided correctly', () {  
        expect(calculator.divide(6, 3), equals(2));  
    });  
}
```

A test file is a Dart program (notice the `main()` entry point) that uses one or more times the `test` method to run unit tests. In practice, “testing an application” means writing a lot of independent Dart programs that use the methods from the `test` package to verify the behavior of your code. Running the `dart test` command from the project root outputs this result:

```
C:\Dart\demo_project> dart test  
00:00 +4: All tests passed!
```

The test runner automatically looks for all Dart files whose name ends with `_test` and reports the result of executing the `test` functions. The first parameter is a string that describes what we're going to test, while the second one is a callback with the body of the test:

```
test('Making sure that numbers can be added correctly', () {  
  expect(calculator.sum(2, 3), equals(5));  
});
```

We have used proper wording to describe what we're going to test. The `expect` function is used to assert that a certain piece of code (on the left) returns the expected result (on the right). To make things a little cleaner, we could group tests by scope:

```
void main() {  
  const calculator = Calculator();  
  
  group('Testing the API', () {  
    test('Making sure that numbers can be added correctly', () {  
      expect(calculator.sum(2, 3), equals(5));  
    });  
  
    test('Making sure that numbers can be subtracted correctly', () {  
      expect(calculator.subtract(2, 3), equals(-1));  
    });  
  
    test('Making sure that numbers can be multiplied correctly', () {  
      expect(calculator.multiply(2, 3), equals(6));  
    });  
  
    test('Making sure that numbers can be divided correctly', () {  
      expect(calculator.divide(6, 3), equals(2));  
    });  
  });  
  
  group("'Object' overrides", () {  
    test('Testing the toString override', () {  
      expect('$calculator', equals('Calculator class'));  
    });  
  });  
}
```

The `group` function groups a series of tests with similar purposes under the same “category”. This is especially useful when you run tests on the IDE since they give you more context about the test

that failed. In *chapter 20 – Creating and maintaining a package*, you will find a complete example of a well-tested Dart application with all the best practices we recommend.

### 9.2.1 Matchers

A matcher is a function that specifies the expectations of a test. Matchers are passed to the `expect` function to ensure that a certain piece of code produces the expected result. For example:

```
test('Making sure that numbers can be divided correctly', () {
  expect(calculator.divide(8, 4), equals(2)); // 'equals' matcher
});

test('Making sure that 0/0 is an undefined value', () {
  expect(calculator.divide(0, 0), isNaN); // 'isNaN' matcher
});

test('Making sure that toString outputs with a capital letter', () {
  expect('$calculator', startsWith('C'))); // 'startsWith' matcher
});

test('Making sure that toString outputs with a capital letter', () {
  expect(calculator.sum(8, 4), greaterThan(4)); // 'greaterThan' matcher
});
```

For example, the `isNaN` (where NaN stands for Not a Number) matcher ensures that the function returns `double.nan` and `startsWith` asserts that a string is prefixed with a certain value. There are dozens of matchers you can use<sup>68</sup>. Here's a list of some matchers you may often see:

- `isNull / isNotNull`: Use them to check whether a value is `null` or not `null`:

```
test("Verifying 'isNull' and 'isNotNull'", () {
  int? value1;
  int value2 = 0;

  expect(value1, isNull);
  expect(value2, isNotNull);
});
```

- `allOf`: This matcher ensures that all of the matchers you've passed in (up to 7) verify the condition:

---

<sup>68</sup> <https://pub.dev/documentation/matcher/latest/matcher/matcher-library.html>

```

test("Verifying the 'allOf' matcher", () {
    final value = 10.45;

    expect(value, allOf([
        greaterThan(-10),
        isA<double>(),
        isPositive,
    ]));
});

```

The `isA<T>` matcher ensures that the object to be tested is of type `T`.

- `orderedEquals / unorderedEquals`: these matchers verify that the two lists have the same length. The ordered version makes sure that items have the same positions in both lists while the unordered version just checks the presence of the items in the iterables.

```

test('Verifying the ordered and unordered list matchers', () {
    const list = <int>[3, 6, 9, 12, 15];

    expect(list, orderedEquals(const [3, 6, 9, 12, 15]));
    expect(list, unorderedEquals(const [6, 3, 15, 12, 9]));
    expect(list, unorderedEquals(const []));
});

```

There are a lot of exception-specific matchers you can use to detect when an exception is thrown. For example:

```

int parseInteger(String value) {
    final parsedInt = int.tryParse(value);

    if (parsedInt == null) {
        throw FormatException('Some message');
    }

    return parsedInt;
}

test('Verifying that exception types can be evaluated', () {
    expect(() => parseInteger(''), throwsFormatException);
});

```

There is a matcher for any exception and error type in the SDK. If you created a custom exception by implementing `Exception` for example, we recommend to the `throwsA` and `isA` matchers. For example:

```
test('Verifying that custom exception types can be evaluated', () {
  expect(() => parseInteger(''), throwsA(isA<SomeCustomException>()));
});
```

The `isA<T>` matcher ensures that the object has the given runtime type while `throwsA` verifies that an exception is thrown. If you extend the `Matcher` class, you can create your custom matchers. For example, consider this case:

```
test('Verify value', () {
  const value = 5.0004675 / 3.15; // 5.0004675 / 3.15 = 1.58745
  expect(value, equals(1.587)); // Doesn't pass because 1.587 != 1.58745
});
```

This test fails because `equals` precisely checks whether the two numbers are equal or not. We can create a reusable matcher that compares floating point values with some tolerance. For example, this is a possible implementation:

```
class MoreOrLessEquals extends Matcher {
  final double value;
  final double precision;
  const MoreOrLessEquals(
    this.value,
    this.precision = 1.0e-12,
  ) : assert(precision >= 0, 'The precision must be >= 0');

  @override
  bool matches(Object? item, Map<Object?, Object?> matchState) {
    if (item is double) {
      // Compares two floating-point values with a tolerance
      return (item - value).abs() <= precision;
    }

    // Anything else that is not 'double' defaults to 'operator=='
    return item == value;
  }

  @override
  Description describe(Description description) =>
    description.add('$value ($precision)');
}
```

The `matches` function determines if the actual value matches the expected one. In this case, we check if the given value is within the tolerance range. We can use `MoreOrLessEquals` in our tests as with any other matcher. For example:

```
expect(value, const MoreOrLessEquals(1.587, precision: 1.0e-3));
```

The `describe` method of a matcher is used to build a textual description of the matcher. An even better implementation would also override the `describeMismatch` function:

```
@override  
Description describe(Expression desc) => desc.add('$value ($precision)');  
  
@override  
Description describeMismatch(  
    Object? item,  
    Description mismatchDescription,  
    Map<Object?, Object?> matchState,  
    bool verbose,  
) =>  
    super.describeMismatch(  
        item,  
        mismatchDescription,  
        matchState,  
        verbose,  
    )..add('$item is not in the range of $value ($precision).');
```

This is very helpful for the developer because a detailed report is printed to the console when the matcher fails. For example, thanks to the `describeMismatch` override, when `MoreOrLessEquals` fails to match a value, the test console prints this report:

```
Expected: 1.577 (±0.001)  
Actual: <1.58745>
```

```
Which: 1.58745 is not in the range of 1.577 (±0.001).
```

The `describe` method controls what is printed after “Expected:” while `describeMismatch` controls what is printed after “Which:”.

## 9.2.2 Testing futures and streams

When you write tests with `async` and `await`, there are no special considerations to do. For example:

```
Future<int> computeValue(int value) {  
    return Future.delayed(const Duration(seconds: 2), () => value * 2);  
}  
  
test('Verifying the result of a future', () async {  
    final value = await computeValue(5);  
    expect(value, equals(10));  
});
```

If you don't want to use `async` and `await`, then you need to manually ensure that the `test` function does not exit before the future completion. To do this, you need the `completion` matcher (which waits for a `Future<T>` to successfully complete). For example:

```
test('Verifying the result of a future without await', () {
  final value = computeValue(5);
  expect(value, completion(equals(10)));
});
```

We think that `async` and `await` make the code more readable and intuitive, so we recommend using them. When it comes to streams, there is a special `emitsInOrder` matcher to ensure that the stream emits the values you have specified. For example:

```
void main() {
Stream<int> generator() async* {
  for(var i = 0; i < 5; ++i) {
    await Future.delayed(const Duration(seconds: 1));
    yield i;
  }
}

test('Verifying the ordered and unordered list matchers', () {
  final stream = generator();

  expect(stream, emitsInOrder(const [0, 1, 2, 3, 4]));
});
```

You will see that the test takes five seconds to complete because it waits for all events to be emitted before terminating. This is an easy example because the stream produces values of the same type (`int`). Imagine you had a stream that emitted custom objects:

```
class Value {
  final int number;
  final String string;
  const Value(this.number, this.string);
}

Stream<Value> generator() async* {
  for(var i = 0; i < 5; ++i) {
    await Future.delayed(const Duration(seconds: 1));
    yield Value(i, 'Value: $i');
  }
}
```

This case is more complicated to test because we need to check the internal properties of the object to see if they're valid. In this case, the `expectAsync1` matcher is a great solution. For example:

```
Stream<Value> generator() async* {
  for(var i = 0; i < 5; ++i) {
    await Future.delayed(const Duration(seconds: 1));
    yield Value(i, 'Value: $i');
  }
}

test('Verifying the emitted events of a stream', () {
  final stream = generator();
  var counter = 0;

  stream.listen(expectAsync1<void, Value>(
    value) {
    expect(value.number, equals(counter));
    expect(value.string, equals('Value: $counter'));

    ++counter;
  },
  count: 5,
);
});
```

In this example, we want to test the stream, but we also want to check the contents of the `Value` objects yielded by the generator. The `expectAsync1` function does three important things:

1. The callback, which takes a single parameter, contains a reference to the object emitted by the stream. In our case, `value` is of type `Value`, so we can use normal `expect` calls to check its internal properties.
2. Thanks to the `count` parameter, `expectAsync1` ensures that the stream exactly emits the specified number of events.
3. `expectAsync1` ensures that the `test` function waits for the stream to finish emitting events before finishing.

The number one at the end of the matcher name (`expectAsync1`) indicates how many parameters the callback function has to pass. In our case, the `listen` function of the stream expects a callback with a single parameter (the event being yielded) so we only need to pass a single parameter (hence the `expectAsync1`).

These matchers can also be used outside of the stream test scope. For example, if you wanted to ensure that a function was called a certain number of times and it took two parameters, you could use `expectAsync2`. For example:

```
void main() {
    double meanValue(int a, int b) => (a + b) * 0.5;

    test('Making sure that meanValue is called 3 times', () async {
        final function = expectAsync2<double, int, int>(meanValue, count: 2);

        await Future.delayed(const Duration(milliseconds: 200), () {
            expect(function(2, 6), equals(4));
        });
        await Future.delayed(const Duration(milliseconds: 200), () {
            expect(function(1, 2), equals(1.5));
        });
    });
}
```

This test successfully executes: it waits for all the futures to finish and ensures that `meanValue` is called precisely two times. If you changed the count to something that is not `2`, the test would fail because the function wouldn't be called two times. There are five variants of the `expectAsyncN` function.

### 9.2.3 Tests setup

In *chapter 22 – HTTP servers and low-level HTML* we will see how to create and test HTTP servers in Dart. For now, it's enough for you to know that the `HttpServer` class from `dart:io` is used to create an HTTP server and the `close()` function stops the server. If we had to write some tests, they would look like this:

```
void main() {
    test('Testing feature A of the server', () async {
        final server = await HttpServer.bind('localhost', 3500);
        testFeatureA(server);
        await server.close();
    });

    test('Testing feature B of the server', () async {
        final server = await HttpServer.bind('localhost', 3500);
        testFeatureB(server);
        await server.close();
    });
}
```

As you can see, for each test we're duplicating the logic that starts and stops the server. If we added more tests, we would need to duplicate even more code. In such cases, where tests share some common initialization and/or finalization code, you can use the `setUp` and `tearDown` functions. For example:

```
void main() {
    late HttpServer server;

    // This is executed before any test
    setUp(() async {
        server = await HttpServer.bind('localhost', 3500);
    });

    // This is executed after any test
    tearDown(() async {
        await server.close();
    });

    test('Testing the A feature of the server', () async {
        testFeatureA(server);
    });

    test('Testing the B feature of the server', () async {
        testFeatureB(server);
    });
}
```

This approach is more readable and avoids code duplication. The body of the `setUp` function is executed before each test and the body of the `tearDown` function is executed after each test. You can see them as the “constructor” and the “destructor” of a test.

### Good practice

The `setUpAll` function is executed once before all tests and `tearDownAll` is executed once after all the tests are completed.

When you need to run initialization and finalization code for each test, use `setUp` and `tearDown` respectively. When you need to run initialization and finalization code for the whole test file, use `setUpAll` and `tearDownAll`.

Each single test case can also be configured with specific settings, such as:

```

test(
  'Making sure that numbers can be added correctly', () {
    expect(calculator.sum(2, 3), equals(5));
  },
  // When running on Windows, this test is skipped
  skip: Platform.isWindows,
  // By default, tests time out after 30 seconds of inactivity but here
  // you can change the time
  timeout: Timeout(const Duration(seconds: 50)),
  // In case the test fails, it is retried the given number of times
  retry: 2,
);

```

The special `onPlatform` parameter allows tests to be configured on a platform-by-platform basis. Note that map's keys are predefined `Strings` whose name must match the host platform name. For example:

```

test(
  'Making sure that numbers can correctly be added', () {
    expect(calculator.sum(2, 3), equals(5));
  },
  onPlatform: {
    'windows': Timeout(const Duration(seconds: 5)),
    'browser': Skip('No browser support yet!'),
  },
);

```

The `Timeout` class is used to change the default timeout of a test. The `Skip` class is used to skip the test execution with a message. For a complete overview of all combinations of parameters, refer to the official test package documentation<sup>69</sup>.

## 9.2.4 Code coverage

Code coverage is a very popular software testing metric that determines how many lines of code have been validated by a test. If you want to collect code coverage for your tests, first add the `coverage`<sup>70</sup> package as `dev_dependency` and then run, in order, these commands:

<sup>69</sup> <https://pub.dev/packages/test#configuring-tests>

<sup>70</sup> <https://pub.dev/packages/coverage>

1. `dart pub global activate coverage`: activates the coverage tool and allows using it from anywhere, even when not inside a Dart package.
2. `dart pub global run coverage:test_with_coverage`: runs all unit tests, creates a LCOV coverage report, and stores it in the `coverage/` folder. This specific command must be run inside the package root (the same folder as the `pubspec` file).

This is the recommended way of collecting code coverage for a Dart package. It runs all tests and gathers the coverage reports inside a `.lcov` file format. LCOV files are not human-readable, but if you install the popular `genhtml` command line tool, it can generate a nice website to easily visualize the coverage status for each file. For example:

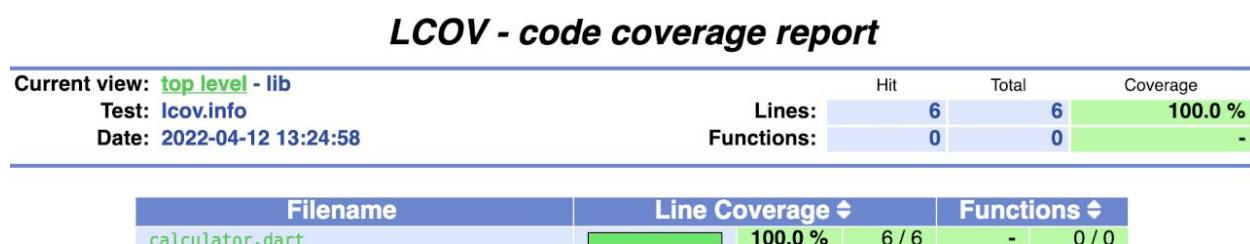


Figure 9.1: The coverage report home page

This is an HTML page created by the `genhtml` tool. It parsed the LCOV file and generated a colored, human-readable local website. Clicking on the Dart files shows a new page with the coverage details for each line:

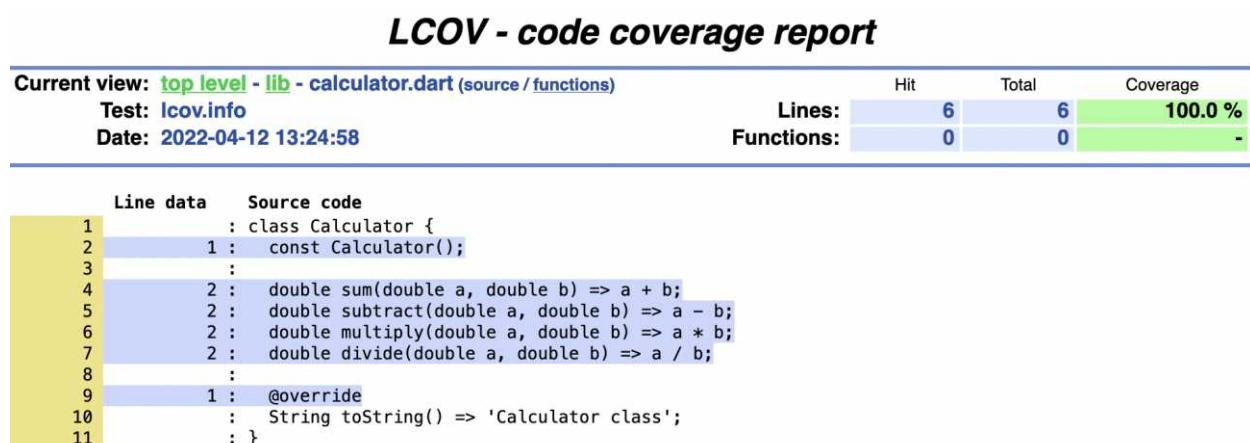


Figure 9.2: The coverage report of the `calculator.dart` file

The numbers next to the highlighted code indicate how many times a test hit that specific line. On the top-right corner, we see that the coverage is 100% so all of the code we've written is hit at least once by a test. The coverage tool can ignore some parts of the code with special comment strings:

- Use `coverage:ignore-line` to exclude a single line from the report. For example:

```
if (value.floor() > 10) {  
    print('value = $value'); // coverage:ignore-line  
}
```

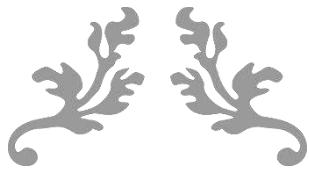
- Use `coverage:ignore-start` and `coverage:ignore-end` to exclude a range of lines from the report. For example:

```
// coverage:ignore-start  
if (value.floor() > 10) {  
    print('value = $value');  
}  
// coverage:ignore-end
```

- Use `coverage:ignore-file` to exclude an entire file from the report. This comment should be placed at the top of the Dart file you want to ignore. For example:

```
// coverage:ignore-file  
import 'dart:math';  
  
void main() { /* code ... */ }
```

These commands don't have spaces and must always be defined in a comment block.



## Part 2: The Flutter framework

---

10 – Flutter, widgets and trees

11 – Material, Cupertino and CustomPaint

12 – State management

13 – Routes and navigation

14 – Layouts and responsiveness

15 – Internationalization and accessibility

16 – Assets and images

17 – Animations

18 – Forms and gestures

19 – Testing Flutter applications



# 10 – Flutter, widgets and trees

---

## 10.1 Flutter overview

Flutter is nothing more than a Dart package with some external tools that build binaries for various platforms. To create a new Flutter project, you should use the `flutter create` command (rather than `dart create`) because it generates a new regular Dart project plus some more content:

- the `android/` folder contains a new Android project that is integrated with the Flutter code when using the `flutter build` command. The default language is Kotlin;
- the `ios/` folder contains a new iOS project that is integrated with the Flutter code when using the `flutter build` command. The default language is Swift;
- the `web/` folder contains an HTML file and some static resources needed to launch Flutter on web browsers;
- the `macos/` folder contains a new macOS project that is integrated with the Flutter code when using the `flutter build` command;
- the `windows/` folder contains a new Windows project that is integrated with the Flutter code when using the `flutter build` command;
- the `linux/` folder contains a new Linux project that is integrated with the Flutter code when using the `flutter build` command;

As usual, the application code is located in `lib/`, and tests go inside `test/`. Whenever you need to configure platform-specific settings, you can navigate to one of the above folders and modify the projects. We recommend using either Android Studio or VS Code.



Figure 10.1: Android Studio controls of a running Flutter application.

For example, in the above image you see how the Android Studio buttons look when working on a Flutter project (in VS Code, buttons look similar). Rather than manually building and running the application from the command line, you can use these colored buttons:

- the leftmost triangle builds and runs the application on the given target platform (in debug mode). It's the same as calling `flutter run lib/main.dart` on the console. The rightmost square instead immediately stops the running app;
- the lightning icon triggers the hot reload feature, which injects the updated source code into the Dart VM so that you can instantly see the changes in the running device.

You can use the hot reload feature when a Flutter project is executed in debug mode. Whenever you change the Dart code, you can press the yellow button on the IDE to immediately “refresh” the application in your running device to see the changes.

### Note

When you make changes to HTML code for example, you just need to save the file and refresh the browser to see the new UI immediately. Hot reload in Flutter works in the same way. After you've made some changes to the Dart code, you can refresh the device in which the application is running to see the new UI immediately.

There are some special cases where hot reload won't work, and so you need to stop the application and re-run it again. For example, when you convert an `enum` into a `class` (or vice versa) or you make changes to a `static` variable, hot reload does not work. There are three “refresh” modes overall:

1. **Hot reload.** It loads changes into the Dart VM and refreshes the application very quickly. The `main()` entry point is not re-run and the application state is preserved.
2. **Hot restart.** It loads changes into the Dart VM and restarts the application. In this case, the `main()` entry point is re-run and the application state is lost.
3. **Full restart.** This is a manual step that stops the application and re-compiles it. To perform a full restart, you have to close the application and re-run it (which then takes longer than the other two modes).

At the time of writing this book, Flutter web doesn't support hot reload. All the other platforms instead support hot reload, hot restart, and full restart. For better cross-platform support, use the `debugPrint` function instead of Dart's `print` to print messages to the console. Both functions are fine, but Flutter's `debugPrint` function has a better platform-specific behavior.

### 10.1.1 Build modes

Flutter projects can be compiled in three modes. Depending on where you are in the development stage, you should decide which mode is the best for your needs. In debug mode, the application is built for debugging on an emulator, a simulator, or a physical device with the following settings:

- assertions made with `assert` are evaluated;
- debugging tools (which are also well-integrated on IDEs) are enabled;
- large binary size and slower execution speed, but hot reload is enabled;
- no minification or tree shaking.

The `flutter run` command compiles your project in debug mode, which is the same as using the green triangle in Android Studio or VS Code. Release mode is for deploying the application, and it offers the maximum optimization level:

- assertions are disabled;
- no debugging information;
- fast startup, fast execution, and small package size;
- tree shaking and code minification are performed where needed.

The `flutter run --release` command compiles your project in release mode. Alternatively, you can use the `flutter build` command to compile in release mode for a specific target:

- `flutter build appbundle` or `flutter build apk` to deploy for Android devices;
- `flutter build ios` to deploy for iOS devices;
- `flutter build web` to deploy for the web;
- `flutter build windows` to deploy for Windows;
- `flutter build macos` to deploy for macOS;
- `flutter build linux` to deploy for Linux;

Profile mode has minimal debugging abilities and its performance are very close to a release mode build. It can only be run on physical devices, and it's used to measure speed and performance using DevTools, which will be covered in *Appendix – Section 1 “Working with DevTools”*. The `flutter run --profile` command compiles your project in profile mode.

### 10.1.2 Tree shaking

Tree shaking is an optimization process that eliminates dead code. In Flutter, it's enabled by default (in release or profile mode), and it's automatically performed by the Flutter tool when you build the project. Look at this example:

```

const isGood = true;

if (isGood) {
  print('Good');
} else {           // Dead code
  print('Bad');  // Dead code
}                 // Dead code

```

The compiler is smart enough to understand that the `else` branch can never be reached because `isGood` is a constant. When running in profile or release mode, the compiler removes dead code to reduce the final executable size. Even if we had declared the variable using `final` or `var`, the `else` branch would have still been eliminated. This is the compiler output in the various modes:

Debug mode	Profile mode	Release mode
<pre> const isGood = true;  if (isGood) {   print('Good'); } else {   print('Bad'); } </pre>	<pre> const isGood = true;  print('Good'); </pre>	<pre> const isGood = true;  print('Good'); </pre>

A more interesting example of dead code elimination is in the Flutter foundation library. It has some useful constant values that represent the build mode of your application. For example:

```

// The foundation library
import 'package:flutter/foundation.dart';

Server buildHttpServer() {
  // 'kDebugMode' determines if the application is run in debug mode or not
  if (kDebugMode) {
    return FakeServer();
  } else {
    return RealServer();
  }
}

```

There are four `const` values:

- `kReleaseMode`: true if the application was compiled in release mode;
- `kProfileMode`: true if the application was compiled in profile mode;
- `kDebugMode`: true if the application was compiled in debug mode;
- `kIsWeb`: true if the application was compiled to run in the web.

In our example, we use `kDebugMode` to determine if we should return a local server for internal testing or the production server. In profile and release mode, the compiler is guaranteed to discard the branch that returns `FakeServer` because it can never be reached. For example, this is how the compiled code would look in different modes:

Debug mode	Profile mode	Release mode
<pre>Server buildHttpServer() {     if (kDebugMode) {         return LocalServer();     } else {         return RealServer();     } }</pre>	<pre>Server buildHttpServer() {     return RealServer(); }</pre>	<pre>Server buildHttpServer() {     return RealServer(); }</pre>

Another case where code gets automatically removed is when one or more members of a class are never used. For example:

```
class Example {
    const Example();

    void _doSomething() => print('Something');
    void printValue() => print('10');
}

void main() {
    const Example().printValue();
}
```

The `_doSomething` method is private, and then it can only be used within the scope of the `Example` class. Since it's never used, the compiler can safely assume that it can be removed without breaking the program flow.

All lines after a `return` statement are considered “dead code” because they will never be reached. For example:

```
int randomValue() {
    return Random().nextInt(10);

    final a = 10; // dead code
    return a; // dead code
}
```

This code successfully compiles, but everything after the first `return` statement gets tree shaken.

## 10.2 Widgets

When you create a new Flutter project, the `flutter create` tool generates the classic `main.dart` file with the `void main()` entry point, as for any other regular Dart application. For example, this is what a basic Flutter application looks like:

```
void main() {  
  runApp(  
    const MaterialApp(  
      home: Text('Hello world!'),  
    ),  
  );  
}
```

Any Flutter application must call the `runApp` method inside the `main` function to “inflate” the given object and render it to the screen. A **widget** is a Dart class that either extends `StatelessWidget` or  `StatefulWidget`. In Flutter, user interfaces are created by nesting widgets one inside the other to create a sort of “widgets hierarchy” called *widget tree*. For example:

Dart code	Widget tree representation
<pre>runApp(   const MaterialApp(     home: Center(       child: Text('Hello world!'),     ),   ), );</pre>	<pre>graph TD; A[MaterialApp] --- B[Center]; B --- C[Text]</pre>

The `runApp` method takes a single parameter (a widget) which becomes the root of the widget tree. Widgets are nested one inside the other using the `child` named parameter, which is often called in this way by convention (but it may also have another name, such as `home` or `body`).

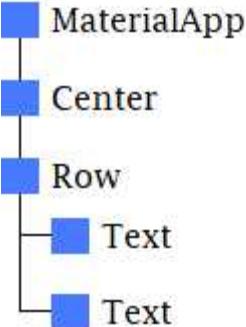
### Note

The “widget tree” is internally created and maintained by the Flutter framework for performance reasons. You only need to care about creating widgets and nesting them one inside the other to compose the UI. We will analyze the widget tree in detail in section 10.3 – “*Widget, Element and RenderObject trees*”.

A widget can also have two or more children. In this case, by convention, there should be a named parameter called `children` which takes a list of widgets. For example:

```
runApp(  
  MaterialApp(  
    home: Center(  
      child: Row(  
        // A row places multiple widgets along the horizontal axis  
        children: const [  
          Text('Hello'),  
          Text('world!'),  
        ],  
      ),  
    ),  
  ),  
);
```

The widget tree from this example looks a bit different because there are two leaves represented by `Text` widgets:

Dart code	Widget tree representation
<pre>runApp(MaterialApp(   home: Center(     child: Row(       children: const [         Text('Hello'),         Text('world!'),       ],     ),   )), );</pre>	

We could infinitely nest other widgets inside `Row` to add more content to the UI. This is the general idea that rules Flutter development. It's very easy: reuse or create new widgets and nest them one inside the other.

### Note

The `MaterialApp`, `Row`, and `Center` widgets are part of the Flutter framework and we will explore them in the next chapter.

### 10.2.1 Stateless and Stateful widgets

A Dart class becomes a Flutter widget when it extends `StatelessWidget` or  `StatefulWidget` and overrides the `build` method. Before creating a new widget, you have to decide whether its state will change over time. For example, the state may change when:

- the user taps on a widget and its color changes;
- a part of your application is listening to a stream that updates the internals of a widget every time a new event is emitted;
- an external dependency of the widget changed and you must update some properties in response to that change.

In other words, you have to ask yourself if the widget is *static* (it will never change once created) or if it's *dynamic* (it can change its internal configuration in response to events). The answer translates into Dart code by extending one of these two classes:

- `StatelessWidget`. Subclass this type to create a widget whose state does not change after the creation. Stateless widgets can be seen as “standalone” blocks that always maintain the same internal configuration over time. For example:

```
class MyName extends StatelessWidget {  
  const MyName();  
  
  @override  
  Widget build(BuildContext context) {  
    return const Center(  
      child: Text('Alberto Miola'),  
    );  
  }  
}
```

This widget only wants to render a string at the center of the screen. It's a “static” object that, once created, will never change.

The `build` method is where you nest other widgets inside `MyName` to compose the UI. Even if we had passed the name from the outside, the widget wouldn't have changed anyway because it's immutable. For example:

```

class MyName extends StatelessWidget {
    final String name;
    const MyName({
        required this.name,
    });

    @override
    Widget build(BuildContext context) {
        return Center(
            child: Text('My name is: $name'),
        );
    }
}

```

By convention (and as a good practice) all stateless widgets must define a `const` constructor and thus have all of its fields declared as `final`. Even if we pass the name from the outside, the widget configuration will not change over time unless we rebuild it with a different `name`.

- `StatefulWidget`. Subclass this type to create a widget whose internal configuration will change over time. Stateful widgets are used when widgets returned by the `build` method need to change in response to one or more events. For example:

```

class Counter extends StatefulWidget {
    const Counter();

    @override
    State<Counter> createState() => _CounterState();
}

class _CounterState extends State<Counter> {
    var counter = 0;

    @override
    Widget build(BuildContext context) {
        // Renders a button with centered text
        return ElevatedButton(
            onPressed: () => setState(() {
                counter++;
            }),
            child: Text('$counter'),
        );
    }
}

```

The stateful widget (`Counter`) is immutable but its state, represented by a class that extends `State<T>`, is mutable. The `setState` method is defined in `State<T>`: it's used to update the internals of the state class and call `build` again to update the UI. In this case, when the user taps the button, the `counter` variable is increased and then the entire `build` method is re-run to update the text.

We want to emphasize again that both stateless and stateful widgets are meant to be immutable classes and so you should always define a `const` constructor. In particular, a `StatefulWidget` is immutable but its associated `State<T>` class is mutable and allows dynamic changes of properties.

### Note

*Chapter 11 – State management* makes a deep-dive on `setState` and the other state management solutions for Flutter. There is a lot to say about it so we've dedicated an entire chapter. For now, it's enough for you to know that `setState` is used to “refresh” the state of a `StatefulWidget` to repaint the widgets it returns.

The `State<T>` class is mutable, lives along with the associated `StatefulWidget`, and “survives” to rebuilds. In fact, when the stateful widget is rebuilt, only the `build` method is re-evaluated. Its instance members are not rebuilt and keep their values. For example:

```
class _CounterState extends State<Counter> {
    var counter = 0;

    @override
    Widget build(BuildContext context) {
        // The contents inside here instead are rebuilt
        return ElevatedButton(
            onPressed: () => setState(() {
                counter++;
            }),
            child: Text('$counter'),
        );
    }
}
```

Anything outside of the `build` method is not rebuilt and keeps its value. As such, the `setState` call only rebuilds the widgets returned by the `build` method, but it does not also reset the `counter` variable. In other words, widgets are rebuilt from scratch to show the new value, but `counter` increases the old value by one. This is an example of a broken implementation:

```

class _CounterState extends State<Counter> {
  @override
  Widget build(BuildContext context) {
    var counter = 0; // Now the counter is INSIDE the build method

    return ElevatedButton(
      onPressed: () => setState(() {
        counter++;
      }),
      child: Text('$counter'),
    );
  }
}

```

Remember what we said a few lines ago: “Anything outside of the `build` method is not rebuilt and keeps its value”. In this case, the `counter` variable is inside the `build` method so each time a rebuild is triggered, the counter is always re-initialized to zero. Pressing the button actually increases the counter value but a few moments later `build` is re-evaluated and `counter` is set to `0` again.

### 10.2.2 The lifecycle of the `State<T>` object

The `State<T>` class is automatically created by Flutter and attached to a `StatefulWidget` class. There are a series of interesting methods you can override to control its lifecycle:

```

class _ExampleState extends State<Example> {
  @override
  void initState() {
    super.initState(); // Initialize resources here
  }

  @override
  void didUpdateWidget(covariant Example oldWidget) {
    super.didUpdateWidget(oldWidget); // State update on widget changes
  }

  @override
  void dispose() {
    super.dispose(); // Dispose resources here
  }

  @override
  Widget build(BuildContext context) {
    return const Something();
  }
}

```

There are two other methods you can override, but we will talk about them in *chapter 12 – State management*. For now, let's focus on the most frequently used methods to understand when they should be overridden. This list describes, in order, the lifecycle events of a state object:

1. The Flutter framework creates a new `State` object by calling the `createState()` method of the stateful widget:

```
class Example extends StatefulWidget {  
  const Example();  
  
  @override  
  State<Example> createState() => _ExampleState();  
}
```

2. A property called `mounted` is set to true, which indicates that the state object is ready. From this moment onwards, we can safely call the `setState` method without problems.
3. The framework calls `initState()`, which is a sort of “constructor” of the widget state. This method is called only once when the state object is created so this is the perfect place for one-time initializations.
4. The framework calls `didChangeDependencies()`, which is covered in detail in *chapter 12 – State management*.
5. At this stage, the state is fully initialized. For example, you can call `setState()` and the framework is free to rebuild the widget whenever it needs. Whenever the stateful widget is changed, the `didUpdateWidget` method is called with a reference to the old object. You can also override the `reassemble()` method, which is only invoked in debug mode whenever a hot reload occurs.
6. When the widget is not needed anymore, the framework calls `dispose()`. It is the opposite of `initState` so it can be seen as the “destructor” of the widget state. This method is called only once before the stateful widget is removed from the widget tree. This is the perfect place to dispose any resource.
7. After the `dispose` call, the state object is defunct and the `mounted` property is set to `false`. At this point, calling `setState` throws a runtime exception because you cannot rebuild a widget that is not alive anymore.

Let's see an example where we override some state lifecycle methods. This code updates a text widget with a new integer value at the given interval:

```
class Example extends StatefulWidget {
    final Duration duration;
    const Example({
        required this.duration,
    });

    @override
    State<Example> createState() => _ExampleState();
}

class _ExampleState extends State<Example> {
    late Timer timer;

    @override
    void initState() {
        super.initState();
        timer = Timer.periodic(widget.duration, (timer) {
            setState(() {});
        });
    }

    @override
    void didUpdateWidget(covariant Example oldWidget) {
        super.didUpdateWidget(oldWidget);

        if (widget.duration != oldWidget.duration) {
            timer.cancel();
            timer = Timer.periodic(widget.duration, (timer) {
                setState(() {});
            });
        }
    }

    @override
    void dispose() {
        timer.cancel();
        super.dispose();
    }

    @override
    Widget build(BuildContext context) {
        return Text('${timer.tick}');
    }
}
```

Since `initState()` is called only once (right after the state creation), it is the best place to initialize resources. As such, we use it to initialize the timer. In the same way, we use `dispose()` to make sure that the timer is closed when the widget gets destroyed. A few interesting things to note:

- The `widget` getter is defined by the state class and returns a reference to the associated stateful widget. In the example, we have used `widget.duration` to access the `Duration` object of the associated stateful widget.
- The `super.initState` and `super.didUpdateWidget` calls should always go first. This is not a strict requirement but a widely adopted convention.
- The `super.dispose` call should always go last.
- To avoid using nullable values, we've used `late` and initialized the timer variable inside the `initState` method. However, we could have reduced the code length a bit by declaring the variable immediately like this:

```
late var timer = Timer.periodic(widget.duration, (timer) {  
    setState(() {});  
});
```

In this way, we write less code (no `initState` needed) but we still need a `late` variable to be able to use the `widget` getter.

- The `setState` call inside the periodic timer contains an empty callback because we have no values to update, we just want to rebuild the widget. In particular, we want to rebuild (or “refresh”) the `Text` widget to make sure that it always paints the latest timer value.

The `didUpdateWidget` method is used to update the state when the widget configuration changes. For example, when you rebuild the `Example` widget and you pass a different `duration` value, this method is triggered:

```
super.didUpdateWidget(oldWidget);  
  
if (widget.duration != oldWidget.duration) {  
    timer.cancel();  
    timer = Timer.periodic(widget.duration, (timer) {  
        setState(() {});  
    });  
}
```

When `Example` is rebuilt with a different `duration`, we want to stop the timer and re-start it with the new interval. The `oldWidget` parameter holds a copy of the previous widget, which is generally compared with the actual widget to spot differences.

In our case, we want to detect if the duration has changed and update the timer accordingly. In other words, `didUpdateWidget` is used to detect when the widget rebuilds and if one or more of its parameters have changed.

### 10.2.3 Keys

Both stateful and stateless widgets have an optional nullable parameter called `key`. In Flutter, keys are used to uniquely identify a widget in the tree. It's the same idea as assigning a key to a field in a relational database. For example:

```
class Example extends StatelessWidget {
  const Example({
    super.key,
  });

  @override
  Widget build(BuildContext context) {
    return const Text('Example');
  }
}
```

From now on, we will always add the `key` parameter in the widget's constructor because it's widely recognized as good practice from the Flutter team<sup>71</sup> <sup>72</sup>. There are three main types of keys you can create:

- `ValueKey<T>`. Use this key when your widgets can be uniquely identified by a single value. For example:

```
const list = <Widget>[
  Language(key: ValueKey<String>('Dart')),
  Language(key: ValueKey<String>('Python')),
  Language(key: ValueKey<String>('C#')),
];
```

---

<sup>71</sup> [https://dart.dev/tools/linter-rules#use\\_key\\_in\\_widget\\_constructors](https://dart.dev/tools/linter-rules#use_key_in_widget_constructors)

<sup>72</sup> [https://github.com/flutter/packages/blob/master/packages/flutter\\_lints/lib/flutter.yaml](https://github.com/flutter/packages/blob/master/packages/flutter_lints/lib/flutter.yaml)

Here we have a list of `Language` widgets that can uniquely be identified by their name. As such, a `ValueKey<String>` is fine for this example.

- `ObjectKey`: Use this key when your widgets cannot be uniquely identified by a single value, and you need a composite object to represent them. For example, you may need to identify a language by name and version:

```
const list = <Widget>[
  Language(
    key: ObjectKey(const LanguageKey('Dart', '2.11')),
  ),
  Language(
    key: ObjectKey(const LanguageKey('Dart', '2.18')),
  ),
  Language(
    key: ObjectKey(const LanguageKey('C#', '10.0')),
  ),
  Language(
    key: ObjectKey(const LanguageKey('C#', '9.0')),
  ),
];
```

The `ObjectKey` class accepts a generic `Object` type and makes comparisons using the `identical` method from the Dart SDK (which checks whether two references are the same or not). In this example, we have created a custom object and used it as a key:

```
class LanguageKey {
  final String name;
  final String version;
  const LanguageKey(this.name, this.version);
}
```

In *chapter 5 – Section 2 “The Object class”* we cover object comparison in detail.

- `GlobalKey`. Use this key when you want to uniquely identify your widgets and get access to their state and context. In the next section *10.3 Widget, Element and RenderObject trees* we will understand what `BuildContext` is and why it's crucial.

We will see other specific implementations in the next chapters. All keys in Flutter have a super class in common called `Key`; it has a `factory` constructor that defaults to a `ValueKey<String>`. Keys in Flutter are often defined with the `Key` constructor, which is shorter and can be `const`:

```

void main() {
  runApp(
    const MaterialApp(
      home: Center(
        // It's the same as ValueKey<String>('Center-Widget-Key'),
        key: Key('Center-Widget-Key'),
        child: Text('Flutter'),
      ),
    ),
  );
}

```

You can always use a `Key` to identify your widgets as long as the string you use is unique. A good idea may be to use a string describing the key location, such as the `Center-Widget-Key` value of the example. In general:

- `GlobalKeys` are used to work with forms (*chapter 18 – Forms and gestures*) and in some cases where you need to access the context or the state of a specific widget (*section 10.3.3 – “Considerations on BuildContext”*)
- When testing widgets (*chapter 19 – Testing Flutter applications*) you will see that `Key` is used to quickly get a reference to a specific widget in the tree.
- A `PageStorageKey<T>` is used to save and restore values that can outlive the widget. For example, when you scroll on a tab and then swipe to another one, this key saves the scroll position.

#### 10.2.4 Good practices

A Flutter developer is very often asked to decide whether to create a stateless or a stateful widget. Rather than a strict rule, this is a general guideline:

- Subclass  `StatelessWidget` to create a widget that is not going to change after its creation. Stateless widgets can be seen as “standalone” blocks that always maintain the same internal configuration over time.
- Subclass  `StatefulWidget` to create a widget whose internal configuration will change over time. Stateful widgets are used when the widgets returned by the `build` method need to change in response to some updates or when you need access to lifecycle methods.

Sometimes, you may need to create a widget that never changes after its creation and also requires some initialization and finalization cleanup. In this case, a stateful widget is the right choice because you need access to lifecycle methods. For example:

```
class Example extends StatefulWidget {
  final Stream<bool> connectionStatus;
  const Example({
    super.key,
    required this.connectionStatus,
  });

  @override
  State<Example> createState() => _ExampleState();
}

class _ExampleState extends State<Example> {
  late StreamSubscription<bool> subscription;

  @override
  void initState() {
    super.initState();
    subscription = widget.connectionStatus.listen(() { /* code ... */ });
  }

  @override
  void dispose() {
    subscription.cancel();
    super.dispose();
  }

  @override
  Widget build(BuildContext context) { /* code ... */ }
}
```

Pretend that the `connectionStatus` stream emits `true` or `false` whenever the availability of the internet connection changes. The configuration never changes, but the widget needs to subscribe and unsubscribe to the stream. As such, you must create a stateful widget because you have to use the `initState` and `dispose` methods.

### Practical note

Stateless widgets are good when you don't need to use lifecycle methods and the widget internal configuration never changes. In all the other cases, you'll need a stateful widget.

## 10.3 Widget, Element and RenderObject trees

At the beginning of the chapter, we have seen that when you nest widgets one inside the other, the framework internally creates and maintains a tree representation (the widget tree). In reality, there is a bigger machinery under the hood. Consider this simple widget:

```
class Example extends StatelessWidget {  
  const Example({super.key});  
  
  @override  
  Widget build(BuildContext context) {  
    return Container(  
      color: Colors.orange,  
      child: Row(  
        children: [  
          Text('Hello... '),'Earth!'),  
        ],  
      ),  
    );  
  }  
}
```

A `Container` is a generic container widget that provides standard painting, positioning, and sizing properties (it's like the `<div>` tag of the HTML world). When Flutter has to render the `Example` widget, we've already seen that it calls the `build` method and it internally generates a tree data structure called `widget tree`:

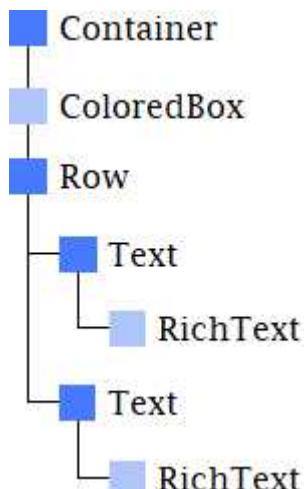


Figure 10.2: Simplified widget tree representation of the `Example` widget. Dark boxes represent classes added by you, while light boxes represent classes added by Flutter.

The `ColoredBox` and `RichText` widgets are internally created by `Container` and `Text` so they're inserted in the tree even if you haven't manually added those. There would be even more widgets to show (created by the `Container`), but we've ignored them to keep the example easy. During the build phase, Flutter also internally builds and maintains another tree called the element tree:

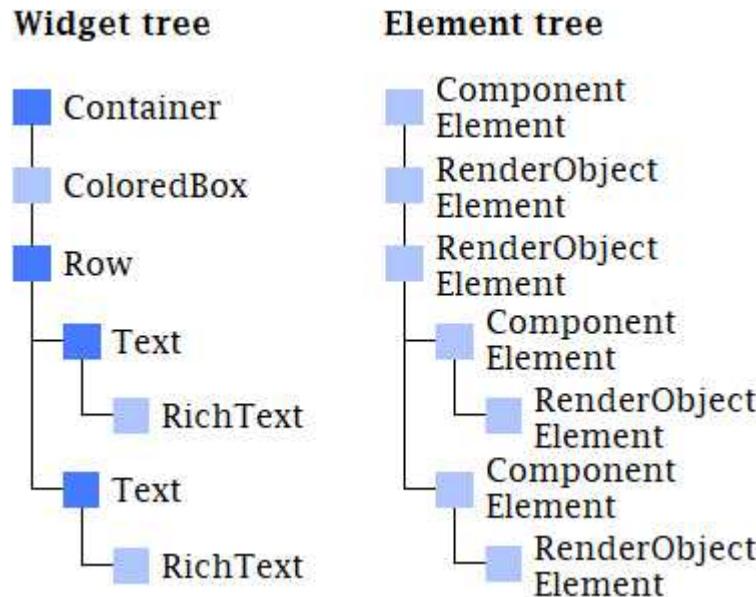


Figure 10.3: The widget tree and the element tree together. Dark boxes represent classes added by you, while light boxes represent classes added by Flutter.

A **widget** is an immutable class that describes how we want a piece of UI to look. An **element** is the lifecycle manager that represents a widget in a specific location of the tree. Any widget always has an associated `Element` object, which can be accessed with the `BuildContext` parameter of the `build` method. There are two kinds of elements:

1. `ComponentElement`: an element that can contain one or more elements inside;
2. `RenderObjectElement`: an element having an associated `RenderObject`, used for laying out, painting, and hit testing.

A `RenderObject` contains all the logic to render the associated element on the screen. During the build phase, Flutter internally builds and maintains a third tree called `render object tree`. For each render object element, a render object is created. In the end, for each widget you make, Flutter internally maintains three tree structure:

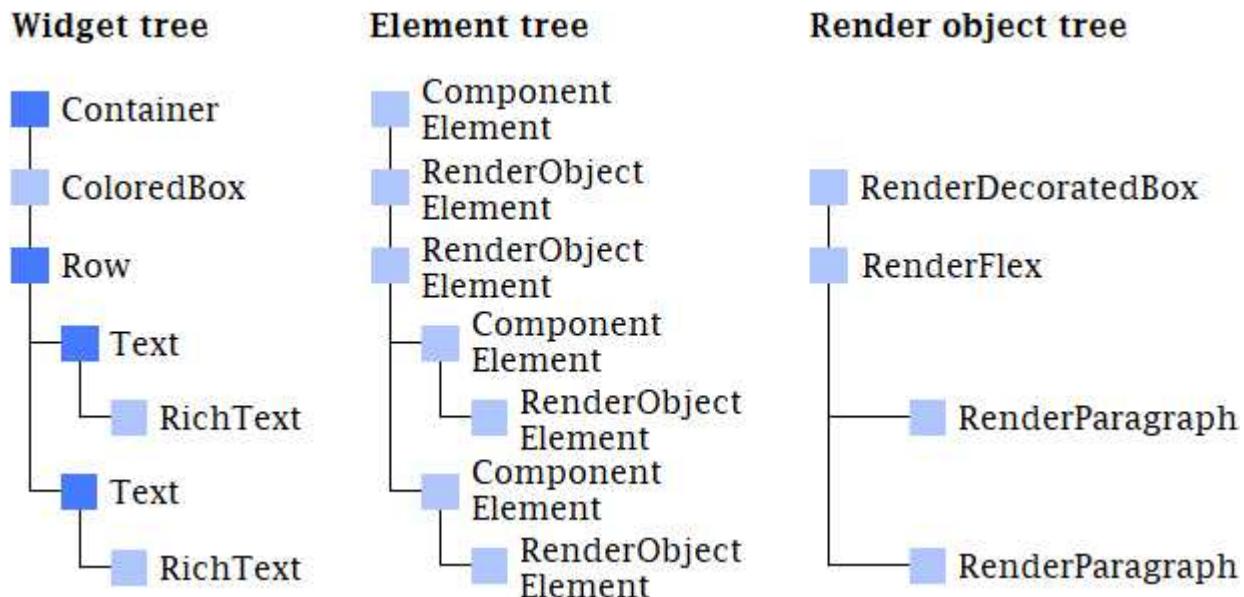


Figure 10.4: Widget, element, and render object trees altogether. Dark boxes represent classes added by you, while light boxes represent classes added by Flutter.

These trees are used by Flutter to efficiently render the UI. A [ComponentElement](#) doesn't directly have an entry in the render object tree because it creates one indirectly with the other elements it contains. To summarize, here are their purposes:

- **Widget tree.** You are the one that builds this tree by nesting stateful or stateless widgets one inside the other. A widget paints nothing to the UI: it's just a description of how you want to see something on the screen.
- **Element tree.** An element is the internal representation of a widget that holds references to the parent, children (if any), and the associated render object. Elements also are the ones that hold the state object of stateful widgets. The purpose of the element tree is to manage the lifecycle of the widget and keep it in sync with the underlying render object.
- **Render object tree.** This tree is used to render the associated elements (and thus the widgets you created) to the UI by taking care of painting, layout constraints, hit testing, and much more.

Widgets are immutable, including their parent/child relationship, so they're always destroyed and re-created in case of changes. Flutter has been designed to be very fast at rebuilding widgets so this is totally fine. Element and render object trees instead persist across rebuilds and are updated only

when really needed. The element tree manages the updating of the UI, while the render object tree actually colors pixels on the screen. If you want to see the differences from a practical point of view, we may say that:

- the widget tree is used to configure how the UI has to look;
- the element tree “installs” a widget in the framework and acts as a lifecycle manager;
- the render object tree is used to color pixels on the screen based on the configuration that you created.

The `main` entry point of a Flutter project always calls the `runApp` method, which is responsible for building all of these trees and initializing the framework machinery. Let’s see what happens to the trees when a widget is rebuilt to better understand how the mechanism works.

### 10.3.1 Rebuilds and trees updates

We’re now going to see under the hood how a rebuild affects the various trees. This is the `build` method of the widget we’re taking as an example:

```
@override  
Widget build(BuildContext context) {  
  return Container(  
    color: Colors.orange,  
    child: Text('Hello earth!'),  
  );  
}
```

The first time the widget is rendered, Flutter calls the `build` method and internally creates all the three trees we already know:

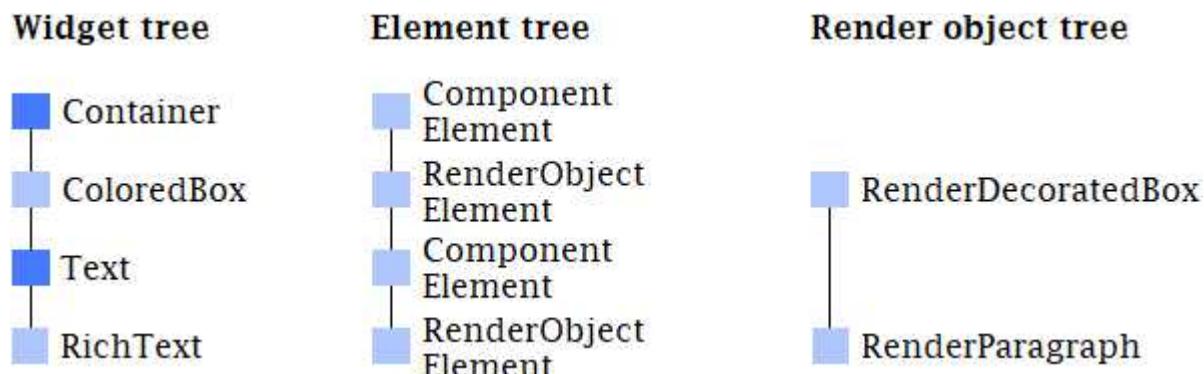


Figure 10.5: Widget, element, and render trees.

Now imagine that a `setState` call changed the string of the `Text` widget. Our widget then may look like this:

```
@override  
Widget build(BuildContext context) {  
  return Container(  
    color: Colors.orange,  
    child: Text('Hello world!'), // Before it was 'Hello earth!'  
  );  
}
```

When `setState` is called, the framework rebuilds the current widget and its subtree to refresh the UI. Elements play a key role because they're used to detect changes (by comparing the old and the new widget) and update/rebuild the associated render object only when needed:

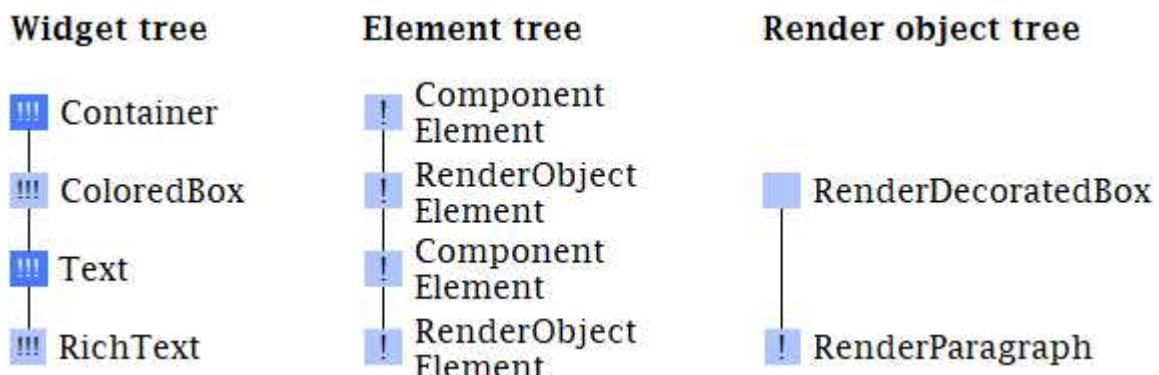


Figure 10.6: “!!!” indicates a rebuild, “!” indicates an update.

Both `Container` and `Text` widgets, along with their children, are rebuilt to update the parent/child relationships and some internal configurations (such as the string in the `Text` widget). We already know that, by design, Flutter is very fast at rebuilding widgets so this is totally fine. The important part of this process is:

1. Thanks to the element tree, Flutter can compare the old and the new `Container` widgets. Since there are no changes, the framework updates the references to the new `Container` object in the element tree. This is an efficient operation because only the widget is rebuilt. The element is only updated, not rebuilt. The associated render object is untouched.
2. Thanks to the element tree, Flutter can compare the old and the new `Text` widgets. Since there are changes, Flutter updates the reference to the new `Text` widget in the element and also updates the associated render object to paint the new text on the screen. Again, notice

that elements and render objects have been updated and not rebuilt. This is the efficient part of the process.

The key point is that render objects are (relatively) expensive to create, so Flutter keeps them alive as much as possible. Rebuilding elements and render objects is more expensive than just updating. The core idea of the framework is to reuse objects as much as possible.

## Note

You may now see why Flutter has been designed to work with three trees. Each part of the UI (configuration, lifecycle, painting) has been split into trees (widget, element, and render object) to cache objects and reuse them as much as possible.

The widget tree is meant to be frequently updated. It's an extremely cheap operation since widgets don't paint anything on the screen. Elements instead decide when it's time to update the associated render object, which is the most expensive entity (because it interacts with the screen for painting and hit testing for example).

The framework considers an element "reusable" if the associated widget keeps the same runtime type and the same key. In particular, the `Widget` class has a static method called `canUpdate` which is used by Flutter to decide if an element and a render object should be updated or rebuilt. It looks like this:

```
static bool canUpdate(Widget oldWidget, Widget newWidget) {  
    return oldWidget.runtimeType == newWidget.runtimeType  
        && oldWidget.key == newWidget.key;  
}
```

When a rebuild happens, the framework calls this method to compare the old and the new widget. If the expression returns `true`, the `Element` can be updated; if the expression returns `false`, the `Element` has to be rebuilt along with its render object. Let's now consider the case where a widget is completely replaced with another one after a rebuild triggered by `setState`:

```
return Container(  
    color: Colors.orange,  
    child: Text('Hello!'),  
);
```

—————>

```
return Center(  
    child: Text('Hello!'),  
);
```

Here the `Container` is replaced with a `Center` widget after the rebuild. This is how the trees look like before the `setState` call:

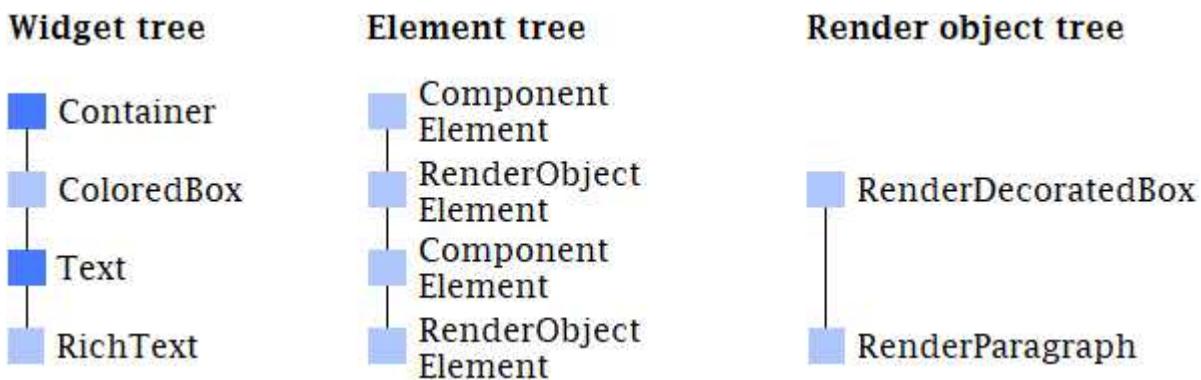


Figure 10.7: How the trees look like before calling `setState`.

After the `setState` method call, the `build` method is re-executed. Thanks to the element tree, Flutter compares the old widget (`Container`) with the new one (`Center`) and notices that they are different. For this reason, not only the widget is rebuilt but also the associated element and render object:



Figure 10.8: How the trees look like after calling `setState`. "!!!!" indicates a rebuild, "!" indicates an update.

When `Center` is created and inserted in the widget tree, the old element is removed and replaced with a new one. Consequently, the render object is also rebuilt. The `Text` widget instead is still the same so the existing element can be updated and the associated render object stays untouched. In this example, the approach is still the same: widgets are often rebuilt, but elements and render objects are reused as much as possible.

### 10.3.2 Performance considerations

The very first thing we want to point out is that there are absolutely NO performance differences between a stateless and a stateful widget. Both kinds of widgets are often rebuilt from the tree and they're treated in the same way. The only difference is that the element of a stateful widget also holds a reference to a `State<T>` object, but it doesn't affect the runtime performance.

## Note

Always using a `StatefulWidget` instead of a  `StatelessWidget` is totally fine, but if you don't need the state, you're just writing more code than needed. This is a coding style concern which has no performance implications.

Flutter is very efficient at traversing trees and reusing existing elements or render objects. However, this is not an excuse to avoid using some good practices to improve the runtime performance even more. For example, consider this code:

```
class Example extends StatefulWidget {
  const Example({super.key});

  @override
  State<Example> createState() => _ExampleState();
}

class _ExampleState extends State<Example> {
  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: () {
        setState(() {});
      },
      child: HelloWorld(),
    );
  }
}

class HelloWorld extends StatelessWidget {
  const HelloWorld({super.key});

  @override
  Widget build(BuildContext context) {
    return Text('Hello world')
  }
}
```

The `setState` call internally marks the element as *dirty* and adds it to a global list of elements that will be rebuilt in the next frame. This is how Flutter precisely knows which parts of the tree need to be rebuilt and which ones don't. In this example, the `build` methods of `Example` and `HelloWorld` are always executed. However, you can save computational time with a small but very effective fix:

```

class _ExampleState extends State<Example> {
  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: () {
        setState(() {});
      },
      child: const HelloWorld(), // Notice the 'const' constructor
    );
  }
}

```

We have added the `const` keyword to the `HelloWorld` widget. Thanks to this fix, the `setState` call will not rebuild the `HelloWorld` widget and its subtree anymore. Using constant constructors on widgets is strongly recommended by the Flutter team<sup>73 74 75</sup> because it skips most of the rebuilding work and therefore saves computational time. We strongly recommend doing this:

- Use `const` constructors as much as possible on your widgets. This is very useful, especially on larger subtrees, to save rebuilds and thus computational time.
- Do not use functions to return widgets because they can never be constant. Always prefer widgets over functions. For example:

```

Widget buildExample() {
  return const Text('Hello world');
}

void main() {
  runApp(const MaterialApp(
    home: buildExample(), // Compiler error here!
  ));
}

```

The object returned by a Dart function cannot be constant by design. As such, the widgets you return from a function can never be `const`. Since we want to be able to create as many constant widgets as possible, don't use functions and prefer stateless or stateful widgets. For example:

<sup>73</sup> <https://docs.flutter.dev/perf/best-practices#control-build-cost>

<sup>74</sup> <https://api.flutter.dev/flutter/widgets/StatefulWidget-class.html> (see the “Performance considerations” part)

<sup>75</sup> <https://api.flutter.dev/flutter/widgets/StatelessWidget-class.html> (see the “Performance considerations” part)

```

class Example extends StatelessWidget {
  const Example({super.key});

  @override
  Widget build(BuildContext context) {
    return const Text('Hello world');
  }
}

void main() {
  // This is great because the entire tree is now a constant
  runApp(const MaterialApp(
    home: Example(), // No error anymore
  ));
}

```

This is better because we can use the `const` keyword and make the entire tree constant. Functions just let you write less code. They return non-constant objects so they aren't really worth using.

- Sometimes you might be in a situation where you would like to make a constant widget but you cannot because it has an external dependency. For example:

```

class Example extends StatelessWidget {
  final String name;
  const Example({
    super.key,
    required this.name,
  });

  @override
  Widget build(BuildContext context) {
    return const Center(
      child: LargeSubtreeWidget('$name'), // Compiler error here
    );
  }
}

```

The `LargeSubtreeWidget` widget could be constant if we passed a compile-time constant string. However, `name` is `final` and it could also change if `Example` is rebuilt with a different value. Even if we can't use the `const` keyword, we can produce the same result with a bit more code:

```

class Example extends StatefulWidget {
  final String name;
  const Example({
    super.key,
    required this.name,
  });

  @override
  State<Example> createState() => _ExampleState();
}

class _ExampleState extends State<Example> {
  late Widget lsw = LargeSubtreeWidget(widget.name);

  @override
  void didUpdateWidget(covariant Example oldWidget) {
    super.didUpdateWidget(oldWidget);

    if (widget.name != oldWidget.name) {
      lsw = LargeSubtreeWidget(widget.name)
    }
  }

  @override
  Widget build(BuildContext context) {
    return Center(
      child: lsw, // uses the “cached” widget instance
    );
  }
}

```

Declaring a widget inside the state class and using its variable in the `build` method is the same as using a `const` constructor. This process is also known as “manual” caching. It’s very important to override `didUpdateWidget` because we need to update the “cached” widget whenever `Example` is rebuilt with a new `name` value. With this approach, only the `Center` widget is rebuilt but `LargeSubtreeWidget` stays untouched.

When Flutter starts rebuilding a widget and its subtree, it stops when it finds the same instance of the child widget as the previous frame. This happens in the case of constant widgets or manually cached ones for example.

Whenever Flutter needs to rebuild a widget and its subtree, it calls the `updateChild` method of the `Element` class, which is at the core of the widgets system. This is the main structure of `updateChild` (we have removed assertions and checks to simplify the code and focus on the key points):

```

// 'child' is the old widget that needs to be removed.
// 'newWidget' is the new widget to be inserted in the tree.
// 'newSlot' is to handle parent/child relationships
Element? updateChild(Element? child, Widget? newWidget, Object? newSlot) {
  if (newWidget == null) { // 1.
    return null;
  }

  final Element newChild;
  if (child != null) {
    if (child.widget == newWidget) { // 2
      newChild = child;
    } else if (Widget.canUpdate(child.widget, newWidget)) { // 3.
      child.update(newWidget);
      newChild = child;
    } else {
      deactivateChild(child);
      newChild = inflateWidget(newWidget, newSlot); // 4.
    }
  }
  return newChild;
}

```

This method is extremely important because it rules the entire lifetime of the three trees. It decides if an object stays untouched, if it has to be updated, or if it has to be rebuilt. In particular, the four `if` statements determine the following behavior:

1. If the new widget that has to replace the old one is `null` (`newWidget == null`), then it was removed from the tree. As such, we return a `null` element to remove the widget from the tree in the next frame. `newChild` is the new widget that could need a rebuild, but it's not yet sure.
2. If you have used a `const` constructor or you have manually cached the widget in the state class, then `child.widget == newWidget` is always `true`. Consequently, no rebuilds happen to the widget itself and all of its subtree. `newChild` is assigned to the already existing widget (`child`) without doing anything. This is the fastest operation!
3. At this point of the function, one or more parts of the tree need an update. However, if the `Widget.canUpdate` method returns `true`, it means that the new widget is of the same type and has the same key. As such, `child.update(newWidget)` updates the widget but leaves the element and the render object untouched.

- This is the last branch. If the framework arrives here, it means that there were no `const` constructors, no manual caching, and the new widget has a different type than the old one. The `deactivateChild` method removes the given element and its render object from the trees. Then, `inflateWidget` creates a new element and a new render object.

This low-level analysis of how the framework rebuilds the trees is useful to understand why `const` constructors on widgets and manual caching are helpful. These two techniques can save entire subtrees from being rebuilt.

### 10.3.3 Considerations on BuildContext

Flutter has been designed to work with widgets. Elements and render objects are meant to be used internally by Flutter, and you shouldn't directly work with them. For this reason, the `Element` of a widget is “hidden” behind the `BuildContext` type. This is how an element is defined:

```
abstract class Element extends DiagnosticableTree implements BuildContext
```

The `BuildContext` type was created to discourage the “direct” manipulation of `Element` objects. Regardless, you can still retrieve a reference to the element behind your widget with a typecast. For example:

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

  @override
  Widget build(BuildContext context) {
    final element = context as Element;

    // Text contains the 'Example' string
    return Text('${element.widget.runtimeType}');
  }
}
```

In practice, there's no reason for you to do this because calling methods on an element is very likely going to mess up the tree and lead to undefined behaviors. The `BuildContext` type itself is very useful for other kinds of purposes, such as:

- Passing the state down in the widget tree, as we will see in *chapter 12 – State management*.
- If a widget is mounted in the tree, you can always access its internals from anywhere using keys. For example, imagine having this simple tree and a `GlobalKey`:

```

final key = GlobalKey();

void main() {
  runApp(
    MaterialApp(
      home: Text(
        'Flutter!',
        key: key,
      ),
    ),
  );
}

```

When a  `GlobalKey` is associated to a widget, you can get a reference to its build context or even its state (in the case of a  `StatefulWidget`). Anywhere in your code, you can use the key as follows:

```

final BuildContext? context = key.currentContext;
final State<Some StatefulWidget>? state = key.currentState;

```

If the widget is removed from the tree, the getters returns `null` hence why the nullability of the return types.

- The `BuildContext` is the “window” you have to access the underlying `Element` and it can change if the widget is moved. For example, you may implement a drag-and-drop for your application that moves a widget from one portion of the subtree to another. This “position change” in the tree is also reflected under the hood by a new `BuildContext` that references the new position of the widget in the tree.

It's essential to keep in mind that the build context may change during the widget's lifetime. By consequence, when you're using a  `StatefulWidget`, never cache the `BuildContext` reference in the state because it might point to a deactivated widget in some cases.

## Deep dive: The framework internals

Motion on a screen is an illusion created by quickly changing static images at a constant speed. Each “static image” is called **frame**. While you are watching a video on a screen for example, you are actually seeing a lot of static images (frames) that, in sequence, change very quickly. The human eyes cannot clearly distinguish frames when they change too quickly, so you have the “illusion” of seeing objects in motion. This mechanism is at the core of the Flutter architecture:

- Any motion in a Flutter application, such as a rotating widget or a fading transition between two pages, is created with a quick sequence of frames. In practice, Flutter creates 60 (or more) frames per second and quickly paints them in sequence to render a smooth animation on the screen.
- The number of frames that are displayed each second has a direct impact on how smooth the animations are. Mobile devices generally work at 60 fps, meaning that they can generate a maximum of 60 frames each second. Powerful desktop hardware may have a higher frame rate (such as 120) and thus Flutter animations could be even smoother.
- If the screen was able to display 60 fps (frames-per-second), Flutter would only have 16.67 milliseconds to generate each static image (frame). In other words, the engine would have up to 16.67 milliseconds to create 60 frames within a second.

In *chapter 8 –Section 1.1 “Event loop and queues”* we saw that a Dart application has an event loop and two queues. For smooth rendering, the Flutter framework pushes “repaint” events in the event queue 60 times per second (or even more if the hardware is powerful enough). Look at this image:

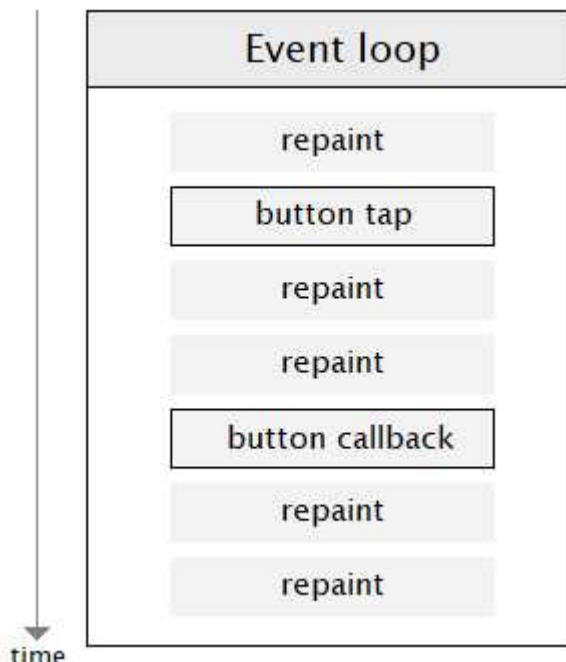


Figure 10.9: The event loop of a Flutter application.

The “repaint” events you see in *Figure 10.9* are automatically added by Flutter to create frames. For example, to run the application at 60 fps, the framework adds 60 repaint events each second to the event queue. In addition, the event queue also holds all those user-generated events such as button taps or scrolling requests. If the event loop was blocked on a specific operation for too long, you would get a frame drop and the UI would become unresponsive. Consider this case:

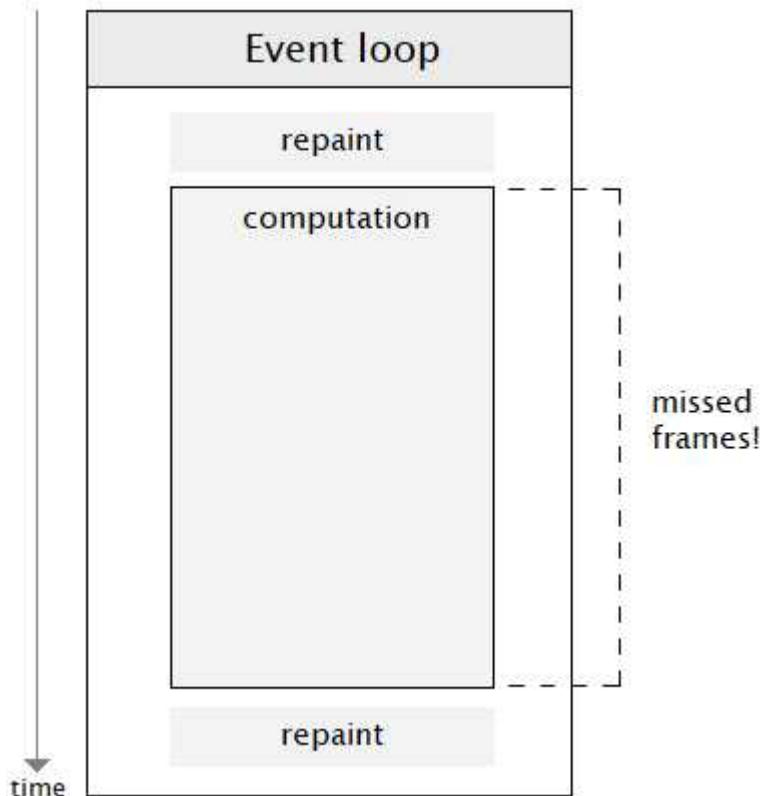


Figure 10.10: A Flutter application that drops frames and makes the UI not responsive.

In the example, a synchronous event (“computation”) is blocking the event loop for some time. The problem is that “repaint” events cannot be processed until the computation completes. The fact that some “repaint” events were missed makes the UI unresponsive, and the frame rate drops.

### Note

Dropping frames is bad because your application may not be responsive for some time. Animations, for example, might not run smoothly and the user could notice a temporary UI “freeze”.

To solve this issue, you could try to move the time-consuming computation to a separate isolate. In this way, the event loop of the main isolate will have enough time to render all frames and keep the UI responsive. This is what happens under the hood with a worker isolate, for example:

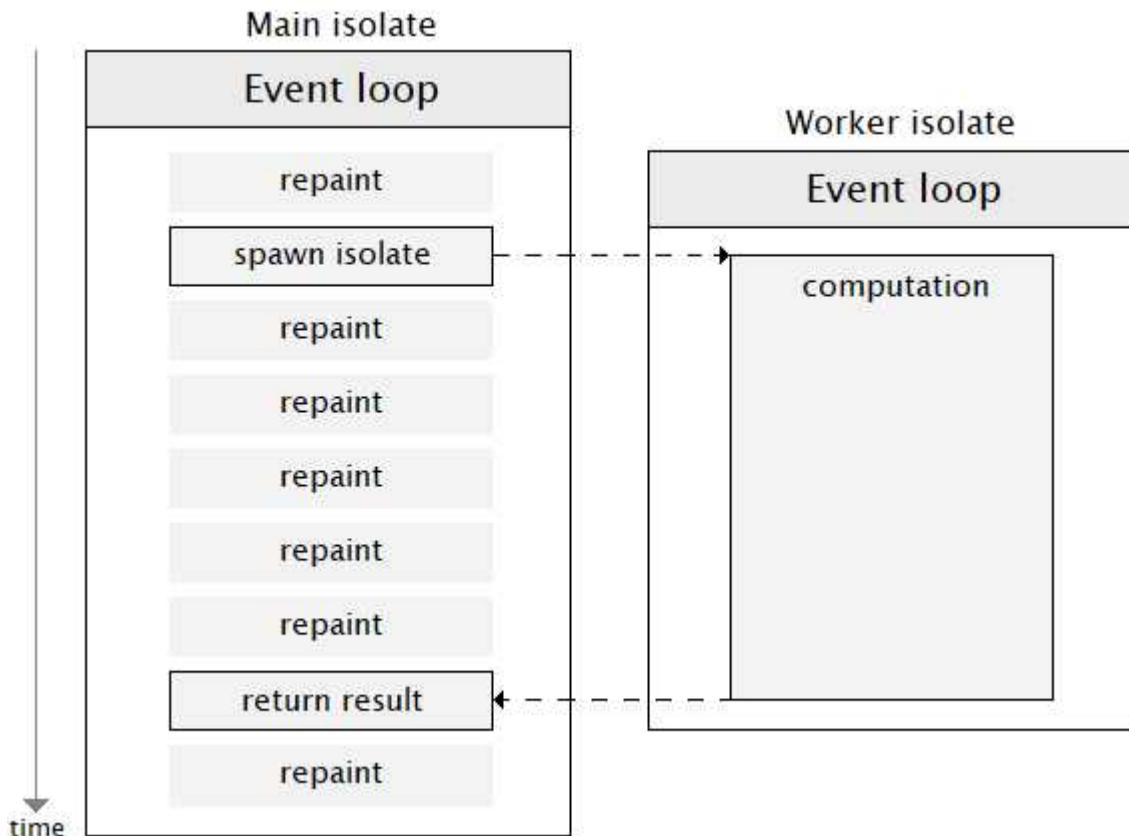


Figure 10.11: A time-consuming computation is executed on a worker isolate to avoid frame drops.

As you can see from *Figure 10.11*, the event loop of the main isolate has enough time to execute all “repaint” events and keep the UI responsive. The time-consuming task concurrently executes on a worker isolate, which does not cause frame drops. To see how the framework interacts with the engine and how it draws frames, let’s start from the very beginning:

```
void main() {  
    runApp(const MyRootWidget());  
}
```

To execute any Flutter application, you always need to call `runApp` and pass it the root widget. This function takes care of building the widget, element, and render objects trees before scheduling the very first frame to draw the UI on the screen. Here’s what the method does:

```

void runApp(Widget app) {
  final WidgetsBinding binding = WidgetsFlutterBinding.ensureInitialized();

  binding
    ..scheduleAttachRootWidget(binding.wrapWithDefaultView(app))
    ..scheduleWarmUpFrame();
}

}

```

The `WidgetsFlutterBinding` singleton class is the glue between the engine layer and the Flutter framework. Once the `binding` object is created, the other two method calls are used to configure the widget tree (`scheduleAttachRootWidget`) and render the very first frame as soon as possible (`scheduleWarmUpFrame`). This is the class signature:

```

class WidgetsFlutterBinding extends BindingBase with
  GestureBinding,
  SchedulerBinding,
  ServicesBinding,
  PaintingBinding,
  SemanticsBinding,
  RendererBinding,
  WidgetsBinding

```

To avoid making the class too big, the Flutter team decided to split each responsibility into a mixin. In particular, each mixin represents an engine feature:

- **`GestureBinding`**: This binding manages the gesture system, which (for example) is used to detect drag & drops, taps, double taps, and much more. Gestures are covered in *chapter 18 – Section 3 “Gestures”*.
- **`SchedulerBinding`**: This binding is used for frame management operations. For example, it synchronizes the application’s behavior with the system’s display and manages operations that execute between frames.
- **`ServicesBinding`**: This binding is used to listen for platform-specific messages from the OS and forward them to the framework. Platform-specific interactions are covered in *chapter 23 – “Platform interactions”*.
- **`PaintingBinding`**: This binding wraps the engine’s painting API for particular purposes, such as scaling images, interpolating shadows, or painting borders around boxes. It is a sort of Dart “view” of the underlying C++ API.

- [SemanticsBinding](#): This binding manages semantics interactions and accessibility features between the framework and the engine. Accessibility is covered in *chapter 15 – Section 2 “Accessibility”*.
- [WidgetsBinding](#): This binding manages the interactions between the widget tree and the engine. This mixin also creates and manages a [BuildOwner](#) object, which is used to control the entire rebuild system of the framework.
- [RenderBinding](#): This binding manages the interactions between the render object tree and the engine. This mixin also creates and manages a [PipelineOwner](#) object, which is used to manage the rendering pipeline and control render objects.

Before moving on to the actual frame creation, let's focus a bit more on the [BuildOwner](#) class. It is extremely important because it manages the widget and the element trees while the application is running. Here's an overview of how rebuilds work in Flutter:

1. When a widget needs to be rebuilt, the [dirty](#) property of the associated element is set to [true](#). When you call the [setState](#) function for example, it executes its callback and then it sets [\\_dirty = true](#) for the associated element. In *chapter 12 – “State management”* we will see how to control rebuilds efficiently.
2. After an element is marked as “dirty”, it is also added to the [BuildOwner](#)'s dirty elements list. A [BuildOwner](#) object is unique, managed by the [WidgetsBinding](#) singleton, and keeps track of which widgets need rebuilding.
3. The [BuildOwner](#) class defines an important method called [buildScope](#). During the frame creation process, [buildScope](#) is called to go through the “dirty elements” list to rebuild all dirty elements. Once the element is rebuilt, its [dirty](#) property goes back to [false](#) and it's removed from the “dirty elements” list.

Thanks to the [BuildOwner](#) class, the framework knows which parts of the tree need a rebuild and which ones don't. The [PipelineOwner](#) class has a similar purpose: any render object that needs an update is marked as “dirty” so that it will be updated in the next frame. In practice, [BuildOwner](#) manages element updates, while [PipelineOwner](#) manages render object updates.

Now that you know of how the rebuild system works, you can better understand how frames are generated. The Flutter engine is the one that decides when it's time to create a new frame. This is an ordered sequence of what the framework does to render a frame:

1. The “animation” phase. Any active animation advances by a tick and callbacks coming from tickers or `AnimationController` instances that completed (if any) are executed. After this, all microtasks that were scheduled by frame callbacks are executed.
2. The “build” phase. The `BuildOwner` singleton rebuilds all dirty elements so that the widget tree is updated with the latest configuration. We have described this process in detail on the previous page.
3. The “layout” phase. The `PipelineOwner` singleton computes sizes and constraints of dirty render objects but does not (yet) paint them to the screen. This step basically makes sure that dirty render objects will have the correct (and updated) geometry for the next steps.
4. Any render object that needs an update to its composite layers is processed. Afterward, dirty render objects are painted into a low-level `Canvas` class. All the drawings instructions made in this phase are used to build a layer tree.
5. The layer tree is turned into a `Scene` object and sent to the GPU to draw the pixels on the screen. At this point, render objects with dirty semantics are updated. Semantic widgets are covered in *chapter 15 – section 2 “Accessibility”*.
6. The `State.dispose` method is called on any object that was removed from the widget tree in this frame. To do this, the framework uses the `finalizeTree` method of the `BuildOwner` class. When the finalization step is done, callbacks registered with `addPostFrameCallback` are executed.

All of these steps are commonly executed 60 times per second on mobile devices (powerful desktop hardware can even produce 120 frames per second). The `runApp` function takes care of scheduling the first frame generation, and then the engine begins to request new frames automatically. If you want to execute some operations after the `build` method is run, use `addPostFrameCallback`:

```
@override
void initState() {
  super.initState();
  WidgetsBinding.instance.addPostFrameCallback((_) => debugPrint('Message'));
}

@Override
Widget build(BuildContext context) => const Text('Hello!');
```

The binding mixin internally holds a list of all post-frame callbacks that will be executed in order on the next frame. In this case, the message will be printed to the console after the execution of the `build` method. To see an example of when `addPostFrameCallback` might be useful, see “*Deep dive: Understanding RenderObjects*”.

# 11 – Material, Cupertino and CustomPaint

---

## 11.1 Basic widgets

As we've already seen, building UIs is a matter of nesting widgets ones inside the others. The Flutter framework comes with a massive set of pre-built widgets<sup>76</sup> you can use to create beautiful user interfaces. We could split them into three main categories:

1. **Basic widgets.** They're used to place other widgets along the horizontal/vertical axis, inside grids, or give them particular positions or space constraints. Some popular widgets for this category are [Container](#), [Text](#), [Row](#), [Column](#) or [Image](#).
2. **Material widgets.** Material<sup>77</sup> is a design system created by Google for Android, iOS, Flutter, and web applications. The framework has built-in support for widgets that have been styled following Material design guidelines.
3. **Cupertino widgets.** Cupertino is the name of a group of Flutter widgets that implement the current iOS design language. These widgets look exactly like their native iOS counterparts.

Basic widgets could be used to create any UI, but they'd require a lot of work and customization. In general, Flutter developers customize the existing Material or Cupertino widgets rather than creating new ones from scratch, which is faster and more maintainable.

### Note

When you want to create a very particular UI, using Material or Cupertino widgets may be very complicated. When a UI element is very particular, built-in Flutter widgets might not be customizable enough. In this case, you should use a [CustomPaint](#).

[CustomPaint](#) is a “low-level” widget that provides a canvas on which you can draw using pixel coordinates. In section 11.4 “Custom painters” we will see how a [CustomPainter](#) makes it possible to freely draw anything on a canvas.

---

<sup>76</sup> <https://docs.flutter.dev/development/ui/widgets>

<sup>77</sup> <https://material.io/design>

As you already know, the `runApp` method is always used in the Dart's `main()` entry point to initialize Flutter and start rendering frames. To correctly initialize the framework, you also have to make sure to place an “app widget” at the root. For example, this code is going to throw a runtime exception:

```
void main() {  
  runApp(const Text('Hello world!'));  
}
```

The reason is that Flutter requires a series of standard widgets to be located high in the widget tree to make the application work. Rather than inserting all of them by hand (which is quite complicated and error-prone), you should use one of these three wrappers:

- [WidgetsApp](#): A widget that wraps a series of other widgets that are commonly required for an application to work.
- [MaterialApp](#): A widget that wraps a series of other widgets that are commonly required for material design applications. It's built on top of a [WidgetsApp](#) and adds Material-design specific functionalities.
- [CupertinoApp](#). A widget that wraps a series of other widgets that are commonly required for iOS design applications. It's built on top of a [WidgetsApp](#) by adding iOS-design-specific functionalities.

You must make sure that one of these three widgets is always passed to the `runApp` method. In practice, [WidgetsApp](#) isn't generally used because it requires lots of manual configurations, and it's already internally used by both [MaterialApp](#) and [CupertinoApp](#). As such, you'll always end up using one of these:

```
void main() {  
  runApp(MaterialApp(  
    home: const Text('Flutter!'),  
  ));  
}  
  
void main() {  
  runApp(CupertinoApp(  
    home: const Text('Hello world!'),  
  ));  
}
```

None of them are “good” or “bad”. Choose which one you want to use according to the design you need to implement. For example, you may wish to use [CupertinoApp](#) if your application has an iOS look and feel.

Flutter comes with a very large number of widgets, most of them from the Material and Cupertino libraries. We won't go through all of them because they would require a dedicated book! In this chapter, we will showcase the most popular ones but you'll discover more in the following chapters.

Let's start by looking at some widgets you need to know about. They are at the core of any Flutter UI, and you will always see them in any application.

### 11.1.1 Text

`Text` is probably one of the most famous widgets of the entire framework. As you may have guessed from the name, it displays a string of text with a certain style. If the enclosing layout is constrained, the next wraps to a new line. For example:

```
const Text(  
  'Flutter!',  
  textAlign: TextAlign.center,  
  style: TextStyle(  
    color: Colors.blue,  
    fontSize: 16,  
    letterSpacing: 3,  
,  
,
```

The `Colors` class is defined in the Material library and it has dozens of pre-defined color names. Nevertheless, you can still define your own colors using the `Color` class (without the trailing `s`):

- The default constructor accepts a hexadecimal value, such as `Color(0xFFEE99CC)`, where `FF` is the transparency, `EE` is for the red, `99` for the green, and `00` for the blue.
- The other named constructor creates a color object from the lower 8 bits of four integers. For example, `Color.fromARGB(255, 238, 153, 204)` is the same color as in the previous bullet point but without the hexadecimal representation.

The `Text` widget expands to fill the entire horizontal space and then goes to a new line. If you want the text to stay on a single line, you can use the `maxLines` parameter and set it to `1`:

```
Text('Hello world!',  
  maxLines: 1,  
  overflow: TextOverflow.ellipsis,  
,
```

The `ellipsis` enumeration adds an ellipsis at the end of the text when it overflows. The nature of a `Text` widget is to render the entire string with the same style, but this is not always the desired behavior. For example, you may want to paint some text and highlight specific words in bold or with a different color. To do so, you have to use the `Text.rich` constructor:

```

const Text.rich(
  TextSpan(
    text: 'We love ',
    style: TextStyle(
      fontSize: 18,
    ),
    children: [
      TextSpan(
        text: 'Dart',
        style: TextStyle(
          fontWeight: FontWeight.bold,
          color: Colors.blue,
        ),
      ),
      TextSpan(
        text: ' and ',
      ),
      TextSpan(
        text: 'Flutter',
        style: TextStyle(
          fontStyle: FontStyle.italic,
          color: Colors.lightBlue,
        ),
      ),
    ],
  ),
  overflow: TextOverflow.ellipsis,
  maxLines: 1,
  textAlign: TextAlign.center,
),

```

The `Text.rich` constructor is used to display together strings that use different styles. The text is created by nesting `TextSpan` objects, where each one can have its own unique style configuration. The above code produces this output:



Figure 11.1: The output of a string styled with `Text.rich`.

The code is indeed a bit verbose but it allows you to print a string with different stylings. You should use a `TextSpan` whenever the text has to be rendered with non-default styling. Under the hood, the `Text.rich` constructor uses `RichText`, which is a lower-level widget to draw text.

### 11.1.2 Container

The other extremely popular widget is called `Container`, which is used for painting, positioning and sizing purposes. You can see this as the Flutter counterpart of the `<div>` tag in the HTML world. It's extremely customizable. For example:

```
Container(  
  width: 150,  
  height: 60,  
  margin: const EdgeInsets.all(10),  
  decoration: BoxDecoration(  
    color: Colors.lightGreenAccent,  
    borderRadius: const BorderRadius.all(  
      Radius.circular(15),  
    ),  
    border: Border.all(  
      color: Colors.black,  
      width: 2,  
    ),  
  ),  
  child: const Center(  
    child: Text('Hello world!'),  
  ),  
)
```



Figure 11.2: The container produced by the code on the left.

If we didn't specify the width and the height, the container would have taken all of the available space within its parent because (when not constrained) it always is as big as possible. Some cases where the container is shrunk to fit its contents are:

- When it's inside another widget that imposes constraints, such as another `Container` or a `SizedBox`:

```
SizedBox(  
  width: 100,  
  height: 40,  
  child: Container(  
    margin: const EdgeInsets.all(10),  
    child: const Text('Hello world!'),  
  ),  
)
```

- When it's inside `Center`, which expands to be as big as possible while constraining its child to its smallest dimensions:

```
Center(  
  child: Container(  
    margin: const EdgeInsets.all(10),  
    child: const Text('Hello world!'),  
  ),  
,
```

Any child of `Center` (including `Container`) is centered and takes the least possible amount of space.

The `BoxDecoration` class is an immutable description of a `Container` has to be painted. It can also be used to clip contents but be aware that clipping may be costly in terms of performance.

### 11.1.3 Row, Column and Align

The `Row` and `Column` widgets are the first choice when it comes to laying down two or more widgets, respectively, on the horizontal or vertical axis. They basically are the same widget, with the only difference being the axis on which they lay down the children. For example:

```
Row(  
  mainAxisAlignment: MainAxisAlignment.center,  
  children: [  
    Container(  
      margin: const EdgeInsets.all(5),  
      width: 40,  
      height: 30,  
      color: Colors.lightGreen,  
,  
    Container(  
      margin: const EdgeInsets.all(5),  
      width: 40,  
      height: 30,  
      color: Colors.red,  
,  
  ],  
,
```

A `Row` expands to fill all of the available width, but it doesn't scroll if there isn't enough horizontal space for the children. Thanks to the `mainAxisAlignment` parameter, you can decide how children can be positioned within the horizontal axis. In our example, the values of the `MainAxisAlignment` enum would paint the widgets with the following positions:

Enumeration type	Effect	Example
<code>MainAxisAlignment.center</code>	Widgets are aligned as close to the center as possible.	
<code>MainAxisAlignment.start</code>	Widgets are aligned as close to the start as possible.	
<code>MainAxisAlignment.end</code>	Widgets are aligned as close to the end as possible.	
<code>MainAxisAlignment.spaceBetween</code>	Free space is added evenly between the widgets.	
<code>MainAxisAlignment.spaceAround</code>	Free space is evenly added between children and half of that in the borders.	
<code>MainAxisAlignment.spaceEvenly</code>	Free space is equally added between the children.	

A `Column` basically is a `Row` that places its children on the opposite axis (the vertical one), but aside from that, there aren't differences. Both widgets share the same parameters, and they don't scroll in case of overflow. For example:

```
Column(
  mainAxisAlignment: MainAxisAlignment.spaceEvenly,
  children: [
    Container(
      margin: const EdgeInsets.all(5),
      width: 14,
      height: 14,
      color: Colors.lightGreen,
    ),
    Container(
      margin: const EdgeInsets.all(5),
      width: 14,
      height: 14,
      color: Colors.lightGreen,
    ),
  ],
),
```

The `mainAxisAlignment` parameter works in the exact same way we've just seen but in the other direction:

Enumeration type	Effect	Example
<code>MainAxisAlignment.center</code>	Widgets are aligned as close to the center as possible.	
<code>MainAxisAlignment.start</code>	Widgets are aligned as close to the start as possible.	
<code>MainAxisAlignment.end</code>	Widgets are aligned as close to the end as possible.	
<code>MainAxisAlignment.spaceBetween</code>	Free space is added evenly between the widgets.	
<code>MainAxisAlignment.spaceAround</code>	Free space evenly added between children and half of that in the borders.	
<code>MainAxisAlignment.spaceEvenly</code>	Free space is equally added between the children.	

If you want a child to expand to fill all of the available space, use an `Expanded` widget. Both `Row` and `Column` should be used to contain two or more children. If you have a single child, consider using the `Align` widget. For example:

```
Container(  
    width: 80,  
    height: 80,  
    decoration: BoxDecoration(  
        border: Border.all(  
            color: Colors.grey,  
        ),  
    ),  
    child: const Align(  
        alignment: Alignment.bottomRight,  
        child: SomeWidget(),  
    ),  
,
```

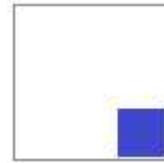


Figure 11.3: How the Align widget places widgets

The `Alignment` enum lets you determine where the child should be placed. In our case, the `Align` widget expands to be as big as possible so it requires a parent whose sizes are well defined (in fact, we've specified both width and height for the enclosing container).

### Note

By default, the `Align` widget aligns its child to the center of the screen. In such case, you can either use...

```
const Align(  
    child: SomeWidget(),  
,
```

... or the most popular `Center` widget:

```
const Center(  
    child: SomeWidget(),  
,
```

We recommend to use `Center` which is more popular. Under the hood, it just returns `Align` so the two above example are equivalent.

When you put too many widgets in a `Row` or in a `Column`, it often happens that you see an overflow error at runtime. This is caused by the fact that there isn't enough space to lay out all the children. In these cases, the most common solutions are:

1. Use a scrollable widget, as we will describe in *chapter 14 – Section 3 “Scrollable widgets”*.

2. Use the `Expanded` widget, which tells a child of a `Row` or `Column` to fill the remaining space. For example, imagine being in this situation:

```
SizedBox(  
    width: 80,  
    height: 60,  
    child: Row(  
        children: [  
            Container(  
                width: 10,  
                height: 20,  
                color: Colors.LightBlue,  
            ),  
            const Text('Very long text here!'),  
            Container(  
                width: 20,  
                height: 20,  
                color: Colors.LightGreen,  
            ),  
        ],  
    ),  
,),
```

Both containers and the text don't fit in the `Row`. The problem is that the `Text` widget takes too much horizontal space because its string is very long. This results in an overflow error and some warning messages on the console. The fix is simple:

```
const Expanded(  
    child: Text('Very long text!'),  
,)
```

In this way, `Expanded` tells the `Row` that `Text` should be given the remaining available room and nothing more. In other words, the `Text` widget does not expand as much as it wants anymore. It just fills all the available space and the text that doesn't fit goes into a new line.

Note that `Expanded` forces the child to fill all the available space. A `Flexible` widget instead still gives the child the possibility to expand to fill all the available space but, unlike `Expanded`, does not require the child to also fill the available space.

### Note

The `Flexible` and `Expanded` widgets can only be used inside `Row` or a `Column`.

In general, when you have two or more children, we suggest to use a [Row](#) or a [Column](#). When you have a single child that needs to be positioned in a certain way, use [Align](#). Keep in mind that rows and columns are the only places where [Expanded](#) can be used.

#### 11.1.4 Stack

The [Stack](#) widget is useful when you need to overlap several children in a simple way. For example, you may want to place some text in front of a container:

```
SizedBox(  
  width: 100,  
  height: 100,  
  child: Stack(  
    children: [  
      Container(  
        width: 50,  
        height: 50,  
        color: Colors.grey,  
      ),  
      const Positioned(  
        top: 15,  
        left: 26,  
        child: Text(  
          'Hello!',  
          style: TextStyle(fontSize: 16),  
        ),  
      ),  
    ],  
  ),  
,
```

The stack paints its children in order. In our example, the [Container](#) is in the background because it comes (in order) before the [Text](#), which is then painted in the foreground. In other words, the last widget of the list is in the foreground. Here's what Flutter renders:

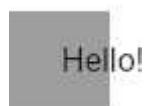


Figure 11.4: What Flutter renders when the container comes before the text in the [Stack](#) widget.

Each child of a Stack is either [positioned](#) or [non-positioned](#). Positioned children are wrapped in the [Positioned](#) widget, which places the child at the given position (relative to the top-left edge). Non-

positioned children are not wrapped by `Positioned` and thus they're placed according with the `alignment` parameter of the `Stack` (which defaults to the top-left corner).

## 11.2 Material library

As we've already seen, Flutter has an extensive set of pre-built widgets that implement the Material design guidelines. Even if you're mainly used to see Material components on Android devices, this design system is meant to work across all platforms (mobile, web, and desktop). This is what a basic material application looks like:

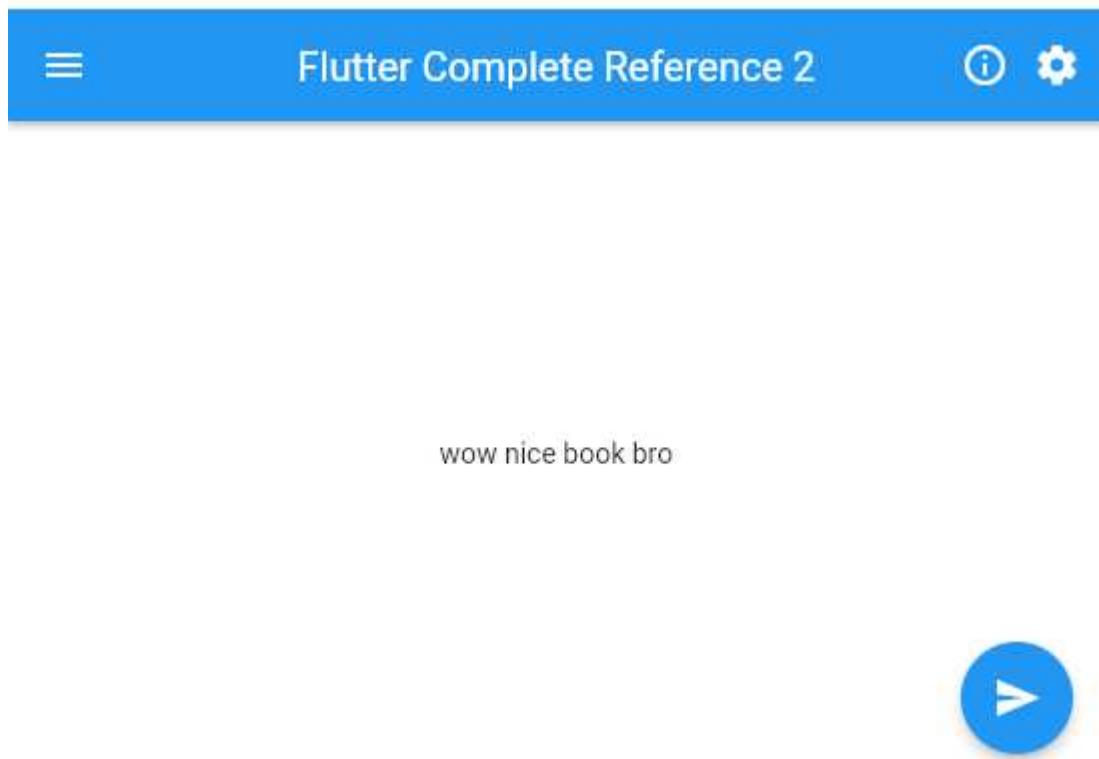


Figure 11.5: A basic material app with an app bar, a navigation drawer and some buttons.

Of course, you could create this kind of UI by yourself without using Flutter's material library, but it would be very time-consuming. Rather than reinventing the wheel, just make sure to import the material library and you're ready to go:

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

The material library is already bundled in the SDK so you don't need to add external dependencies in the `pubspec.yaml` file.

### 11.2.1 Scaffold

To create the design we've shown in *Figure 11.5*, you need to use the `Scaffold` widget (right after the `MaterialApp` one), which implements the basic Material layout structure. For example:

```
class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: Colors.white,
      appBar: AppBar(
        title: const Text('Flutter Complete Reference 2'),
        centerTitle: true,
        actions: const [
          Padding(
            padding: EdgeInsets.symmetric(
              horizontal: 6,
            ),
            child: Icon(Icons.info_outline),
          ),
          Padding(
            padding: EdgeInsets.only(
              left: 6,
              right: 12,
            ),
            child: Icon(Icons.settings),
          ),
        ],
      ),
      body: const Center(
        child: Text('wow nice book bro'),
      ),
      drawer: const Drawer(),
      floatingActionButton: FloatingActionButton(
        child: const Icon(Icons.send),
        onPressed: () {},
      ),
    );
  }
}
```

A **Scaffold** is a sort of general-purpose container that hosts your Material design application. It's used to wrap other Material components, such as:

- **AppBar**: This widget is always placed at the top of the screen. It consists of a toolbar plus some other optional widgets. For example:

```
AppBar(  
    title: const Text('Flutter Complete Reference 2'),  
    actions: [  
        IconButton(  
            onPressed: () {},  
            icon: const Icon(Icons.info_outline),  
        ),  
    ],  
)
```

Actions generally are tappable icons on the right side of the application bar. As the name suggests, an **IconButton** is a tappable icon with a Material design style. The **Icon** class has dozens of pre-built icon values defined inside the **Icons** class.

- **Drawer**: A drawer is a material design panel that horizontally slides from the left edge of a scaffold to show navigation links. When you add a **Drawer** inside a **Scaffold**, the **AppBar** automatically adds a hamburger button to open the panel:

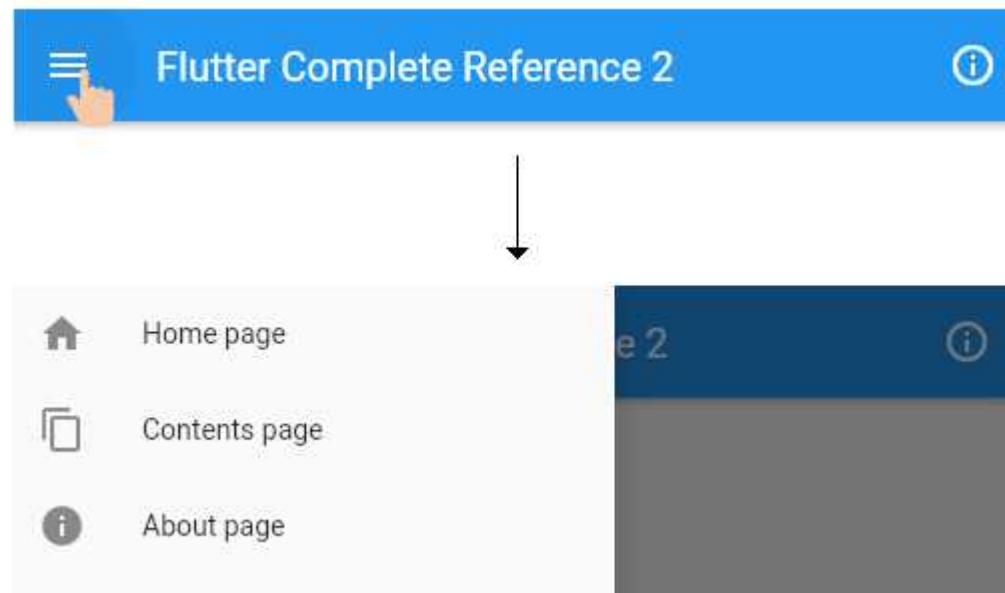


Figure 11.6: The hamburger button and an opened navigation drawer.

You can also open a drawer by sliding from the left edge towards the center. To obtain the result in *Figure 11.6*, use a `Drawer` and add some `ListTile` widgets inside:

```
Scaffold(  
  drawer: Drawer(  
    child: Column(  
      children: [  
        ListTile(  
          title: const Text('Home page'),  
          leading: const Icon(Icons.home),  
          onTap: () {}),  
        ListTile(  
          title: const Text('Contents page'),  
          leading: const Icon(Icons.content_copy),  
          onTap: () {}),  
        ListTile(  
          title: const Text('About page'),  
          leading: const Icon(Icons.info),  
          onTap: () {}),  
      ],  
    ),  
  ),  
,  
,
```

We already know that a `Column` doesn't scroll so when you have a lot of tiles, they might overflow. To add the scrolling behavior, replace the column with a `ListView` widget, which will be covered in *chapter 14 – Section 3 “Scrollable widgets”*.

- `BottomNavigationBar`: A material widget displayed at the bottom for selecting a restricted number of views, typically between two or five:



Figure 11.7: An example of a bottom navigation bar with some items.

Each item is represented by an icon and a label. You have to store the currently selected index somewhere to make the component work. For example, the `State` class of a stateful widget is a good place:

```
class Example extends StatefulWidget {
  const Example({super.key});

  @override
  State<Example> createState() => _ExampleState();
}

class _ExampleState extends State<Example> {
  var index = 0;

  void onItemTap(int newIndex) {
    setState(() {
      index = newIndex;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: Colors.white,
      body: Center(
        child: Text('Selected page ${index + 1}'),
      ),
      bottomNavigationBar: BottomNavigationBar(
        items: const <BottomNavigationBarItem>[
          BottomNavigationBarItem(
            icon: Icon(Icons.home),
            label: 'Home',
          ),
          BottomNavigationBarItem(
            icon: Icon(Icons.business),
            label: 'Business',
          ),
        ],
        currentIndex: index,
        selectedItemColor: Colors.amber[800],
        onTap: onItemTap,
      ),
    );
  }
}
```

- **FloatingActionButton**: Also known as “FAB”, it is a special rounded button that usually appears in the bottom-right corner of the screen. It’s used to promote a primary action in the application, and it hovers the contents underneath (if any):

```
floatingActionButton: FloatingActionButton(
  child: const Icon(Icons.send),
  onPressed: () {},
),
```

You can see this button in the bottom right corner of *Figure 11.5*. To change its position, just pass a new value to the `floatingActionButtonLocation` property of the `Scaffold`.

- **TabBar** and **TabBarView**: When you want to divide a single page into multiple tabs, you can use a `TabBar` to define the tabs and `TabBarView` to associate a page to each tab. You also have to make sure to wrap everything in a `DefaultTabController` widget:

```
DefaultTabController(
  length: 2,
  child: Scaffold(
    backgroundColor: Colors.white,
    appBar: AppBar(
      title: const Text('Flutter Complete Reference 2'),
      bottom: const TabBar(
        tabs: [
          const Tab(
            text: 'Home',
            icon: const Icon(Icons.home),
          ),
          const Tab(
            text: 'Info',
            icon: const Icon(Icons.info),
          ),
        ],
      ),
    ),
    body: TabBarView(
      children: const [
        Center(child: Text('Home page')),
        Center(child: Text('Info page')),
      ],
    ),
  ),
),
```

The `length` parameter of the `DefaultTabController` defines how many children both `Tab` and `TabBarView` must have. In our case, each list must have exactly 2 children. The final result looks like this:



Home page

Figure 11.8: An example of a tabbed layout.

You can change the view by tapping on the tab buttons in the app bar. On mobile devices, you can also change the view with a swipe gesture. To override this behavior and also enable sliding on desktop and web, you need to change the default configurations:

```
class CustomScrollBehavior extends MaterialScrollBehavior {  
  const CustomScrollBehavior();  
  
  @override  
  Set<PointerDeviceKind> get dragDevices => {  
    PointerDeviceKind.touch,  
    PointerDeviceKind.mouse,  
  };  
}  
  
void main() {  
  runApp(const MaterialApp(  
    scrollBehavior: const CustomScrollBehavior(),  
    home: SomeTabs(),  
  ));  
}
```

Thanks to this override, all material widgets supporting swiping can be swiped on mobile, desktop and web platforms.

Make sure to visit the official widget catalog<sup>78</sup> and the inline documentation for all the possible widgets and configurations you can make on material components.

### 11.2.2 Buttons

Buttons don't need introductions since they are probably the most basic UI component. There are a few implementations in the material design system:

- **TextButton**: a flat button with a transparent background without a border outline.

Widget	Normal	Hovered
<pre>TextButton(     onPressed: () {},     child: const Text('Press me!'), ) ,</pre>		

- **OutlinedButton**: essentially a **TextButton** with an outlined border.

Widget	Normal	Hovered
<pre>OutlineButton(     onPressed: () {},     child: const Text('Press me!'), ) ,</pre>		

- **ElevatedButton**: a filled button whose background elevates when pressed.

Widget	Normal	Hovered
<pre>ElevatedButton(     onPressed: () {},     child: const Text('Press me!'), ) ,</pre>		

<sup>78</sup> <https://docs.flutter.dev/development/ui/widgets/material>

- [IconButton](#): an icon on a material widget that reacts to touches by filling the background with a color.

Widget	Normal	Hovered
<code>IconButton(     onPressed: () {},     icon: const Icon(Icons.home), ) ,</code>		

Each button has an `onPressed` callback that is required but nullable. Whenever `onPressed` is set to `null`, the button is disabled and thus it cannot be pressed.

### 11.2.3 Dialogs

A dialog is a modal window that appears in front of the application contents to provide information or, more commonly, ask for a decision. Low-priority information can be displayed in a snackbar, but for important communications a dialog is preferred. There are various dialog widgets:

- [AlertDialog](#): An alert dialog informs the user about a situation that could require action. It has a title, a body, and one or more action buttons:

```
Future<void> openDialog(BuildContext context) async {  
    await showDialog<void>(  
        context: context,  
        builder: (context) {  
            return AlertDialog(  
                title: const Text('Info'),  
                content: const Text('Dialog contents'),  
                actions: [  
                    ElevatedButton(  
                        onPressed: doSomething,  
                        child: Text('OK'),  
                    ),  
                ],  
            );  
        },  
    );  
}
```

The [AlertDialog](#) is simply a stateless widget that holds text and a few buttons. To make it appear above the application contents with entrance and exit animation, we need to use the `showDialog` method, which produces this result:

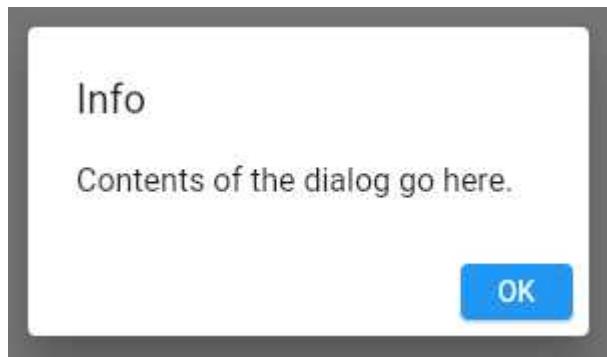


Figure 11.9: A Material alert dialog with a semi-transparent background.

To close this dialog, you can either tap/click in the grayed area underneath the box or add a `Navigator.of(context).pop()` call to one of the buttons. We will see what a navigator is in *chapter 13 – Routes and navigation*. If you set `barrierDismissible` to `false`, the dialog won't close when tapping in the grayed area:

```
await showDialog(  
    barrierDismissible: false,  
    context: context,  
    builder: (context) {},  
)
```

In such case, the dialog will never close unless you explicitly call the `pop()` method..

- `SimpleDialog`: Offers the user a choice between several options and it typically is a child of the `showDialog` method. It's very similar to an alert dialog, but it's not meant to inform the user about something. For example:

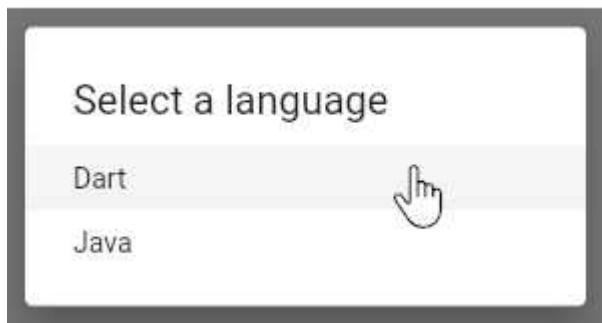


Figure 11.10: A simple dialog with two options.

In this case, can use the returned value from `showDialog` to retrieve the user selection:

```

Future<void> openDialog(BuildContext context) async {
    // note the nullable return type
    final String? value = await showDialog<String>(
        context: context,
        builder: (context) {
            return SimpleDialog(
                title: const Text('Select a language'),
                children: <Widget>[
                    SimpleDialogOption(
                        onPressed: () => Navigator.of(context).pop('Dart'),
                        child: const Text('Dart'),
                    ),
                    SimpleDialogOption(
                        onPressed: () => Navigator.of(context).pop('Java'),
                        child: const Text('Java'),
                    ),
                ],
            );
        },
    );
}

// This can be 'null' if the user closes the dialog without choosing.
debugPrint(value);
}

```

If the user closed the dialog without choosing one of the options, the dialog returns `null` (hence the nullable `String?` type).

- **SnackBar:** A “snackbar” holds a lightweight message with an optional action on the right, which briefly displays at the bottom of the screen. It’s used within a `Scaffold` and it should be displayed using the `ScaffoldMessenger` class. For example:

```

void openSnackbar(BuildContext context) {
    ScaffoldMessenger.of(context).showSnackBar(const SnackBar(
        content: Text('A message goes here!'),
        duration: Duration(seconds: 2),
        elevation: 5,
    ));
}

```

A `ScaffoldMessenger` manages a `SnackBar` within a `Scaffold`. This code shows a black stripe, with a sliding transition, at the bottom of the screen for two seconds:

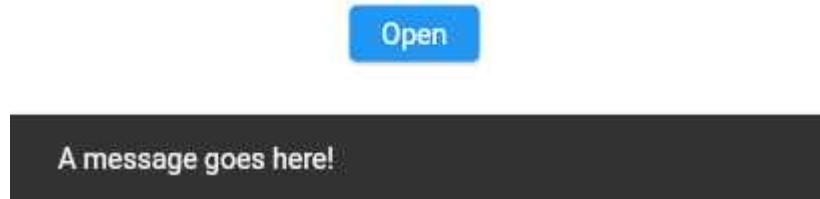


Figure 11.11: A button at the bottom of a Scaffold that opens a snackbar.

The `SnackBar` you see in the image is displayed at the bottom of a `Scaffold`, and the button triggers the `openSnackbar(context)` method we've created on the previous page. With a bit of customization, we can also make the container “float” on the screen and add a button:

```
void openSnackBar(BuildContext context) {  
    ScaffoldMessenger.of(context).showSnackBar(const SnackBar(  
        content: Text('A message goes here!'),  
        duration: Duration(seconds: 2),  
        elevation: 5,  
        behavior: SnackBarBehavior.floating,  
        width: 350,  
        backgroundColor: Colors.grey,  
    ));  
}
```

The snackbar is now “detached” from the bottom and has a fixed width. Its background color has also changed from black (the default) to gray:

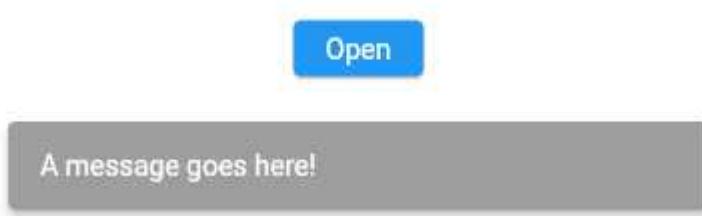


Figure 11.12: A customized snackbar

- `BottomSheet`: A *persistent* bottom sheet shows information that supplements the primary content of the application. It slides from the bottom towards the center of the screen and stays visible even when the user interacts with other parts of the application:

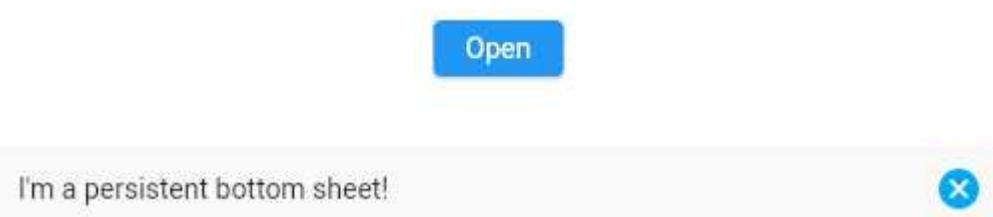


Figure 11.13: A persistent bottom sheet

The `showBottomSheet` function is used to show a persistent bottom sheet at the bottom of the screen. To close it, you can use the `Navigator` class for example:

```
showBottomSheet<void>(
    context: context,
    elevation: 4,
    builder: (context) {
      return Row(
        mainAxisAlignment: MainAxisAlignment.spaceBetween,
        children: [
          const Padding(
            padding: EdgeInsets.all(10),
            child: Text("I'm a persistent bottom sheet!"),
          ),
          IconButton(
            onPressed: () => Navigator.of(context).pop(),
            icon: const Icon(Icons.cancel, color: Colors.lightBlue),
          ),
        ],
      );
    },
);
```

This function returns a controller that can be used to programmatically close the bottom sheet without using a navigator. For example, you could also close the menu in this way:

```
// Get a reference to the controller
final controller = showBottomSheet<void>(
    context: context,
    builder: (_) => const SizedBox.shrink(),
);

// Use it later to close it
controller.close();
```

Material also defines another kind of bottom sheet called *modal*, which is an alternative to a dialog and prevents the user from interacting with the rest of the app. It can be closed programmatically (with a `pop()` call) or by tapping in the grayed area:

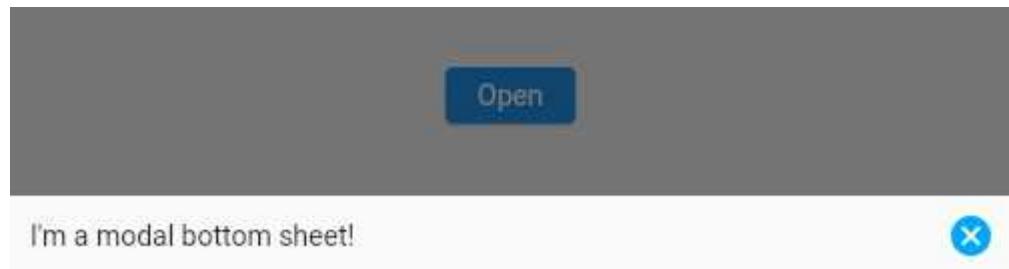


Figure 11.14: A modal bottom sheet

The code is the same as the one we've already seen for a persistent bottom sheet, with the only difference being the function to call:

```
showModalBottomSheet<void>(
    context: context,
    elevation: 4,
    builder: (context) {
        return Row(
            mainAxisAlignment: MainAxisAlignment.spaceBetween,
            children: [
                Padding(
                    padding: const EdgeInsets.all(10),
                    child: const Text("I'm a modal bottom sheet!"),
                ),
                IconButton(
                    onPressed: () => Navigator.of(context).pop(),
                    icon: const Icon(
                        Icons.cancel,
                        color: Colors.lightBlue,
                    ),
                ),
            ],
        );
    },
);
```

We're using `showModalBottomSheet` instead of `showBottomSheet`. A modal bottom sheet does not return a controller, so you have to close it using the navigator.

#### 11.2.4 Themes

To share colors and font styles across the application without duplicating code, themes are the right choice. You can define application-wide themes using [ThemeData](#), or local themes that only apply to a particular portion of the widget tree using [Theme](#). For example:

```
MaterialApp(  
  theme: ThemeData(  
    primaryColor: Colors.amber,  
    fontFamily: 'Georgia',  
    textTheme: const TextTheme(  
      headline1: TextStyle(fontSize: 60, fontWeight: FontWeight.bold),  
      headline3: TextStyle(fontSize: 42, color: Colors.grey),  
    ),  
  ),  
  home: const MyApp(),  
)
```

To share a theme across the entire application, pass a [ThemeData](#) to the [MaterialApp](#) constructor. There are more than sixty constructor parameters you can use to customize your Material widgets' appearance. You can style the application with “dark” and “light” variants using [ThemeData](#)'s named constructors:

- [ThemeData.light\(\)](#) is the default theme when no [theme](#) is defined.
- [ThemeData.dark\(\)](#) is the dark theme of the Material design library.

If you like the Material dark theme but there's something you'd like to adjust, you can use [copyWith](#) to only customize specific settings. For example:

```
void main() {  
  final myDarkTheme = ThemeData.dark().copyWith(  
    scaffoldBackgroundColor: Colors.blueGrey,  
  );  
  
  runApp(  
    MaterialApp(  
      theme: myDarkTheme,  
      home: const Example(),  
    ),  
  );  
}
```

If no [theme](#) is provided to [MaterialApp](#), the framework uses the light Material theme. If you want to apply a specific theme to a specific portion of your application, use the [Theme](#) widget. It overrides

the global theme configuration and applies the changes only to its children. For example, this is how you override a theme of a floating action button only:

```
MaterialApp(  
    theme: ThemeData.dark(),  
    home: Scaffold(  
        appBar: AppBar(  
            title: const Text('Example'),  
        ),  
        floatingActionButton: Theme(  
            data: ThemeData.light(),  
            child: FloatingActionButton(  
                onPressed: () {},  
                child: const Icon(Icons.info_outline),  
            ),  
        ),  
    ),  
,
```

The entire application uses the dark theme configuration, but the floating action button (along with its child) uses the light configuration instead. Sometimes, it may make more sense to override only a few configurations of the default theme. You can use `Theme.of` to retrieve the current Material theme and change it:

```
MaterialApp(  
    theme: ThemeData.dark(),  
    home: Scaffold(  
        appBar: AppBar(  
            title: const Text('Example'),  
        ),  
        floatingActionButton: Theme(  
            data: Theme.of(context).copyWith(  
                floatingActionButtonTheme: FloatingActionButtonThemeData(  
                    backgroundColor: Colors.LightGreen,  
                ),  
            ),  
            child: FloatingActionButton(  
                onPressed: () {},  
                child: const Icon(Icons.info_outline),  
            ),  
        ),  
    ),  
,
```

The `Theme.of` method call relies on an `InheritedWidget`, a special kind of widget that is analyzed in *chapter 12 – State management*. This code produces the following result:

## Example

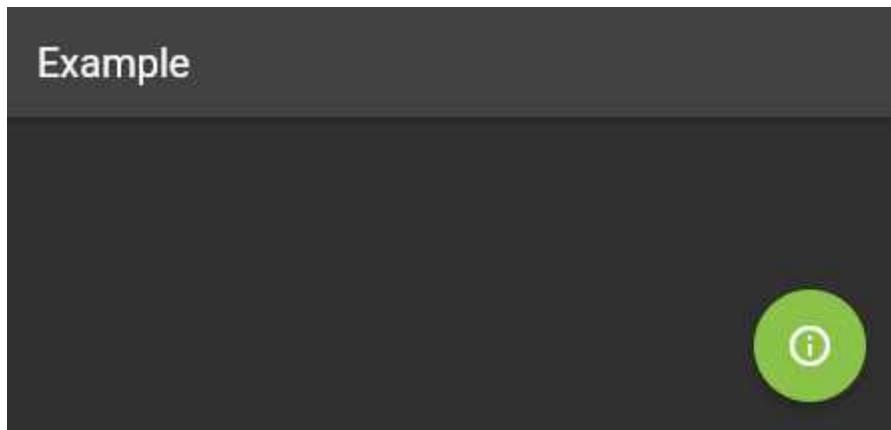


Figure 11.15: A dark Material app with a custom FAB color.

As you can see, we have changed the background color of the floating action button while keeping the rest of the application with a dark theme.

## 11.3 Cupertino library

As we've already seen, Flutter comes with some pre-built widgets that implement the iOS design language. Even if these widgets best suit on Apple's mobile devices, nothing stops you from also using them on Android or desktop platforms for example. This is an example of Cupertino style:

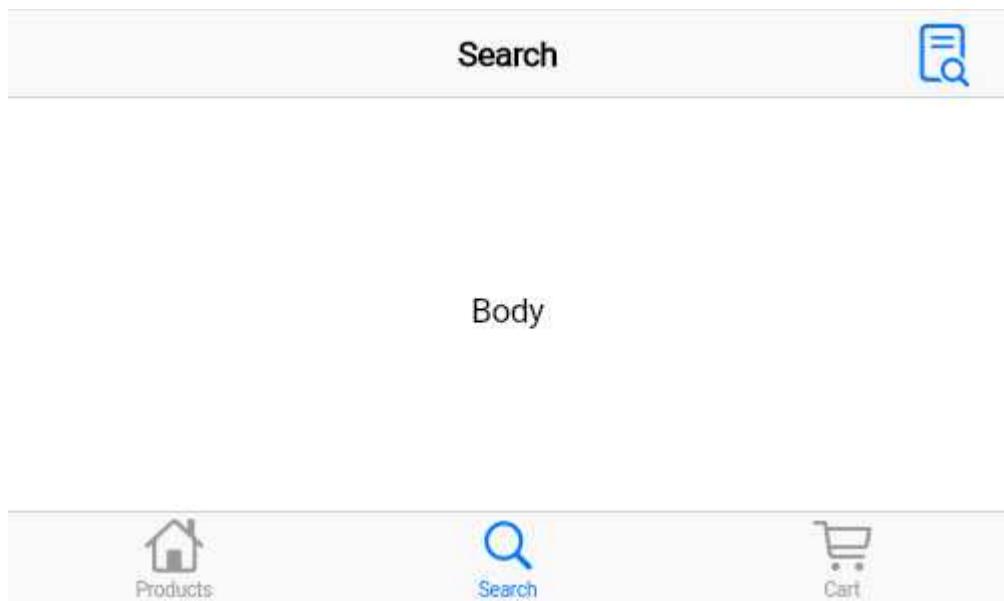


Figure 11.16: A basic Cupertino app with a scaffold and a navigation bar

Of course, you could create this kind of UI by yourself without using Flutter's Cupertino library but it would be very time-consuming. Rather than reinventing the wheel, just make sure to import the Cupertino library and you're ready to go:

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

The Cupertino library is already bundled in the SDK so you don't need to add external dependencies in the `pubspec.yaml` file. Cupertino has less widgets than Material but it doesn't mean that it has a lower quality or it's less valuable.

### 11.3.1 Cupertino scaffolds

With Cupertino, there are various kinds of scaffolds you can use. A `CupertinoPageScaffold` is the most basic one that implements a single-page layout. It simply defines a navigation bar at the top and the contents below:

```
CupertinoApp(  
  home: const CupertinoPageScaffold(  
    navigationBar: CupertinoNavigationBar(  
      middle: Text('Simple page'),  
      trailing: Icon(CupertinoIcons.add),  
    ),  
    child: Center(  
      child: Text('Page contents'),  
    ),  
  ),  
);
```

To keep consistency with the design, you may want to use `CupertinoIcons` instead of `Icons`. This code produces the following iOS-like design:

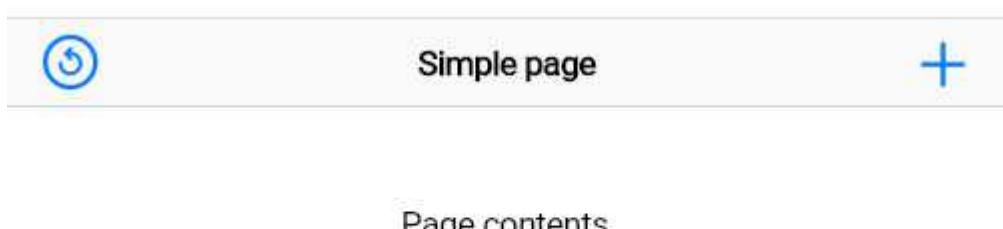


Figure 11.17: A `CupertinoPageScaffold` with a navigation bar and some buttons.

A `CupertinoTabScaffold` scaffold is used to show multiple pages on a single screen. It lays out the tab bar at the bottom of the page and automatically listens to tap events to change the currently visible contents. For example:

```
CupertinoTabScaffold(  
  tabBar: CupertinoTabBar(  
    items: const [  
      BottomNavigationBarItem(  
        icon: Icon(CupertinoIcons.home),  
        label: 'Products',  
      ),  
      BottomNavigationBarItem(  
        icon: Icon(CupertinoIcons.search),  
        label: 'Search',  
      ),  
    ],  
  ),  
  tabBuilder: (context, index) {  
    CupertinoTabView returnValue;  
    switch (index) {  
      case 0:  
        returnValue = CupertinoTabView(  
          builder: (context) {  
            return const Center(  
              child: Text('Products page'),  
            );  
          },  
        );  
        break;  
      default:  
        returnValue = CupertinoTabView(  
          builder: (context) {  
            return const Center(  
              child: Text('Search page'),  
            );  
          },  
        );  
        break;  
    }  
    return returnValue;  
  },  
);
```

While `CupertinoTabBar` handles tabs at the bottom of the screen, a `CupertinoTabView` widget is used to build a page. The `index` property of the `tabBuilder` is updated to reflect the currently

selected index, which can be changed by tapping/clicking on a `BottomNavigationBarItem` widget. The code produces the following output:

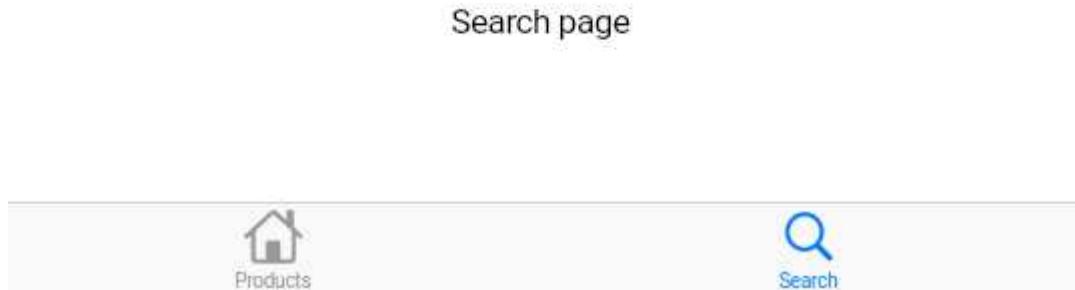


Figure 11.18: A CupertinoTabScaffold with two tabs.

### 11.3.2 Buttons

In the iOS design system, there is a single kind of button; it may change color and size, but, in the end, it's still the same component. The child of a `CupertinoButton` widget generally is a `Text` or an `Icon`. For example:

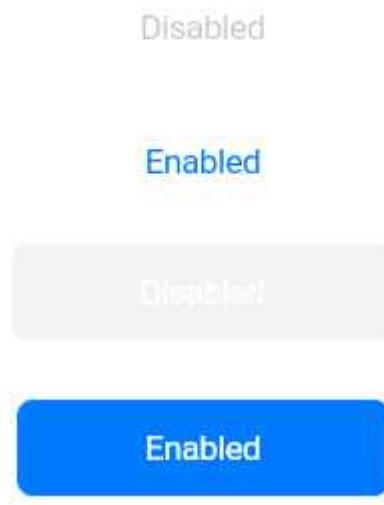
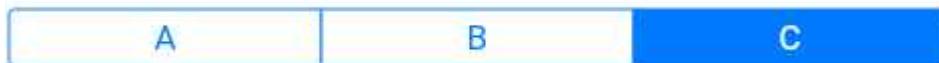


Figure 11.19: Standard and filled versions of Cupertino buttons.

The first two buttons of *Figure 11.19* are created using the default `CupertinoButton` constructor. The other two use the `CupertinoButton.filled` named constructor, which adds background and some more padding around the child:

```
Column(  
  children: [  
    const CupertinoButton(  
      onPressed: null,  
      child: Text('Disabled'),  
    ),  
    const SizedBox(height: 15),  
    CupertinoButton(  
      onPressed: () {},  
      child: const Text('Enabled'),  
    ),  
    const SizedBox(height: 15),  
    const CupertinoButton.filled(  
      onPressed: null,  
      child: Text('Disabled'),  
    ),  
    const SizedBox(height: 15),  
    CupertinoButton.filled(  
      onPressed: () {},  
      child: const Text('Enabled'),  
    ),  
  ],  
,),
```

As always, whenever the `onPressed` callback is set to `null`, the button is disabled and thus cannot be clicked/tapped. When you need more buttons that require a mutually exclusive choice, rather than using a series of `CupertinoButtons`, prefer a `CupertinoSegmentedControl` widget. It has a series of choices aligned along the horizontal axis, and only one of them can be selected at the same time:



*Figure 11.20: A CupertinoSegmentedControl with three buttons.*

Each button is represented by a `Map` whose `value` holds the children to be placed inside the button. For example, the component in *Figure 11.20* is rendered with the following code:

```

class _ExampleState extends State<Example> {
  int? currentValue;

  final Map<int, Widget> children = const <int, Widget>{
    0: Text('A'),
    1: Text('B'),
    2: Text('C'),
  };

  @override
  Widget build(BuildContext context) {
    return CupertinoSegmentedControl<int>(
      children: children,
      onValueChanged: (newValue) {
        setState(() {
          currentValue = newValue;
        });
      },
      groupValue: currentValue,
    );
  }
}

```

Each key of the `children` map is the value that uniquely identifies a button. Whenever the currently selected button changes, the `onValueChanged` callback is triggered. The `groupValue` property is very important because it keeps track of the currently selected value. With a the `currentValue` state variable we can simply manage the component state and remember which button is currently selected.

## 11.4 CustomPaint and CustomPainter

Flutter is shipped with hundreds of pre-built widgets; you can generally reuse them to build the UI as you wish. However, there are some cases where you may need to render complex shapes such as arcs or particular polygons. For example, imagine you had to create this progress indicator bar:



Figure 11.21: A rounded progress indicator with a grey track underneath.

You may want to start with a `Stack` to overlap two `Containers`, rotate them, and somehow try to stick everything together with relative coordinates. You'll quickly realize how hard this can become. In these cases, using a canvas and some mathematical skills is the best solution. A `CustomPainter` object is used to execute painting commands on a canvas. For example:

```
class SimplePainter extends CustomPainter {  
  const SimplePainter();  
  
  @override  
  void paint(Canvas canvas, Size size) {  
    final paint = Paint()  
      ..strokeWidth = 2  
      ..color = Colors.grey.withOpacity(80)  
      ..style = PaintingStyle.fill;  
  
    final center = Offset(  
      size.width / 2,  
      size.height / 2,  
    );  
  
    canvas.drawCircle(center, 10, paint);  
  }  
  
  @override  
  bool shouldRepaint(covariant ProgressPainter oldDelegate) {  
    return false;  
  }  
}
```

This code is used to paint a circle with a grey background color. Custom painters are the “low-level” way to tell Flutter what you want to render on the screen. There are two methods you always need to override:

1. `paint`: This is the place where you’re given a canvas on which you can freely draw. In this method, you generally use the `Paint` class to define the colors and the style of what you paint. The `canvas` object is used to draw shapes, lines, or polygons.
2. `shouldRepaint`: This method is used to determine under which circumstances the canvas should be repainted. In this case, we directly return `false` because the painter has no external dependencies so it should never be repainted.

Let's try to work on a more complicated example to see how painters can be used. We're now going to create a custom painter to create the radial progress indicator we have shown in *Figure 11.21*. To get started, we always need to subclass the `CustomPainter` type:

```
class ProgressPainter extends CustomPainter {
    final double progressValue;
    const ProgressPainter({
        required this.progressValue,
    });

    @override
    void paint(Canvas canvas, Size size) {
        // Drawing arcs here...
    }

    @override
    bool shouldRepaint(covariant ProgressPainter oldDelegate) {
        return progressValue != oldDelegate.progressValue;
    }
}
```

In this case, there is an external dependency (`progressValue`) that is used in the `paint` method. We have overridden the `shouldRepaint` to return `true` only if the progress value changes. Let's move to the `paint` method:

```
@override
void paint(Canvas canvas, Size size) {
    final railPainter = Paint()
        ..strokeWidth = 5
        ..color = Colors.grey.withOpacity(80)
        ..style = PaintingStyle.stroke;

    final progressPainter = Paint()
        ..strokeWidth = 5
        ..color = Colors.blue
        ..style = PaintingStyle.stroke;

    final center = Offset(size.width / 2, size.height / 2);

    // Arcs drawing here...
}
```

First of all, we need to define the painters that will color the grey track and the blue progress indicator. Note that `PaintingStyle.stroke` paints the borders, and `PaintingStyle.fill` paints the background color of the shapes. Here's the full implementation of `paint`:

```

@Override
void paint(Canvas canvas, Size size) {
    final railPainter = Paint()
        ..strokeWidth = 5
        ..color = Colors.grey.withOpacity(80)
        ..style = PaintingStyle.stroke;

    final progressPainter = Paint()
        ..strokeWidth = 5
        ..color = Colors.blue
        ..style = PaintingStyle.stroke;

    // The coordinates of the box center
    final center = Offset(size.width / 2, size.height / 2);

    // Drawing the grey track underneath the progress status
    canvas.drawArc(
        Rect.fromCenter(center: center, width: size.width, height: size.height),
        0,
        math.pi * 2,
        false,
        railPainter,
    );

    // Drawing the progress track
    canvas.drawArc(
        Rect.fromCenter(center: center, width: size.width, height: size.height),
        -math.pi / 2,
        math.pi * 2 * progressValue / 100,
        false,
        progressPainter,
    );
}

```

As the name suggests, the `drawArc` method is used to draw arcs and takes a few parameters:

1. The coordinates of the arc center, provided by a `Rect` object. A `Rect` is an immutable, 2D, floating point rectangle whose coordinates are relative to a given origin.
2. The starting angle from which painting the arc.
3. The sweep angle, which determines “how long” the arc has to be.
4. When this parameter is `true`, the arc is closed with a line that connects the edges to the center. We set it to `false` because we want to paint an open arc.
5. The painter, which tells the engine how to paint the arc.

The `CustomPaint` widget is used to render custom painters to the UI. In other words, it “installs” a custom painter in the widget tree. `CustomPaint` sizes itself according to the child dimensions so we can easily return a `SizedBox` to force constraints around the text:

Widget	Result
<pre>class ArcProgress extends StatelessWidget {   final double progress;   const ArcProgress({     super.key,     required this.progress,   }) : super(key: key);    @override   Widget build(BuildContext context) {     return CustomPaint(       painter: ProgressPainter(         progressValue: progress       ),       child: SizedBox(         width: 100,         height: 100,         child: Center(           child: Text('\$progress%',             style: const TextStyle(               fontSize: 20,             ),           ),         ),       ),     );   } }</pre>	<pre>ArcProgress(   progress: 63, ),</pre> 

If there wasn’t a `child`, you could have used the `size` parameter to size the painter. Using custom painters isn’t trivial because they require good mathematical skills. Our example wasn’t too tricky because we’ve only drawn two arcs. If you wanted to render waves or very particular shapes for example, you would need knowledge about Bézier curves and trigonometry. However, for basic shapes and lines, using the `canvas` is not too hard:

- `canvas.drawCircle`: draws a circle centered at the given point with the given radius;

- `canvas.drawLine`: draws a straight line from a starting point to an ending point;
- `canvas.drawRect`: draws a rectangle;
- `canvas.drawRRect`: draws a rectangle whose corners are rounded by the given amount;
- `canvas.drawParagraph`: draws some text into the canvas.

You can tell the `CustomPaint` widget whether the painter has to be placed in front or behind the child, if any. For example:

- If you use the `painter` parameter, the `child` is placed in front of the custom painter you created:

```
CustomPaint(
  painter: ProgressPainter(
    progressValue: 63,
  ),
  child: Container(
    width: 90,
    height: 90,
    color: Colors.grey,
  ),
),
```



- If you use the `foregroundPainter` parameter, the `child` is placed in front of the custom painter you created:

```
CustomPaint(
  foregroundPainter: ProgressPainter(
    progressValue: 63,
  ),
  child: Container(
    width: 90,
    height: 90,
    color: Colors.grey,
  ),
),
```



We have replaced the text with a gray container so that you can clearly see the difference. In our example, the arc doesn't fill the internal space so the `Text` is always visible (regardless we use a `painter` or a `foregroundPainter`).

## 11.4.1 Effects

In some cases, you can apply effects to your widgets without directly using custom painters. The framework provides some useful classes you can use out of the box, but they may require attention due to potential performance drawbacks.

### 11.4.1.1 Opacity

The `Opacity` widget can be used to make its child partially transparent. For example:

```
const Opacity(  
    opacity: 0.7,  
    child: Text('Hello!'),  
)
```

This class paints its child into an intermediate buffer and then blends the child back into the scene partially transparent. As such, it's worth pointing out some performance considerations:

- when `opacity` is 0.0, the child isn't painted at all;
- when `opacity` is 1.0, the child is directly painted without intermediate buffers;
- for values of `opacity` other than 0.0 and 1.0, this widget is relatively expensive because it paints the child into an intermediate buffer.

The best idea would be to use a `const` constructor on this widget or (if not possible) manually cache it as we've explained in *chapter 10 – Section 3.2 “Performance considerations”*. We don't want to discourage the usage of this widget, but we want to point out that there are some cases where it's not a good choice. For example:

```
Opacity(  
    opacity: 0.5,  
    child: Container(  
        width: 50, height: 50,  
        color: Colors.blue,  
)  
)
```

Here the engine has to apply an opacity value to the blue color. A better and much faster solution is to use the `Color` or `Colors` class utilities:

```
// Better because colors with alpha don't internally use intermediate buffers.  
Container(  
    width: 50, height: 50,  
    color: Colors.blue.withOpacity(0.5),  
)
```

This approach is more efficient than using `Opacity` on top the `Container` them because `Opacity` could apply the opacity to a group of widgets (and therefore a costly offscreen buffer will be used). Drawing content into the offscreen buffer may also trigger render target switches.

#### 11.4.1.2 Clip widgets

Clip widgets are used to clip their children using various shapes. The term “clipping a widget” means preventing a child from painting outside of its bounds. Here are some examples with a `Container` and various clipping widgets:

Widget	Result
<pre>Container(     width: 70,     height: 70,     color: Colors.blue, )</pre>	
<pre>ClipRRect(     borderRadius: BorderRadius.all(         Radius.circular(10)), ,     child: Container(         width: 70,         height: 70,         color: Colors.blue, , , )</pre>	
<pre>ClipOval(     child: Container(         width: 70,         height: 70,         color: Colors.blue, , , )</pre>	

A `ClipOval` inscribes an oval into its layout dimensions and prevents its child from painting outside of the oval. The same idea also works for the `ClipRRect`, with the only difference being that it clips with a rounded rectangle rather than a circle. They both preserve the child size, but we can of course

customize this behavior with a custom clipper. For example, this is how you can create a clipper that imposes custom constraints on the child:

```
class MyClipper extends CustomClipper<Rect> {
  const MyClipper();

  @override
  Rect getClip(Size size) {
    return const Rect.fromLTWH(0, 0, 50, 50);
  }

  @override
  bool shouldReclip(covariant CustomClipper<Rect> oldClipper) => false;
}
```

Similarly to custom painters, the `shouldReclip` method should always return `false` if the clipper has no external dependencies. Keep in mind that the `getClip` override might be optimized away by the framework if `shouldReclip` always returns `false`.

### Note

The `Rect.fromLTWH` constructor builds a rectangle from its left and top edges, its width, and its height. In our example, we want the clipped child to be as close to the top-right edge as possible (the first two `0, 0` parameters) and have a size of `50x50` (the last two `50, 50` parameters).

Since `MyClipper` returns fixed clip sizes (`50x50`), if we wrapped our original `Container` (`70x70`) in a clip widget it would be smaller:

```
ClipOval(
  // 'MyClipper' forces the child to be 50x50
  clipper: const MyClipper(),
  child: Container(
    width: 70,
    height: 70,
    color: Colors.blue,
  ),
),
```

Note that `ClipOval`'s size is still `70x70`, but the child is constrained and painted using a `50x50` box. In the following image, we have added a grey background to visualize the difference better. As you can see, `MyClipper` imposes smaller constraints, but they're only applied to the child:



Figure 11.22: The without and with the clipper

Of course, you can also use custom clippers with `ClipRRect`, `ClipRect` or `ClipPath` and they work in the same way. There are no special performance considerations you should be aware of.

## Deep dive: Asynchronous widgets

When you need to wait for a future to complete, use the `FutureBuilder` widget. It builds a subtree according with the future's current state. In other words: rather than using `async` and `await` inside the `build` method, use a `FutureBuilder`. Here's an example:

```
class _MyWidgetState extends State<MyWidget> {
    final randomFuture = Future<int>.delayed(
        const Duration(seconds: 2),
        () => Random().nextInt(10),
    );

    @override
    Widget build(BuildContext context) {
        return FutureBuilder<int>(
            future: randomFuture,
            builder: (context, snapshot) {
                // The future completed with success and 'data' is not null
                if (snapshot.hasData) {
                    return Text('${snapshot.data}');
                }
                // If the future completed with an error, show an error message
                if (snapshot.hasError) {
                    return const Text('Something went wrong!');
                }

                return const Text('Wait...');
            },
        );
    }
}
```

In this example, we're using a `FutureBuilder` widget to wait for the future to complete and build the UI accordingly. The `snapshot` variable holds information about the future state:

- `data`: contains the latest data received by the asynchronous computation and can be `null`.
- `hasData`: it is `true` only when the future completed with success and it returned a non-null value. For this reason, if the future completed with success but it returned `null`, this getter would return `false`.
- `connectionState`: it determines the current state of the future. This `enum` is generally used when the future may return `null` and you cannot only use `hasData` to check the completion status. For example:

```
builder: (context, snapshot) {  
  if (snapshot.connectionState == ConnectionState.done) {  
    if (snapshot.hasError) {  
      return const Text('Failure');  
    }  
    if (snapshot.hasData) {  
      return const Text('Success and non-null value');  
    }  
  
    return const Text('Success and null value');  
  }  
  
  return const Text('Wait...');  
}
```

The `done` value means that the future completed, either with a value or an error.

- `hasError`: it is `true` if the asynchronous computation's last result was a failure or the future completed with an error. When `true`, you can access the error object that was thrown using the `snapshot.error` getter.

The future object passed to the `FutureBuilder` widget must live in the state class<sup>79</sup>. This is to avoid that each rebuild would cause the future to be re-evaluated. Notice that in the example we stored the future in the state class:

---

<sup>79</sup> <https://api.flutter.dev/flutter/widgets/FutureBuilder-class.html>

```

// The future is assigned to a variable in the state class
final randomFuture = Future<int>.delayed(
  const Duration(seconds: 2),
  () => Random().nextInt(10),
);

@Override
Widget build(BuildContext context) {
  return FutureBuilder<int>(
    future: randomFuture, // <-- We use the variable that holds the future
    builder: (context, snapshot) { /* ... code ... */ },
);
}

```

Even if the widget is rebuilt, the future is not re-evaluated because it is “cached” in the state class. This is the correct way to use the `FutureBuilder` widget to wait for `Futures` to complete. It would have been very wrong if we used the future directly inside the `build` method like this:

```

@Override
Widget build(BuildContext context) {
  return FutureBuilder<int>(
    // This is bad
    future: Future<int>.delayed(
      const Duration(seconds: 2),
      () => Random().nextInt(10),
    ),
    builder: (context, snapshot) { /* ... code ... */ },
);
}

```

The problem is that whenever `FutureBuilder` is rebuilt, the future is also rebuilt (and restarted). In other words, passing the future directly in the `build` method causes the future to be re-executed every time the widget is rebuilt. Here is another example:

```

FutureBuilder<String>(
  // Assume that 'makeHttpRequest()' returned a 'Future<String>'
  future: makeHttpRequest(),
  builder: (context, snapshot) { /* ... code ... */ },
);

```

Since the returned future is not stored in a state variable, the HTTP request is re-executed on every rebuild. This is certainly not efficient and also not performant. These considerations are also valid for the `StreamBuilder` widget, which is used to listen on a stream and build the UI accordingly. For example:

```

class _MyWidgetState extends State<MyWidget> {
    // The stream must be declared in a state variable (same thing as futures)
    final counterStream = Stream<int>.periodic(
        const Duration(seconds: 1),
        (tick) => tick,
    );

    @override
    Widget build(BuildContext context) {
        return StreamBuilder<int>(
            initialData: 0,
            stream: counterStream,
            builder: (context, snapshot) {
                if (snapshot.hasData) {
                    return Text('value = ${snapshot.data!}');
                }

                if (snapshot.hasError) {
                    return Text('error = ${snapshot.error!}');
                }

                return const Text('No data');
            },
        );
    }
}

```

The `initialData` value is useful to make sure that the first frame has the expected value. If it's not specified, the initial value is set to `null`. Notice that we are not directly creating the stream inside the `build` method, but we're storing it inside the state class. As it happens for the `FutureBuilder`, this approach is wrong:

```

StreamBuilder<int>(
    initialData: 0,
    stream: Stream<int>.periodic(
        const Duration(seconds: 1),
        (tick) => tick,
    ),
    builder: (context, snapshot) {
        // code...
    },
);

```

Since the stream is not stored into a state variable, it is restarted on every rebuild. When you use a `StreamBuilder`, always give it a stream that was created inside the state class and not inside `build`.

## Note

A `FutureBuilder` behaves in the same way as a `StreamBuilder` that is configured with `future.asStream()`.

When you wait for a future outside of a `FutureBuilder`, you have to pay attention to asynchronous gaps with the widget's build context. This problem is caused by the fact that the widget might be removed from the tree while the operation hasn't yet completed. For example:

```
class MyWidget extends StatefulWidget {
  const MyWidget(super.key);

  void buttonCallback(BuildContext context) async {
    await Future<int>.delayed(const Duration(seconds: 6));
    doSomething(context); // This is a dangerous call
  }

  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: () => buttonCallback(context),
      child: const Text('Button'),
    );
  }
}
```

The button callback waits for six seconds and then does something using the `BuildContext` object. After you press the button, if `MyWidget` is removed from the tree after two seconds for example, the `doSomething` callback is still registered. Since `context` holds a reference to a dead widget, the `doSomething` callback will use an invalid context reference (and you'll get a runtime error). To fix the issue, check the widget status before using the context:

```
await Future<int>.delayed(const Duration(seconds: 6));

if (mounted) { // Checks that the widget is still mounted in the tree
  doSomething(context); // This is safe!
}
```

The practical rule is to always protect your code with `if (mounted)` if you're using the context variable after waiting for a future to complete. The `mounted` property can always be accessed from the widget's `BuildContext` object. In this way, you can check whether a widget is mounted or not on a `StatelessWidget` object too. For example:

```
if (context.mounted) {  
  // do something  
}
```

The same rule also applies in case of a `setState` call that comes after having waited for a future to complete. If you forgot about asynchronous gaps, you can configure a linter rule<sup>80</sup> that reminds you to fix your code. We will see how to configure the analyzer and customize linter rules in *chapter 20 – Section 1.3 “Static analysis and linter rules”*.

---

<sup>80</sup> [https://dart-lang.github.io/linter/lints/use\\_build\\_context\\_synchronously.html](https://dart-lang.github.io/linter/lints/use_build_context_synchronously.html)

# 12 – State management

---

## 12.1 Introduction

In an “imperative” style of UI programming, you generally create *mutable* entities which are later modified using methods and setters. In a “declarative” style of UI programming instead, entities are *immutable* and they need to be rebuilt to reflect the changes. Let’s make a concrete example to see the implications:

- **Imperative style.** If we wanted to change some text in an imperative framework, we would need to get a reference to the component and mutate it internally. For example:

```
Text text = findComponent(Text.id.hello_world);
text.setText('Hello!');
text.setFontSize(16);
```

The key point is that objects are long-lived and mutable. The text component doesn’t need to be recreated because setters (such as `setText`) modify the existing object. Android, for example, used to be an imperative UI framework.

- **Declarative style.** If we wanted to change some text in a declarative framework, we would need to entirely rebuild the component with a new configuration because it is immutable. For example:

```
Text(
  'Hello!',
  style: TextStyle(
    color: Colors.indigo,
  ),
),
```

The key point is that objects are short-lived and immutable. The `Text` component needs to be recreated because objects are immutable, and so they cannot be changed using setters. Flutter is a declarative UI framework.

In a very broad sense, the state of an application can be seen as the condition in which the various widgets are. If you change a color, increase a counter, or login a user for example, the state will change. In real-world applications, you need to keep multiple widgets in sync with the current state. It’s also very common creating more sub-states for specific parts of the widget tree. All of this can

be extremely confusing and hard to maintain without knowing how to *manage the state* of your application. During your Flutter development journey, you'll often be asked to create and manage either an *ephemeral state* or an *application state*.

### 12.1.1 Ephemeral and application state

The term “ephemeral state” refers to the state that can be managed in a single stateful widget. In other words, a `StatefulWidget` is all you need to keep track of the state because no other widgets care about it. For example, the ephemeral state is a great fit when you need to keep track of the currently selected index of a widget:

```
class _ExampleState extends State<Example> {
  var index = 0;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: Text('Selected page ${index + 1}'),
      ),
      bottomNavigationBar: BottomNavigationBar(
        items: const <BottomNavigationBarItem>[
          BottomNavigationBarItem(
            icon: Icon(Icons.home),
            label: 'Home',
          ),
          BottomNavigationBarItem(
            icon: Icon(Icons.business),
            label: 'Business',
          ),
        ],
        currentIndex: index,
        onTap: (newIndex) => setState(() => index = newIndex),
      ),
    );
  }
}
```

Using `setState` and the `index` variable is highly natural and easy. No other widgets need to access `index` because it's something that is only used by `Example`. In this example, we could also say that the state is scoped to the stateful widget because no one else cares about it. Other cases where the ephemeral state of a single  `StatefulWidget` may be enough are:

- when you need to keep track of the current progress of an animation;

- when you need to keep track of the currently selected index of a widget;
- when the widget configuration changes (for example, changing the state of a button from “enabled” to “disabled”).

The term application state refers to the state that is shared across multiple widgets. In this case, a single `StatefulWidget` is not enough because multiple widgets need to know about the state and potentially react if it changes. The application state is generally a good fit in these scenarios:

- When your application requires a user to login. For example, certain pages or widgets might be visible only when the user is logged in.
- When you need to create a shopping cart for an e-commerce project. Navigating between screens and interacting with the application doesn’t reset the items you put in the cart; they persist over time.
- When you need to keep track of user preferences or settings. These configurations are often stored on the disk, loaded at startup, and determine how certain widgets or parts of the application behave.

The application state is generally managed using `InheritedWidgets` and listenable classes, such as `ChangeNotifier`. They allow us to locate the state up in the widget tree and make it available to all the children along the hierarchy.

### 12.1.2 Good practices

Let’s start by clearly pointing out that you could use `setState` to manage both the ephemeral and the application state in your project. At the same time, it’s also true that you could use inherited widgets for everything (ephemeral and application state).

There is no clear rule that determines when one is better than the other, but we don’t want to play the devil’s advocate and say “it depends”. Here there are a few considerations we want to share with you, purely based on our professional experience mixed with some technical advice from the official <sup>81</sup> documentation:

---

<sup>81</sup> <https://docs.flutter.dev/perf/best-practices>

- When the state only rules a single widget, then a  `StatefulWidget` and  `setState` are very likely the right choices.
- When the state has to be shared across multiple widgets, an  `InheritedWidget` along with a listenable class is a great fit (also in terms of performance and rebuild optimization).
- The code may change very often due to client requests or just “regular” maintenance. For example, you could start with some obvious ephemeral state, but then you may need more features and the application state may become a better fit.

The point we want to underline is that both  `setState` and  `InheritedWidget` can do everything, but there are some cases where one has more benefits than the other. In general, we recommend to prefer using  `setState` on ephemeral states (the ones that affect a single widget) and inherited widgets otherwise.

In the following two sections, we will analyze some examples of both ephemeral and application state. Since we want to keep the focus on the state management approach, the UI of the example will be minimal.

## 12.2 Using `setState`

We have already introduced  `setState` in *chapter 10 – Flutter, widgets and trees* and discussed, in detail, how Flutter handles rebuilds. In this section, we want to focus on the use cases where this state management technique is best used. Let’s start with a simple counter application to see why  `setState` is extremely helpful for ephemeral state management:

```
void main() {
  runApp(
    const MaterialApp(
      home: Scaffold(
        body: Center(
          child: CounterApp(),
        ),
      ),
    ),
  );
}
```

All of these widgets don’t care about the current counter value and can be  `const`. For this reason, we’ve wrapped the actual counter inside the  `CounterApp` widget:

```

class CounterApp extends StatefulWidget {
  const CounterApp({super.key});

  @override
  State<CounterApp> createState() => _CounterAppState();
}

class _CounterAppState extends State<CounterApp> {
  var counter = 0;

  @override
  Widget build(BuildContext context) {
    return Column(
      mainAxisAlignment: MainAxisAlignment.min,
      children: [
        const Text('Press the button'),
        ElevatedButton(
          onPressed: () => setState(() { counter++ }),
          child: Text('$counter'),
        ),
      ],
    );
  }
}

```

The `setState` call will increase the counter and rebuild the `CounterApp` widget on the next frame to update the button's text. Remember that constant widgets are never rebuilt. While this approach is good, we can improve the code even more. Notice that the `Column` and the `Text` don't care about the state so, in this case, we can move them outside:

```

runApp(
  MaterialApp(
    home: Scaffold(
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.min,
          children: const [
            Text('Press the button'),
            CounterApp(),
          ],
        ),
      ),
    ),
  );
)

```

At this point, we can just leave the `ElevatedButton` inside `CounterApp` because it's the only widget that really cares about the state:

```
class CounterApp extends StatefulWidget {
  const CounterApp({super.key});

  @override
  State<CounterApp> createState() => _CounterAppState();
}

class _CounterAppState extends State<CounterApp> {
  var counter = 0;

  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: () => setState(() {
        counter++;
      }),
      child: Text('$counter'),
    );
  }
}
```

This is a perfect example of ephemeral state management. The example is quite trivial so there isn't a huge performance gain, but the critical point is that our `setState` call only rebuilds those widgets that depend on the state (`ElevatedButton`). This fact is also called out by the `StatefulWidget`<sup>82</sup> official documentation:

### Quote

“Push the state to the leaves. For example, if your page has a ticking clock, rather than putting the state at the top of the page and rebuilding the entire page each time the clock ticks, create a dedicated clock widget that only updates itself”.

The term “leaves” refers to widgets in the widget tree that don’t have a `child`. Since leaf widgets don’t have children, their state is “scoped” specifically to the widget itself, and it’s not exposed to other parts of the tree that don’t need it. Other than reducing the number of rebuilds, pushing the

---

<sup>82</sup> <https://api.flutter.dev/flutter/widgets/StatefulWidget-class.html>

state to the leaves also makes testing and maintenance easier because it localizes the state in those parts of the tree that actually need it. Let's see another example of ephemeral state management:

```
class _HideAndShowState extends State<HideAndShow> {
    var visible = false;

    @override
    Widget build(BuildContext context) {
        final text = visible ? 'Set to invisible' : 'Set to visible';

        return Column(
            children: [
                Text('Visible = $visible'),
                const SizedBox(height: 10),
                ElevatedButton(
                    onPressed: () => setState(() {
                        visible = !visible;
                    }),
                    child: Text(text),
                ),
            ],
        );
    }
}
```

Even if the `Column` and the `SizedBox` don't depend on the state, they are required for our desired layout. They cannot be moved outside, but this is totally fine because your goal is to minimize the number of widgets that get rebuilt. You don't necessarily need to always have a single widget that gets rebuilt.

In this case, we already have the “minimum” configuration possible for this state to work correctly. Flutter has an extremely fast and performant rebuilding process so you don't need to worry about that `Column` rebuild for example. The rebuild system is discussed in detail in *chapter 10 – Section 3.1 “Rebuilds and tree updates”*.

### 12.2.1 Performance considerations

The `setState` call schedules a rebuild for the widget itself and its children (if any) on the next frame. Don't think this is bad in terms of performance because Flutter has been designed specifically to work in this way. Keep in mind that rebuilds also happen when you don't use `setState`. We have already seen the two techniques to save widgets from being rebuilt:

- create widgets using `const` constructors;

- create widgets inside the state class and reuse the variable inside the `build` method.

Even without caching, the child of a widget might not be rebuilt if you call `setState`. Look at this example:

```
void main() {
  runApp(const MaterialApp(
    home: WidgetA(
      child: WidgetB(),
    ),
  ));
}

class WidgetA extends StatefulWidget {
  final Widget child;
  const WidgetA({super.key, required this.child});

  @override
  State<WidgetA> createState() => _WidgetAState();
}

class _WidgetAState extends State<WidgetA> {
  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        ElevatedButton(
          onPressed: () => setState(() {}),
          child: const Text('Press me'),
        ),
        widget.child,
      ],
    );
  }
}

class WidgetB extends StatelessWidget {
  const WidgetB({super.key});

  @override
  Widget build(BuildContext context) {
    debugPrint('WidgetB rebuild'); // <-- called only once!
    return Container();
  }
}
```

Even if the button was pressed hundreds of times, the child (`WidgetB`) wouldn't be rebuilt. If you ran the example, you'd see that the "*WidgetB rebuild*" message is not printed to the console each time `setState` is called. Flutter is able to detect that the child is still the same across frames and so only `WidgetA` will be rebuilt. Since the `Text` inside the button is `const`, it also doesn't get rebuilt.

This example shows that `setState` doesn't always rebuild the entire subtree, nor it's a bad practice to use it. The real point is that it's not always the best solution for all the possible state management use cases. We're now going to introduce `InheritedWidget` and listenable classes, which are used for "application" state management (as an alternative to `setState`).

## 12.3 Using InheritedWidget

The `InheritedWidget` abstract class is designed to be subclassed to create widgets that efficiently propagate information down in the widget tree. Imagine you had an object to share with multiple widgets in the subtree like this:

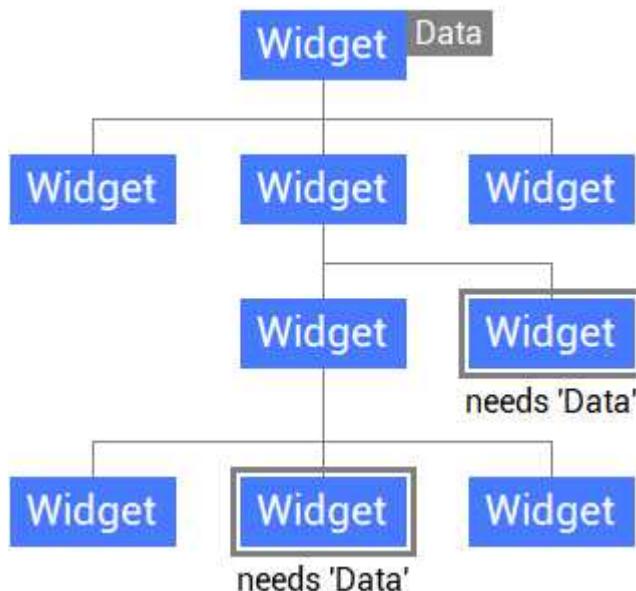


Figure 12.1: A widget holding data that needs to be shared down in the subtree.

From *Figure 12.1*, we see that two widgets across the tree need the gray piece of data and probably more if, in the future, the codebase evolves. The most straightforward approach would be to pass the data via constructor and then in the `build` method, until we arrive to the target widgets. For example, look at this simplified version of the tree:

```

class WidgetWithData extends StatelessWidget {
  final String value;
  const WidgetWithData({super.key, required this.value});

  @override
  Widget build(BuildContext context) {
    return WidgetA(
      data: Data(value), // This is where 'data' is defined
    );
  }
}

class WidgetA extends StatelessWidget {
  final Data data;
  const WidgetA({super.key, required this.data});

  @override
  Widget build(BuildContext context) => WidgetB(data: data);
}

class WidgetB extends StatelessWidget {
  final Data data;
  const WidgetB({super.key, required this.data});

  @override
  Widget build(BuildContext context) => WidgetC(data: data);
}

class WidgetC extends StatelessWidget {
  final Data data; // This is where 'data' is needed
  const WidgetC({super.key, required this.data});

  @override
  Widget build(BuildContext context) => UseData(data: data);
}

```

The only widget that needs the data is `WidgetC`, but we need to pass through `WidgetB` and `WidgetA` to arrive there. The problem is that we are adding to `WidgetB` and `WidgetA` a dependency that they don't use. In addition, we're not able to use constant constructors because of the data dependency, which is not a `const` object.

As you may have guessed, this is not the best way to propagate data in the widget tree. The correct solution is to use `InheritedWidgets` to get a direct reference to the object up in the tree without having to pass information via constructors. In other words, an `InheritedWidget` is a dependency injection tool that makes an object “available” for any of its children. Look at this image:

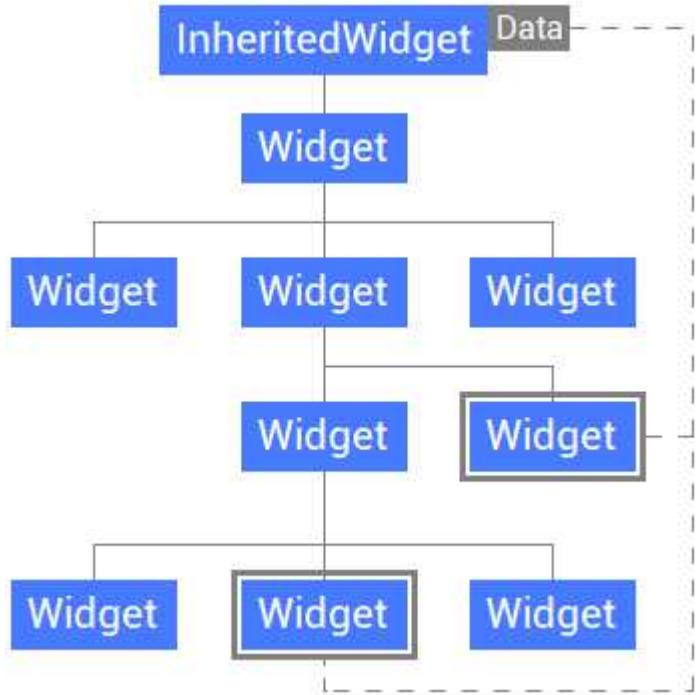


Figure 12.2: An inherited widget that shares data to all widgets in the subtree.

The data we need to share in the widget tree is held by an `InheritedWidget`, which is not a stateful or a stateless widget. It is a particular version of a widget that exposes information to its children using the `BuildContext` variable of the `build` method. This is how we can create it:

```

class InheritedData extends InheritedWidget {
  final Data data;
  const InheritedData({
    super.key,
    required this.data,
    required super.child,
  });

  static InheritedData of(BuildContext context) {
    return context.dependOnInheritedWidgetOfExactType<InheritedData>()!;
  }

  @override
  bool updateShouldNotify(covariant InheritedData oldWidget) {
    return data != oldWidget.data; // Include here all dependencies
  }
}
  
```

The convention is to provide a static method called `of` which is used by all children widgets (below `InheritedWidget`) to get a reference to the data. We can remove the `data` parameter from the constructors and rely on the `BuildContext` to retrieve the object from above the tree:

```
class WidgetWithData extends StatelessWidget {
  final String value;
  const WidgetWithData({super.key, required this.value});

  @override
  Widget build(BuildContext context) {
    // Makes 'data' available for all widgets in the subtree
    return InheritedData(
      data: Data(value),
      child: const WidgetA(),
    );
  }
}

class WidgetA extends StatelessWidget {
  const WidgetA({super.key}); // No longer needs 'data'

  @override
  Widget build(BuildContext context) => const WidgetB();
}

class WidgetB extends StatelessWidget {
  const WidgetB({super.key}); // No longer needs 'data'

  @override
  Widget build(BuildContext context) => const WidgetC();
}

class WidgetC extends StatelessWidget {
  const WidgetC({super.key}); // No longer needs 'data'

  @override
  Widget build(BuildContext context) {
    // Retrieves the 'data' object from the 'InheritedWidget' up in the tree
    return UseData(
      data: InheritedData.of(context).data,
    );
  }
}
```

Notice that `WidgetA` and `WidgetB` don't need to ask for `data` via constructor anymore. They can now be declared with a `const` constructor. The `of(context)` method looks up in the widget tree

for the closest `InheritedData` instance and returns a reference. The diagram in *Figure 12.2* shows what is happening: any widget below `InheritedData` can access the `data` object using the `context` variable. From this example, we can understand that:

1. `InheritedWidget` injects information into the widget tree and makes it available for the entire subtree. In our example, all children of `WidgetWithData` can get a reference to the `InheritedData` widget located above the tree and access `data`.
2. Flutter's `BuildContext` is a beautiful dependency injection system that passes information down the widget tree. All of this is possible because `context` actually is a representation of the `Element` behind the widget, which holds parent/child relationships. As such, Flutter can quickly pass information up and down.
3. If you put two or more inherited widgets of the same type, no runtime errors would happen. In this case, Flutter would just use the first match it finds while traversing the tree upwards. As such, try to only have a single inherited widget of a certain type for a given subtree.

When using an inherited widget, you need to ensure it's located at least one level above the first widget that needs it. As such, this code is fine because `InheritedData` is at least one level above the first widget that relies on it (`WidgetC`):

```
runApp(  
  const MaterialApp(  
    home: InheritedData( // <-- OK! One Level above 'WidgetC'  
      data: Data(),  
      child: WidgetC(),  
    ),  
  ),  
);
```

We could have also moved `InheritedData` above `MaterialApp` and it would have still been fine. In fact, in this example the inherited widget is two levels above the widget that consumes it:

```
runApp(  
  const InheritedData(  
    data: Data(),  
    child: MaterialApp(  
      home: WidgetC(),  
    ),  
  ),  
);
```

If you forget to place the inherited widget above the first dependency that needs it, you will get a runtime error because Flutter will traverse the entire tree (up to the root) without finding the instance. The `dependOnInheritedWidgetOfExactType` is used to look for the given instance up in the widget tree. For example:

```
static InheritedData of(BuildContext context) {
  final ref = context.dependOnInheritedWidgetOfExactType<InheritedData>();
  assert(ref != null, "No 'InheritedData' found above in the tree.");
  return ref!;
}
```

This aptly named method (and maybe too long) internally uses a `Map` to retrieve the closest instance of the given type up in the tree. If it's found, the inherited widget object is returned and, in case of changes, the `didChangeDependency` override on the state is triggered. For example, consider this case:

```
class Example extends StatefulWidget {
  const Example({super.key});

  @override
  State<Example> createState() => _ExampleState();
}

class _ExampleState extends State<Example> {
  var data = Data();

  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        ElevatedButton(
          onPressed: () => setState(() {
            data = Data();
          }),
          child: const Text('Rebuild'),
        ),
        InheritedData(
          data: data,
          child: const SubExample(),
        ),
      ],
    );
  }
}
```

When the button is pressed, `data` is given a new object and `InheritedData` gets rebuilt. In other words, the data exposed by the inherited widget changes each time `setState` is called. All children that call `dependOnInheritedWidgetOfExactType` are notified when the inherited widget is rebuilt with a different configuration (in this case, a new `Data` object). For example:

```
class SubExample extends StatefulWidget {
  const SubExample({super.key});

  @override
  State<SubExample> createState() => _SubExampleState();
}

class _SubExampleState extends State<SubExample> {
  @override
  void didChangeDependencies() {
    super.didChangeDependencies();
    debugPrint('The InheritedWidget changed!');
  }

  @override
  Widget build(BuildContext context) {
    final data = InheritedData.of(context).data;
    return Container();
  }
}
```

When `InheritedData` is rebuilt with a different configuration, `didChangeDependencies` is called (so you'll always see "*The InheritedWidget changed!*" printed on the console) and the widget itself is also rebuilt. This approach isn't widely used because listenable objects are often a better option, as we will see in the next section. You can control how an inherited widget rebuild notifies listening widgets. For example:

```
bool updateShouldNotify(covariant InheritedData oldWidget) => false
```

In this case, even if the inherited widget is rebuilt with a different configuration, it will not notify its listeners. As such, the `didChangeDependencies` method won't be invoked. This kind of override is only good when the inherited widget has no external dependencies. Otherwise, you should check them all like this:

```
bool updateShouldNotify(covariant MyInheritedWidget oldWidget) =>
  oldWidget.dependency1 != dependency1;
  oldWidget.dependency2 != dependency2;
```

### 12.3.1 App state management using listeners

We have already seen that “application state” refers to the state that it shared across multiple widgets. Since an `InheritedWidget` exposes (or “shares”) an object across multiple widgets down in the tree, we can use it to manage the application state. The idea is to expose a listenable object (which holds the state) with an inherited widget and then rebuild only those widgets that depend on the state. Let’s create this simple counter application:

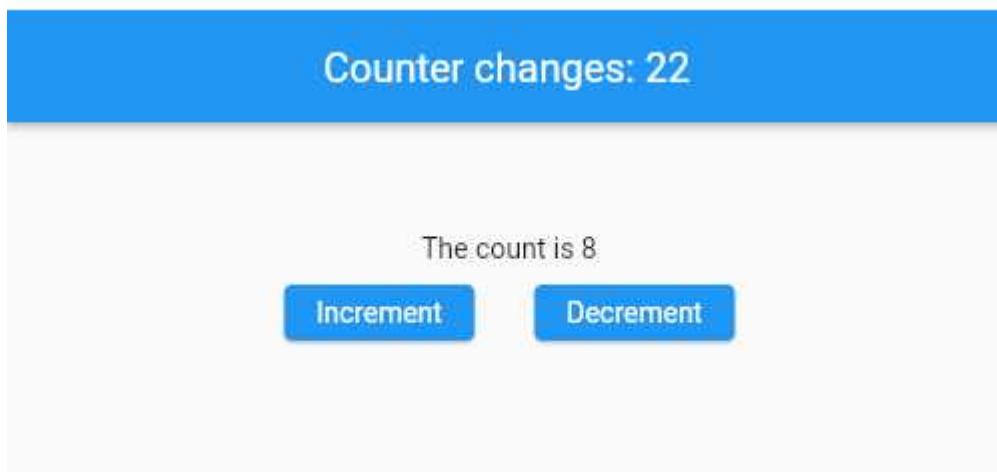


Figure 12.3: A counter app whose state is managed by an inherited widget and a listenable object.

The `Scaffold` at the top keeps track of how many times the two buttons have been pressed, while the text at the center shows the difference between increments and decrements. Let’s start by creating the listenable class that holds the counter state:

```
// 'ChangeNotifier' is a 'mixin class' and you can either extend or mix it
class CounterState extends ChangeNotifier {
    int _increment = 0;
    int _decrement = 0;
    int get incrementsCount => _increment;
    int get decrementsCount => _decrement;

    void increase() {
        _increment++;
        notifyListeners();
    }
    void decrease() {
        _decrement++;
        notifyListeners();
    }
}
```

The `Listenable` class in Flutter is used to maintain a list of listeners and notify all of them when needed. A `ChangeNotifier`, which is a `Listenable` subclass, is widely used in the framework to notify widgets about changes. Look closely at how we have implemented the API:

```
int get incrementsCount => _increment;

void increase() {
  _increment++;
  notifyListeners();
}
```

The `_increment` variable counts how many times we have performed a “+1” in the counter. The `notifyListeners()` method from `ChangeNotifier` is essential because it notifies listeners that the state has changed (and widgets should be rebuilt to the latest state).

Value reads don’t need to be listened to (hence the simple getter), but whenever the counter value changes, all listening widgets must rebuild (since the state changed). The inherited widget exposes this class in the usual way:

```
class InheritedCounter extends InheritedWidget {
  final CounterState counterState;
  const InheritedCounter({
    super.key,
    required this.counterState,
    required super.child,
  });

  static InheritedCounter of(BuildContext context) {
    final iw = context.dependOnInheritedWidgetOfExactType<InheritedCounter>();
    // The assertion is nice-to-have but not essential
    assert(iw != null, "No 'InheritedCounter' found above in the tree.");
    return iw!;
  }

  @override
  bool updateShouldNotify(covariant InheritedCounter oldWidget) {
    return counterState != oldWidget.counterState;
  }
}
```

We have added an assertion in the `of` method for debugging purposes (which is also what Flutter often does internally). The purpose of `InheritedCounter` is to expose the state to all widgets in the subtree, but only the interested ones will listen for changes. In fact, this is what the `Scaffold` looks like:

```

void main() {
  runApp(
    MaterialApp(
      home: InheritedCounter(
        counterState: CounterState(),
        child: const CounterApp(),
      ),
    ),
  );
}

class CounterApp extends StatelessWidget {
  const CounterApp({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const AppTitle(),
      ),
      body: const AppBody(),
    );
  }
}

```

The `Scaffold` and the `AppBar` do not depend on the state and so they do not listen for changes. In other words, they will not get rebuilt when the state changes. The title of the `AppBar` instead cares about the state and so we have created a dedicated widget (`AppTitle`) to listen for changes:

```

class AppTitle extends StatelessWidget {
  const AppTitle({super.key});

  @override
  Widget build(BuildContext context) {
    final state = InheritedCounter.of(context).counterState;

    // Rebuilds its children whenever 'CounterState' uses 'notifyListeners()'
    return ListenableBuilder(
      listenable: state,
      builder: (context, _) {
        final totalChanges = state.incrementsCount + state.decrementsCount;
        return Text('Counter changes: $totalChanges');
      },
    );
  }
}

```

A `ListenableBuilder` widget listens for changes on `Listenable` objects, like `ChangeNotifier`. In particular, it is used to only rebuild certain parts of the tree (leaving others untouched). In our case, `CounterState` is the source that “sends” notifications whenever the state changes and `ListenableBuilder` is the “receiver” that listens for updates and rebuilds its children accordingly. This is a simplified view of the widget tree of our counter application:

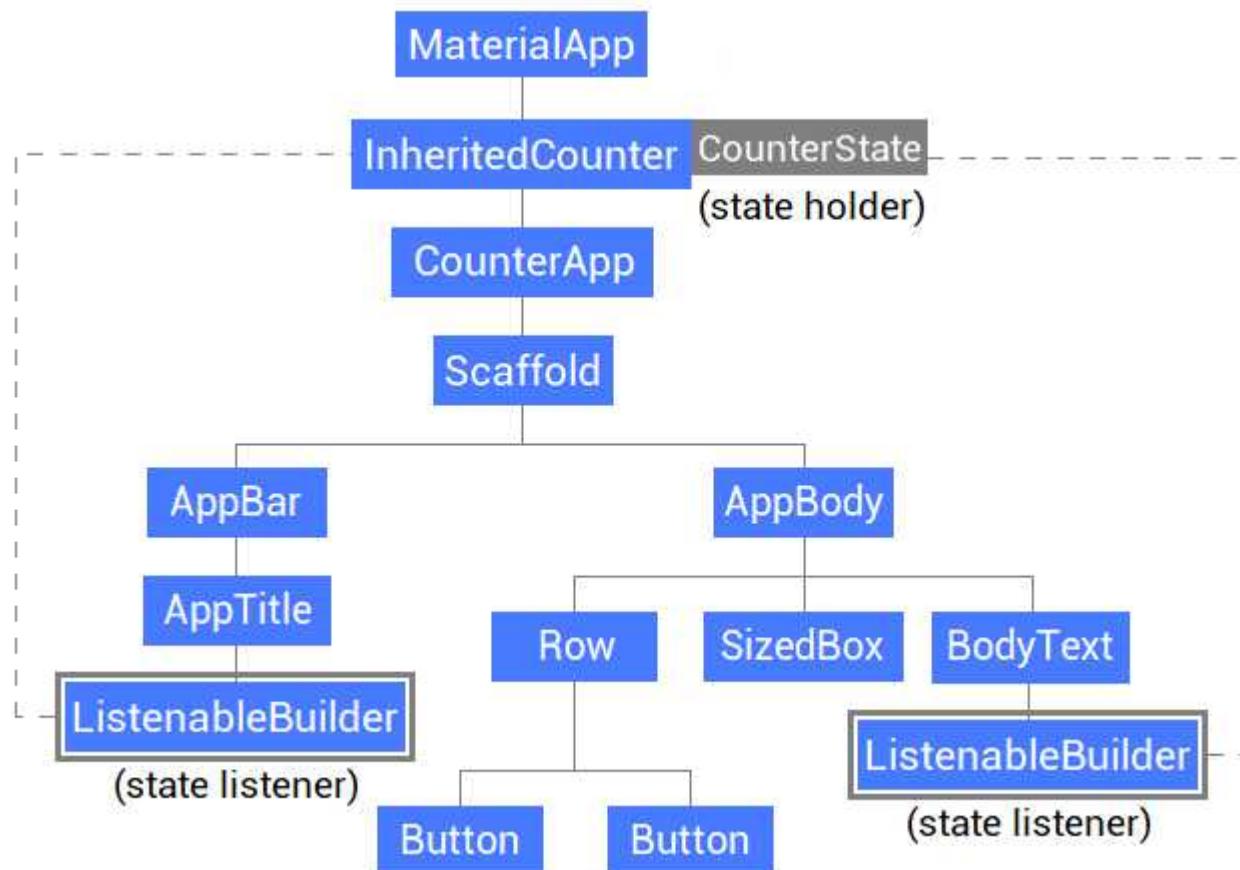


Figure 12.4: A simplified view of the app’s widget tree.

This image shows that `ListenableBuilder` uses the `InheritedCounter` inherited widget to get a reference to `CounterState` and rebuilds its subtree whenever `notifyListeners` is called. In this way, only the children of the builder will be rebuilt. All the other widgets aren’t listening will not be rebuilt.

For example, the `AppBody` widget doesn’t care about the counter state. As such, it will not be rebuilt when `CounterState` calls `notifyListeners` because there are no `ListenableBuilders` that are listening for changes:

```

class AppBody extends StatelessWidget {
  const AppBody({super.key});

  @override
  Widget build(BuildContext context) {
    final state = InheritedCounter.of(context).counterState;

    return Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.min,
        children: [
          // The counter text at the center of the screen
          const BodyText(),

          // Some spacing
          const SizedBox(height: 10),

          // The two tappable buttons
          Row(
            mainAxisAlignment: MainAxisAlignment.center,
            children: [
              // increases the counter and calls 'notifyListeners'
              ElevatedButton(
                onPressed: state.increase,
                child: const Text('Increment'),
              ),
              const SizedBox(width: 30),

              // decreases the counter and calls 'notifyListeners'
              ElevatedButton(
                onPressed: state.decrease,
                child: const Text('Decrement'),
              ),
            ],
          ),
        ],
      );
  }
}

```

As you can see, there are no `ListenableBuilders` because this part of the widget tree does not need updates in response to state changes. However, the two buttons need to change the counter state by increasing or decreasing it, so we still need to get a reference to `CounterState`. We have wrapped the mutable part (the one that depends on the state) in the `BodyText` widget:

```

class BodyText extends StatelessWidget {
  const BodyText({super.key});

  @override
  Widget build(BuildContext context) {
    final state = InheritedCounter.of(context).counterState;

    return ListenableBuilder(
      listenable: state,
      builder: (context, _) {
        final diff = state.incrementsCount - state.decrementsCount;
        return Text('The count is ${diff}');
      },
    );
  }
}

```

This is the exact same thing we've already done for the app bar text. The part that depends on the state is put in a `const` widget (`BodyText`) to return a listener (`ListenableBuilder`) that rebuilds the subtree whenever `notifyListeners` is called. To sum up, this is what we've done to manage the application state of the counter:

1. We have created a listenable class (`CounterState`) that extends `ChangeNotifier` to hold the state. In this way, listeners can be notified whenever the state changes.
2. We have created an inherited widget (`InheritedCounter`) to expose the listenable class so that any widget in the subtree can listen for state changes.
3. All widgets that need to be rebuilt in response to state changes are wrapped in a constant, stateless widget that returns a `ListenableBuilder`. The builder takes a reference to the listenable (using the inherited widget) and rebuilds its children whenever `notifyListeners` is called.

Take these as the general steps required to manage the application state efficiently. Let's now see some performance considerations for this state management approach.

### 12.3.2 Performance considerations

This state management approach allows you to selectively rebuild only those parts of the widget tree that need to be updated, which is very efficient. Getting a reference to an inherited widget up in the tree is an `O(1)` (constant) operation. Whenever the state changes, the `builder` function of

the `ListenableBuilder` is the only part of the tree that is rebuilt. You can even cache parts of the builder itself using the `child` parameter. Look at this example:

```
class Example extends StatelessWidget {
  final int value;
  const Example({
    super.key,
    required this.value,
  });

  @override
  Widget build(BuildContext context) {
    final state = InheritedState.of(context).myState;

    return ListenableBuilder(
      listenable: state,
      builder: (context, _) {
        return Container(
          color: state.color,
          child: OtherWidget(
            value: value,
          ),
        );
      },
    );
  }
}
```

The `Container` depends on the state to change its background color. `OtherWidget` instead does not depend on the state, but we cannot make it `const` because of the `value` dependency. When a parent widget depends on the state but the child doesn't, we can use the `child` parameter to avoid unnecessary rebuilds. For example:

```
return ListenableBuilder(
  listenable: state,
  builder: (context, child) { // <-- this 'child' references 'OtherWidget'
    return Container(
      color: state.color,
      child: child!,
    );
  },
  child: OtherWidget( // this 'child' is passed to the builder above
    value: value,
  ),
);
```

The `child` parameter is passed to the `builder` function and the widget it holds (`OtherWidget`, in our case) does not get rebuilt. The `builder` function is the only part of the `ListenableBuilder` that is rebuilt when the listenable class uses `notifyListeners`. Since `child` is passed from outside the build function, the reference to the widget does not change across frames and so Flutter does not rebuild it. This is what the inline documentation of `ListenableBuilder` says:

### Note

If your `builder` function contains a subtree that does not depend on the listenable, it's more efficient to build that subtree once instead of rebuilding it on every change of the listenable.

If a pre-built subtree is passed as the `child` parameter, the `ListenableBuilder` will pass it back to the builder function so that it can be incorporated into `build`.

Using this pre-built child is entirely optional, but can improve performance significantly in some cases and is therefore a good practice.

We recommend wrapping `ListenableBuilders` in dedicated constant widgets and using the `child` parameter whenever possible. This is the most you can do to minimize widgets being rebuilt on state changes. The downside of this approach is that you have to write a lot of code. To sum up, application state management is mainly split into three different categories:

1. Creation, test and management of a listenable class, which often is a `ChangeNotifier` (we will see another kind of listenable in the next section).
2. Creation, test and management of an inherited widget. It must always be placed at a proper location in the widget tree to avoid runtime errors.
3. Creation of `ListenableBuilder` widgets that rebuild a piece of widget tree in response to changes, notified by in the listenable class.

You can surely use inherited widgets and builders for ephemeral state management too, but that'd be too much code for something that `setState` greatly does (with less code and less maintenance).

Using `ListenableBuilder` and caching parts that don't change with `child` is not so different from creating a dedicated stateful widget, using `setState` and using `const` constructors (or manually caching widgets in the state class).

At the same time, you could use `setState` for application state management and achieve the same performance benefits as the inherited widget approach. However, you would need to pass the state via constructors (which disallows `const` constructors on widgets) and make a lot of manual caching. This quickly becomes very hard to maintain because you would have to manually cache many parts of the tree and. The key point is:

- Both approaches (`setState` and `InheritedWidget + ChangeNotifier`) are totally fine and could be used for all kinds of state management.
- When it comes to ephemeral state management, `setState` is easier to test and maintain.
- When it comes to application state management, inherited widgets and listenable classes are easier to test and maintain.
- Take our suggestions with a grain of salt because there is no clear distinction that applies to all use cases. We aren't giving a *good practice* here but more of a *general guideline* on how to approach state management. Always consider your specific use case and evaluate which option suits better!
- Don't create a single `ChangeNotifier` that holds all of your application state! For example, you may want to create different notifier classes for authentication status, theme settings, cart items, user's preferences and so on.

Before moving on, we want to share a little improvement you can make to avoid explicitly calling an inherited widget every time. Rather than always creating a final variable in the state just to take a reference of the inherited widget...

```
Widget build(BuildContext context) {
  final state = InheritedCounter.of(context).counterState;

  return ListenableBuilder(
    listenable: state,
    builder: (context, child) {
      // use the 'state' variable here ...
    },
  );
}
```

... you could create an extension on `BuildContext` to reduce the code to write and make it a little clearer (in our opinion) that you're accessing the context to look for the reference:

```
extension BuildContextExt on BuildContext {  
    CounterState get counterState => InheritedCounter.of(this).counterState;  
}
```

Now you can access the state with less code, using a getter on the `BuildContext` itself. This is an “aesthetic” change to avoid repeating `InheritedCounter.of` on each widget:

```
Widget build(BuildContext context) {  
    return ListenableBuilder(  
        listenable: context.counterState, // <-- this!  
        builder: (context, _) {  
            // use 'context.counterState' here ...  
        },  
    );  
}
```

There are no performance benefits from using this extension method.

### 12.3.3 ValueNotifier and ValueListenableBuilder

The combination of an `InheritedWidget`, a `ChangeNotifier`, and a `ListenableBuilder` is the most generic solution for application state management. The main problem with this approach is the high amount of code to write to expose a single object to a subtree. For example, imagine you had to hold the theme configuration for your application:

```
class ThemeState extends ChangeNotifier {  
    ThemeData _themeData = ThemeData.light();  
  
    ThemeData get themeData => _themeData;  
  
    set themeData(ThemeData themeData) {  
        if (themeData != _themeData) {  
            _themeData = themeData;  
            notifyListeners();  
        }  
    }  
}
```

This is the standard way of creating a listenable object. Imagine having a lot of these listenable classes that hold a single value. You would basically need to copy/paste the code and just change the variable type. For these use cases, the framework has the `ValueNotifier<T>` class. It has the following signature:

```
/// A [ChangeNotifier] that holds a single value.  
class ValueNotifier<T> extends ChangeNotifier implements ValueListenable<T>
```

A `ValueNotifier<T>` is nothing more than a `ChangeNotifier` that holds a single value. Instead of creating your own listenable classes that hold a single value, use a `ValueNotifier<T>` and expose it as usual with an inherited widget:

```
class InheritedTheme extends InheritedWidget {  
    final ValueNotifier<ThemeData> themeState; // <-- 'ValueNotifier' here  
    const InheritedTheme({  
        super.key,  
        required this.themeState,  
        required super.child,  
    });  
  
    @override  
    bool updateShouldNotify(covariant InheritedTheme oldWidget) {  
        return themeState != oldWidget.themeState;  
    }  
}
```

There are no performance concerns because a `ValueNotifier` is a `ChangeNotifier` that internally manages a single object for you. To consume a `ValueNotifier` in a stateful or stateless widget, use the dedicated `ValueListenableBuilder<T>` instead of a `ListenableBuilder`:

```
class MyApp extends StatelessWidget {  
    const MyApp({super.key});  
  
    @override  
    Widget build(BuildContext context) {  
        return ValueListenableBuilder<ThemeData>(  
            valueListenable: InheritedTheme.of(context).themeState,  
            builder: (context, value, child) {  
                return MaterialApp(  
                    theme: value,  
                    home: child!,  
                );  
            },  
            child: const Scaffold(  
                body: AppBody(),  
            ),  
        );  
    }  
}
```

Of course, there must be an `InheritedTheme` object somewhere above the `MyApp` widget. It's all very similar to the usual with the little exceptions that:

1. Rather than creating your own notifier class, you use the already existing `ValueNotifier`.

2. Rather than using `ListenableBuilder`, we've used `ValueListenableBuilder` which has the same usage and purpose. The `child` parameter is used to cache the subtree that doesn't depend on the state.

When you have a single value, prefer `ValueNotifier` because it makes you write less code. When you have more than a single value, go for a `ChangeNotifier` and create your own API.

#### 12.3.4 Alternatives

Flutter's built-in state management system is meant to work with `setState`, `InheritedWidgets` and listenable classes. You could also use streams rather than listenable classes but in the end, the logic would still be the same. You can find lots of alternatives to inherited widgets and listenable classes at [pub.dev](https://pub.dev), which you may like or not. The official Flutter website maintains a page<sup>83</sup> with a series of state management packages created by the community you many want to try.

### Deep dive: State restoration

Mobile operating systems like iOS and Android give users the possibility to put their applications in the background and come back to any of them later. It may seem that backgrounded applications continue to execute forever and when re-opened they (should) still have the same state the user left them in.

#### Note

The Flutter state restoration API has been designed for mobile devices and browsers. It does not affect desktop platforms.

Since modern mobile phones have limited memory, to allow launching multiple applications, the operating system may need to kill a background one to free space. Before that happens, the OS gives a chance to the application being killed to “serialize” its state and restore when it will be re-opened. Let's make an example to understand better how state restoration works:

1. You have you Flutter application opened in a mobile device. At a certain point, you decide to put it in the background (along with other applications).

---

<sup>83</sup> <https://docs.flutter.dev/development/data-and-backend/state-mgmt/options>

2. When you open another application, there might not be enough free space. As such, the OS might decide to kill your Flutter application but keep it in the “backgrounded applications” list (state restoration is not enabled by default in Flutter).
3. The operating system gives your Flutter application (right before being killed) the chance to save its state information. This is where the restoration API kicks-in because it serializes various state information for a later use.
4. When you’ll come back to your application, before loading it, the operating system passes back the state information it saved earlier. In this way, the user can see the application in the state where it was left giving the illusion that it was always running in the background.

To see how state restoration works in Flutter, let’s consider this simple counter application:

```
void main() {
  runApp(
    const MaterialApp(
      home: Scaffold(
        body: Center(
          child: CounterApp(),
        ),
      ),
    ),
  );
}

class CounterApp extends StatefulWidget {
  const CounterApp({super.key});

  @override
  State<CounterApp> createState() => _CounterAppState();
}

class _CounterAppState extends State<CounterApp> {
  var counter = 0;

  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: () => setState(() => counter++),
      child: Text('$counter'),
    );
  }
}
```

As it stands, if we backgrounded this application on a mobile device and the operating system killed it, the current state would be lost. Thanks to `RestorationMixin` and `RestorableProperty`, we can take advantage of state restoration make sure the state survives in case the app was killed.

## Using RestorationMixin and restorable values



[https://github.com/albertodev01/flutter\\_book\\_examples/chapter\\_12/restoration/](https://github.com/albertodev01/flutter_book_examples/tree/master/chapter_12/restoration)

Flutter internally manages all the communications with the underlying platform, so you don't need to worry about it. To enable state restoration in the counter application example of the previous section, for example, we just need to pass a string to the `MaterialApp` or `CupertinoApp` widget:

```
const MaterialApp(  
  restorationScopeId: 'root', // Just set this value!  
  home: Scaffold(  
    body: Center(  
      child: CounterApp(),  
    ),  
  ),  
,
```

Providing a `restorationScopeId` creates the storage to serialize state information, but it doesn't also write data for us. We need to update the state of our `CounterApp` widget a bit:

```
class _CounterAppState extends State<CounterApp> with RestorationMixin {  
  final counter = RestorableInt(0);  
  
  @override  
  String? get restorationId => 'counter_page';  
  
  @override  
  void restoreState(RestorationBucket? oldBucket, bool initialRestore) {  
    registerForRestoration(counter, 'counter');  
  }  
  
  @override  
  Widget build(BuildContext context) {  
    return ElevatedButton(  
      onPressed: () => setState(() => counter.value++),  
      child: Text('${counter.value}'),  
    );  
  }  
}
```

Whenever we want to be able to restore the state of a widget, we need to mix the state with the `RestorationMixin` mixin and replace “primitive” values (such as numbers, booleans, or strings) with a restorable value. In particular

- We have changed the type of the counter variable from `int` to `RestorableInt`, a particular type that knows how to store and restore an integer value.
- The `restorationId` is internally required to correctly manage the widget restoration. Make sure to give it a unique value.
- The `restoreState` method is called right after `initState` and when new restoration data has been provided to the mixin. As such, this is the perfect place to register those values we want to be restored using `registerForRestoration`.

Thanks to `registerForRestoration`, our property can be saved every time it changes and it will also be restored whenever the application is killed while in the background. We recommend creating a dedicated file that contains all the restorations ids. For example:

```
/// Restoration id for [MaterialApp].  
const restorationScopeId = 'root';  
  
/// Restoration id for the [CounterApp] widget.  
const counterPageRestorationId = 'counter_page';  
  
/// Restoration id for the [RestorableInt] property.  
const counterRestorationId = 'counter';
```

To avoid problems, always make sure to create unique ids for your restorable values and widgets. There are a lot more types other than `RestorableInt`, such as:

- `RestorableBool`
- `RestorableDateTime`
- `RestorableDouble`
- `RestorableString`
- `RestorableTimeOfDay`
- `RestorableNum`

All of these classes are subtypes of `RestorableValue<T>`, which is also is a `ChangeNotifier` so you could listen for changes on it. You can even create your own restorable types if the framework doesn’t already have one. For example, the above list is missing a `RestorableDuration` type, which wraps a `Duration` object. You can create your own in this way:

```

class RestorableDuration extends RestorableValue<Duration> {
  @override
  Duration createDefaultValue() => Duration.zero;

  @override
  void didUpdateValue(Duration? oldValue) {
    if (oldValue?.inMicroseconds != value.inMicroseconds) {
      notifyListeners();
    }
  }

  @override
  Duration fromPrimitives(Object? data) {
    if (data != null) {
      return Duration(microseconds: data as int);
    }
    return Duration.zero;
  }

  @override
  Object toPrimitives() => value.inMicroseconds;
}

```

There is a `T? value` property in `RestorableValue` that holds the currently stored value. Whenever it changes, `didUpdateValue` is called, so in our implementation, we need to notify listeners. Note that some Flutter widgets have built-in restoration support, and so they don't need to be wrapped in a `RestorableValue<T>`. Some popular examples are:

- All scrollable widgets, which are covered in *chapter 14 – Section 3 “Scrollable widgets”*, have a `restorationId` property that remembers the current scroll position. For example:

```

ListView(
  restorationId: 'listview-id',
  children: const [
    Text('Hello'),
    Text('Flutter'),
    Text('World'),
  ],
),

```

You can see `ListView` as a scrollable row or column. Thanks to the restoration id, when the backgrounded application is killed and then restarted, the scroll offset is restored. Other scrollable widgets (for example) are `GridView` or `SingleChildScrollView`.

- The `Scaffold` widget also allows setting a restoration id, which is useful (for example) to persist and restore whether the `Drawer`, if any, was opened or closed.
- User input widgets, which will be covered in *chapter 18 – Section 1 “Input widgets”*, have a `restorationId` property that is used to save and restore the form field validation state.

All of these widget-specific ids assume that you've provided a `restorationScopeId` value in your `MaterialApp` or `CupertinoApp` root widget.

## Considerations and debugging

When working with state restoration, the Flutter team recommends to only save primitive values or objects that don't take too much space<sup>84</sup>. The reason is that some platforms restrict the size of restoration data and may cause exception if the limit is exceeded.

### Note

On Android for example, there is a 1 MB size limit<sup>85</sup>. That's a lot of space for sure but you should still be aware that an exception might be thrown if you exceed that limit.

Data that can be retrieved from external services (such as a database) shouldn't be included in the restoration data. Flutter handles restoration in the following way:

- Restoration data is organized in a tree of `RestorationBucket` entries. To store new data, a Flutter widget claims the ownership of a new bucket from this tree and then it arbitrarily stores and reads data.
- Values are stored in the bucket using a restoration ID as a key (whose type is `String`). The ID must be unique and that's the reason why we recommended to always use unique values for IDs.
- Flutter communicates with Android and iOS using the `StandardMessageCodec` class, which serializes and deserializes state data that need to be stored and restored.

---

<sup>84</sup> <https://api.flutter.dev/flutter/services/RestorationManager-class.html>

<sup>85</sup> <https://developer.android.com/reference/android/os/TransactionTooLargeException>

To test state restoration, always fully re-compile your application after making a change. Hot reload and hot restart do not persist changes on the device because they're lost when the application is terminated or restarted. If you want to test state restoration on Android, follow these steps:

1. Go to the developer options section and enable the *Don't keep activities* flag. It's used to always destroy activities as soon as they're backgrounded.
2. Run the application on the device and ensure to change the state you want to be restored.
3. Background the application and then return to it. It will restart and restore its state.

On iOS instead it's a bit trickier:

1. Open `ios/Runner.xcworkspace` in Xcode.
2. For iOS 14+, switch to build in profile or release mode since launching an application from the home screen is not supported in debug mode.
3. Build and run the application from XCode and make sure to change the state you want to be restored.
4. Background the application on the phone (for example, by going back to the home screen), and press the stop button on XCode (to terminate the application while it's running in the background).
5. Open the application again on the phone, without using XCode, and you will see that the state will be restored.

Of course, you can make tests on virtual and physical devices. When it comes to widget testing, you can use the `restartAndRestore` method. It simulates restoring the state of the widget tree after the application is restarted. For example:

```
// Simulates state restoration
await tester.restartAndRestore();
```

The state restoration example in our GitHub repository uses `restartAndRestore` in a widget test to check that state restoration works correctly.

# 13 – Routes and navigation

---

## 13.1 Introduction

In the Flutter world, application's pages are called routes and thus the *route management* term is about the techniques you can use to handle multiple routes. Any widget can be used to represent a page but in general [Scaffold](#) or [CupertinoPageScaffold](#) are the most convenient and popular choices. For example:

```
class HomePage extends StatelessWidget {
  const HomePage({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('My app'),
        centerTitle: true,
      ),
      body: const HomePageContents(),
      floatingActionButton: FloatingActionButton(
        onPressed: () {},
        child: const Icon(Icons.add),
      ),
    );
  }
}
```

A scaffold is a pre-built widget that allows you to quickly setup a new route (or “page”) with a title, a body and some other common elements such as buttons or menus. Nevertheless, you can create your custom scaffolds from scratch to fully customize the page layout. When creating new routes, we encourage you to:

- Evaluate if you can use a [Scaffold](#) or a [CupertinoPageScaffold](#) instead of creating a new page structure layout from scratch. These two widgets are very flexible and customizable so you may find them easy to reuse.
- If you need to create a very custom implementation and you cannot easily reuse [Scaffold](#) or [CupertinoPageScaffold](#), then create your own scaffold. In this case, we recommend to append the “*Scaffold*” word at the end of the class name to make its purpose clear to other Flutter developers.

We recommend to place scaffolds inside widgets whose name ends with either “route” or “page”. For example, `SettingsRoute` or `SettingsPage` might be an intuitive name for a widget that has a scaffold to build the “settings” page of your application.

### 13.1 Imperative and declarative navigation

The framework has an imperative route management system, conventionally called `navigator 1.0`, and a declarative one, conventionally called `navigator 2.0`. Both are used to navigate from a route to another but they have substantial implementation and usages differences:

- The imperative system uses the `Navigator` widget while the declarative system uses the `Router` widget.
- For small applications, the `Navigator` API is typically enough. Larger applications that also need deep linking support, sub-routes and arguments parsing, the `Router` API is generally a better choice. Even if you can do everything with both approaches, there are tradeoffs to consider.

The declarative system is built on top of the imperative system, but one is absolutely not meant to be the replacement of the other. As such, (even if it’s a terrible idea) you could mix the declarative and the imperative systems in the same application.

#### Note

The `Navigator` API was part of Flutter since its early days hence it’s often referenced as `navigator 1.0`.

The `Router` API was created later to add a new declarative API that also facilitates some tasks, hence the `navigator 2.0` name.

The declarative and the imperative systems can both do everything. The `Router` API for example uses a `Navigator` under the hood. However, there are some cases where one has more advantages than the other.

To be clear: if you’re happy with the imperative approach you can continue using it. However, it might make some things hard to implement and maintain. The declarative system could make those things much easier to create but, at the same time, it could seem too complicated for simpler tasks. For example, navigating back and forth between routes is simple with both systems. However, for

a finer page stack control or deep linking management, the declarative [Router](#) API is easier to use and requires less code. The next sections cover in detail how the imperative and the declarative navigation systems work in Flutter.

## 13.2 Imperative navigation using Navigator

We will create a straightforward application with two routes to see how the imperative navigation system works. The first route is [HomePage](#), which is immediately shown when the application starts, and the second route is [RandomPage](#), which can be opened with a button tap and outputs a random number. For example:

```
class HomePage extends StatelessWidget {
  const HomePage({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: ElevatedButton(
          onPressed: () { /* Navigate to RandomPage */ },
          child: const Text('Open random page'),
        ),
      ),
    );
  }
}

class RandomPage extends StatelessWidget {
  static final _random = Random();
  const RandomPage({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: ElevatedButton(
          onPressed: () { /* Navigate back to HomePage */ },
          child: Text('${_random.nextInt(10)}'),
        ),
      ),
    );
  }
}
```

Even if `HomePage` and `RandomPage` are widgets, we call them “routes” (or “pages”) because they are used to represent a route in the application. We also want to create the `ErrorPage` route, whose purpose will be clear soon:

```
class ErrorPage extends StatelessWidget {
  const ErrorPage({super.key});

  @override
  Widget build(BuildContext context) {
    return const Scaffold(
      body: Center(
        child: Text('Error!'),
      ),
    );
  }
}
```

Now that all routes are created, we need to assign them a name so that the navigator can uniquely identify them. We strongly recommend creating a dedicated file, called `routes.dart` for example, to handle routes and names:

```
// 'routes.dart' file
const homeRouteName = '/';
const randomRouteName = '/random_page';

Route<Object?> generateRoutes(RouteSettings settings) {
  switch (settings.name) {
    case homeRouteName:
      return MaterialPageRoute<void>(
        builder: (_) => const HomePage(),
      );
    case randomRouteName:
      return MaterialPageRoute<void>(
        builder: (_) => const RandomPage(),
      );
    default:
      return MaterialPageRoute<void>(
        builder: (_) => const ErrorPage(),
      );
  }
}
```

Each route is associated with a path, which is formatted as if it was a relative URL, and the names must be unique. The `generateRoutes` method is passed to the `MaterialApp` widget to “install” the routes and enable navigation. For example:

```

class MyApp extends StatelessWidget {
  const MyApp();

  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      initialRoute: homeRouteName,
      onGenerateRoute: generateRoutes,
    );
  }
}

```

The `MaterialApp` widget internally manages a `Navigator` object and “installs” all routes using the `generateRoutes` function. To navigate to a new route, we need to use the navigator as if it was an inherited widget:

```

ElevatedButton(
  onPressed: () async=>await Navigator.of(context).pushNamed(randomRouteName),
  child: const Text('Open random page'),
),

```

Thanks to `pushNamed`, we can pass the name of the route we want to open, and the navigator will display it. This is the reason why we had to use `RouteSettings`:

```

Route<Object?> generateRoutes(RouteSettings settings) {
  switch (settings.name) {
    case homeRouteName: // case when 'pushNamed' passes '/'
      return MaterialPageRoute<void>(
        builder: (_) => const HomePage(),
      );
    case randomRouteName: // case when 'pushNamed' passes '/random_page'
      return MaterialPageRoute<void>(
        builder: (_) => const RandomPage(),
      );
    default: // case when 'pushNamed' passes an unknown route name
      return MaterialPageRoute<void>(
        builder: (_) => const ErrorPage(),
      );
  }
}

```

The `settings.name` property is the route name passed by `pushNamed`, used to determine which route will be displayed. We have also created an error page that is only displayed when you try to navigate to a non-existing route. For example, this call would show the `ErrorPage` route:

```
await Navigator.of(context).pushNamed('/non_existing_route_name');
```

A [Route](#) is an entry in the navigation stack managed by the [Navigator](#). As such, we have always wrapped our page widgets in a [MaterialPageRoute<T>](#) object. Alternatively, you can use another page route object to get different behaviors:

- [MaterialPageRoute<T>](#): replaces a route with another using platform-adaptive transitions. For example, on Android there's a zoom-and-fade animation while on iOS the page slides from a side to the other.
- [CupertinoPageRoute<T>](#): replaces a route with another using an iOS transition. As such, on all platforms, pages are replaced with a slide transition.
- [PageRouteBuilder<T>](#): replaces a route with another using a custom transition. This type is used when you need to implement a custom transition between pages. Animations will be discussed in detail in *chapter 17 – “Animations”*.

In *Section 2.3 – Passing data between routes* you will see why these classes define the [T](#) parameter. In short, [T](#) specifies the return type of the [pop\(\)](#) method when it's used to return data from a route.

### 13.2.1 Good practices

We strongly encourage the creation of a dedicated file that holds your entire route management configurations. All you need is the route names, stored as compile-time constants, and a method that generates routes. For example:

```
const firstRouteName = '/';
const secondRouteName = '/second';
const errorRouteName = '/error';

Route<Object?> generateRoutes(RouteSettings settings) {
  switch (settings.name) {
    case firstRouteName:
      return MaterialPageRoute<void>(builder: (_) => const FirstPage());
    case secondRouteName:
      return MaterialPageRoute<void>(builder: (_) => const SecondPage());

    // add more cases if needed...

    default:
      return MaterialPageRoute<void>(builder: (_) => const ErrorPage());
  }
}
```

This approach makes testing and maintenance easy because it centralizes the routing system in a single file. Alternatively, you could define all of your routes directly inside the `MaterialApp` widget itself:

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp( // This could have also been a 'CupertinoApp' widget
      initialRoute: '/',
      routes: {
        '/': (_) => const FirstPage(),
        '/second_page': (_) => const SecondPage(),
      },
      onUnknownRoute: (routeSettings) => MaterialPageRoute<void>(
        builder: (_) => const ErrorPage(),
      ),
    );
  }
}
```

This approach may be suitable for fast prototyping or demo purposes, but it doesn't scale well. It's not a matter of performance because it's the same as using the route generator function. The main concern is about maintainability and testing difficulties. If your application only had a single route, then you don't need to care about routes management at all:

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      home: HomePage(),
    );
  }
}
```

Just pass the only route the `home` parameter, and it will be assigned the '`/`' path by default. When using `home`, you cannot also define `initialRoute` otherwise a runtime exception will be thrown. To sum up, imperative navigation is done with the following steps:

1. Create all routes, preferably using a `Scaffold` or a custom scaffold-like widget that is reused across the entire application.
2. Define the routing system, preferably in a dedicated file that declares both route names (as top-level `const` strings) and a generator function.

3. Install the routes in a `MaterialApp` or `CupertinoApp` widget. Navigate to a new route with `pushNamed` and go back with `pop()`.

Let's move on to see how the navigation API actually works under the hood.

### 13.2.2 Navigating between routes

The `Navigator` class allows you to move from one route to another using `BuildContext`. The pages you navigate to are “overlapped” and the navigator internally keeps track of their position using a stack. For example, when you call `pushNamed(abc)` Flutter overlaps the destination page with the source one:

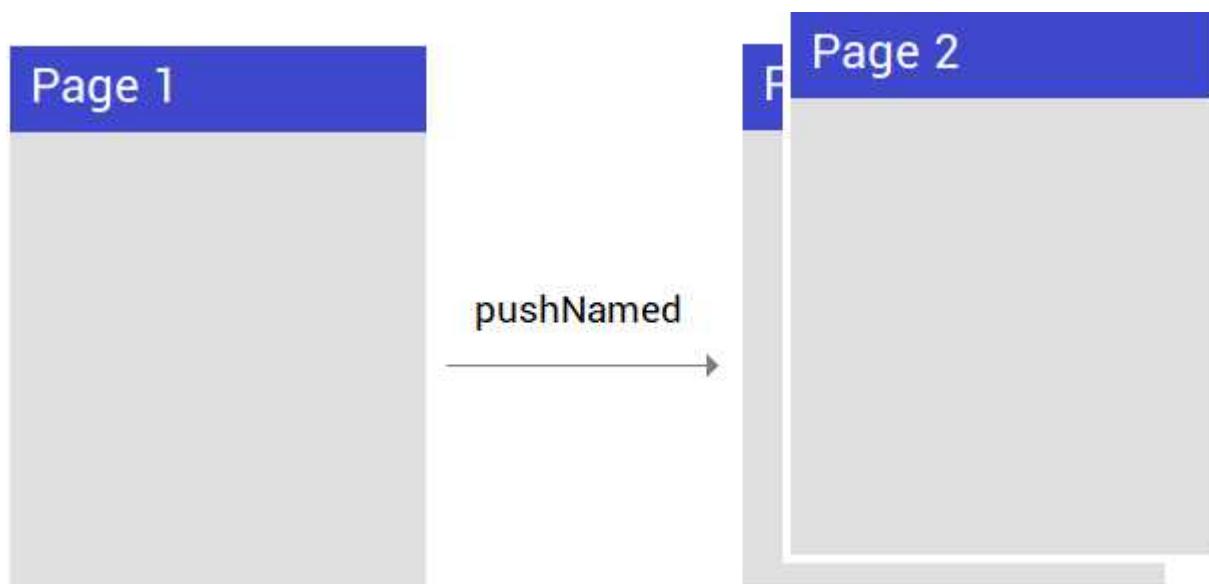


Figure 13.1: Routes being stacked when using `Navigator.of(context).pushNamed`

As you can see, the `Navigator.of(context).pushNamed` method adds a new route onto the stack and thus overlaps the new widget with the old one. You can use this equivalent syntax to move to a new page:

```
await Navigator.pushNamed(context, someRouteName),
```

If you want to come back from a page you've pushed, call `Navigator.of(context).pop()` which removes the topmost widget from the stack:

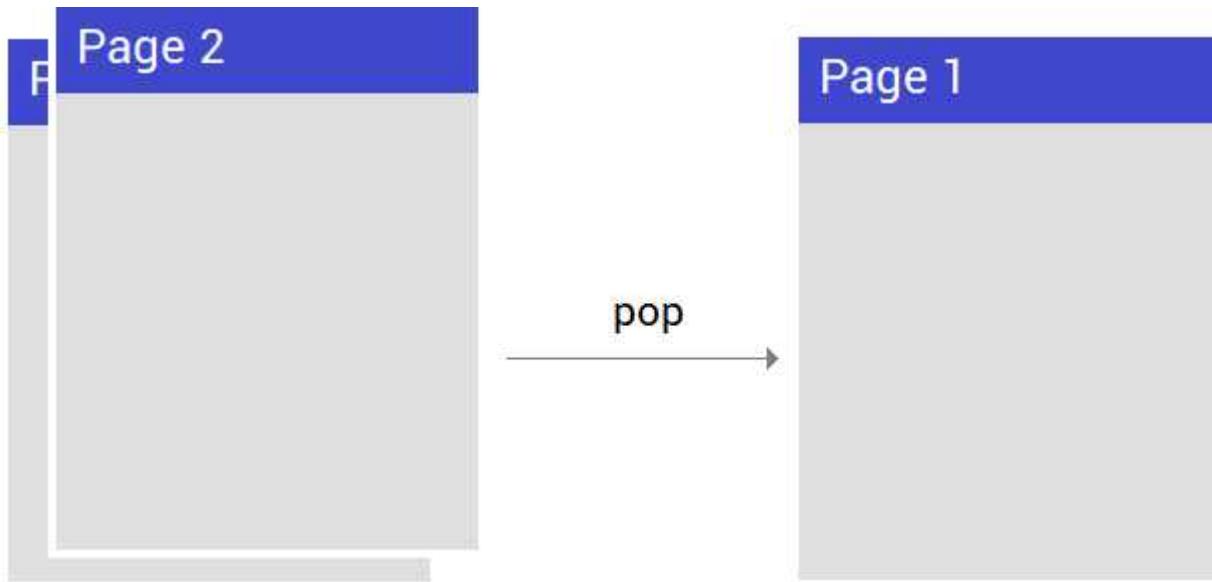


Figure 13.2: The topmost route being popped from the stack using `Navigator.of(context).pop()`.

If you wanted to come back to a certain route down in the stack, you'd need to repeatedly call `pop` multiple times. Note that in our previous example, we could have also pushed a route object directly into the stack without using the name. So yes, there are two ways to navigate to a new route (both are equivalent):

1. Use named routes, as we've already seen:

```
await Navigator.of(context).pushNamed(randomPageRoute);
```

2. Use anonymous routes with the push method:

```
await Navigator.of(context).push(
  MaterialPageRoute<void>(
    builder: (_) => const RandomPage(),
  ),
),
```

In the second approach, an anonymous route always requires you to wrap the widget with a `Route` object, which leads to code duplication. The first approach is better because it just asks for the route name (and the `Route` wrapper was already defined in the route generator function). The navigator API defines a few other interesting methods to handle the pages stack:

- `pushReplacement`: replaces the current route of the navigator with the new one. In other words, the topmost entry of the stack is disposed and replaced with the given one.

```
await Navigator.of(context).pushReplacement(
    MaterialPageRoute<void>(builder: (_) => const HomePage()),
);
```

- `pushReplacementNamed`: the same as `pushReplacement` with but it takes a string (the route name) rather than a route object.

```
await Navigator.of(context).pushReplacementNamed('/settings_page');
```

- `popAndPushNamed`: pops the current route off the navigator and pushes a named route in its place. This is the same as calling `pop` and `pushNamed` in sequence. This method is very similar to `pushReplacement` with the only difference that here `pop` is called.

```
await Navigator.of(context).popAndPushNamed('/settings_page');
```

- `popUntil`: repeatedly calls `pop` until the given predicate returns `true`. If the predicate never returned `true`, because you entered an invalid route name for example, you'd end up with an empty screen (because you would clear the entire page stack).

```
await Navigator.of(context).popUntil(ModalRoute.withName('/home_page'));
```

- `pushAndRemoveUntil`: pushes the given route in the navigator and pops all the previous ones until the predicate returns `true`. It's basically the same of `popUntil` with the only difference that it first pushes a new route.

```
await Navigator.of(context).pushAndRemoveUntil(
    MaterialPageRoute<void>(builder: (_) => const HomePage()),
    ModalRoute.withName('/'),
);
```

- `pushNamedAndRemoveUntil`: the same as `pushAndRemoveUntil` with the only difference that it takes a string (the route name).

```
await Navigator.of(context).pushNamedAndRemoveUntil(
    '/settings_page', ModalRoute.withName('/'),
),
```

### 13.2.3 Passing data between routes

The `Navigator` object gives the possibility to navigate to another route and pass data. For example, imagine you had a `TodoList` route with a list of to-dos and a `TodoItem` route to show the details of the selected to-do item. The simplest approach would first require a model class to hold the data:

```
final class Todo {  
    final String title;  
    final String contents;  
    const Todo(this.title, this.contents);  
}
```

Then, we could push an anonymous route and build the `TodoItem` widget by passing the to-do data via constructor. For example:

```
class _TodoListState extends State<TodoList> {  
    final todos = const [  
        Todo('Todo 1', 'Content 1'),  
        Todo('Todo 2', 'Content 2'),  
    ];  
  
    Future<void> _pushRoute(int index) async {  
        // Pushing an anonymous route  
        await Navigator.of(context).push(  
            MaterialPageRoute<void>(  
                builder: (_) => TodoItem(  
                    title: todos[index].title,  
                    content: todos[index].content,  
                ),  
            ),  
        );  
    }  
  
    @override  
    Widget build(BuildContext context) {  
        return ListView.builder(  
            itemCount: todos.length,  
            itemBuilder: (context, index) {  
                return ListTile(  
                    title: Text(todos[index].title),  
                    onTap: async () => _pushRoute(index),  
                );  
            },  
        );  
    }  
}
```

Even if this approach works, we don't really like it for two reasons:

1. Named routes are easier to maintain and also are compatible with deep linking. In this case, we must pass an anonymous route using `push` because the destination page takes two parameters via constructor.
2. You cannot create the `TodoItem` page with a `const` constructor because it has two external dependencies.

A better idea would be to use `pushNamed` and its `arguments` optional parameter to provide the data we want to pass to the other page. In this way, we can remove the constructor dependency from the `TodoItem` page and pass data using the context. This is what the new method looks like:

Old method	New method
<pre>void _pushRoute(int index) {     await Navigator.of(context).push(         MaterialPageRoute&lt;void&gt;(             builder: (_) =&gt; TodoItem(                 title: todos[index].title,                 content: todos[index].content,             ),         ),     ); }</pre>	<pre>void _pushRoute(int index) {     await Navigator.of(context).pushNamed(         todoItemRouteName,         arguments: todos[index]     ); }</pre>

The main difference is that we pass data with the `Navigator` rather than using `TodoItem` directly. We are now pushing a named route rather than an anonymous route, which also reduces code duplication. Thanks to `ModalRoute`, we can retrieve the data we've pushed to the destination route:

```
class TodoItem extends StatelessWidget {
    const TodoItem({super.key});

    @override
    Widget build(BuildContext context) {
        final todo = ModalRoute.of(context)!.settings.arguments! as Todo;

        return Scaffold(
            body: Text('${todo.title} - ${todo.contents}'),
        );
    }
}
```

The `Navigator` takes the object we've passed in the `arguments` parameter and makes it available to the destination route using the `ModalRoute` object. You can also pass data back from one page to the previous one. For example, if you wanted to return a string from a page you would need to pass a value to `pop`:

```
await Navigator.of(context).pop('Hello');
```

The string value is returned to the `push` (or `pushNamed`) method that previously pushed the route. In practice, to get the returned string of the example, you need to `await` the `push` or `pushNamed` method:

```
// 'push' returns a future that completes when 'pop' is called
final result = await Navigator.of(context).push(
    MaterialPageRoute<String>(
        builder: (_) => const SomePageItem(),
    ),
);
debugPrint('$result'); // Hello
```

Notice the type annotation in `MaterialPageRoute<String>`, which indicates the expected type returned by a `pop` call from the pushed page.

#### 13.2.3.1 Passing data using `InheritedWidget`



[https://github.com/albertodev01/flutter\\_book\\_examples/tree/chapter\\_13/13.2.3.1](https://github.com/albertodev01/flutter_book_examples/tree/chapter_13/13.2.3.1)

Using `arguments` and `ModalRoute` is surely an excellent way to exchange data between two pages, but it's not always the best choice. In some cases, multiple routes might be interested in receiving information.

#### Note

For example, you might have to push a sequence of routes for a step-by-step flow and gather all of the information at the end in a final route. Passing data each time might make the code hard to read and also become tedious to maintain.

*Lifting the state up* is a great idea when you need to pass data across multiple pages. In other words, when you need to pass data to various routes, it might be worth using an `InheritedWidget`. In our

to-do list example, if we had more routes or widgets interested in the selected item, we could lift the state up in a notifier class and expose it with an inherited widget. For example:

```
class TodoListState extends ChangeNotifier {
    final todos = List<Todo>.unmodifiable(const [
        Todo('Todo 1', 'Content 1'),
        Todo('Todo 2', 'Content 2'),
    ]);

    var _index = 0;
    int get index => _index;

    set index(int newValue) {
        if (_index != newValue) {
            _index = newValue;
            notifyListeners();
        }
    }
}
```

As you already know, exposing `TodoListState` with an inherited widget will make it available for an entire portion of the tree. Consequently, no more `ModalRoute` or route arguments are needed since we just need to rely on the inherited widget to get a reference to the data object. For example, in the `build` method we could use this code:

```
final todoState = InheritedTodoList.of(context).todoListState;

return Scaffold(
    body: ListenableBuilder(
        listenable: todoState,
        builder: (context, _) {
            return ListView.builder(
                itemCount: todoState.todos.length,
                itemBuilder: (context, index) {
                    return ListTile(
                        title: Text(todoState.todos[index].title),
                        onTap: () async {
                            todoState.index = index; // Update the state value
                            await Navigator.of(context).pushNamed(todoItemRouteName);
                        },
                    );
                },
            );
        },
    ),
);
```

The `TodoItem` page only needs a `ListenableBuilder` to listen to the notifier class exposed by the inherited widget. This approach is more complicated but more flexible when multiple widgets and routes need to access the data. Make sure to place the inherited widget above the `MaterialApp` (or `CupertinoApp`):

```
InheritedTodoList(  
    todoListState: TodoListState(), // before 'MaterialApp' or 'CupertinoApp'  
    child: MaterialApp(  
        initialRoute: todoListRouteName,  
        onGenerateRoute: generateRoutes,  
        home: const TodoList(),  
    ),  
,  
)
```

Since `Navigator` is internally handled by `MaterialApp` (or `CupertinoApp`), all pushed and popped pages are located below the app widget. As such, the inherited widget has to stay at least one level above the app widget.

### 13.2.4 Deep linking

In the web world, “deep linking” is the use of an URL that links to a specific piece of a website rather than the home page. When you compile a Flutter application for the web, you can activate deep linking support with a straightforward addition:

```
Route<Object?> generateRoutes(RouteSettings settings) {  
    switch (settings.name) {  
        case homeRouteName:  
            return MaterialPageRoute<void>(  
                settings: settings, // <- pass 'settings' here  
                builder: (_) => const HomePage(),  
            );  
        case randomRouteName:  
            return MaterialPageRoute<void>(  
                settings: settings, // <- pass 'settings' here  
                builder: (_) => const RandomPage(),  
            );  
        default:  
            return MaterialPageRoute<void>(  
                settings: settings, // <- pass 'settings' here  
                builder: (_) => const ErrorPage(),  
            );  
    }  
}
```

That's it! Passing `Navigator`'s `setting` to the returned material page enables deep linking for each route name. In other words, with this addition you can directly navigate to the random page using the route name in the URL:

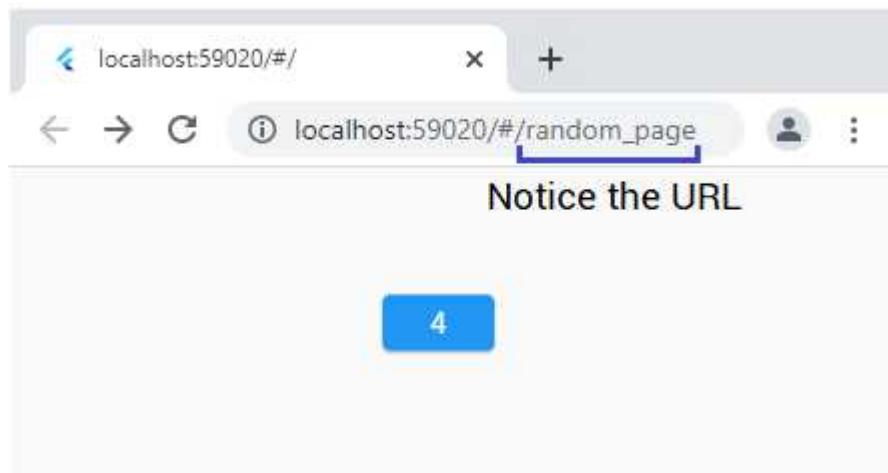


Figure 13.3: Using a deep link to directly navigate to the random numbers page.

If you didn't pass `settings` to `MaterialPageRoute`, deep linking wouldn't be enabled. In fact, the URL would always point to the home page even if you navigated to another route. If you tried to remove the `settings` parameter, you would notice that you can still change pages, but the URL doesn't update to reflect the actual path:

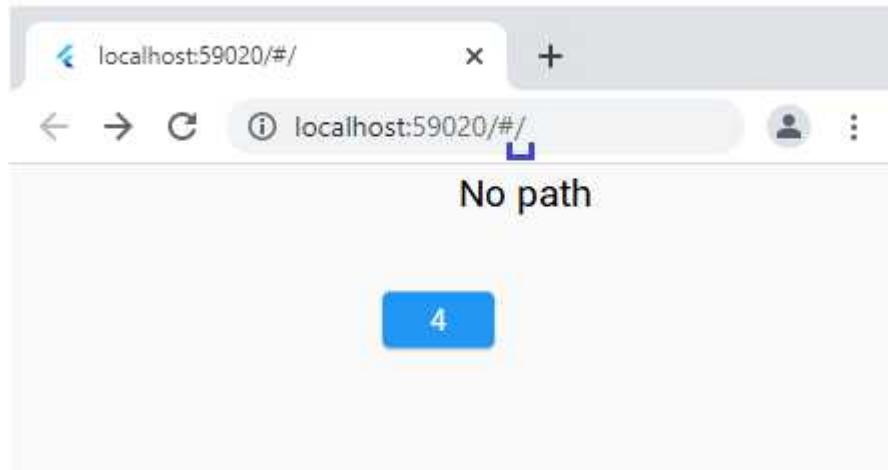


Figure 13.4: Without deep linking, the URL doesn't update to reflect the actual position.

Thanks to deep linking, we can inject data into the page using query parameters. With reference to *Figure 13.3*, we could for example pass an explicit value for the random number as such:

```
http://localhost:59020/#/random_page?value=2023
```

We can access query parameters and parse the value inside the route generator function thanks to the `RouteSettings` parameter. For example, this is a possible implementation:

```
Route<Object?> generateRoutes(RouteSettings settings) {
    // ['/random_page', 'value=2023']
    final List<String> parts = settings.name!.split('?');

    // {'value': 2023}
    final Map<String, String>? args =
        parts.length == 2 ? Uri.splitQueryString(parts[1]) : null;

    // 'parts.first' contains the route name
    switch (parts.first) {
        case homeRouteName:
            return MaterialPageRoute<void>(
                settings: settings,
                builder: (_) => const HomePage(),
            );
        case randomRouteName:
            return MaterialPageRoute<void>(
                settings: settings,
                builder: (_) {
                    return RandomPage(
                        initialValue: int.tryParse(args?['value'] ?? ''),
                    );
                },
            );
        default:
            return MaterialPageRoute<void>(
                builder: (_) => const ErrorPage(),
            );
    }
}
```

We first split `settings.name` by `?` to obtain both the current path and the query parameters list, if any. Then, we convert the query parameters list into a map (thanks to the `Uri.splitQueryString` static method). In this way, we can update the `RandomPage` widget to ask for the initial value and pass it directly from the query parameter value (if there's one). For example, the page could look like this:

```

class RandomPage extends StatelessWidget {
  static final _random = Random();

  final int? initialValue;
  const RandomPage({super.key, this.initialValue});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: ElevatedButton(
          onPressed: Navigator.of(context).pop,
          child: Text('${initialValue ?? _random.nextInt(10)}'),
        ),
      ),
    );
  }
}

```

Since the example is trivial, we have placed the URL parsing logic directly inside the route generator function. On larger codebases, you should create a specific class in a separated file that handles the URL parsing.

### 13.3 Declarative navigation using Navigator and Router



[https://github.com/albertodev01/flutter\\_book\\_examples/tree/main/chapter\\_13/13.3](https://github.com/albertodev01/flutter_book_examples/tree/main/chapter_13/13.3)

With the imperative approach (“navigator 1.0”), you only need to define and install routes because the `MaterialApp` or `CupertinoApp` widget will automatically handle the navigator for you. In the declarative approach instead (“navigator 2.0”), you are the one that manages the navigator’s state. Consequently, deep linking is also more complicated to set up and maintain. In this section, we’re going to call `pages` (rather than `routes`) each screen of the application.

#### Note

We aren’t discouraging the usage of the declarative approach. We’re just pointing out the fact that it’s more complicated than the imperative one. The declarative navigation system has a lot of benefits and thanks to the `go_router` package (which will be covered in the next section) it’s almost as easy to use as the `Navigator` API.

We're going to build the same example as the previous section. The first page is `HomePage`, which is immediately shown when the application starts, and the second one is `RandomPage`, which can be opened with a button tap and outputs a random number:

```
class HomePage extends StatelessWidget {
  const HomePage({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            /* Navigate to 'RandomPage' */
          },
          child: const Text('Open random page'),
        ),
      ),
    );
  }
}

class RandomPage extends StatelessWidget {
  static final _random = Random();
  const RandomPage({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            /* Navigate back to 'HomePage' */
          },
          child: Text('${_random.nextInt(10)}'),
        ),
      ),
    );
  }
}
```

With the declarative approach, the `MaterialApp` widget does not internally manage a `Navigator` object automatically. As such, we need to keep track ourselves of the navigation stack using a state management solution. Since many parts of the tree need to be able to retrieve information about the navigator state, we're going for the *application state* management technique:

```

// Contents of the 'routes.dart' file
enum ActivePage {
  homePage('/'),
  randomPage('/random_page');

  final String path;
  const ActivePage(this.path);
}

class InheritedNavigatorState extends InheritedWidget {
  final ValueNotifier<ActivePage> pagesState;
  const InheritedNavigatorState({
    super.key,
    required this.pagesState,
    required super.child,
  });

  static InheritedNavigatorState of(BuildContext context) {
    final ref =
      context.dependOnInheritedWidgetOfExactType<InheritedNavigatorState>();
    assert(ref != null, "Couldn't find the InheritedNavigatorState.");
    return ref!;
  }

  @override
  bool updateShouldNotify(InheritedNavigatorState oldWidget) {
    return pagesState != oldWidget.pagesState;
  }
}

extension InheritedNavigatorStateExt on BuildContext {
  ValueNotifier<ActivePage> get pagesState =>
    InheritedNavigatorState.of(this).pagesState;
}

```

The `ActivePage` enum keeps track of the currently visible page and carries along the associated path. The `Navigator` state is managed by the `pagesState` listenable object, exposed to the subtree by the `InheritedNavigatorState` inherited widget:

```

runApp(
  InheritedNavigatorState(
    pagesState: ValueNotifier<ActiveRoute>(ActiveRoute.homePage),
    child: const MyApp(),
  ),
);

```

Since the configuration will become more complicated, here's an image that summarizes what we will do:

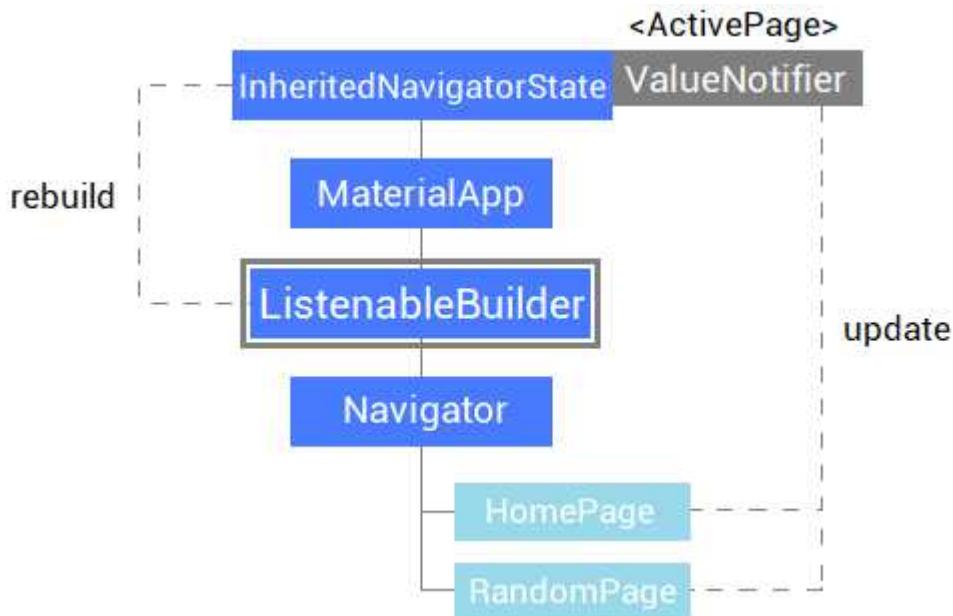


Figure 13.5: How the navigator state is handled.

Thanks to `InheritedNavigatorState`, the navigator state is available to all the widgets below in the tree. This is extremely important because:

- The `Navigator` widget itself can be wrapped in a `ListenableBuilder` to listen for changes about the currently visible page. This is essential to update the page stack and thus change the visible page on the screen. Each time the active page changes, the `Navigator` is rebuilt with the new pages stack.
- Any widget (from any location in the tree) can move to another route by changing the value exposed by the inherited widget. For example, the button callback in the `HomePage` widget can navigate to `RandomPage` with this simple call:

```
ElevatedButton(  
    onPressed: () => context.pagesState.value = ActiveRoute.randomPage,  
    child: const Text('Open random page'),  
,
```

This call uses the inherited widget to reference the `ValueNotifier<ActiveRoute>` object and updates its value, which causes a rebuild of the `Navigator` (see *Figure 13.5*).

The `Navigator` widget is created and wrapped in the `ListenableBuilder`, which changes the contents of the page whenever the selected page is updated:

```
class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: ListenableBuilder(
        listenable: context.pagesState,
        builder: (context, _) {
          return Navigator(
            pages: [
              const MaterialPageRoute(
                child: HomePage(),
              ),
              if (context.pagesState.value == ActiveRoute.randomPage)
                const MaterialPageRoute(
                  child: RandomPage(),
                ),
            ],
            onPopPage: (route, result) {
              if (!route.didPop(result)) {
                return false;
              }

              context.pagesState.value = ActiveRoute.homePage;
              return true;
            },
          );
        },
      ),
    );
  }
}
```

In the `pages` list, the home page comes first because we always want it to be at the bottom of the stack. However, `RandomPage` is only added if the active route points to the random number page. From this, you notice that the last entry in the list is the currently visible page in the screen.

When using the `pages` parameter, the `onPopPage` callback must always be defined. It's triggered whenever `Navigator.pop` is called so, since we only have two pages, we just always set the state back to the home page value. It's very important to return `false` because it tells the framework that the route didn't actually pop.

### 13.3.1 Navigating and passing data between pages

To navigate from one page to another, you need to get a reference to the state object (using the inherited widget) and change the visible page value. For example:

```
class HomePage extends StatelessWidget {
  const HomePage({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            // Navigate to the random number page
            context.pagesState.value = ActivePage.randomPage;
          },
          child: const Text('Open random page'),
        ),
      ),
    );
  }
}

class RandomPage extends StatelessWidget {
  static final _random = Random();
  const RandomPage({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            // Navigate to the random home page
            context.pagesState.value = ActivePage.homePage;
          },
          child: Text('${_random.nextInt(10)}'),
        ),
      ),
    );
  }
}
```

From `RandomPage`, you could have also navigated back to the home page using the `pop` function. While it still works, we don't recommend doing it for consistency:

```
 onPressed: () {
  Navigator.of(context).pop();
  // context.pagesState.value = ActivePage.homePage;
},
```

This call triggers the `onPopPage` callback we defined in the `Navigator` widget. We ensured that the state would be updated back to the home page configuration whenever `pop` was called. As such, popping automatically sets the state back to `ActivePage.homepage` anyway:

```
onPopPage: (route, result) {
  if (!route.didPop(result)) {
    return false;
  }

  context.pagesState.value = ActivePage.homePage;
  return true;
}
```

Even if possible, we recommend avoiding mixing declarative and imperative approaches. When it comes to passing data from one page to another, we recommend avoiding placing route-specific data in the navigator state. If you created a shopping application for example, you could make the following:

- create a `routes.dart` file where you manage the navigator state;
- create another file where you manage the user authentication state;
- create another file where you manage the shopping cart state and so on.

Don't put everything inside `routes.dart` or don't have your navigator state to also handle other things (such as the user authentication state). To pass data from a page to another, you can take advantage of `InheritedWidgets` and create state classes that can share data between multiple widgets.

### Note

Even if you're using another state management solution, try to create state classes that follow the single responsibility principle. We believe that having multiple small state classes is better (and more maintainable/testable) than having a single one that handles everything.

We still can push routes using `Navigator.of(context).push` but it puts out of sync the state with the current stack. By consequence, we recommend again to NOT use together the declarative and

navigator approaches (even if it's possible). Declarative route management is meant to manually manage the pages stack and thus you should only use the navigator with the state management solution we've chosen (in our case, inherited widgets and listenable classes).

### 13.3.2 Deep linking



[https://github.com/albertodev01/flutter\\_book\\_examples/chapter\\_13/13.3.2/](https://github.com/albertodev01/flutter_book_examples/chapter_13/13.3.2/)

Even if we've already written a lot of code, we can only define a stack of pages and manage it using an `InheritedWidget` (and a `ValueNotifier`). We are not yet able to handle the platform's back button deep linking is not enabled. To complete the work, we need to refactor our code and create more classes:

- `RouteInformationParser`: parses routes information that come from the platform and converts them into user-defined data types.
- `RouterDelegate`: manages the navigator state by building and maintaining the page stacks. We'll need to move our state inside here.
- Remove the state class and the inherited widget because all the logic will be moved inside the router delegate.
- We could also create a `RouteInformationProvider` object but it's not required to make deep links work. To not complicate the example even more, we won't create it. This class is just a `ValueNotifier` that listens for route information changes.

We will write a lot of code just to enable deep linking in a two pages application. You can imagine how complicated this can become in a real-world application.

#### Note

We're still showing the same example because we think it's important to understand the `Router` API and how it works under the hood. In the next section, we will show how to use the official Flutter `go_router` package to make declarative navigation easy and quick to implement.

We're not using inherited widgets and listenable classes anymore because the `Router` object will handle everything. We could still take advantage of an inherited widget to not pass callbacks to page widgets directly, but it would make the example even more complicated. This is what our new main function looks like:

```
void main() {
  runApp(
    MaterialApp.router(
      routeInformationParser: const MyRouteInformationParser(),
      routerDelegate: MyRouterDelegate(),
    ),
  );
}
```

The `router` named constructor internally uses a `Router` rather than a `Navigator`, and it requires the creation of two specific classes. Before moving on, we need to remove the entire contents of the `routes.dart` file and create this new enumeration:

```
enum AppRoutePath {
  home('/'),
  random('/random'),
  error('/error');

  final String path;
  const AppRoutePath(this.path);

  bool get isHomePage => index == 0;
  bool get isRandomPage => index == 1;
  bool get isError => index == 2;
}
```

This is useful for page stack management inside the router delegate. Let's move on and see how the parser and the delegate classes are implemented.

#### 13.3.2.1 The information parser class

The `RouteInformationParser` class is used to parse incoming routes and convert them into a user-defined data type. In practical terms, this class parses a URL and maps it to a specific `AppRoutePath` instance. There are two methods to override:

- `parseRouteInformation`: converts a `Uri` into a user-defined data type. In our example, we have to convert `/` to `AppRoutePath.home()` and `/random` to `AppRoutePath.random()`.

- `restoreRouteInformation`: converts a user-defined data type into a `Uri`. It does the reverse operation of the above method: for example, it converts `AppRoutePath.random()` to the `/random` path.

You can see this parser as a converter that translates route information (coming from the OS or the browser) into Dart data types. We have implemented it in this way:

```
class MyRouteInformationParser extends RouteInformationParser<AppRoutePath>{
  const MyRouteInformationParser();

  @override
  Future<AppRoutePath> parseRouteInformation(
    RouteInformation routeInformation,
  ) async {
    if (routeInformation.uri.path == AppRoutePath.home.path) {
      return AppRoutePath.home;
    }

    if (routeInformation.uri.path == AppRoutePath.random.path) {
      return AppRoutePath.random;
    }

    return AppRoutePath.error;
  }

  @override
  RouteInformation? restoreRouteInformation(AppRoutePath configuration) {
    if (configuration.isError) {
      return RouteInformation(
        uri: Uri.parse(AppRoutePath.error.path),
      );
    }
    if (configuration.isHomePage) {
      return RouteInformation(
        uri: Uri.parse(AppRoutePath.home.path),
      );
    }
    if (configuration.isRandomPage) {
      return RouteInformation(
        uri: Uri.parse(AppRoutePath.random.path),
      );
    }

    return null;
  }
}
```

The first method converts a route into a Dart type and the second override does the opposite. This logic handles the communication between the platform (Android, iOS or a web browser) and the `Router` class in the Dart code.

### 13.3.2.2 The router delegate

The `RouterDelegate` class is used by the `Router` to build and configure a page stack with the usual `Navigator` widget. Rather than managing the `Navigator` state in a dedicated notifier class, we have to do it inside here:

```
class MyRouterDelegate extends RouterDelegate<AppRoutePath>
    with ChangeNotifier, PopNavigatorRouterDelegateMixin<AppRoutePath> {
    final GlobalKey<NavigatorState> _navigatorKey = GlobalKey<NavigatorState>();
    var _isError = false, _isHome = true;

    @override
    GlobalKey<NavigatorState> get navigatorKey => _navigatorKey;

    @override
    AppRoutePath get currentConfiguration {
        if (_isError) { return AppRoutePath.error; }
        if (_isHome) { return AppRoutePath.home; }
        return AppRoutePath.random;
    }

    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Navigator( /* Navigator setup here... */ ),
        );
    }

    @override
    Future<void> setNewRoutePath(AppRoutePath configuration) async {
        if (configuration.isError) {
            _isError = true;
        } else if (configuration.isHomePage) {
            _isError = false;
            _isHome = true;
        } else {
            _isError = false;
            _isHome = false;
        }
    }
}
```

A lot is going on here:

- We first define a global key to hold the navigator state (a required override).
- The `currentConfiguration` getter is used by `Router` to determine which is the currently active page after it's rebuilt. If you don't override this getter, the router has no way to report route information.
- The `setNewRoutePath` is a callback: it's triggered whenever a new route has been pushed to the application by the operating system. This callback updates the internal state to reflect changes in the stack.

The `build` method must return a `Navigator` with the same key we passed to the `navigatorKey` override:

```
Navigator(  
  key: _navigatorKey,  
  pages: [  
    MaterialPage(  
      key: const ValueKey('HomePage'),  
      child: HomePage(  
        openRandomPage: () {  
          _isHome = false; _isError = false;  
          notifyListeners();  
        },  
      ),  
    ),  
    ),  
    if (_isError)  
      const MaterialPage(  
        key: ValueKey('ErrorPage'),  
        child: ErrorPage(),  
      ),  
    if (!_isHome)  
      MaterialPage(  
        key: const ValueKey('RandomPage'),  
        child: RandomPage(  
          openHomePage: () {  
            _isHome = true; _isError = false;  
            notifyListeners();  
          },  
        ),  
      ),  
  ],  
);
```

Giving a [key](#) to each page is very important because it allows [Navigator](#) to distinguish between different [MaterialPage](#) objects, especially when the same page could be created with different arguments.

We're directly passing a callback to page widgets for simplicity, but we could have created another state class with an inherited widget to handle these changes. With this huge setup, we finally have deep linking enabled in the application:

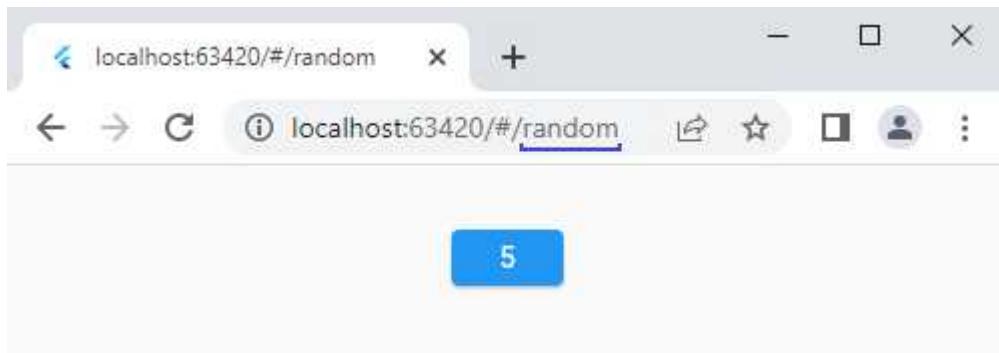


Figure 13.6: Deep linking in action with the [Router API](#)

The [Navigator](#) widget also required the [onPopPage](#) callback to be defined. We did not override it to not make the code too long, but you must remember to do that.

#### 12.3.2.3 Considerations

Implementing all of this is quite complex and time-consuming, especially if you will always need to repeat the whole process for any new project. The separation of responsibilities gives the developer full control over deep linking, sub-routes, and much more but at the cost of complexity.

#### Note

Imagine implementing all of this logic in a real-world application with maybe a dozen of routes. Deep linking is a feature you very likely want because Flutter can also run on the web. You'd add even more complexity if you also wanted to parse query parameters, create handle rub-routes and handle special redirect cases.

We aren't discouraging the usage of the [Router API](#). We just believe that it's very flexible but at the same time too hard to maintain. The Flutter team created an official package called [go\\_router](#) to

reduce the complexity of the declarative approach and we strongly recommend to use it. It has a straightforward API that builds the parser and the delegate for you.

## 13.4 The go\_router package

The official `go_router` package uses the `Router` API under the hood and greatly simplifies its usage. You will need a very minimal route setup to build the same two-screen application of the previous sections. This is the content of the `routes.dart` file:

```
// routes.dart
const homePath = '/';
const randomPath = '/random';

GoRouter generateRouter() {
  return GoRouter(
    initialLocation: homePath,
    routes: [
      GoRoute(
        path: homePath,
        builder: (context, state) => const HomePage(),
      ),
      GoRoute(
        path: randomPath,
        builder: (context, state) => const RandomPage(),
      ),
    ],
  );
}
```

We don't need to pass callbacks to our page widgets anymore. The `GoRouter` object we have built can be “installed” in the `MaterialApp` or `CupertinoApp` widget using the classic `router` named constructor:

```
void main() {
  // We prefer creating the router here, but you could have also passed this
  // function directly to 'routerConfig'
  final router = generateRouter();

  runApp(
    MaterialApp.router(
      debugShowCheckedModeBanner: false,
      routerConfig: router,
    ),
  );
}
```

The `GoRouter` setup is minimal because it just requires a list of routes. It internally generates the delegate, parser, and information objects the `router` constructor needs. In other words, you don't have to manage the router and the navigator state anymore because `GoRouter` does everything for you. This is how you navigate from one page to the other:

```
class HomePage extends StatelessWidget {
  const HomePage({
    super.key,
  });

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: ElevatedButton(
          onPressed: () => context.go(randomPath),
          child: const Text('Open random page'),
        ),
      ),
    );
  }
}

class RandomPage extends StatelessWidget {
  static final _random = Random();
  const RandomPage({
    super.key,
  });

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: ElevatedButton(
          onPressed: () => context.go(homePath),
          child: Text('${_random.nextInt(10})'),
        ),
      ),
    );
  }
}
```

Thanks to extension methods, you can use `context.go` to navigate from a route to another easily. Deep linking is enabled by default and thanks to the `errorBuilder` parameter, the `ErrorPage` widget is shown whenever the user tries to navigate to an unknown route. For example:

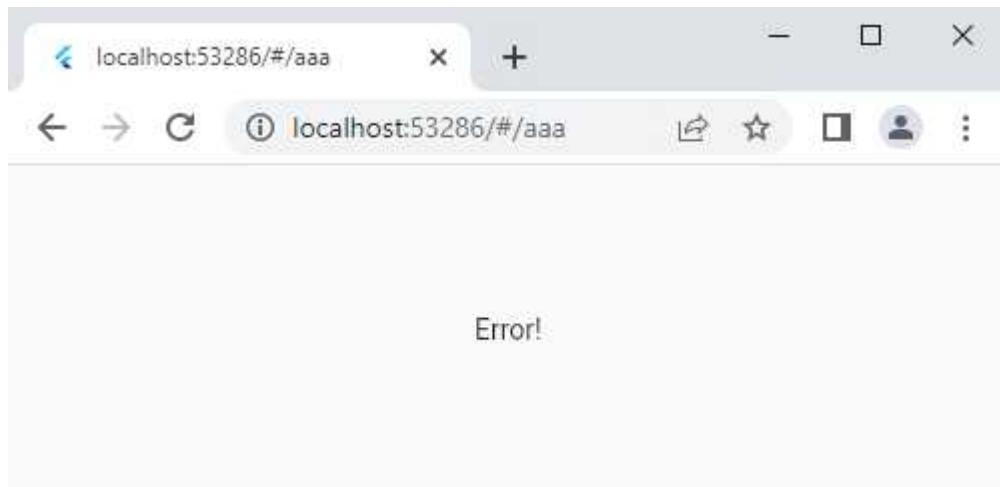


Figure 13.7: The `GoRouter` package redirects to the error page in case of an invalid URL

The `GoRouterState` object is always passed to builder functions and it's used to extract information about the route. For example, you could use it to get the current location and redirect the user to another page under certain conditions:

```
return GoRouter(  
  routes: [  
    GoRoute(  
      path: '/',
      builder: (context, state) => const HomePage(),  
    ),  
    GoRoute(  
      path: '/login',  
      builder: (context, state) => const LoginPage(),  
    ),  
    GoRoute(  
      path: '/profile',  
      builder: (context, state) => const ProfilePage(),  
    ),  
  ],  
  redirect: (context, state) async {  
    // Assume that 'loginState' is an external object that keeps track of the  
    // authentication status  
    if (state.location.contains('profile') && !loginState.isUserLogged) {  
      return '/login';  
    }  
    return null;  
  },  
);
```

The `state` parameter in the `redirect` callback is used to redirect back to the login page when the user tries to access the profile page without being authenticated. The `redirectLimit` value is used to configure the maximum number of redirects that are expected to occur in the application. There are two kinds of redirection:

- Top-level redirection: defined by `GoRouter` and called before any navigation event:

```
GoRouter(  
  routes: [  
    GoRoute(  
      path: '/',
      builder: (context, state) => const HomePage(),  
    ),  
    ],  
    redirect: (context, state) async {  
      // code...  
    },  
  ),
```

- Route-level redirection: defined by `GoRoute` and called when a navigation event is about to display the route:

```
GoRouter(  
  routes: [  
    GoRoute(  
      path: '/',
      builder: (context, state) => const HomePage(),  
      redirect: (context, state) async {  
        // code...  
      }  
    ),  
    ],  
  ),
```

By default, `GoRouter` has a pre-built error screen (for `MaterialApp` and `CupertinoApp` widgets) that is shown when the user navigates to an unknown route. You can customize this behavior using the `errorBuilder` callback:

```
return GoRouter(  
  errorBuilder: (context, state) => const ErrorPage(),  
  routes: [ /* routes... */ ],  
);
```

If you want to use Material-like or Cupertino-like styled pages, you can use `pageBuilder` instead of `builder`. For custom transitions, use the `CustomTransitionPage` object. For example:

```

return GoRouter(
  routes: [
    GoRoute(
      path: pathA,
      pageBuilder: (context, state) => MaterialPage(
        key: state.pageKey,
        child: const PageA(),
      ),
    ),
    GoRoute(
      path: pathB,
      pageBuilder: (context, state) => CupertinoPage(
        key: state.pageKey,
        child: const PageB(),
      ),
    ),
    GoRoute(
      path: pathC,
      pageBuilder: (context, state) => CustomTransitionPage(
        key: state.pageKey,
        transitionsBuilder: (context, animation, _, child) {
          // This is where you create custom animations
          return ScaleTransition(
            scale: animation,
            child: child,
          );
        },
        child: const PageC(),
      ),
    ),
  ],
);

```

When creating a `Page` object yourself using the `pageBuilder` callback, it is very important that you remember to set the `key` property with `state.pageKey`. The key is used to uniquely identify the page widget on the stack, which is an important internal detail. The “animation framework”, which also includes transitions, is covered in *chapter 17 – “Animations”*.

### 13.4.1 Understanding navigation

To navigate from one page to another you can either use `context.go` (which we recommend) or the more verbose (but equivalent) `GoRouter.of(context).go` version. Under the hood, `GoRouter` is a listenable class because it extends `ChangeNotifier`. You can use it to listen for navigation events with `ListenableBuilder`:

```

class MyWidget extends StatelessWidget {
  const MyWidget({super.key});

  @override
  Widget build(BuildContext context) {
    final router = GoRouter.of(context);

    return ListenableBuilder(
      listenable: router,
      builder: (_, __) => Text(router.location),
    );
  }
}

```

There actually are two methods you can use to navigate to another page. Going to a page means *replacing* the old one with the new one, pushing a page means *adding* a new entry to the stack. To better understand the difference, imagine you called `push` to go from `Page1` to `Page2`:

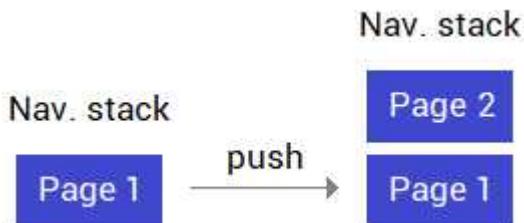


Figure 13.8: Using `context.push` to add a new page in the stack.

Notice that pushing makes `Page2` visible by adding it to the stack before the previous page. Now let's make a comparison between `go` and `push` when navigating to `Page3`:

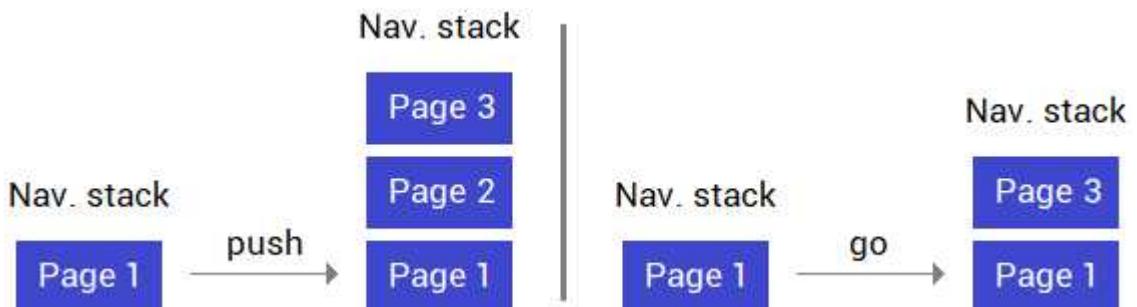


Figure 13.9: `context.push` adds the page to the stack, `context.go` replaces the old one.

Both `go` and `push` have the same effect: they show the `Page3` route on the screen. However, they do it in two different ways:

- Pushing a route means adding new entries in the stack. In the example, when you call `pop()`, you come back to the previous page (`Page2`) in a linear way.
- Going to a route means replacing the new entry with the old one in the stack. In the example, when you call `pop()`, you go to the penultimate page in the stack (`Page1`) because `Page2` was replaced with the currently visible page.

Pushing pages allows you to build the stack programmatically rather than declaratively. You might want to use `push` and `go` together to distinguish between top-level navigation and sub-routes. For example, imagine that your application had two buttons: one to authenticate already existing users and one to register new ones. You could do this:

- If you successfully authenticate, you don't want to go back to the login page unless you press the "logout" button. As such, you could use `context.go` to replace the login page with the landing page. In this way, if the user presses the back button, the application closes since the landing page replaced the login page.
- If you decided to register, you could `context.push` to open the registration page and then use `context.go` for the various registration steps. In this way, you can press the back button at any step to close the entire registration flow and come back to the login page directly.

Generally, `push` is used to create "top level" routes (since it adds a new page to the stack) while `go` is used to create "sub" routes (since it replaces the previous page). Regardless, both are good for any kind of navigation purpose. You may decide to always use `go`, for example, but sometimes it could make your navigation logic harder to implement than `push`.

### 13.4.2 Passing data between pages and using query parameters

To pass data from one route to another, we still recommend using a state management solution (such as `InheritedWidgets` and listenable classes) to share information across multiple widgets. However, you could also pass data between routes using the `go` or `push` methods. For example:

```
onPressed: () => context.go(randomPath, extra: 10),
```

This code passes an initial value (`10`) to the random number page using `Object? extra`. To receive this value in `RandomPage`, the route definition needs a change:

```

GoRoute(
  path: randomPath,
  builder: (context, state) {
    final value = state.extra as int?;

    return RandomPage(
      initialValue: value,
    );
  },
),
),

```

This is useful to directly pass objects to other routes without exposing the information via URI. The `extra` object cannot be used in deep linking. For example, it's lost when you press the browser's back button so we don't recommend using `extra` it on Flutter web applications. Another more convenient way to pass information from a page involves query parameters:

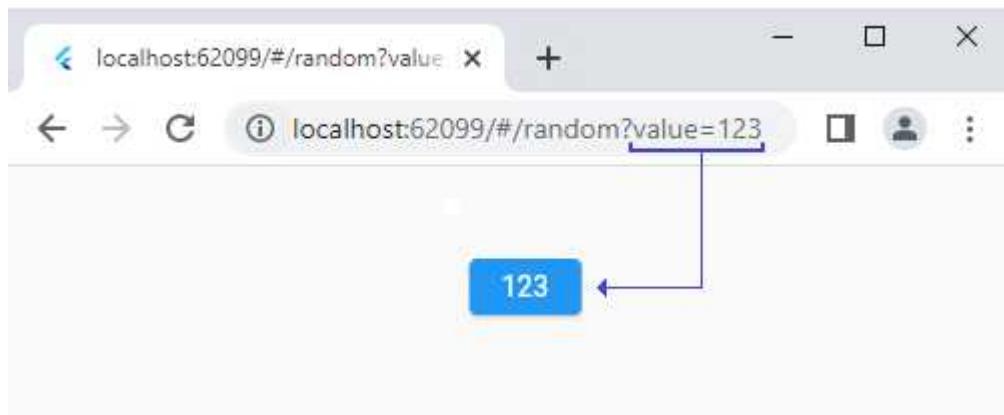


Figure 13.10: Parsing query parameters using `go_router`.

Query parameters are a set of key-value pairs passed at the end of a URI after the `?` symbol. To obtain the result in *Figure 13.10*, you need to use the router's state to access the query parameters list and pass the data to the page. For example:

```

GoRoute(
  path: randomPath,
  builder: (context, state) {
    final int? value = int.tryParse(state.queryParams['value'] ?? '');

    return RandomPage(
      initialValue: value,
    );
  },
),
),

```

The `queryParams` property is a `Map<String, String>` you can use to retrieve the value of a query parameter. The nice thing is that you can use query parameters even in the application itself when pushing or going to a new route:

```
 onPressed: () => context.go('$randomPath?value=123'),  
 onPressed: () => context.push('$randomPath?value=123'),
```

Since query parameters are optional, you should always handle the case where they are not passed (generally with default values or null checks). The idea of dynamic linking instead is used to pass required data to another route so that it gets uniquely identified. For example, we could change our logic to use dynamic links for the random numbers page:

```
GoRoute(  
    path: '/random/:value',  
    builder: (context, state) {  
        final value = int.tryParse(state.params['value']!);  
  
        return RandomPage(  
            initialValue: value,  
        );  
    },  
,),
```

With this change, you will have to always pass a value to the URI. In this case, the random number has to be generated elsewhere, such as from the route that calls `go/push` or directly from the URI. On the browser for example, it works as follows:

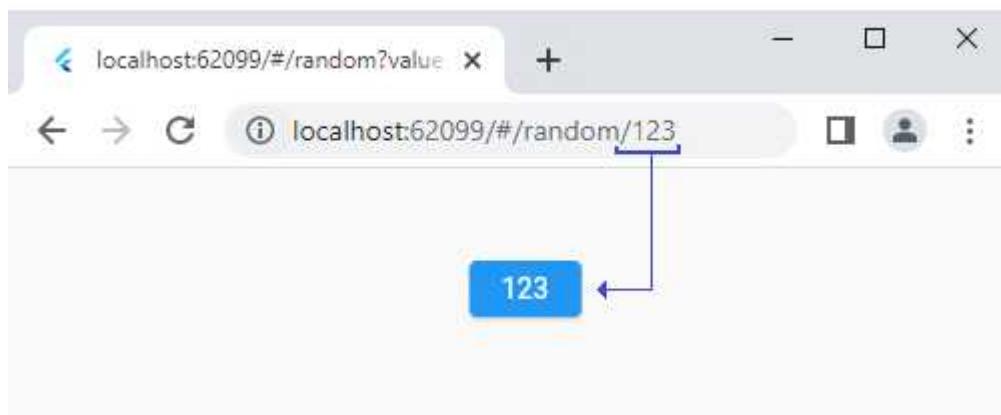


Figure 13.11: Usage of dynamic links to pass data to another route

The value is part of the URI itself and is not optional. The value can be changed on the browser or you can pass it to `go` or `push`. For example:

```

// Example callback with 'go'
 onPressed: () {
  final value = Random().nextInt(100);
  context.go('$randomPath/$value');
}

// Example callback with 'push'
 onPressed: () {
  final value = Random().nextInt(100);
  context.push('$randomPath/$value');
}

```

Notice that the separator symbol is `/` and not `?`. To sum up, these are all the possible ways you have to pass data between routes:

1. Use the `extra` parameter of `go` or `push`. We do not recommend this approach because it doesn't take advantage of deep linking and almost always requires a runtime cast (because `extra` is of `Object?` type).
2. Use a state management solution such as inherited widgets with a listenable class or a value notifier. We recommend this approach when the page state is needed by multiple widgets and it combines with the router's state.
3. Use query parameters to pass optional data. We recommend this approach when your page can be created without optional information. For example, query parameters can be useful to let your application know about a possible redirect to another source. If you don't pass redirect information, the redirect doesn't happen.
4. Use dynamic linking to pass required data. We recommend this approach when your page cannot be created without specific information. For example, it might be helpful for a library website to show particular data about an author or a book using a unique id.

Both query parameters and dynamic linking are fully compatible with deep linking.

## 13.5 Good practices

We have described three techniques to handle navigation between pages. It may be helpful to have a small comparison table that shows the main differences between the imperative and declarative navigation approaches:

## Imperative

- Also known as navigator 1.0
- The `Navigator` is internally created and maintained by `MaterialApp` or `CupertinoApp` widgets
- Deep linking is enabled by passing the `settings` parameter to the page route widget.
- Query parameters are manually parsed with the (recommended) `Uri` class.
- Data can be passed from a page to another using a state management solution or directly within the page route object.

## Declarative

- Also known as navigator 2.0
- The `Navigator` has to be created and maintained by the developer manually
- Deep linking is enabled by creating a parser and a delegate class.
- Query parameters are manually parsed with the (recommended) `Uri` class.
- Data can be passed from a page to another using a state management solution, possibly associated to the router delegate class.

The `go_router` package can be seen as “declarative navigation made easy” because it creates the router delegate and parser classes for you. In other words, the Router API’s complexity is hidden behind the `GoRouter` object. This package has a lot of strong points:

- No need to manually manage the `Navigator` state, the router class, deep linking, query parameters and much more. Everything is internally handled by the package for you.
- Query parameters don’t need to be parsed because they’re automatically handled by the `GoRouterState` class.
- Data between pages can be passed using a state management solution, query parameters or dynamic linking.

Keep in mind that the declarative approach is built on top of the imperative one so you could mix their usages. Since `go_router` is built on top of the declarative system, it can be combined with the other two as well. Whatever solution you choose, don’t mix one with the others: be consistent and stick with your choice across the entire application.

### 13.5.1 Changing the URL path strategy

By default, when you run your Flutter application on a web browser, it always adds a hash (#) in the URL. For example:

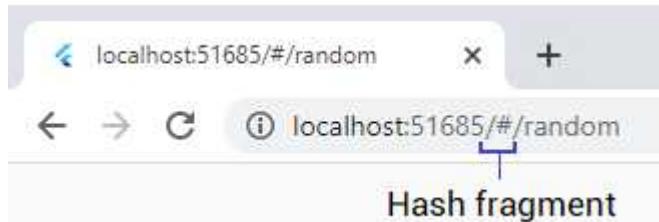


Figure 13.12: The hash fragment in a Flutter web application.

You must change the URL path strategy if you want to remove the hash after the hostname. To do so, import the `flutter_web_plugins` package (which is part of the Flutter SDK) and set the path strategy before building the widget tree. For example:

```
import 'package:flutter_web_plugins/url_strategy.dart'; // <-- import this

void main() {
  final router = generateRouter();
  usePathUrlStrategy(); // Call this before 'runApp'

  runApp(
    MaterialApp.router(
      routeInformationProvider: router.routeInformationProvider,
      routerDelegate: router.routerDelegate,
      routeInformationParser: router.routeInformationParser,
    ),
  );
}
```

With this change for example, the hash goes away and you'll have a path-based URL:



Figure 13.13: Removing the hash fragment from the URL.

Regardless you're using `go_router`, navigator 1.0 or navigator 2.0, you may need another server-side configuration. When deploying your web application, configure your server so that any URL ends up at your `index.html` file, otherwise Flutter won't be able to route your pages.

### 13.5.2 State restoration and navigation

In *Deep dive – “State restoration”* we have covered how state restoration works in Flutter. There are specific considerations to make regarding restoring the navigation state. For the imperative navigation system (*navigator 1.0*) it is quite easy:

```
Navigator.of(context).restorablePushNamed(settingsPath);
```

Push operations have a restorable version (always prefixed by the “*restorable*” word) that can be used to restore the navigation state. For the declarative approach instead (navigator 2.0), we need to give a few unique IDs to the navigator and its pages:

- We've already seen that the `build` method of the delegate returns a `Navigator` widget. To correctly enable state restoration, we need to assign it a unique string:

```
const Navigator(  
    restorationScopeId: 'navigator_id',  
    pages: [  
        // pages list here...  
    ],  
,),
```

- Each navigator page requires a unique id, which is passed to the `restorationId` property:

```
const Navigator(  
    restorationScopeId: 'navigator_id',  
    pages: [  
        MaterialPage(  
            key: ValueKey('HomePage'),  
            restorationId: 'home_page_id', // unique id  
            child: HomePage(),  
        ),  
        MaterialPage(  
            key: ValueKey('SettingsPage'),  
            restorationId: 'settings_page_id', // unique id  
            child: SettingsPage(),  
        ),  
    ],  
,),
```

When it comes to the `go_router` package, you need to add a restoration id to the `GoRouter` object:

```
GoRouter(  
  routes: [  
    GoRoute(  
      path: homePath,  
      builder: (context, state) => const HomePage(),  
    ),  
    GoRoute(  
      path: settingsPath,  
      builder: (context, state) => const SettingsPage(),  
    ),  
  ],  
  restorationScopeId: 'go_router_id'  
)
```

All of the considerations for these three navigation approaches assume that your `MaterialApp` or `CupertinoApp` widgets have a `restorationScopeId` defined.

## Deep dive: Navigation and overlays

The `Overlay` widget is a special version of a `Stack` that floats on top of other widgets. For example, overlays are often used when you need a widget that stays visible even when navigating another route. Imagine you had two routes in your application:

route\_a.dart

```
class PageA extends StatelessWidget {  
  const PageA({super.key});  
  
  @override  
  Widget build(BuildContext ctx) {  
    return Scaffold(  
      body: Center(  
        child: ElevatedButton(  
          onPressed: () => ctx.go('/b'),  
          child: const Text('Page B'),  
        ),  
      ),  
    );  
  }  
}
```

route\_b.dart

```
class PageB extends StatelessWidget {  
  const PageB({super.key});  
  
  @override  
  Widget build(BuildContext ctx) {  
    return Scaffold(  
      body: Center(  
        child: ElevatedButton(  
          onPressed: () => ctx.go('/a'),  
          child: const Text('Page A'),  
        ),  
      ),  
    );  
  }  
}
```

For example, we couldn't use a `Stack` to show a popup message. If we created a `Stack` on `PageA`, it would disappear as soon as we navigated to `PageB`. To solve this problem, we can use overlays.

An overlay is a special `Stack` widget placed “in front” of your application. As such, the overlay stays visible even if you navigate to a different route. For example, this is how you could create a popup message on `PageA` that remains visible even if you navigate to `PageB`:

```
class _PageAState extends State<PageA> {
  OverlayEntry? entry;

  void removePopup() {
    if (entry != null) {
      entry!.remove();
      entry = null;
    }
  }

  void createPopupMessage() {
    removePopup();

    entry = OverlayEntry(builder: (context) {
      return Positioned(
        left: 30,
        right: 30,
        bottom: 10,
        child: Container(
          height: 20,
          color: Colors.lime,
        ),
      );
    });
  }

  Overlay.of(context).insert(entry!);
}

@Override
Widget build(BuildContext context) {
  // code...
}
```

The `OverlayEntry` listenable class represents a widget that floats in front of the screen. Its builder function often returns a `Positioned` widget to position the widget that floats on the screen at the given location. In this example:

- The `Overlay` widget is used to add `OverlayEntry` elements on the screen.

- The `Navigator` internally creates an `Overlay` widget so that you can safely call `Overlay.of` from anywhere. Although you can manually create an `Overlay` and insert it on the widget tree, it's most common to use the one created by the `Navigator`.
- The `OverlayEntry` object generally lives within the state class because, to remove an entry from the screen, you need to call its `remove` method. `OverlayEntry` is a `Listenable` that notifies when the widget built by its builder function is mounted or unmounted.
- As soon as you call `remove()`, the children widgets are disposed and the overlay is removed.

The `build` method of our example has three buttons: one to show the overlay, one to navigate to the other route, and one to remove the overlay:

```
Row(
  mainAxisAlignment: MainAxisAlignment.spaceAround,
  children: [
    ElevatedButton(
      onPressed: createPopupMessage, // Shows the overlay
      child: const Text('Show'),
    ),
    ElevatedButton(
      onPressed: () => context.go(randomPath),
      child: const Text('Page B'),
    ),
    ElevatedButton(
      onPressed: removePopup, // Closes the overlay
      child: const Text('Hide'),
    ),
  ],
),
```

When you press the `Show` button, the `OverlayEntry` object is created and added to the `Overlay` children list using `insert`. As a result, the green rectangle is displayed on the screen:



Figure 13.14: The green rectangle is the `OverlayEntry` that appears after having tapped on “Show”.

The interesting part is: when you press the “*Page B*” button, you navigate to the [PageB](#) route and the overlay stays at the same position on the screen:

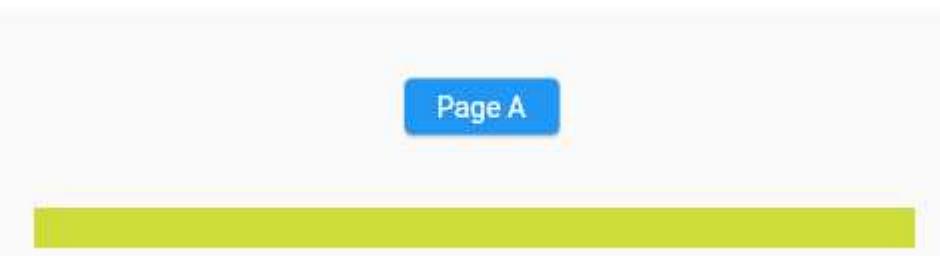


Figure 13.15: The [PageB](#) route with the [OverlayEntry](#) that was inserted by the previous route ([PageA](#)).

In practice, the green rectangle is “in front” of your application and so, even if you move to another route, it stays visible. In the example, the entry will always remain visible unless the `remove` method is called from the [OverlayEntry](#) object (located in the [PageA](#) route).

### Note

The [Navigator](#) widget internally uses [Overlay](#) to add or remove routes from the screen and also allows you to manually insert entries (thanks to `Overlay.of(context)`). So yes, routes are “full-screen” [OverlayEntry](#) objects that are automatically managed by the [Navigator](#) widget.

This is why it seems that the green rectangle of the example is floating “in front” of the application. The [Navigator](#) adds one or more [OverlayEntry](#) objects to its internal [Overlay](#) widget to show routes on the screen. In addition, we added one more entry to show our custom widget (the rectangle).

As result, you see two overlapped [OverlayEntry](#): the route, added by [Navigator](#), and the green rectangle, added by you.

When an [OverlayEntry](#) occupies the entire screen, you should set its `opaque` property to `true` for efficiency. If an entry is *opaque*, the [Overlay](#) widget will not build all entries below. For example:

```
fScreenEntry = OverlayEntry(  
    opaque: true,  
    builder: (context) => Positioned.fill( /* ... code ... */ ),  
);
```

Since `fScreenEntry` takes the entire screen (thanks to the `Positioned.fill` named constructor), all the contents below this entry will not be visible. As such, setting the `opaque` parameter to `true` ensures that all entries below `fScreenEntry` will not be built unnecessarily (because you wouldn't see them).

### Note

Even if the entry doesn't cover the entire screen, you can still set `opaque: true`. In this case, you will only see your entry and the rest of the screen will be black (because all the other entries will not be built).

If you want to disable this mechanism and make sure that an entry is always built (even if there is an opaque one above), set the `maintainState` property to `true`:

```
myEntry = OverlayEntry(  
  maintainState: true,  
  builder: (context) => Positioned.fill( /* ... code ... */ ),  
);
```

With this change `myEntry` will always be built, even if another opaque entry will later be inserted into the `Overlay`. However, keep in mind that using `maintainState` is more expensive and should be used with care<sup>86</sup>, because it forces widgets to be rendered even when it may not be needed.

In general, `maintainState` is only used by the `Navigator` (and `Route` objects) to make sure that routes are kept around even when in the background. This is to make sure that futures coming from subsequent routes will be handled properly when they complete.

---

<sup>86</sup> <https://api.flutter.dev/flutter/widgets/OverlayEntry/maintainState.html>

# 14 – Layouts and responsiveness

## 14.1 Layouts in Flutter

When you set a widget using `width: 100`, it does not mean that it's 100 pixels wide. Flutter instead calculates the platform's pixel ratio, which provides a flexible way to accommodate a design across platforms. For example, look at this image:

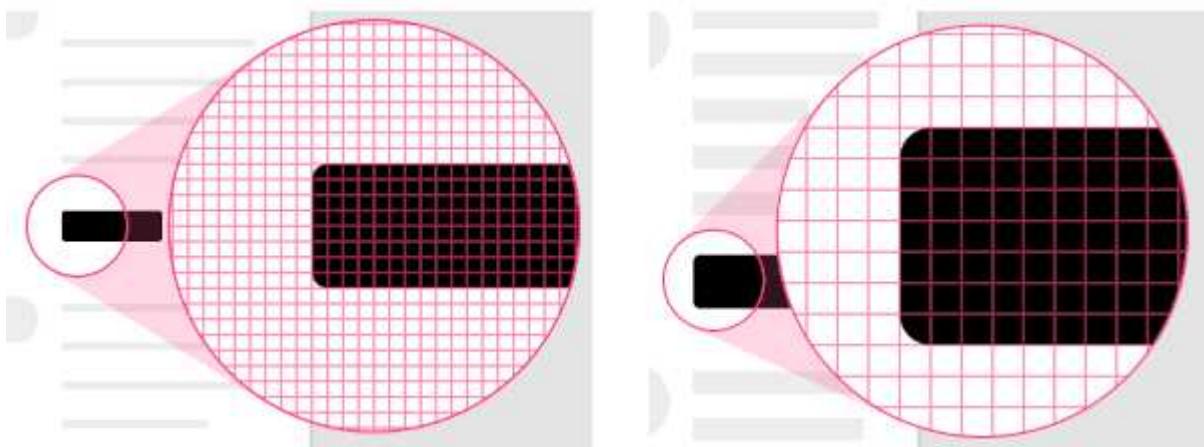


Figure 14.1: The same application on a high-density display (left) and a low-density display (right).

A high-density display (on the left of *Figure 14.1*) has more pixels per millimeter than a low-density display (on the right of *Figure 14.1*). Flutter uses **logical pixels**: a flexible measure unit that scales to have uniform dimensions on any screen. In other words, rather than using pixels, Flutter relies on a ratio based on pixel density to display elements consistently (regardless the resolution).

### Note

Since Flutter runs on both mobile and desktop platforms, it has to support any kind of screen regardless the resolution. If Flutter used pixels, the UI on a high-density display wouldn't be consistent with the UI on a low-density display (because there would be less pixels). Logical pixels are roughly the same visual size across different devices.

Logical pixels are basically the same thing as **dps** (density-independent pixels). Note that one logical pixel maps to a certain number of pixels, according with the resolution.

From a practical point of view, it's enough for you to know that Flutter does not work with pixels. Instead, it uses a scalable unit called “logical pixel” which allows consistency across different kind of resolutions.

### 14.1.1 Laying out widgets

In *chapter 11 – Material, Cupertino and CustomPaint* we've seen how to use widgets to create UIs in Flutter individually. This chapter will explain how multiple widgets can be “packed” together to compose UIs. The most common layout pattern is to arrange widgets along the vertical or horizontal axis. In Flutter, there are two basic classes for this purpose:

- `Row` is used to lay out one or more widgets horizontally; `Column` is used to lay out one or more widgets vertically.
- You can specify how a `Row` or a `Column` widget align their children. Both widgets are basically identical: the only difference is the axis in which they place the children.

These two widgets are good for a broad category of use cases, but they shouldn't be always used. If you wanted to place a single widget at the center of a parent, for example, you could use a `Row` but (for this specific use-case) a `Center` widget would be better:

Not so good	Very good
<pre>SizedBox(     width: 150,     height: 150,     child: Row(         mainAxisAlignment:             MainAxisAlignment.center,         children: const [             Text('Hello!'),         ],     ), ,</pre>	<pre>const SizedBox(     width: 150,     height: 150,     child: Center(         child: Text('Hello!'),     ), ,</pre>

It is a matter of balancing code readability and complexity. Both versions are good and produce the same result, but rows and columns are meant to be used when two or more children exist. As such, in this case, `Center` fits better and also makes the code cleaner.

`Row` and `Column` are particularly useful when you want to pack two or more widgets together. For example, imagine you wanted to center three icons in a box:



Figure 14.2: Packing three icons at the center of a box with `Row`.

By default, rows and columns take as much space as possible along their main axis. However, you can set the `mainAxisSize` property to `MainAxisSize.min` to make them as small as possible. To get the result in *Figure 14.2*, we need to “pack” three icons within a `Row`:

```
Container(  
  width: 200,  
  height: 60,  
  decoration: BoxDecoration(  
    border: Border.all(  
      color: Colors.lightBlue,  
      width: 2,  
    ),  
  ),  
  child: Center(  
    child: Row(  
      mainAxisAlignment: MainAxisAlignment.min, // packs children together  
      children: const [  
        Icon(Icons.ac_unit),  
        SizedBox(width: 10),  
        Icon(Icons.access_alarm_rounded),  
        SizedBox(width: 10),  
        Icon(Icons.account_balance_outlined),  
      ],  
    ),  
  ),  
,
```

Thanks to `MainAxisSize.min`, we’re telling the `Row` to be as small as possible so that it occupies the least amount of space possible. This technique is often used with `Column` as well to pack two or more widgets but in the vertical axis.

### Note

Both `Row` and `Column` do NOT scroll if the widgets don’t fit the viewport. In section 14.3 *Scrollable widgets* we will see how to handle this particular case.

When a layout is too large to fit in a [Row](#) or in a [Column](#), a yellow and black striped box appears along the problematic edge. Consider this example where the boxes are too wide for the available space:

```
@override  
Widget build(BuildContext context) {  
    final container = Container(  
        color: Colors.grey,  
        width: 50,  
        height: 50,  
        margin: const EdgeInsets.all(5), // because of this, the size is 60x60  
    );  
  
    return Center(  
        child: SizedBox(  
            width: 160,  
            child: Row(  
                children: [  
                    container,  
                    container,  
                    container,  
                    ],  
                ),  
            ),  
        );  
}
```

The total available space is determined by the [SizedBox](#) widget, and the containers don't fit since we also need to consider their margins. Other than reporting an error, the framework also adds this striped box to indicate where the overflow took place:



Figure 14.3: An overflow error due to be absence of enough space to render the widgets.

If you don't want to scroll, you can solve this problem by wrapping the containers in an [Expanded](#) widget. It expands its child so that it fills all of the available space without going out of bounds:

```
Row(  
  children: [  
    Expanded(child: container),  
    Expanded(child: container),  
    Expanded(child: container),  
  ],  
,
```

Alternatively, you can also use the `Flexible` widget. It has the same purpose as `Expanded` but it doesn't require the child to fill all of the available space:

```
Row(  
  children: [  
    Flexible(child: container),  
    Flexible(child: container),  
    Flexible(child: container),  
  ],  
,
```

If you don't want to scroll and you don't want to constrain the children using either `Expanded` or `Flexible`, then you can try the `Wrap` widget. It lays outs its children one by one, like a `Column` or a `Row`, but when it runs out of space it just wraps to the next line:

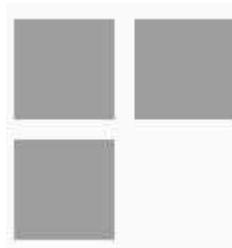


Figure 14.4: `Wrap` puts widgets on a new line rather than letting them overflow.

Since the containers don't fit within the `SizedBox`, we use `Wrap` to avoid overflowing issues:

```
SizedBox(  
  width: 160,  
  child: Wrap(  
    children: [  
      container,  
      container,  
      container,  
    ],  
,  
,
```

It has the `space` property to control spacing between widgets on the same row and `runSpacing` for the spacing between lines.

When you need to render a grid of widgets, rows and columns can help, but they can quickly end up in a complicated and deeply nested structure. In such cases, prefer using the `Table` widget:

```
Table(  
  children: [  
    for (var i = 0; i < 5; ++i)  
      TableRow(  
        children: [  
          for (var j = 0; j < 10; ++j)  
            Center(  
              child: Container(  
                margin: const EdgeInsets.all(5),  
                padding: const EdgeInsets.all(5),  
                color: Colors.black26,  
                child: Text('$i, $j'),  
              ),  
            ),  
          ],  
        ),  
      ],  
    ),  
  ),
```

The `Table` widget can resize children to accommodate all of its contents. You can also specify how widgets are vertically aligned and the relative horizontal widths of the columns. For example, with the above code, the result is the following:

0, 0	0, 1	0, 2	0, 3	0, 4	0, 5	0, 6	0, 7	0, 8	0, 9
1, 0	1, 1	1, 2	1, 3	1, 4	1, 5	1, 6	1, 7	1, 8	1, 9
2, 0	2, 1	2, 2	2, 3	2, 4	2, 5	2, 6	2, 7	2, 8	2, 9
3, 0	3, 1	3, 2	3, 3	3, 4	3, 5	3, 6	3, 7	3, 8	3, 9
4, 0	4, 1	4, 2	4, 3	4, 4	4, 5	4, 6	4, 7	4, 8	4, 9

Figure 14.5: Using a `Table` widget to arrange widgets in a grid.

If you wanted to give a fixed width to the columns, you'd need to set `defaultColumnWidth` with a specific value. For example:

```
defaultColumnWidth: const FixedColumnWidth(60),
```

The `FixedColumnWidth` type, as the name suggests, sizes the column to a specific number. This is the cheapest way to size a column but the less flexible one in terms of responsiveness:

0, 0	0, 1	0, 2	0, 3	0, 4	0, 5	0, 6	0, 7	0, 8	0, 9
1, 0	1, 1	1, 2	1, 3	1, 4	1, 5	1, 6	1, 7	1, 8	1, 9
2, 0	2, 1	2, 2	2, 3	2, 4	2, 5	2, 6	2, 7	2, 8	2, 9
3, 0	3, 1	3, 2	3, 3	3, 4	3, 5	3, 6	3, 7	3, 8	3, 9
4, 0	4, 1	4, 2	4, 3	4, 4	4, 5	4, 6	4, 7	4, 8	4, 9

Figure 14.6: Using a `Table` with a fixed columns width.

You could have also used `IntrinsicColumnWidth`, to size the columns according with the intrinsic dimensions of all the cells, or `FractionalColumnWidth`, to size the column to a fraction of the total table width.

### 14.1.2 Understanding constraints

There are two main categories of widgets in Flutter: the ones that expand to fill the available space and those with fixed sizes. You must always make sure that constraints are well-defined to avoid overflow or layout errors. In particular:

- a widget gets its constraints from the parent;
- a widget can size itself but has to stay within the constraints imposed by the parent;
- a widget gets positioned from its parent;
- a widget doesn't know its position on the screen (because the position is always determined by the parent);
- a widget tells its parent about its own size.

You can remember these rules with the following motto: "*Constraints go down, sizes go up, parent sets position*"<sup>87</sup>. The parent widget tells its child (or children) how much space they have available

---

<sup>87</sup> Constraints go down. Sizes go up. Parent sets position.

and where they need to be placed. Let's start with a simple case where the parent is the screen and the `Container` tries (without success) to be 100x100:

```
void main() {  
  runApp(  
    MaterialApp(  
      home: Container(  
        width: 100,  
        height: 100,  
        color: Colors.lightGreen,  
      ),  
    ),  
  );  
}
```

You'd probably expect the container to be of the given size, but it actually covers the entire page. Even if `Container` wants to be 100x100, `MaterialApp` forces its child to fill all the available space:

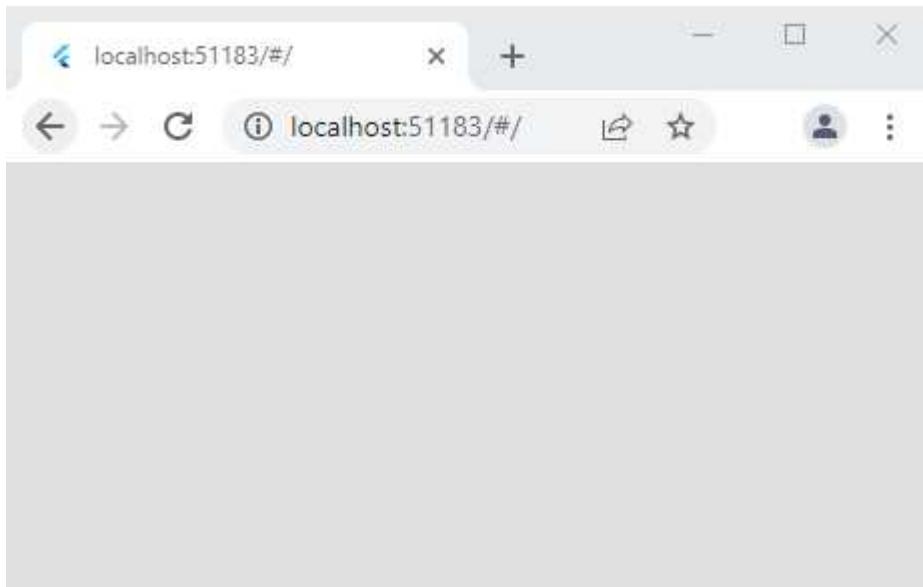


Figure 14.7: The `Container` takes all of the available space.

To fix this, we need to wrap our `Container` in a widget that lets it have any size it wants (but within the screen's dimension). There are a few solutions to this problem, but none is the best because it depends on what you want to achieve. For example:

1. You could wrap it with `Center`, which expands to fill all of the available space and lets its child have its own size:

```
Center(  
  child: Container(  
    width: 100,  
    height: 100,  
    color: Colors.black12,  
  ),  
,
```

The same result could have also been achieved with `Align`, which `Center`'s super type.

2. Wrap it within a `Scaffold`, which also expands to fill all of the available space and lets the body have its own sizes:

```
Scaffold(  
  body: Container(  
    width: 100,  
    height: 100,  
    color: Colors.black12,  
  ),  
,
```

3. Even if it's not a great idea, you could have also wrapped the container in a `Row` or a `Column`. They expand to fill all of the available space and let the children have their own sizes.

```
Row(  
  children: [  
    Container(  
      width: 100,  
      height: 100,  
      color: Colors.black12,  
    ),  
  ],  
,
```

Whatever choice you make, the point is still the same: you need a widget that expands to fill all of the available space and lets the child set its own size.

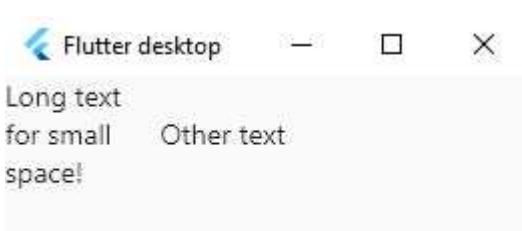
#### Note

Knowing this rule is good, but it's not enough. Each widget can behave differently so it's always worth to read the inline documentation or visiting [docs.flutter.dev](https://docs.flutter.dev).

Remember that `Row` and `Column` don't impose any constraints on the children, so they may just be too big for the viewport. For example, if you placed a long string within a `Row`, you might see the overflow error because `Text` is free to size itself:

Code	Result
<pre>SizedBox(   width: 150,   child: Row(     children: const [       Text('Long text for '           'small space!'),       SizedBox(width: 5),       Text('Other text'),     ],   ), ),</pre>	 <p>Flutter desktop</p> <p>Long text for small space! Other text</p> <p>RIGHT OVERFLOW!</p>

We can use `Expanded` or `Flexible`, as we have already seen, to impose additional constraints on the children so that they're "aware" of the total available space:

Code	Result
<pre>SizedBox(   width: 150,   child: Row(     children: const [       Expanded(         child: Text('Long text for '             'small space!'),       ),       SizedBox(width: 5),       Expanded(         child: Text('Other text'),       ),     ],   ), ),</pre>	 <p>Flutter desktop</p> <p>Long text for small space!</p> <p>Other text</p>

`Expanded`, in this case, automatically gives equal space to both widgets. Its `flex` property can be used to prioritize the available space for specific children (the same also applies for `Flexible`). When you are not able to size a widget as you wish, the reason often is that its parent is not allowing

it to do so. Similarly, overflow errors are caused by the child not being constrained by the parent to we need to wrap the widget with one that actually imposes constraints.

#### 14.1.3 Box constraints

You can easily impose specific constraints on your widgets using `ConstrainedBox`. Any child won't be able to be smaller or bigger than the given dimensions. For example:

```
Scaffold(  
  body: ConstrainedBox(  
    constraints: const BoxConstraints(  
      minWidth: 80,  
      maxWidth: 150,  
      minHeight: 80,  
      maxHeight: 150,  
    ),  
    child: Container(  
      width: 500,  
      height: 500,  
      color: Colors.black12,  
      child: const Text('Hello'),  
    ),  
  ),  
,
```

Even if the `Container` size is 500x500, it ends up being 150 wide and 80 tall because the parent "decides" the constraints for the child. Keep in mind that `ConstrainedBox` just adds constraints to the existing ones; it does not replace them. As such, if we removed the `Scaffold`, the container would take the entire space again:

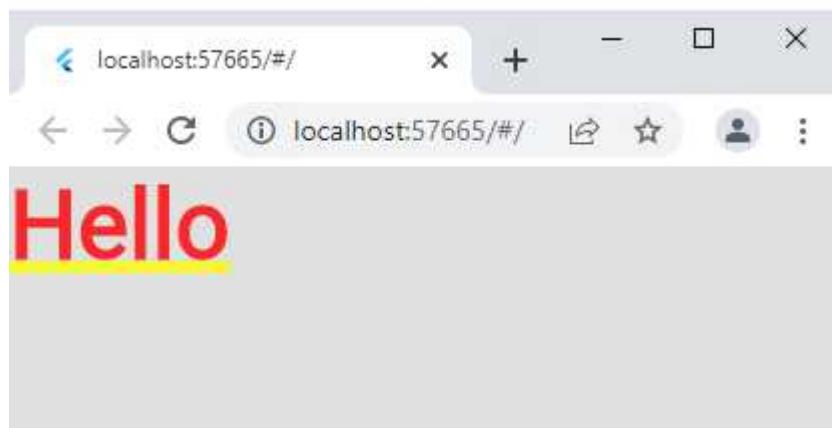


Figure 14.10: A constrained box without an outer `Scaffold`.

The reason is still the same: there is no parent that fills the entire space and makes the child have its own size. To do so, we would need to add the [Scaffold](#) back or use another similar widget, such as [Align](#) or [Center](#). There are two other constructors you may want to consider:

- [BoxConstraints.tight](#): A tight constraint requires the box to have an exact size and thus the children don't have room to change their dimensions.
- [BoxConstraints.loose](#): A loose constraint only sets the maximum width and height but gives room to the children to be as small as it desires.

With a loose constraint of 150 for example, the [Container](#) size could range from 0x0 to 150x150. If we tried to make it bigger, such as 310x255, no errors would happen but the widget size would exactly be 150x150 because of the rules imposed by the parent (always remember that “*constraints go down*”).

### Note

---

In some cases, you may get an error caused by unbounded constraints from the parent widget. It means that either the maximum width or height are set to [double.infinity](#). In these cases, you need to add constraints.

You could for example use [Center](#), a [SizedBox](#), a [ConstrainedBox](#) or any other widget that imposes fixed boundaries to its child.

Overall, there are three categories of widgets:

1. The ones that try to be as big as possible, such as [Center](#) or [ListView](#) (more on scrollable lists in [14.3 Scrollable Widgets](#)).
2. The ones that try to be as small as possible (to have the same size as their child or children), such as [Opacity](#) or a [Row/Column](#) created with [MainAxisSize.min](#).
3. The ones that try to have a fixed size, such as [Text](#) or [Image](#) (more on images and assets in [chapter 16 – Assets and images](#)).

There also are a few exceptional cases, such as [Container](#): it tries to be as big as possible unless you give it a width and/or a height. If you are unsure about the constraints and the behavior of a widget, check out the inline documentation which always provides precious information.

## 14.2 Building responsive layouts

High-quality applications are responsive because they automatically rearrange their contents to best fit the device and screen they're running on. This is particularly important in Flutter because it runs on various platforms. For example, imagine having to build an application with a scrollable list like this:



Figure 14.11: A scrollable list on a mobile device (vertically oriented).

Figure 14.11 shows the scrollable list on a mobile phone when in portrait mode, which looks quite good. However, as soon as the device is rotated in landscape mode, notice that there is too much space on the right:



Figure 14.12: The same scrollable list on the rotated device.

Furthermore, fewer items are visible because of the reduced viewport height. It would be better to arrange items to fill the entire space rather than stay on a single column. In other words, we would like the application to behave like this:



Figure 14.13: The list is replaced by a grid when there is enough horizontal space.

This is much better because the application distributes its UI elements across the entire area, even when dimensions change. This is not only the case of a device rotation; the screen sizes may also change when your application runs on a desktop device and you resize the window for example. To obtain this result, we should start by creating a file called `breakpoints.dart`:

```
// 'breakpoints.dart' file  
  
const doubleColumnBreakpoint = 450;
```

We can put any breakpoint we want within the file. In our case, we have decided that `450` is a good value to determine when using a single-column or a double-column layout. The `LayoutBuilder` widget is what we need to apply this breakpoint:

```
@override  
Widget build(BuildContext context) {  
    return LayoutBuilder(  
        builder: (BuildContext context, BoxConstraints constraints) {  
            if (constraints.maxWidth >= doubleColumnBreakpoint) {  
                return const DoubleColumnList();  
            }  
  
            return const SingleColumnList();  
        },  
    );  
}
```

The `LayoutBuilder` widget has a `builder` function that is called by the framework whenever the parent widget constraints change. In other words, this widget is used to listen for screen dimension changes and update the UI accordingly. Its builder is called in various situations:

- the first time the widget is laid out;
- when the parent passes different layout constraints (because the parent size changed);
- when the parent rebuilds the `LayoutBuilder`;
- when a dependency of the builder (such as an inherited widget) changes.

The `builder` function is never called if the parent repeatedly passes the same constraints. Our recommendation is to create a dedicated file with all breakpoints needed by your application. Use those constants in the `LayoutBuilder` widget to make the UI responsive.

#### 14.2.1 The `MediaQuery` type

The `MediaQuery` type is useful to obtain data about the application as a whole, such as the total screen size or its orientation. While a `LayoutBuilder` exposes the parent widget constraints, the `MediaQuery` widget doesn't care about its parent. For example, you can use it to obtain the actual screen size:

```
@override  
Widget build(BuildContext context) {  
  final size = MediaQuery.of(context).size;  
  
  // The overall window width and height  
  return Text('Screen size (logical pixels): ${size.width}x${size.height}');  
}
```

As we've said at the beginning of this chapter, Flutter uses logical pixels (not raw pixels). You can also use `MediaQuery` to tell if the current device is in landscape or portrait mode, for example:

```
@override  
Widget build(BuildContext context) {  
  if (MediaQuery.of(context).orientation == Orientation.landscape) {  
    return const Text('Landscape mode');  
  }  
  
  return const Text('Portrait mode');  
}
```

This value does not tell you if you're using a mobile or a desktop device. It just indicates whether the current screen width is equal to or smaller than the height, and that's it. If `BuildContext` is not available, you can use `WidgetsBinding` to retrieve some information too:

```
// Requires 'context'  
final ts1 = MediaQuery.of(context).textScaleFactor;  
  
// Does not require 'context'  
final ts2 = WidgetsBinding.instance.platformDispatcher.textScaleFactor;  
  
debugPrint('$ts1 | $ts2'); // Both print the same value
```

Note that `MediaQuery` is an inherited widget and it's constantly updated whenever its properties change. As such, you can listen for changes using a `ListenableBuilder`. The platform dispatcher object cannot be listened to and has a lower lever API. We recommend using `MediaQuery` and the context as much as possible.

#### 14.2.2 Good practices

To make an application responsive (breakpoints aside), you shouldn't hard-code layout values such as sizes or fonts. We recommend using `LayoutBuilder` when you need to calculate the free space within the widget tree. In particular:

- Use `LayoutBuilder` to create responsive widgets that change their contents according with the parent's constraints. This widget is able to tell you the exact free space you have thanks to the `BoxConstraint` object passed by the parent.
- Use `MediaQuery` to get information about the device and the application "as a whole". This type is an inherited widget that knows nothing about widgets and box constraints. As such, it generally isn't helpful to make widgets responsive.

Let's make an example to emphasize the difference between the two classes. Look at the below image, which reports two different sizes for the same window:



Figure 14.14: The width of a `MediaQuery` vs the width of a `LayoutBuilder`.

While `MediaQuery` reports the total window size, the `LayoutBuilder` widget reports the available space considering the parent constraints. This is the code for *Figure 14.14*:

```
@override
Widget build(BuildContext context) {
  final windowSize = MediaQuery.of(context).size;

  return Scaffold(
    body: Center(
      child: Padding(
        padding: const EdgeInsets.all(20),
        child: LayoutBuilder(
          builder: (BuildContext context, BoxConstraints constr) {
            final window = '${windowSize.width}x${windowSize.height}';
            final parent = '${constr.maxWidth}x${constr.maxHeight}';

            return Text('$window vs $parent');
          },
        ),
      ),
    ),
  );
}
```

Notice that the builder is aware of the actual free space because it has a `BoxConstraints` object, which considers the `Padding` constraints (hence why 460x20). `MediaQuery` instead is not aware of `Padding` because it holds information about the application screen (hence why 500x60, which ignores the padding). More classes can help you make responsive applications, such as:

- **AspectRatio**: This widget tries to size the child to a specific aspect ratio. For example, if you wanted a container to have a 16:9 (width : height) aspect ratio, you would need to use this widget as follows:

```
const Center(
  child: AspectRatio(
    aspectRatio: 16 / 9,
    child: ColoredBox(
      color: Colors.blue,
    ),
  ),
),
```

A `ColoredBox` is basically a `Container` where you've only set its `color` property. With this code, the box size is 16/9 (width : height ratio) of the available space.

- **FittedBox**: This widget scales and positions its child according to the fit logic. For example, you might want to render an image in a box that fills the entire space (and distorts the aspect ratio):

```
SizedBox(
  width: 256,
  height: 85,
  child: FittedBox(
    fit: BoxFit.fill,
    child: Image.asset('image.png'),
  ),
),
```

The `BoxFit.cover` instead makes the image as small as possible but preserves the aspect ratio and fills the remaining space. Make sure to check the official documentation<sup>88</sup> for all the possible fitting configurations. Images and assets will be covered in *chapter 16 – Assets and images*.

- **FractionallySizedBox**: This widget sizes its child to a fraction of the total available space. It's a `SizedBox` that uses relative dimensions rather than hard-coded ones:

```
const FractionallySizedBox(
  widthFactor: 1 / 2,
  heightFactor: 1 / 2,
  child: ColoredBox(
    color: Colors.blueAccent,
  ),
),
```

The box in the example takes half of the parent size.

- **CustomPaint**: Consider rendering complex pieces of UI by freely drawing on the canvas provided by the `CustomPainter` class (*chapter 11.4 – CustomPaint and CustomPainter*).

Don't forget about `Expanded` or `Flexible`, which are used in rows and columns to arrange the free space without hard-coding values. In general, hard-coded dimensions are only good when a widget has fixed dimensions by design.

<sup>88</sup> <https://api.flutter.dev/flutter/painting/BoxFit.html>

## 14.3 Scrollable widgets

The scrollable version of `Row` and `Column` is called `ListView`, which is extremely popular and easy to use. It displays its children one after the other in the given direction and if they don't fit within the viewport, you can scroll to reveal the others. For example:

```
@override  
Widget build(BuildContext context) {  
  return Scaffold(  
    body: ListView(  
      children: [  
        for (var i = 0; i < 90000; ++i)  
          Text('Index $i'),  
      ],  
    ),  
  );  
}
```

Since all those `Text` widgets won't fit within the viewport, you can scroll to reveal all the others. In the example, scrolling happens along the vertical axis, which is the default one. However, you can also horizontally scroll the list with a simple change:

```
@override  
Widget build(BuildContext context) {  
  return Scaffold(  
    body: ListView(  
      scrollDirection: Axis.horizontal, // Horizontal scroll  
      children: [  
        for (var i = 0; i < 90000; ++i)  
          Text('Index $i'),  
      ],  
    ),  
  );  
}
```

You can see `ListView` as the scrollable version of `Column` and `ListView` with `Axis.horizontal` as the scrollable version of `Row`. Other than using the default constructor (like in the above example), there also are other two popular and more efficient ways to initialize a list:

1. `ListView.builder`: This constructor is used to build children on demand and is particularly efficient<sup>89</sup> when the source list is long (potentially infinite). It is a very good practice to use

---

<sup>89</sup> <https://api.flutter.dev/flutter/widgets/ListView-class.html>

`ListView.builder` (rather than the default `ListView` constructor) with large lists because the builder is called only for those children that are visible. For example:

```
class _ExampleState extends State<Example> {
    final random = Random();

    late final values = List<int>.generate(
        8000,
        (index) => random.nextInt(index + 1),
    );

    @override
    Widget build(BuildContext context) {
        return Scaffold(
            body: ListView.builder(
                itemCount: values.length,
                itemBuilder: (context, index) => Text('${values[index]}'),
            ),
        );
    }
}
```

If you could specify the `itemExtent` of the list, the rendering would be even more efficient. In the example, each child determines the area to occupy but if `itemExtent` is given, the scroll machinery would save computational time. For example:

```
Scaffold(
    body: ListView.builder( // Lazily builds widgets
        itemExtent: 20, // <-- this improves performance
        itemCount: values.length,
        itemBuilder: (context, index) {
            return Text('${values[index]}');
        },
    ),
),
```

It's not always easy to calculate the `itemExtent` value but, whenever you can, try to define it. A `ListView` always renders all of its children, even those that are not visible. However, the `ListView.builder` constructor only builds the currently visible children. All the other children will only be built when (and if) they will become visible.

2. `ListView.separated`: It does the same as `ListView.builder` but it has an additional (and required) argument called `separatorBuilder`. It returns a widget that is inserted between two consecutive children. Consider this example:

```

Scaffold(
  body: ListView.separated( // does Lazy builds like ListView.builder
    itemCount: values.length,
    separatorBuilder: (context, index) {
      return const Divider(); // added between consecutive widgets
    },
    itemBuilder: (context, index) {
      return Text('${values[index]}');
    },
  ),
),
),

```

A [Divider](#) widget (which just paints a grey line) is added between two consecutive children. This constructor does not have the possibility to define the extent of the children.

When you have a widget that is normally visible but there may be some special cases where it might not, you could use a [SingleChildScrollView](#). For example, consider this packed column:

```

Scaffold(
  body: Column(
    mainAxisSize: MainAxisSize.min,
    children: const [
      Text('Hello'),
      SizedBox(height: 200),
      Text('World'),
    ],
  ),
),
)

```

This is generally entirely visible on a mobile device but on desktop you can resize the window as much as you want. If you reduce the window height until it almost collapses to the title bar, the widget will overflow. The fix is simple:

```

Scaffold(
  body: SingleChildScrollView(
    child: Column(
      mainAxisSize: MainAxisSize.min,
      children: const [
        Text('Hello'),
        SizedBox(height: 100),
        Text('World'),
      ],
    ),
  ),
),
)

```

With this change, when you shrink the window height on desktop or web, no overflow errors occur because of the scrolling behavior imposed by `SingleChildScrollView`. Make sure to check out the official documentation<sup>90</sup> for a complete coverage of all the edge cases.

### 14.3.1 Using a ScrollController

The `ScrollController` class is used to programmatically control the scroll behavior of a scrollable widget. It's generally a good practice to create one controller per scrollable widget. For example:

```
class ExampleState extends State<Example> {
    final controller = ScrollController();

    @override
    void dispose() {
        controller.dispose();
        super.dispose();
    }

    @override
    Widget build(BuildContext context) {
        return Scaffold(
            body: ListView.builder(
                controller: controller,
                itemExtent: 25,
                itemCount: values.length,
                itemBuilder: (context, index) => Text('${values[index]}'),
            ),
        );
    }
}
```

As it happens with any other controller, a `ScrollController` has to be disposed. As such, you should always define it in a stateful widget and override the `dispose` method. A controller is helpful for various use cases, such as programmatically scrolling the list up to a certain point:

```
controller.animateTo(
    300,
    duration: const Duration(seconds: 1),
    curve: Curves.ease,
);
```

---

<sup>90</sup> <https://api.flutter.dev/flutter/widgets/SingleChildScrollView-class.html>

If you make this call with a button press, for example, the list is animated to scroll from its current value to the given one. In our case, we're scrolling the list by `300` logical pixels along the scroll axis. Alternatively, we could also have used `jump`:

```
controller.jumpTo(300);
```

It has the same behavior as `animateTo` but with the only difference that `jump` does not animate the scroll (it immediately brings to the target position). Controllers are helpful for making calculations based on the current scroll position. For example:

- `controller.offset`: returns the distance from the start to the current scroll position;
- `controller.position`: returns a `ScrollPosition` object that holds information about the current position of the scrollable widget.

Since a `ScrollController` is a `ChangeNotifier`, you can use a `ListenableBuilder` widget to listen for changes in the scroll position. Some operations, such as reading the scroll `offset`, require the controller to be used by a single widget. For example, this code will throw a runtime exception:

```
children: [
  Expanded(
    child: ListView.builder(
      controller: controller,
      itemCount: values.length,
      itemBuilder: (_, index) => Text('${controller.offset}'),
    ),
  ),
  Expanded(
    child: ListView(
      controller: controller,
      children: const [],
    ),
  ),
],
```

The problem is that `controller` is assigned to two lists, and they both read the `position` value. If you read any other property that doesn't use the position, then no runtime errors would happen. This code for example works fine:

```
ListenableBuilder(
  listenable: controller,
  builder: (_, __) => Text('${controller.hasClients}'),
),
```

The `hasClients` property tells whether the scroll controller is attached to a scrollable widget or not. Two or more widgets could have read `hasClients` without errors. In general, scroll controllers are mostly used to move to a specific location or read position data.

### 14.3.2 Other scroll-related widgets

This section showcases a series of scrollable widgets that are part of the framework. Keep in mind that not all of them can have a `ScrollController` attached so make sure to check out the official documentation for more information on whether it's available or not.

#### 14.3.2.1 Scrollbar

On mobile devices, scrollbars never appear. On desktop platforms, scrollbars only appear while the user is scrolling. This is the default behavior, but it can be changed with the `Scrollbar` widget. For example:

```
Scrollbar(  
    thumbVisibility: true,  
    child: ListView.builder(  
        itemExtent: 25,  
        itemCount: values.length,  
        itemBuilder: (context, index) {  
            return Text('${values[index]}');  
        },  
    ),  
,  
)
```

The `thumbVisibility` parameter overrides the default behavior and makes the scrollbar always visible, even on mobile platforms. You can also customize the widget to always show the track and change the thumb's border radius:

```
const Scrollbar(  
    thumbVisibility: true,  
    thickness: 6,  
    radius: const Radius.circular(0),  
    trackVisibility: true,  
    child: MyScrollableWidget(),  
)
```

You shouldn't show scrollbars on mobile devices. The `Scrollbar` widget is good when you want to customize the scroll bar of individual scrollable widgets. In some cases, you might need to apply a custom scroll bar across the entire application. In this case, you must create a new global scroll configuration and assign it to your root `MaterialApp` or `CupertinoApp` widget. For example:

```
MaterialApp.router(  
  scrollBehavior: const DesktopAlwaysVisibleBehavior(),  
)
```

The `scrollBehavior` parameter is used to configure all scrollable widgets in your application. In our example, we want to hide scrollbars on mobile devices but make them always visible on desktop platforms (even if the user is not scrolling). This is how the configuration class might look like:

```
class DesktopAlwaysVisibleBehavior extends MaterialScrollBehavior {  
  const DesktopAlwaysVisibleBehavior();  
  
  @override  
  Widget buildScrollbar(  
    BuildContext context,  
    Widget child,  
    ScrollableDetails details,  
  ) {  
    // Do nothing with horizontal scroll  
    if (axisDirectionToAxis(details.direction) == Axis.horizontal) {  
      return child;  
    }  
  
    // Horizontal scroll configuration for desktop and mobile  
    switch (Theme.of(context).platform) {  
      case TargetPlatform.linux:  
      case TargetPlatform.macOS:  
      case TargetPlatform.windows:  
        return Scrollbar(  
          controller: details.controller,  
          thumbVisibility: true, // Makes it always visible on desktop  
          child: child,  
        );  
      case TargetPlatform.android:  
      case TargetPlatform.fuchsia:  
      case TargetPlatform.iOS:  
        return child;  
    }  
  }  
}
```

We copied the `buildScrollbar` implementation from the superclass and just made the scrollbar always visible on desktop, even when the user is not scrolling. On mobile platforms instead, the scrollbar never appears. If you're using a `CupertinoApp` widget, the procedure is the same but you have to extend `CupertinoScrollBehavior` instead.

#### 14.3.2.2 NotificationListener<T>

Some widgets, such as scrollable ones, dispatch notifications that move up in the tree and can be listened by parents. For example, `ListView` emits notifications of type `ScrollNotification` which give the parents, if interested, information about the scrolling status. In this example, we are logging into the console several scrolling information:

```
return NotificationListener<ScrollNotification>(
    onNotification: (notification) {
        if (notification is ScrollStartNotification) {
            debugPrint('User started scrolling...');
        }

        if (notification is ScrollUpdateNotification) {
            final currPosistion = notification.metrics.pixels;
            final isAtEdge = notification.metrics.atEdge;
            final scrollDistance = notification.scrollDelta;

            debugPrint(
                'User is scrolling.'
                ' > Position: $currPosistion'
                ' > Is at the edges? $isAtEdge'
                ' > How much scroll: $scrollDistance',
            );
        }

        if (notification is ScrollEndNotification) {
            debugPrint('User stopped scrolling!');
        }
    }

    return true;
},
child: Scaffold(
    body: ListView.builder(
        itemExtent: 25,
        itemCount: values.length,
        itemBuilder: (context, index) {
            return Text('${values[index]}');
        },
    ),
),
);
```

This is very useful when parents need to be quickly aware of the scroll position of a child. Note that `ScrollNotification` is abstract because the scrolling state is determined by its subclasses. There are five of them:

- `ScrollStartNotification` is emitted when a scrollable widget has started scrolling;
- `ScrollUpdateNotification` is emitted while the scrollable widget is changing position;
- `ScrollEndNotification` is emitted when the scrollable widget stops scrolling;
- `UserScrollNotification` is emitted when the user changes the scroll direction;
- `OverscrollNotification` is emitted when the scrollable widget has arrived at the edge but the user is still trying to scroll.

By default, the notification keeps propagating up in the tree unless we explicitly tell it to stop. To do so, you need to return `true` from the callback:

```
return NotificationListener<ScrollNotification>(
  onNotification: (_) => true; // Stops the propagation
  child: const MyScrollableWidget(),
);
```

If we returned `false`, the scroll notification would continue propagating up in the tree. To create custom notifications, extend `Notification` and call `dispatch` to “send” it:

```
// Create your own notification...
class MyNotification extends Notification { /* ... code ... */}

// ...and then use 'dispatch' within the "build" method
ElevatedButton(
  onPressed: () => const MyNotification().dispatch(context),
  child: const Text('Send'),
),
```

While inherited widgets propagate information down in the widget tree, notifications are used to propagate information up in the widget tree.

#### 14.3.2.3 RefreshIndicator

The `RefreshIndicator` widget implements a popular material component that scrolls down from the top of the page, shows a spinning indicator and then disappears. In particular, this is used to let the users know that a list is being refreshed. For example, consider this code:

```
RefreshIndicator(
  onRefresh: _onRefresh,
  child: ListView.builder(
    itemCount: myList.length,
    itemBuilder: (_, index) => Text('Item ${myList[index]}'),
  ),
),
```

The `_onRefresh` callback is used to run some logic that updates the list itself and can be awaited. For example, it might look like this:

```
Future<void> _onRefresh() async {
  try {
    final List<int> newList = await fetchNewData();

    setState(() {
      myList = newList;
    });
  } on Exception {
    // Exception handling...
  }
}
```

In the example, we're awaiting a network call to refresh the list contents and then we use `setState` to update the list. To trigger the refresh callback, you need to be at the top of the list and overscroll to the bottom to see the indicator:

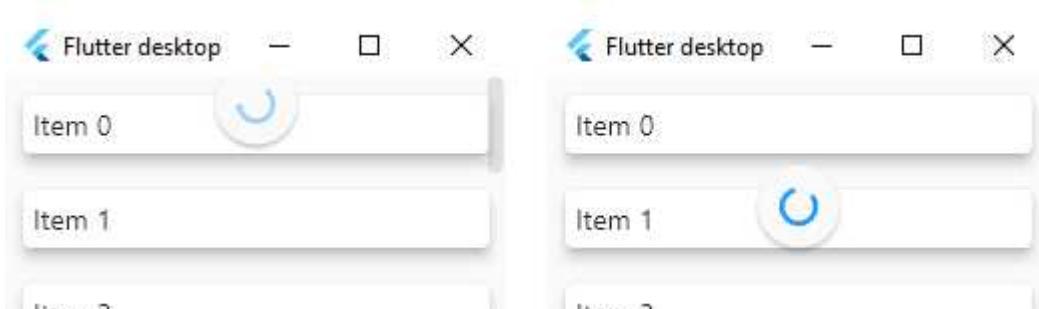


Figure 14.15: The `RefreshIndicator` appearing from the top of a `ListView`.

The indicator is opaque while dragged down but as soon as you release it (if it was dragged enough), the animation starts. When `_onRefresh` completes, the indicator automatically disappears from the screen with a scale animation. The `triggerMode` parameter defines how the indicator can be triggered:

- `RefreshIndicatorTriggerMode.onEdge` (the default) : The indicator can be triggered only if the scrollable widget is at the top edge.
- `RefreshIndicatorTriggerMode.anywhere`: The indicator can be triggered regardless of the position of the scrollable widget.

We recommend sticking with the default value since the refresh indicator is usually known to be draggable only when the list is at the top edge of the screen.

#### 14.3.2.4 ReorderableListView

As the name suggests, `ReorderableListView` is basically a `ListView` whose children can be moved up and down via dragging gestures. The widget automatically adds black “hamburger” icons on the right: you need to tap or click on them and start dragging the item.

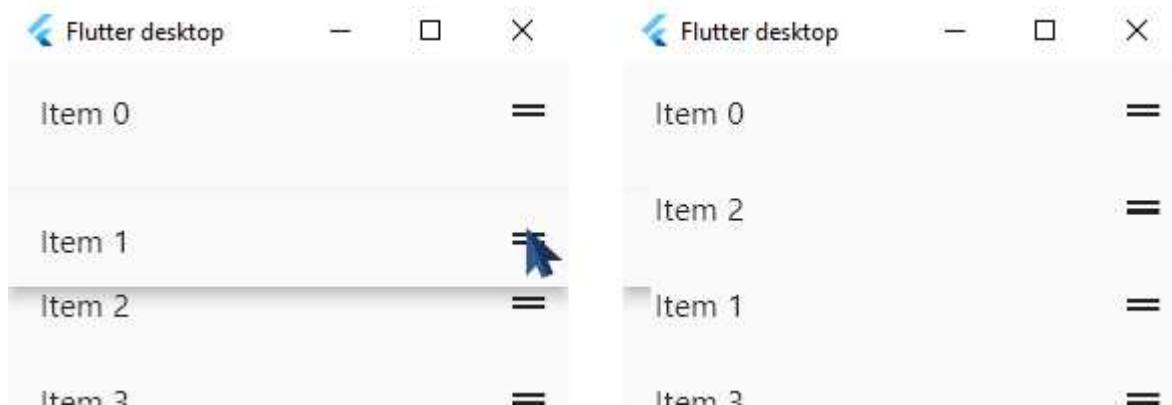


Figure 14.16: An item of a `ReorderableListView` being dragged to a new position.

Each item must have a `ValueKey`, holding a unique string, so that Flutter can correctly identify the entries in the list and rearrange them. We recommend passing the index to the `key` parameter so that each child is uniquely identified by a value. For example:

```
class _ExampleState extends State<Example> {
  final items = List<ListTile>.generate(
    100,
    (index) => ListTile(
      key: Key('$index'), // Builds a 'ValueKey'
      title: Text('Item $index'),
    ),
  );
}

@Override
Widget build(BuildContext context) {
  return Scaffold(
    body: ReorderableListView(
      onReorder: (int oldIndex, int newIndex) { /* reorder logic */ },
      children: items,
    ),
  );
}
```

The `onReorder` callback is used by the list to report that an item has been moved to a new position. This callback assumes that the implementation removes the corresponding item at `oldIndex` and reinserts it at `newIndex`. Because of this precondition, the reordering logic generally looks like this:

```
onReorder: (int oldIndex, int newIndex) {
    if (oldIndex < newIndex) {
        newIndex -= 1;
    }

    setState(() {
        final item = items.removeAt(oldIndex);
        items.insert(newIndex, item);
    });
},
```

When `oldIndex` is smaller than `newIndex`, removing the item at `oldIndex` from the list will reduce the length by one. Inside `setState`, we make sure to remove the item in the old position and insert it in the new one.

#### 14.3.2.5 GridView

While a `ListView` places all of its children in a single column, a `GridView` places its children in two or more columns. The `scrollDirection` property determines whether a grid has to vertically or horizontally scroll. For example:

```
class _ExampleState extends State<Example> {
    final children = List<Widget>.generate(100, (i) => Text('$i'));

    @override
    Widget build(BuildContext context) {
        return Scaffold(
            body: GridView.count(
                crossAxisCount: 4,
                children: children,
            ),
        );
    }
}
```

This example creates a four-column table that scrolls when needed. Children can be spaced using the `mainAxisSpacing` or `crossAxisSpacing` properties. `GridView.extent` is another popular constructor that gives each child the maximum available space in which it can expand. For example:

```
GridView.extent(  
    maxCrossAxisExtent: 50,  
    children: children,  
) ,
```



Figure 14.17: A GridView with an extent of 50.

```
GridView.extent(  
    maxCrossAxisExtent: 25,  
    children: children,  
) ,
```

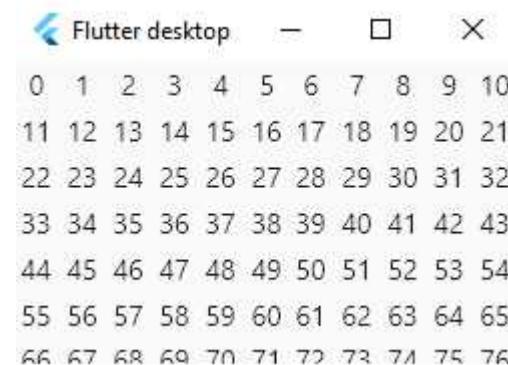


Figure 14.18: A GridView with an extent of 25.

The default constructor uses slivers, which will be covered in the next section.

#### 14.3.2.6 ScrollPhysics

All scrollable widgets have a `physics` parameter to determine how the scrolling machinery should behave. The `NeverScrollablePhysics`, for example, does not allow the user to scroll:

```
// Works with any scrollable widget, not only 'ListView'  
ListView(  
    physics: const NeverScrollableScrollPhysics(),  
    children: [  
        // children...  
    ],  
) ,
```

In this example, the `ListView` does not scroll. The `AlwaysScrollableScrollPhysics` physic is the default value, which always enables scrolling. Flutter (currently) implements five physics for its scrollable widgets:

- `FixedExtentScrollPhysics`: This physics is used to always land directly on an item rather than anywhere in the list. This is often used with `ListWheelScrollView` to obtain a “slot machine” behavior, where the scrolling always ends up with a selected item:

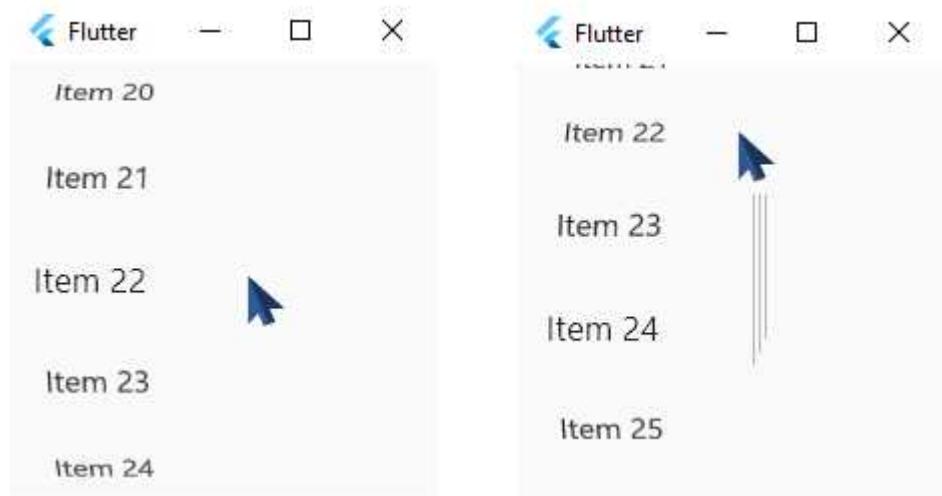


Figure 14.19: The wheel effect of a `ListWheelScrollView` being scrolled.

As you can see, `ListWheelScrollView` creates a box in which children scroll as if they were on a wheel (very similar to a slot machine display). All children must be the same size so the `itemExtent`, which indicates the size of each child, is required:

```
// A normal “ScrollController” would NOT work
final controller = FixedExtentScrollController();

@Override
Widget build(BuildContext context) {
  return ListWheelScrollView(
    controller: controller,
    physics: const FixedExtentScrollPhysics(),
    useMagnifier: true,
    magnification: 1.05,
    itemExtent: 50,
    children: children,
  );
}
```

The magnifier highlights the currently selected item at the center while the custom `physics` ensures that the scrolling always ends up selecting an item.

- `NeverScrollablePhysics`: This physics is used to disable scrolling on a scrollable widget.
- `AlwaysScrollableScrollPhysics`: the default behavior that enables free scrolling. On Android, overscrolls will be clamped with a glow effect. On iOS, overscrolls will load a spring that will return the scroll view to its normal range when released.
- `BouncingScrollPhysics`: This physics is for environments that allow the scroll offset to go beyond the contents but then bounce back to the edge. This is a typical iOS behavior.
- `ClampingScrollPhysics`: This physics prevents the scroll offset from going beyond the content's bounds.

You can combine multiple effects using the `parent` parameter. For example, the bouncing physics doesn't create an overscroll effect if the contents do not extend beyond the viewport size. You can add this behavior by *merging* two physics together:

```
physics: const BouncingScrollPhysics(  
  parent: AlwaysScrollableScrollPhysics(),  
,
```

The Flutter team recommends <sup>91</sup> to nest physics together using `parent` rather than creating your own subclasses, which may be challenging to implement and maintain.

### 14.3.3 Slivers

A sliver is a portion of a scrollable area whose behavior can be customized in a special way. In other words, when a scrollable widget doesn't do what you want, you can manually use slivers to create a particular behavior.

#### Note

Every scrollable widget in Flutter (such as `ListView` and `GridView`) uses slivers under the hood. Slivers themselves aren't very easy to use so that's why the framework has so many scrollable widgets, which just are convenient and ready-to-use wrappers of sliver configurations.

---

<sup>91</sup> <https://api.flutter.dev/flutter/widgets/ScrollPhysics-class.html>

Since scrollable widgets use slivers under the hood, we could for example replace all `ListView`s in our application with its sliver representation. Consider this simple code:

```
class _ExampleState extends State<Example> {
    final children = List<Widget>.generate(100, (i) => Text('$i'));

    @override
    Widget build(BuildContext context) {
        return Scaffold(
            body: ListView.builder(
                itemCount: children.length,
                itemBuilder: (_, index) => children[index],
            ),
        );
    }
}
```

To convert `ListView.Builder` into a sliver, we first need a `CustomScrollView` located above the scrollable widget. Then, we can use `SliverList` (which is what a `ListView` uses under the hood) to place its children on a single column:

```
class _ExampleState extends State<Example> {
    final children = List<Widget>.generate(100, (i) => Text('$i'));

    @override
    Widget build(BuildContext context) {
        return Scaffold(
            body: CustomScrollView(
                physics: const BouncingScrollPhysics(),
                slivers: [
                    SliverList(
                        delegate: SliverChildBuilderDelegate(
                            (context, index) => children[index],
                            childCount: children.length,
                        ),
                    ),
                ],
            );
    }
}
```

It's a bit more complicated to read, but this code is equivalent to the previous one where we used a simple `ListView`. This is a “conversion cheat sheet” of some standard scrollable widgets and their sliver equivalents:

- `ListView` is created using a `SliverList`;
- `GridView` is created using a `SliverGrid`;
- `ListView` with an `itemExtent` is created using a `SliverFixedExtentList`;
- to add opacity, use `SliverOpacity`;
- to add padding, use `SliverPadding`;
- to add a fade transition, use `SliverFadeTransition`.

Note that the children of a `CustomScrollView` widget can only be slivers. For example, you are not allowed to pass a `Column` because it is not a sliver. The following code compiles with success, but it throws a runtime error:

```
const CustomScrollView(
  slivers: [
    Column( // this causes a runtime error because 'Column' is not a sliver
      mainAxisAlignment: MainAxisAlignment.min,
      children: [
        Text('Hello'),
        SizedBox(
          height: 10,
        ),
        Text('World'),
      ],
    ),
  ],
),
```

A `SliverToBoxAdapter` sliver makes a non-sliver widget compatible with a `CustomScrollView`. To make the above run without runtime errors, we have to wrap the `Column` in the sliver adapter widget. For example:

```
slivers: [
  SliverToBoxAdapter(
    child: Column( // no error because 'Column' is inside the adapter
      mainAxisAlignment: MainAxisAlignment.min,
      children: [
        Text('Hello'),
        SizedBox(
          height: 10,
        ),
        Text('World'),
      ],
    ),
  ),
],
```

The code compiles and executes without errors because `SliverToBoxAdapter` adapts a single box widget to the scrollable view. In general:

- Try to reuse existing sliver widgets, such as `SliverList` or `SliverPadding`.
- Use `SliverToBoxAdapter` when you have to use a “custom” widget that does not have a sliver implementation.

Don’t put scrollable lists inside box adapters. It is more efficient if you put slivers directly inside the `CustomScrollView`.

#### 14.3.3.1 Nesting scrollable widget

Both `ListView` and `GridView` are perfect if you want to separately scroll through a list and a grid of items. For example, you could place them together in the same column in this way:

```
class _ExampleState extends State<Example> {
    final children = List<Widget>.generate(100, (i) => Text('$i'));

    @override
    Widget build(BuildContext context) {
        return Column(
            children: [
                Expanded(
                    child: ListView.builder(
                        itemCount: children.length,
                        itemBuilder: (_, index) => children[index],
                    ),
                ),
                Expanded(
                    child: GridView.builder(
                        gridDelegate: const SliverGridDelegateWithFixedCrossAxisCount(
                            crossAxisCount: 4,
                        ),
                        itemCount: children.length,
                        itemBuilder: (_, index) => children[index],
                    ),
                ),
            ],
        );
    }
}
```

Using a `SliverGridDelegateWithFixedCrossAxisCount` is the same as creating a `Grid` with the `count` constructor, but it’s more efficient because builders lazily load children. Note that the list

and the grid do not scroll together. They are two separate widgets with their scrolling machinery and layout constraints. If we wanted to scroll the list and the grid together, as if they were part of the same scrollable widget, we would need slivers:

```
class _ExampleState extends State<Example> {
  final children = List<Widget>.generate(100, (i) => Text('$i'));

  @override
  Widget build(BuildContext context) {
    return CustomScrollView(
      slivers: [
        SliverList(
          delegate: SliverChildBuilderDelegate(
            (context, index) => children[index],
            childCount: children.length,
          ),
        ),
        SliverGrid(
          delegate: SliverChildBuilderDelegate(
            (context, index) => children[index],
            childCount: children.length,
          ),
          gridDelegate: const SliverGridDelegateWithFixedCrossAxisCount(
            crossAxisCount: 4,
          ),
        ),
      ],
    );
}
```

In this way, `SliverList` and `SliverGrid` are “merged” and they scroll together. If we wanted to add padding around the grid, for example, we would have to use `SliverPadding`:

```
slivers: [
  SliverList(
    // code ...
  ),
  SliverPadding(
    padding: const EdgeInsets.all(10),
    sliver: SliverGrid(
      // code ...
    ),
  ),
],
```

#### 14.3.3.2 Using SliverAppBar for dynamic and sticky headers

Slivers are great for creating *sticky headers* for list widgets. The so-called *sticky headers* are widgets that always remain visible in the scroll view even if they would normally go off from the visible viewport. For example:

```
CustomScrollView(  
  slivers: [  
    const SliverAppBar(  
      centerTitle: true,  
      title: Text('ListView #1'),  
      pinned: true, // Notice this property  
    ),  
    SliverList(  
      // code...  
    ),  
    const SliverAppBar(  
      centerTitle: true,  
      title: Text('ListView #2'),  
      pinned: true, // Notice this property  
    ),  
    SliverList(  
      // code...  
    ),  
  ],  
,),
```

The `SliverAppBar` widget is a material widget that, by default, shows a blue stripe with some text that looks very similar to an `AppBar`. It scrolls along with all the other slivers but if you set `pinned` to `true` (as we did in the example), the bar stays fixed at the top of the view:

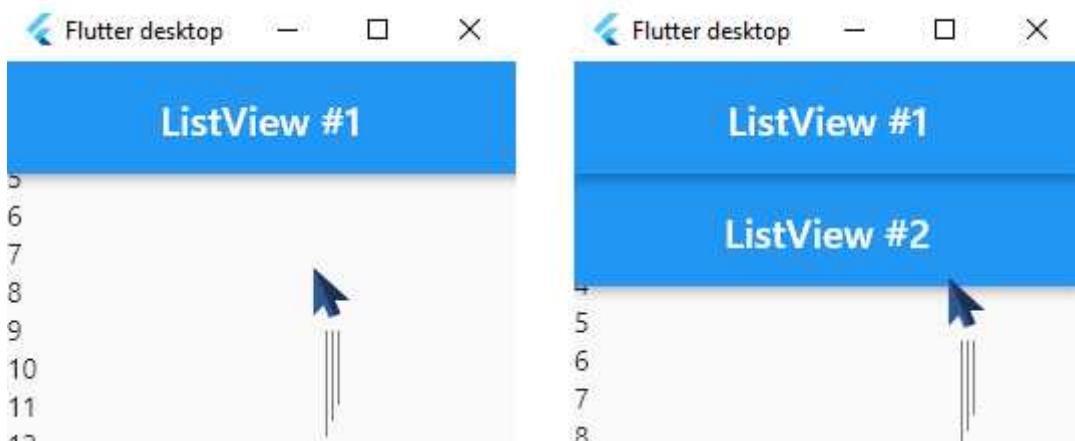


Figure 14.20: A sticky header in a custom scroll view, along with other slivers.

It's called "*sticky header*" because it sticks to the top part of the screen, and it never goes out from the visible viewport. If we don't make it sticky, then it just expands and collapses following the scroll direction:

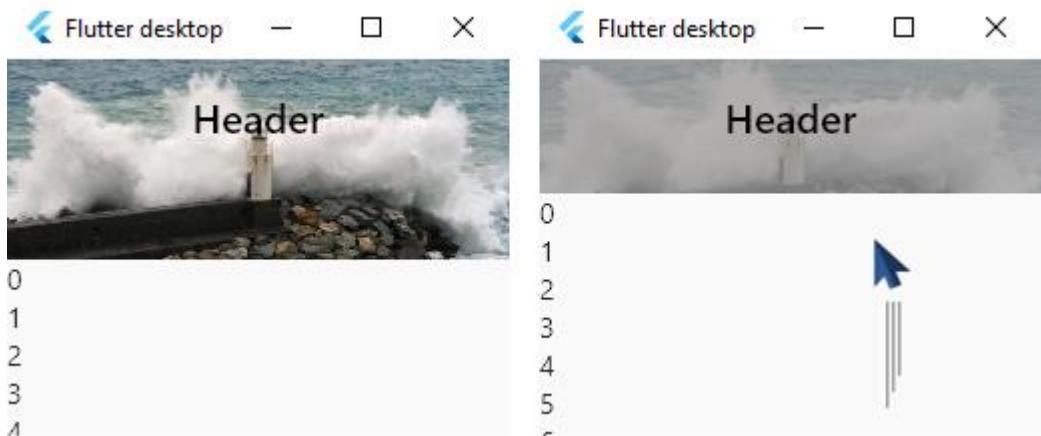


Figure 14.21: A dynamic header that fades away its background when scrolled.

By default, `pinned` is set to `false` so if you don't enable it, you will see the behavior in *Figure 14.21* (the header fades out its `flexibleSpace` and keeps reducing its size while remaining visible). This is the full code:

```
SliverAppBar(  
    centerTitle: true,  
    title: const Text(  
        'Header',  
        style: TextStyle(  
            color: Colors.black,  
        ),  
    ),  
    expandedHeight: 100,  
    backgroundColor: Colors.grey,  
    flexibleSpace: FlexibleSpaceBar(  
        background: Image.asset(  
            'assets/asset.png',  
            fit: BoxFit.cover,  
        ),  
    ),  
)
```

Once the app bar completely faded out the image and the header cannot collapse anymore, the sliver scrolls away from the top of the viewport:

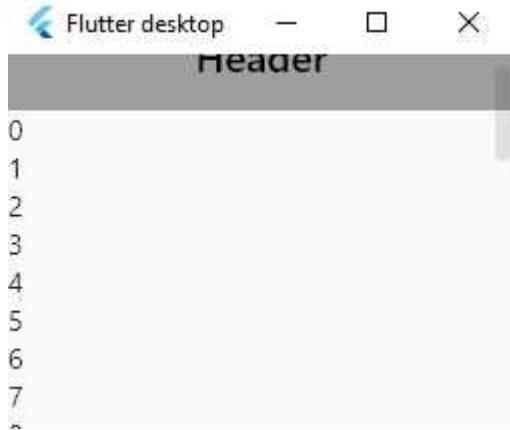


Figure 14.22: The dynamic header goes out of the view when it cannot collapse anymore.

The `SliverAppBar.medium` and `SliverAppBar.large` factory constructors create an app bar with default values that follow Material guidelines for “medium” and “large” components. Make sure to check the videos in the official documentation<sup>92</sup> to see all the possible configuration combinations for `SliverAppBar`.

## Deep dive: Understanding RenderObjects

In *chapter 10 – Section 3 “Widget, Element and RenderObject trees”* we covered what render objects are and how they relate to elements and widgets. One of the key points of that section is that render objects are internally managed by the framework so you don’t have to work with them directly. In general, you can say that:

- “Everything is a widget”, because “widgets” are subclasses of `Widget`;
- “Everything that is painted on the screen is a render object”, because “render objects” are subclasses of `RenderObject`.

The `RenderObject` abstract class does not define a coordinate system or a specific layout protocol. It just implements basic painting and layout algorithms. Rather than using `RenderObject` directly, you will generally find it more useful working with one of its specialized subclasses. There are four of them overall:

---

<sup>92</sup> <https://api.flutter.dev/flutter/material/SliverAppBar-class.html>

- **RenderBox**: This is a render object that introduces a 2D Cartesian coordinate system. What it means is that render objects gain new properties such as width, height, and position. For example, imagine you created this widget:

```
class MyWidget extends StatelessWidget {
  final String text;
  const MyWidget({required this.text, super.key});

  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: () => debugPrint('MyWidget size'),
      child: Text(text),
    );
  }
}
```

Remember that elements are the “glue” between widgets and render objects. To access the **RenderBox** associated with this widget, we have to use the **BuildContext** object (because it represents the **Element** behind **MyWidget**). This allows us to get the widget size:

```
void _printSize(BuildContext context) {
  final RenderObject? renderObject = context.findRenderObject();

  // Not all widgets use 'RenderBox' so we need a type check
  if (renderObject is RenderBox) {
    final size = renderObject.size;
    debugPrint('w = ${size.width} | h = ${size.height}');
  }
}

@Override
Widget build(BuildContext context) {
  return ElevatedButton(
    onPressed: () => _printSize(context),
    child: Text(text),
  );
}
```

The **findRenderObject** method is used to get a reference to the **RenderObject** associated with the widget. Most widgets use **RenderBox** (for its sizing and layout algorithms), but this is not always the case. As such, we need an **if** to check whether the render object really is a **RenderBox**. In that case, we can print the actual widget size. It wouldn’t be safe to assume that any render object is a **RenderBox**:

```
// This is dangerous!
final RenderObject? obj = context.findRenderObject()! as RenderBox;
```

First of all, the `findRenderObject` method could return a different kind of `RenderObject` (such as a `RenderAbstractViewport`, which will be analyzed in the next bullet point). The other problem with that line of code is that we are using the bang operator (!) on a method that could return null. For example:

```
@override
Widget build(BuildContext context) {
    final obj = context.findRenderObject()!; // Runtime exception here
    debugPrint('RenderBox? ${obj is RenderBox}');
    return const Text('Flutter!');
}
```

The `findRenderObject` method will always return `null` during the build phase because the underlying render object isn't yet ready. In practice, calling `findRenderObject` inside `build` causes a runtime error because you're trying to access a render object that is not built (or updated) yet. To solve the problem, you have to invoke `findRenderObject` after the `build` method is executed:

```
class _ExampleState extends State<Example> {

    @override
    void initState() {
        super.initState();

        // OK because this callback is executed after 'build'
        WidgetsBinding.instance.addPostFrameCallback(_);
        final renderObject = context.findRenderObject()!;
        debugPrint('RenderBox? ${renderObject is RenderBox}');
    }

    @override
    Widget build(BuildContext context) => const Text('Flutter!')
}
```

This code does not throw a runtime exception anymore because `addPostFrameCallback` is called after the build phase, which is when all render objects have already been created or updated. By consequence, `findRenderObject` will never be `null`. For more information about `WidgetsBinding`, check “*Deep dive: The framework internals*”.

- `RenderAbstractViewport`: This is a render object that is used by the scrolling machinery. For example, it can be useful to return the offset that is needed to “reveal” an object inside a scrollable widget. Consider this code:

```

class _ExampleState extends State<Example> {
  final keys = List< GlobalKey >.generate(4, (_) => GlobalKey());

  void lastItemOffset() {
    final ro = keys.last.currentContext?.findRenderObject();

    if (ro != null) {
      final viewPort = RenderAbstractViewport.of(ro);
      final revealDist = viewPort.getOffsetToReveal(ro, 0);
      debugPrint('Offset = ${revealDist.offset}');
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: ListView(
        itemExtent: 25,
        children: [
          Text('Item 1', key: keys.first),
          Text('Item 2', key: keys[1]),
          Text('Item 3', key: keys[2]),
          Text('Item 4', key: keys.last),
        ],
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: lastItemOffset,
        child: const Icon(Icons.info_outline),
      ),
    );
  }
}

```

We have assigned a  `GlobalKey` to each `Text` widget in the list so that we can access their `BuildContext` (and the associated render object). The `lastItemOffset` method internally uses `RenderAbstractViewport` to calculate the offset (or “the distance”) required to reveal the last item of the `ListView`. Note that `getOffsetToReveal` is the only method defined by the class (other than calculating the offset, there isn’t much to do). One thing to note:

```
final viewPort = RenderAbstractViewport.of(ro);
```

The `RenderAbstractViewport.of` method is not an inherited widget, but it has a similar purpose. It navigates up in the render object tree until a render viewport object is found. To get more information about the render object of a scrollable widget, a more “specialized” subclass of `RenderAbstractViewport` is required. For example:

```
if (ro != null) {
    final viewPort = RenderAbstractViewport.of(ro);

    // 'RenderViewport' extends 'RenderBox'
    if (viewPort is RenderViewport) {
        debugPrint('Scroll direction = ${viewPort.axis}');
        debugPrint('Item size = ${viewPort.size}');
        debugPrint('Is not visible? = ${viewPort.hasVisualOverflow}');
    }
}
```

A `RenderViewport` is the visual “engine” of the scrolling machinery which sizes and lays its children. In fact, in the example we can get the scroll direction, the child size and whether the object is in the visible area or not. This class is typically used when you need to determine whether an object is currently visible or not.

- `RenderSliver`: This is a render object that implements scrolling effects in viewports. As the name suggests, `RenderSliver` is the abstract class behind all sliver widgets. For example, a `SliverPadding` widget uses a `RenderSliverPadding` render object. This class isn’t directly used because it’s meant to serve as base class for custom sliver implementations.
- `RenderView`: This render object is the root of the render tree. It represents the total output surface of the render tree and manages the rendering pipeline initialization. It requires a single `RenderBox` object to fill the entire output surface. In this way, all children can be sized and positioned in a 2D Coordinate system.

You can subclass any of these four classes to implement your own layout, painting, accessibility or hit testing algorithms. It is almost never a good idea to do so because widgets and custom painters let you do almost everything you want.

Render objects are low level tools that are entirely managed by the framework. You should really focus on the widget layer, which is what Flutter developers are expected to use. There may be some cases where you need `findRenderObject()` to get particular information about a widget, which is fine. But other than that, you’ll hardly ever need to create a render object yourself from scratch.

## Deep dive: Scrolling and overlays



[https://github.com/albertodev01/flutter\\_book\\_examples/chapter\\_14/scroll\\_overlay/](https://github.com/albertodev01/flutter_book_examples/tree/chapter_14/scroll_overlay)

We have already seen in *Deep dive – Navigation and overlays* what overlays are and how they work. In this section, we want to cover how they can integrate with scrollable widgets. For example, let's consider this widget:

```
class HomePage extends StatefulWidget {
  const HomePage({super.key});

  @override
  State<HomePage> createState() => _HomePageState();
}

class _HomePageState extends State<HomePage> {
  final GlobalKey textKey = GlobalKey();

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: ListView(
        children: [
          const SizedBox(height: 1800),
          Center(
            child: const Text(
              'Hello world!',
              key: textKey,
            ),
          ),
          const SizedBox(height: 1800),
        ],
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () {}, // TODO
        child: const Icon(Icons.info_outline),
      ),
    );
  }
}
```

Unless the screen is very tall, the `Text` widget inside the `ListView` is not immediately visible. You will have to scroll down a bit before revealing the “Hello world!” message. In this example, we want to show an overlay below the `Text` widget when the `FloatingActionButton` is pressed. To do so,

we need a  `GlobalKey` to access the widget's render object without using `BuildContext`. Here is a possible way to implement a function that will place an  `OverlayEntry` right below the  `Text` widget:

```
class _HomePageState extends State<HomePage> {
    final textKey = GlobalKey();
    OverlayEntry? entry;

    void showOverlay() {
        // The key gives access to the 'RenderObject' of the 'Text' widget
        final renderObject = textKey.currentContext?.findRenderObject();

        if (renderObject is RenderBox) {
            // Gets the position of the 'Text' widget on the screen
            final position = renderObject.globalToLocal(Offset.zero);

            final entry = OverlayEntry(
                builder: (_) {
                    return Positioned(
                        left: 0,
                        right: 0,
                        top: position.dy + 25, // The '+25' is just to add some space
                        child: Container(
                            color: Colors.grey,
                            width: renderObject.size.width,
                            height: 50,
                        ),
                    );
                },
            );
        }

        Overlay.of(context).insert(entry);
    }
}

@Override
Widget build(BuildContext context) {
    // code...
}
```

Thanks to the  `globalToLocal` function, we can get the position of the  `Text` widget on the screen. We passed the Y coordinate (`position.dy`) to the  `Positioned` widget to place the overlay below the  `Text`. We have added some extra space on the vertical axis (`25`) to make sure that the overlay does not cover the widget. The code works as intended, but the problem is that the overlay will not “follow” the  `Text` widget if we scrolled up or down. Look at this image:

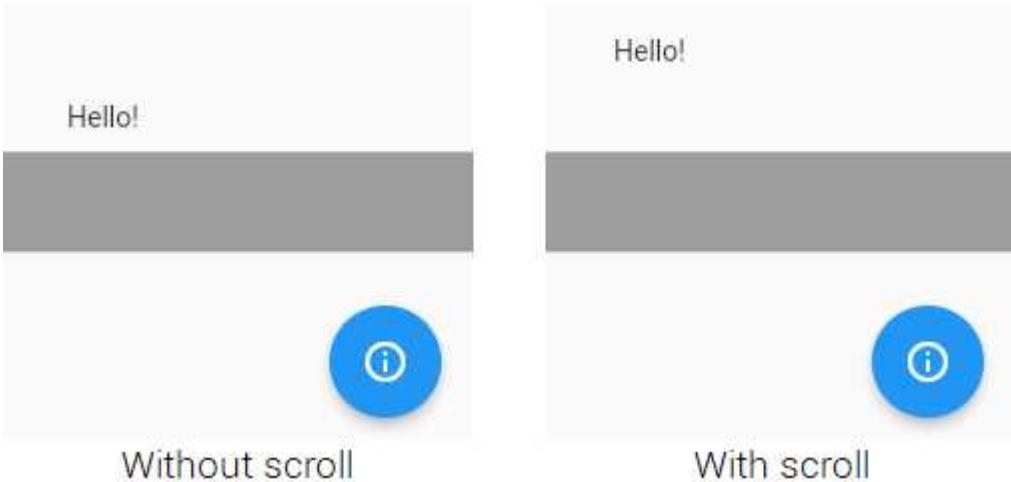


Figure 13.23: Without scroll, the overlay is at the correct position. After having scrolled, it is not.

On the left of *Figure 13.21* you see that the `OverlayEntry` is positioned as expected below `Text`. When the user scrolls (on the right of *Figure 13.21*), the text moves up but the overlay does not because it is static. In other words, the overlay does not “follow” the widget. Here’s how to solve the problem:

1. Inside the state class, create a `LayerLink` object.:

```
final link = LayerLink();
```

2. Wrap the `Positioned` child inside a `CompositedTransformFollower` and give it the `link` you created in the state class. Note that we have moved the padding (25) inside the follower using its `offset` property:

```
return Positioned(
  left: 0,
  right: 0,
  top: position.dy,
  child: CompositedTransformFollower(
    link: link, // This is the 'LayerLink' of the state class
    offset: const Offset(0, 25),
    followerAnchor: Alignment.topCenter,
    targetAnchor: Alignment.topCenter,
    child: Container(
      color: Colors.grey,
      height: 50,
    ),
  ),
);
```

3. Wrap the `Text` inside a `CompositedTransformTarget` and give it the `link` you created in the state class:

```
Center(  
    child: CompositedTransformTarget(  
        link: link,  
        child: Text(  
            'Hello!',  
            key: textKey,  
        ),  
    ),  
,
```

A `CompositedTransformFollower` follows a `CompositedTransformTarget` when it moves on the screen. The `LayerLink` is used as an identifier so that a target widget can be followed by a follower widget. With this fix, the overlay can correctly follow the `Text` widget when it moves up or down:

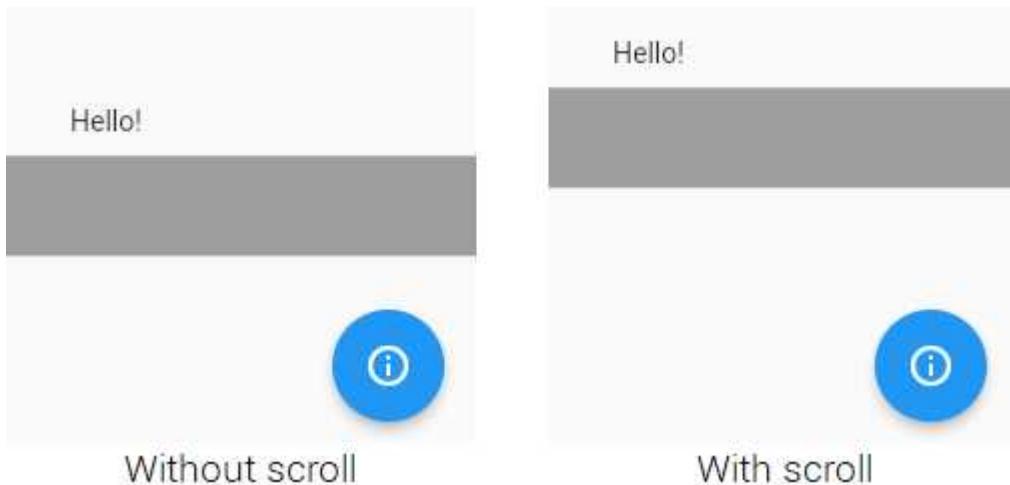


Figure 13.24: The overlay “follows” the widget and always stays at the same distance from it (25).

By default, overlays do not move unless you dynamically change the coordinates of the `Positioned` widget. However, to follow a widget, it is easier to use a `LayerLink` (along with transform widgets) rather than manually updating coordinates by hand.

# 15 – Internationalization and accessibility

---

## 15.1 Internationalization

If your application is distributed to a worldwide audience, it will be used by people with different languages and cultures. In practice, it means that you need to translate the text for each language you want to support and also re-arrange the UI according with the geographical area and culture. For example:

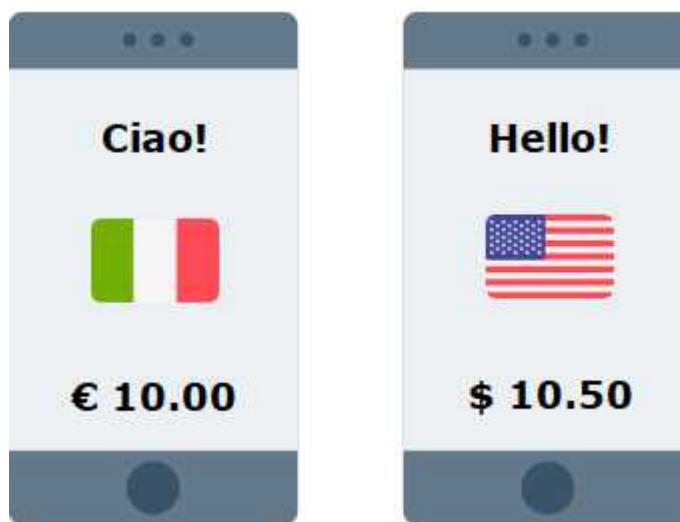


Figure 15.1: The same application running on mobile devices from two different geographical areas.

The image in *Figure 15.1* shows the same application running on an Italian device (on the left) and on an American device (on the right). Note that not only the text is translated but also the flags, the price and the currency also change. When your application is distributed to a worldwide audience, there are many things you should take care of. For example:

- translate text into various languages;
- show prices in the correct currency (euro, dollar, sterling...);
- choose between 24 or 12h format when displaying the time;
- format dates and times in a specific way;
- render text from left-to-right or from right-to-left;
- time zones and daylight savings periods might also kick in, which make the date and time management more complicated;

- images or generic assets may require variants;
- audio tracks might need to change across countries or states;
- some features might be available for a restricted number of countries.

As you can see, making an application suitable for a broad audience is a demanding work. However, being able to communicate with your users effectively is extremely important and the effort is often worth the result. The most relevant benefits that your application will get are:

1. Customers will be satisfied because they will use the application in a way they understand and are comfortable with. In other words, the application adapts to the user.
2. Have a competitive edge against other applications similar to yours that are instead specific to a particular geographical area.
3. The possibility to increase revenues and the brand loyalty because you have higher chances to gain your users' trust.

Before moving on, we want to clarify some terminology we're going to use in this chapter. There is no unique and precise definition to these words but this is what they're used for:

- Internationalization, abbreviated with *i18n*, is the process of creating an application so that it adapts to various languages and regions without manual changes. For example: rather than hard-coding strings, you could create a class with some getters that dynamically load the proper translation. In this way, the same application can automatically adapt to various languages.
- Localization, abbreviated with *L10n*, is the act of adapting an internationalized software for a specific region or language. For example, creating an *Italian localization* of an application includes translating all the text to Italian and formatting dates in the same way as someone living in Italy would expect.

Flutter has built-in support to make internationalization and localization easy to be implemented. For example, the `flutter_localization` is used to localize all Material and Cupertino widgets in more than 78 different languages. It's already bundled in the SDK so you don't even need to update the package version:

```
dependencies:
  flutter_localizations:
    sdk: flutter
```

After having imported `flutter_localization` in the `pubspec.yaml` file, you need to enable it for the entire application. Of course, the same setup would also work for a `CupertinoApp` widget:

```
class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      localizationsDelegates: [
        GlobalMaterialLocalizations.delegate,
        GlobalWidgetsLocalizations.delegate,
        GlobalCupertinoLocalizations.delegate,
      ],
      supportedLocales: [
        Locale('en'), // American english
        Locale('it'), // Italian
      ],
      home: HomeWidget(),
    );
  }
}
```

Localization delegates simply are factories that produce collections of localized values for various languages. The `supportedLocales` list indicates which localizations the application supports. In our example, the application is localized for “American English” and “Italian”.

## Note

The term `locale` refers to a set of parameters that define the user’s language, region and other special configurations. A locale is generally identified by a language code and a country code. For example, here are two different French locales:

- `fr_CH` represents Swiss French
- `fr_CA` represents Canadian French

In those tags, `fr` is the language code while `CH` and `CA` are country codes. Flutter uses the `Locale` class to represent a locale.

At the moment, we’ve only seen how to localize Material and Cupertino widgets. To localize those widgets plus your application’s text, more work is needed. Let’s see how Flutter allows adding our own localized messages within an application.

### 15.1.1 Internationalizing a Flutter application



[https://github.com/albertodev01/flutter\\_book\\_examples/chapter\\_15/15.1.1](https://github.com/albertodev01/flutter_book_examples/tree/chapter_15/15.1.1)

To localize your own strings, other than adding the `flutter_localization` package, you also need the `intl` dependency. This package, maintained by the Dart team, has lots of internationalization and localization features. Make sure to have both of them in your pubspec file:

```
dependencies:  
  flutter:  
    sdk: flutter  
  
  flutter_localizations:  
    sdk: flutter  
  
  intl: ^0.18.0
```

At the bottom of the `pubspec.yaml` file, also make sure to enable the code generation flag. This is needed to internally generate internationalization files:

```
flutter:  
  uses-material-design: true # This is here by default  
  generate: true # Add this line.
```

Now you need to create a new file in the root directory of your project. It has to be called `l10n.yaml` and should have the following (recommended) configuration:

```
arb-dir: lib/l10n/locales  
template-arb-file: app_en.arb  
output-localization-file: app_localizations.dart  
nullable-getter: false
```

These configurations are used by the generator tool to create internationalization and localization classes for the application. In particular:

- The `arb-dir` parameter is used to tell Flutter where ARB files are located. ARB stands for “Application Resource Bundle” and it’s nothing more than a JSON file on steroids. This kind of file is used as a template to localize strings in various languages.
- The `template-arb-file` parameter is used to tell Flutter which language to use as default. In our case, we want English to be the default (hence why `app_en.arb`) and later we will also define Italian (creating `app_it.arb`).

Note that ARB file names should end with the language code. Inside `lib/l10n/locales` we can add all ARB files with the localized strings. Rather than hard-coding strings directly in your Flutter widgets, you have to define them here:

lib/l10n/locales/app_en.arb	lib/l10n/locales/app_it.arb
{ "helloWorld": "Hello World!" }	{ "helloWorld": "Ciao mondo!" }

Now run the `flutter gen-l10n` command from your project root so that code generation takes place. It basically converts the ARB files into Dart getters that return localized string values. Even if you forgot to run the `gen-l10n` command, when you'll run the project, Flutter will automatically execute the code generation step for you. We're almost done with the setup. Even if not required, we strongly recommend to create the `l10n.dart` file with the following contents:

```
export 'package:flutter_gen/gen_l10n/app_localizations.dart';

extension AppLocalizationsExt on BuildContext {
  AppLocalizations get l10n => AppLocalizations.of(this);
}
```

The `AppLocalizations` class was automatically created by the Flutter generation tool. It contains a series of getters that return localized string values. It is the glue that connects ARB files, auto-generated Dart classes, and your application. To “install” it, you need to set two properties on the root `MaterialApp` or `CupertinoApp` widget:

```
const MaterialApp(  
  localizationsDelegates: AppLocalizations.localizationsDelegates,  
  supportedLocales: AppLocalizations.supportedLocales,  
  home: Scaffold(  
    body: MessageWidget(),  
  ),  
,
```

The `localizationsDelegates` property contains the usual Material and Cupertino delegates plus the localization delegate for custom strings. In our example, the `supportedLocales` list contains two `Locale` instances: one for English and one for Italian. For example, this is how we can localize the “Hello world” string anywhere in the widget tree:

```
@override  
Widget build(BuildContext context) => Text(context.l10n.helloWorld);
```

On an Italian device, the `Text` widget will render “*Ciao Mondo!*”. On an English device and any other locale configuration (because English is set as default in the `l10n.yaml` file), the `Text` widget will render “*Hello World!*”. This is the complete code:

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

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      localizationsDelegates: AppLocalizations.localizationsDelegates,
      supportedLocales: AppLocalizations.supportedLocales,
      home: Scaffold(
        body: MessageWidget(),
      ),
    );
  }
}

class MessageWidget extends StatelessWidget {
  const MessageWidget({super.key});

  @override
  Widget build(BuildContext context) {
    return Text(context.l10n.helloWorld);
  }
}
```

We recommend checking the official GitHub repository of this book to clone this example and play with it. To sum up, localizing a Flutter project involves four main steps:

1. Add the official `flutter_localization` and `intl` packages as dependencies. They are used to automatically localize Flutter widgets and your own strings.
2. Create the `l10n.yaml` file (to configure Flutter’s code generation tool).
3. Create ARB files (one for each locale you want to support), and place all of them in the same folder. Run the `flutter gen-l10n` command to start the code generation tool.
4. If you want, create an extension method to access localized strings using `context.l10n`.

The Flutter build tool internally converts each ARB file into a Dart file that has one getter per string. You will notice that, after having run `flutter gen-l10n`, two classes are generated. One contains configurations for the English locale:

```
/// The translations for English (`en`).
class AppLocalizationsEn extends AppLocalizations {
    AppLocalizationsEn([String locale = 'en']) : super(locale);

    @override
    String get helloWorld => 'Hello World!';
}
```

The other contains configurations for the Italian locale:

```
/// The translations for Italian (`it`).
class AppLocalizationsIt extends AppLocalizations {
    AppLocalizationsIt([String locale = 'it']) : super(locale);

    @override
    String get helloWorld => 'Ciao mondo!';
}
```

The `AppLocalizationDelegate` type is used to determine the system locale. It automatically uses one of the two generated classes, according with the locale configuration of the platform. If we added more locales, then the tool would generate more localization classes. All of this is extremely developer friendly: you just need to define your strings in the various ARB files and Flutter will do the rest.

#### 15.1.1.1 Dynamically changing locale



[https://github.com/albertodev01/flutter\\_book\\_examples/tree/chapter\\_15.1.1.1](https://github.com/albertodev01/flutter_book_examples/tree/chapter_15.1.1.1)

By default, both `MaterialApp` and `CupertinoApp` automatically detect the system locale and use it in the application. In some cases, you might need to override the default system locale and use a specific one.

#### Note

We're taking the example of the previous section and adding the possibility to manually change the locale. As such, only English and Italian locales are supported. You can find the complete source code in our GitHub repository.

In order to dynamically change the application locale, we need to create an `InheritedWidget` and expose a `Locale` object. When the locale changes, the `MaterialApp` or `CupertinoApp` widget must rebuild. As such, we need to expose a listenable class:

```
class InheritedLocale extends InheritedWidget {
  static const englishLocale = Locale('en');
  static const italianLocale = Locale('it');

  final ValueNotifier<Locale> locale;

  const InheritedLocale({
    super.key,
    required super.child,
    required this.locale,
  });

  static InheritedLocale of(BuildContext context) {
    final ref = context.dependOnInheritedWidgetOfExactType<InheritedLocale>();
    assert(ref != null, "No 'InheritedLocale' found above in the tree.");
    return ref!;
  }

  @override
  bool updateShouldNotify(covariant InheritedLocale oldWidget) {
    return locale != oldWidget.locale;
  }
}

extension InheritedLocaleExt on BuildContext {
  ValueNotifier<Locale> get locale => InheritedLocale.of(this).locale;
}
```

We created two `static` instances for reusability purposes because we don't want to copy and paste the same code more than once. Now we need to "install" the inherited widget at the root because `MaterialApp` (or `CupertinoApp`) and possibly other widgets may be interested in the current locale value:

```
void main() {
  runApp(
    InheritedLocale(
      locale: ValueNotifier<Locale>(InheritedLocale.englishLocale),
      child: const MyApp(),
    ),
  );
}
```

In this way, “English” is the default locale of the application. To listen for locale changes, we need to wrap `MaterialApp` or `CupertinoApp` in a `ListenableBuilder`. For example:

```
class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return ListenableBuilder(
      listenable: context.locale,
      builder: (context, child) {
        return MaterialApp(
          locale: context.locale.value,
          localizationsDelegates: AppLocalizations.localizationsDelegates,
          supportedLocales: AppLocalizations.supportedLocales,
          home: child!,
        );
      },
      child: const HomeWidget(),
    );
  }
}
```

The `locale` parameter is `null` by default so that `MaterialApp` and `CupertinoApp` are created with the system locale. When a value is given to `locale`, the default behavior is overridden. As such, in the example we set...

```
locale: context.locale.value,
```

... to rebuild the application whenever the `Locale` value changes. In practice, the inherited widget is used to control the application locale and rebuild it accordingly. Inside `HomeWidget` (or anywhere else in the widget tree), we have to add a button to change the locale. For example:

```
ElevatedButton(
  onPressed: () {
    if (context.locale.value == InheritedLocale.englishLocale) {
      context.locale.value = InheritedLocale.italianLocale;
    } else {
      context.locale.value = InheritedLocale.englishLocale;
    }
  },
  child: const Text('Toggle locale'),
),
```

Since `locale` is a `ValueNotifier`, when its value changes the `MaterialApp` widget is rebuilt with the new locale. The current locale can also be obtained with Flutter’s `Localization` class:

```
final Locale locale = Localizations.localeOf(context);
```

The value returned by `localeOf` is the same as the one defined by `locale` in the `MaterialApp` or `CupertinoApp` widgets. However, it cannot be listened to.

#### 15.1.1.2 Manual localization



[https://github.com/albertodev01/flutter\\_book\\_examples/chapter\\_15/15.1.1.2/](https://github.com/albertodev01/flutter_book_examples/tree/master/chapter_15/15.1.1.2/)

Thanks to ARB files and Flutter's internal code generation system, we can easily localize our strings. However, you might not like this way of working and you might prefer doing everything yourself. In this section, we cover how to manually internationalize Flutter applications.

#### Good practice

We recommend to not go for this approach. Prefer using Flutter's built-in system, which doesn't need maintenance. It handles everything automatically and is (in our opinion), easier to understand.

Since we don't want to rely on Flutter tools, we will have to create all classes manually. First of all, we need to create a file called `app_localizations.dart`. It contains the `AppLocalization` class, which is used to define all locales and strings:

```
class AppLocalizations {  
  final Locale locale;  
  const AppLocalizations(this.locale);  
  
  static AppLocalizations of(BuildContext context) {  
    return Localizations.of<AppLocalizations>(context, AppLocalizations)!;  
  }  
  
  static final Languages = _localizedValues.keys.toList();  
  static final Locales = _localizedValues.keys.map(Locale.new).toList();  
  static const _localizedValues = <String, Map<String, String>>{  
    'en': {'title': 'Hello world!'},  
    'it': {'title': 'Ciao mondo!'},  
  };  
  
  String get title => _localizedValues[locale.languageCode]![  
    'title']!;  
}
```

Since we no longer use ARB files, a map of maps (`_localizedValues`) is the best option (since it can also be constant). The biggest problem is that a map of maps is not practical to maintain, and this class also needs testing. In the same file, we also need to add the localization delegate:

```
class AppLocalizationsDelegate extends LocalizationsDelegate<AppLocalizations> {
  const AppLocalizationsDelegate();

  @override
  bool isSupported(Locale locale) =>
    AppLocalizations.languagess.contains(locale.languageCode);

  @override
  Future<AppLocalizations> load(Locale locale) {
    return SynchronousFuture<AppLocalizations>(AppLocalizations(locale));
  }

  @override
  bool shouldReload(AppLocalizationsDelegate old) => false;
}
```

In this file, you could add the extension method to access via context the `AppLocalization` object. Finally, install the delegates in the `MaterialApp` or `CupertinoApp` widget in the usual way:

```
@override
Widget build(BuildContext context) {
  return MaterialApp(
    localizationsDelegates: const [
      AppLocalizationsDelegate(),
      GlobalMaterialLocalizations.delegate,
      GlobalWidgetsLocalizations.delegate,
    ],
    supportedLocales: AppLocalizations.locales,
    home: const Scaffold(
      body: MessageWidget(),
    ),
  );
}
```

The result is the same but we've had to create the delegate and the localization classes manually. There are no performance differences from the generation-based approach. The biggest problem is that we need to handle strings inside a `Map<String, Map<String, String>>`, which is not easy to maintain. While ARB files are portable and can be shared on external platforms (for translation reviews for example), a hard-coded `Map` structure is harder to manage. We recommend relying on Flutter's generation tool and avoiding internationalizing applications manually.

## 15.1.2 Using the intl package

The official `intl`<sup>93</sup> package has a huge number of internationalization and localization facilities. It's used for messages replacement, date and number formatting or string parsing. It also has an API to work with bidirectional text.

### 14.1.3.1 Numbers

To format numbers in a locale-specific way, use the `NumberFormat` class. The format is specified using a subset of the ICU formatting patterns<sup>94</sup>. For example:

```
final formatterUS = NumberFormat('###.0#', 'en_US');
final formatterIT = NumberFormat('###.0#', 'it_IT');

final numberUS = formatterUS.format(14.5432); // 14.54
final currencyUS = formatterUS.currencyName; // USD

final numberIT = formatterIT.format(14.5432); // 14,54
final currencyIT = formatterIT.currencyName; //EUR
```

Notice that strings are formatted differently according to the locale. The decimal separator for the American locale is a period, while the Italian locale uses a comma. Of course, the currency name also changes accordingly. The first parameter of the `NumberFormat` constructor specifies the format pattern. Here is what the various symbols mean:

- `0` A single digit.
- `#` A single digit, which is omitted if the value is zero.
- `.` Decimal separator.
- `-` Minus sign.
- `,` Grouping separator.
- `E` Separates mantissa and exponent
- `+` Before an exponent to say that it should be prefixed with the plus sign.
- `%` In prefix or suffix, multiplies by 100 and shows the percentage.
- `\$` Currency sign, replaced by the currency name.
- `'` Used to quote special characters.
- `;` Used to separate positive and negative patterns if both are present.

---

<sup>93</sup> <https://pub.dev/packages/intl>

<sup>94</sup> <https://pub.dev/documentation/intl/latest/intl/NumberFormat-class.html>

These symbols combined together produce different results. When you don't explicitly provide the locale, it will default to the current system's locale. If the format is not specified, it will print at least one integer digit and three fraction digits. For example:

```
debugPrint(NumberFormat('###.##', 'en_US').format(13.432)); // 13.43
debugPrint(NumberFormat('###.0', 'en_US').format(13.432)); // 13.4
debugPrint(NumberFormat('0.00#', 'en_US').format(13.432)); // 13.432

debugPrint(NumberFormat('-#.#', 'en_US').format(13.432)); // -13.4
debugPrint(NumberFormat('00.00 ₩', 'en_US').format(13.432)); // 13.43 USD
debugPrint(NumberFormat('##,##%', 'en_US').format(13.432)); // 13,43%
```

There are some named constructors that build a `NumberFormat` object with a common pattern. For example, you can format a number with default locale conventions:

```
debugPrint(NumberFormat.currency(locale: 'en_US').format(13.4)); // USD13.40
debugPrint(NumberFormat.currency(locale: 'it_IT').format(13.4)); // 13,40 EUR
```

If you don't want to use the named constructor, you can create your own string pattern and get the same result. However, try to use these named constructors as much as possible for readability.

#### 15.1.2.2 Date and time

The `DateFormat` class is used for formatting and parsing dates in a locale-sensitive manner. The usage is very similar to `NumberFormat`. You need to define a pattern, which follows specific rules, and the locale, which is optional. If you don't specify the locale, `DateFormat` automatically uses the system locale. For example:

```
final date = DateTime(2018, 12, 4, 16, 24);

// 2018-12-4 4:24.0
debugPrint(DateFormat('y-M-d h:m.s', 'en_US').format(date));

// 4 12 2018
debugPrint(DateFormat('d M y', 'en_US').format(date));

// 4-12-2018 16:24.0
debugPrint(DateFormat('d-M-y H:m.s', 'it_IT').format(date));

// 2018 December 4 (Tue)
debugPrint(DateFormat('y MMMM d (E)', 'en_US').format(date));

// 4 dicembre 2018 (mar)
debugPrint(DateFormat('d MMMM y (E)', 'it_IT').format(date));
```

Notice how the locale affects the output. For example, when it's set to `en_US` the month name is *December*, while `it_IT` (Italian) outputs *Dicembre*. Users can use customized patterns and combine tokens. Formatting dates defaults to `en_US` when no locale is specified:

```
debugPrint(DateTime('y MMMM d (E)').format(date)); // 2018 December 4 (Tue)
```

There also are a series of pre-built constructors that apply specific formatting for you. For example, these two constructors are equivalent:

```
debugPrint(DateTime('E, MMM d, y').format(date)); // Tue, Dec 4, 2018
debugPrint(DateTime.yMMEd().format(date)); // Tue, Dec 4, 2018
```

Of course, if you pass a different locale, the result changes. For example:

```
debugPrint(DateTime.yMMEd('fr_CA').format(date)); // mar. 4 déc. 2018
```

Intl uses the ICU/JDK date/time pattern specification with dozens of tokens. We encourage you to visit the official documentation<sup>95</sup> to see all the possible tokens and details about date and time formatting.

## 15.2 Accessibility

A high-quality Flutter application is accessible to the broadest range of users and does not create barriers for people of all ages. We recommend including an accessibility checklist in your project planning as a to-do item before sending the application to the production state. Here are a few key suggestions we think are very important to make your project accessible:

- Large fonts. When a human gets older, the eye also ages and starts having focusing issues. Some people may be born with eye diseases or, in general, with vision defects. You should make sure that the text scales appropriately when users enable text scaling options.

Flutter automatically calculates font sizes based on platform settings. The only thing you have to do is ensure that the layout has enough room to render the text even if it's very big. You can quickly test this yourself:

```
Text('Hello world!',
      textScaleFactor: 1.5,
),
```

---

<sup>95</sup> <https://pub.dev/documentation/intl/latest/intl/DateFormat-class.html>

The `textScaleFactor` property is a multiplier that enlarges the text by the given factor. In our example, 1.5 means that the text is 150% of its normal size. You generally don't want to manually set this value because if you leave it `null`, Flutter will automatically assign a proper value.

- Color contrast. High color contrast makes it easier for the eye to recognize images and text. For example, you should not render white text on a light grey background because the two colors are very similar and do not make much contrast. Controls should also be usable and legible in grayscale and colorblind modes.
- Target sizes. Buttons and other tappable targets should not be too small. The official Flutter documentation recommends 48x48 as smallest size. However, we think that 52x52 would be a better fit.
- Screen readers. A screen reader is an assistive technology essential for blind people and very useful to those with other difficulties such as vision impairment or learning disability. Screen readers enable users to get vocal feedback about screen content and allow for interaction via gestures or keyboard shortcuts. You can test the screen reader compatibility by running it and playing around with your application. Each platform may use different software by default:
  - On macOS and iOS, VoiceOver is the built-in screen reader made by Apple.
  - On Android, TalkBack is the built-in screen reader made by Google.
  - On Windows, there are various choices but NVDA and JAWS are the most popular.
  - On web browsers, each software has its own recommended plugin to use.
  - On Linux, Orca is a popular and open-source screen reader.

For mobile devices, we recommend to try screen readers on real devices for higher fidelity.

The great news is that widgets are compatible with screen readers by default. In the next section, we will see how accessibility works and how you can customize the interaction with screen readers.

### 15.2.1 The Semantic widget

If we looked at how material widgets have been created, we would notice that they often contain the `Semantics` widget somewhere in the tree. For example, look at how Flutter handles toggleable widgets:

```
Semantics(  
    enabled: isInteractive,  
    child: CustomPaint(  
        size: size,  
        painter: painter,  
    ),  
,
```

To make a more specific example, this is how `ListTile` internally looks. We have removed most of the non-related properties to keep the code short and highlight the the `Semantics` widget:

```
return InkWell(  
    // more properties here...  
    child: Semantics(  
        selected: selected,  
        enabled: enabled,  
        child: Ink(  
            // more properties here...  
        ),  
    ),  
,  
);
```

The `Semantics` widget is used to describe the meaning of a sub-tree. Inside the framework, it is handy because accessibility tools, screen readers, search engines, and semantic analysis software leverage it to determine the meaning of the application. Thanks to `Semantics`, a Flutter application can quickly become screen-reader-friendly just by setting a few properties. For example, consider this widget:

```
class Example extends StatelessWidget {  
    const Example({super.key});  
  
    @override  
    Widget build(BuildContext context) {  
        return const Column(  
            children: [  
                CustomPaint(  
                    size: Size(400, 400),  
                    painter: MyPainter(),  
                ),  
                Text('A five edges star.'),  
            ],  
        );  
    }  
}
```

The custom painter is used to draw a star and the text below is a label that describes the output of the `CustomPaint` widget. Overall, we could describe the `Example` widget as “an image that has no interactions and a description”. Let’s translate this meaning into a widget using `Semantics`:

```
Semantics(  
  image: true,  
  label: 'A five edges star.',  
  child: Column(  
    children: const [  
      CustomPaint(  
        painter: MyPainter(),  
        size: Size(400, 400),  
      ),  
      Text('A five edges star.'),  
    ],  
  ),  
,  
);
```

Thanks to this configuration, a screen reader knows that `Example` is an image that represents a five edges star. The `Semantics` widget has more than forty properties you can use to describe a portion of subtree. Make sure to check out the official documentation for a complete coverage.

### Good practice

Don’t wrap each widget you create in a `Semantics` widget because it might produce confusing results for the screen reader or misleading semantic analysis. Make sure to only describe the widget “as a whole” rather than focusing on each single child.

On the technical side, `Semantics` does not define a constant constructor. However, if you use the `fromProperties` named constructor you can make it `const`. We recommend creating `Semantics` widgets with the named constructor whenever you have a const-able expression like this:

```
return const Semantics.fromProperties(  
  properties: SemanticsProperties(  
    image: true,  
    label: 'Painter label',  
,  
    child: CustomPaint(  
      painter: MyPainter(),  
      size: Size(400, 400),  
,  
    );
```

It's the same as using the default `Semantic` constructor but this one can be constant. Some widgets give you the possibility to define a `semanticsLabel`, a string that describes the purpose of a widget. For example, it might be useful to replace abbreviations or fully describe a piece of text that has symbols. For example:

```
const Text(  
  r'$ 10.00',  
  semanticsLabel: 'Ten dollars',  
)
```

The `semanticsLabel` property is not defined for all Flutter widgets but where possible, try to use it. This value overrides any semantic label applied directly to the `TextSpan`. If a widget doesn't have the `semanticsLabel` property, just wrap it with `Semantics` and define it.

### 15.2.2 The semantic tree

We have already seen in *chapter 10 – Section 3 “Widget, Element and RenderObject trees”* that Flutter creates and maintains three trees in parallel for performance purposes. There also is another tree to add to the list: the semantics tree.

#### Good practice

Nodes of the semantics tree are `SemanticsNode` objects and they are used to represent semantic data. The creation of these nodes is handled by Flutter and it's influenced by the presence of `Semantics` widget across the widget tree.

The `Semantics` widgets are used by Flutter to build the semantic tree, which is what screen readers care about. You don't need to work directly with `SemanticsNode` objects or manage the semantic tree creation. You only need to use `Semantics` to describe portions of the tree, and the framework will do the rest. You can also do advanced customization, such as:

- `ExcludeSemantics`. Removes the current widget and all of its children from the semantics tree. For example, this can be used in those cases where you're testing your application and you notice that the screen reader output is confusing. For example:

```
const ExcludeSemantics(  
  child: Example(),  
)
```

- **BlockSemantics**. This widget is useful when you want to hide widgets from accessibility tools that are painted behind another widget. For example, when you popup a dialog at the center of the screen, the contents “behind” the dialog should be disallowed. This is a perfect use-case for **BlockSemantics**.
- **MergeSemantics**. This widget merges the semantics of its descendants in those cases where they make sense to be treated as a single entity. For example, imagine having a checkbox and the associated label on the left:

```
@override
Widget build(BuildContext context) {
  return Row(
    children: [
      Checkbox(
        value: isChecked,
        onChanged: (newValue) => setState(() => isChecked = newValue!),
      ),
      const Text('Enable dark theme?'),
    ],
  );
}
```

With this setup, the label (represented by the `Text`) is presented as a separate feature from the checkbox because both widgets have their own **Semantics**. In other words, the screen reader wouldn’t be sure whether those two are related or not. We can easily solve this issue:

```
MergeSemantics(
  child: Row(
    children: [
      Checkbox(
        value: isChecked,
        onChanged: (newValue) => setState(() => isChecked = newValue!),
      ),
      const Text('Enable dark theme?'),
    ],
  ),
),
```

The `Text` value is merged with the “checked” semantic state of the `Checkbox` into a single node that has both the text and the checked state. Pay attention when using this widget because if two nodes have conflicting semantics, the result will very likely be nonsensical.

All these widgets will affect how the semantics tree will be created and parsed. To tell if they work as intended, run the application on a (preferably real) device and play with the screen reader.

### 15.2.3 The semantics debugger

You should use the semantics debugger to programmatically check which parts of the application are described by a `Semantics` widget and verify how a screen reader would interpret the UI. The debugger is enabled in the `MaterialApp` or `CupertinoApp` widget with a flag:

```
const MaterialApp(  
  showSemanticsDebugger: true, // <- enables the debugger  
  home: Scaffold(  
    body: Center(  
      child: AppBody(),  
    ),  
  ),  
,
```

To understand how to use this feature, we are going to create a widget called `AppBody` and analyze it. This is the state class of the `AppBody` stateful widget:

```
class _AppBodyState extends State<AppBody> {  
  bool isChecked = false;  
  
  @override  
  Widget build(BuildContext context) {  
    return Row(  
      mainAxisAlignment: MainAxisAlignment.min,  
      children: [  
        Checkbox(  
          value: isChecked,  
          onChanged: (newValue) => setState(() {  
            isChecked = newValue!;  
          }),  
        ),  
        const Text('Dark mode?'),  
      ],  
    );  
  }  
}
```

Because of the `showSemanticsDebugger` flag, when the application is run, all widgets are replaced with a series of colored squares and strings. It is the representation of what a screen reader would see while the application is running:

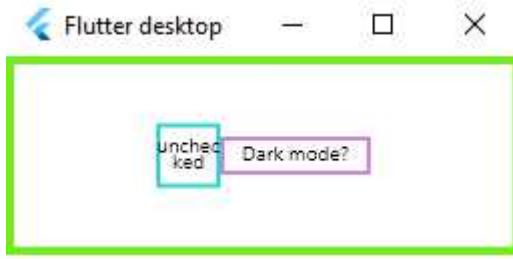


Figure 15.2: The output of the semantic debugger.

The square represents the checkbox while the small rectangle next to it is the `Text` widget. You can visually see that there are two semantic nodes (the checkbox and the text). The screen reader will recognize them as two separate entities. Let's change the code a little bit:

```
Widget build(BuildContext context) {
  return MergeSemantics( // <- We have added this
    child: Row(
      mainAxisAlignment: MainAxisAlignment.min,
      children: [
        Checkbox(/* ... code ... */),
        const Text('Dark mode?'),
      ],
    ),
  );
}
```

We have wrapped `Row` in a `MergeSemantics` widget, which roots the entire subtree under a single semantic node. Consequently, the debugger output is different from before:



Figure 15.3: The output of the semantic debugger with the `MergeSemantics` addition.

There is a single rectangle that encloses both the checkbox and the text. As such, the screen reader will treat those two widgets as a single entity. If we wanted to remove a widget from being “visible” by the screen reader, we could wrap it into `ExcludeSemantics`:

```

Row(
  mainAxisAlignment: MainAxisAlignment.min,
  children: [
    Checkbox(
      value: isChecked,
      onChanged: (newValue) => setState(() {
        isChecked = newValue!;
      }),
    ),
    const ExcludeSemantics(
      child: Text('Dark mode?'),
    ),
  ],
),

```

The `Text` widget is completely removed from the semantics tree so its node is not visible anymore in the debugger. By consequence, a single square (representing the checkbox widget) stays visible in the debugger:



Figure 15.4: The text excluded using `ExcludeSemantics`.

Even if this debugger gives useful visual hints on how the various semantic pieces look like, testing the application on a real device with a screen reader is always the best idea.

## Deep dive: Advanced localization techniques

In section 1.1 – “Internationalizing a Flutter application” we have seen how easy it is to translate strings using ARB files and the Flutter generation tool. For example, imagine you wanted to localize the “Hello” world in various languages and use English as default. You would do this:

1. Go inside the `app_en.arb` file and create a new entry for the “Hello” message (repeat this step for any other language your application supports). For example:

```
"hello": "Hello!",
```

2. Run the `flutter gen-l10n` command to automatically convert ARB files into Dart classes.
3. Use `AppLocalizations` (or, if you created it, an extension method on `BuildContext`) to use the localized string in your widget tree. For example:

```
Text(AppLocalizations.of(context).hello), // Hello!
```

This standard procedure works very well with strings that do not need to interpolate with values you provide at runtime. However, you can also use placeholders to include additional strings at runtime to the localized message. For example:

1. Go inside the `app_en.arb` file and create a new entry for the “Hello” message (repeat this step for any other language that your application supports). For example

```
"hello": "Hello, {userName}!",  
"@hello": {  
  "placeholders": {  
    "userName": {  
      "type": "String"  
    }  
  }  
}
```

Inside curly braces, we have defined the `{userName}` placeholder. Then, the `@hello` key is used to configure how the associated `hello` key is generated (make sure to use the same name). In our case, we have configured the `userName` placeholder to be a string.

2. Run the `flutter gen-l10n` command to convert ARB files into Dart classes automatically
3. Use `AppLocalizations` (or, if you created it, an extension method on `BuildContext`) to use the localized string in your widget tree. For example:

```
Text(AppLocalizations.of(context).hello('Rob')), // Hello, Rob!
```

Note that `hello` is not a getter anymore. It is a function that takes a positional parameter, whose name is `userName`. In other words, each placeholder in the ARB file is converted into a parameter so that the function can localize a string with custom values.

Placeholders are also helpful when distinguishing between singular and plural nouns. For example, say that you have to count people in a room, but they might be in different quantities. There might be “nobody”, one “person”, or multiple “people”. Here’s how you can do it:

```
{
  "test": "{count, plural, =0{noone} =1{one person} other{{count} people}}",
  "@test": {
    "placeholders": {
      "count": {
        "type": "int",
        "format": "compact"
      }
    }
  }
}
```

This syntax is a bit convoluted. First of all, we need to define the `count` placeholder (which could have been any other name) and the required `plural` identifier. Afterward, we can use a series of expressions to pluralize a word:

- `=n{ ... }`, where `n` is a number  $\geq 0$ : It defines which word to use when the value is zero. In our case, we want to use *nobody* because `0` indicates that nobody is there.
- `=other{ ... }`: It defines which word to use when no previous placeholder matched. This is required and, in our case, it's used when we want to describe the presence of two or more people.

Note that the `count` placeholder is defined as `int` (and not as `num`) to ensure that only integers are allowed. This is the result after code generation with `flutter gen-l10n` took place:

```
Text(AppLocalizations.of(context).turnout(0)), // noone.
Text(AppLocalizations.of(context).turnout(1)), // one person.
Text(AppLocalizations.of(context).turnout(2)), // 2 people.
Text(AppLocalizations.of(context).turnout(5)), // 5 people.
```

Other than numbers, you can also work with `String` placeholders. They are often used when the language distinguishes genders with different pronouns. For example:

```
{
  "test": "{gender, select, female{she} male{he} other{it}}",
  "@test": {
    "placeholders": {
      "gender": {
        "type": "String"
      }
    }
  }
}
```

In this example, `gender` is a placeholder but `select` is a required value that must always precede the pronouns list. Keep in mind that `select` is a case-sensitive statement, so spacer or capital letters for example make the difference. This is the result after code generation with `flutter gen-l10n` took place:

```
Text(AppLocalizations.of(context).test('male')), // he  
Text(AppLocalizations.of(context).test('female')), // she
```

As you have seen, all placeholders can define the `format` parameter. It is extremely useful to format numbers but also dates and currencies. For example:

- Dates can be formatted in many different ways according to the currently selected locale. Since the `gen-l10n` tool uses `intl` under the hood, `DateTime` objects can be formatted with the ICU notation (see [section 1.2.2 – “Date and time”](#) for more details). For example:

```
{  
  "birthday": "My birthday is on {date}",  
  "@birthday": {  
    "placeholders": {  
      "date": {  
        "type": "DateTime",  
        "format": "yMd",  
      }  
    }  
  }  
}
```

The `yMd` value of `format` calls the `DateFormat.yMd` named constructor and formats the date as follows:

```
// Prints: 'My birthday is on 5/20/1997'  
AppLocalizations.of(context).birthday(DateTime(1997, 5, 20)),
```

The `DateFormat` class has a small set of named constructors. If you want to build your own format using a combination of custom rules, set the `isCustomDateFormat` to `true`:

```
"date": {  
  "type": "DateTime",  
  "format": "EEE, MMM d y",  
  "isCustomDateFormat": "true",  
}
```

In this case for example, the previous date is formatted as “*Tue, May 20 1997*”.

- Numbers (including those representing currencies) are formatted differently according with the locale. You can change how a numeric placeholder is used with the `format` parameter. For example:

```
{
  "currency": "Balance: {value}",
  "@currency": {
    "placeholders": {
      "value": {
        "type": "double",
        "format": "compactSimpleCurrency",
      }
    }
  }
}
```

There are a lot of `format` values you can use. For example, `compactSimpleCurrency` prints the number with the (localized) currency symbol at the front. It shows two decimal digits unless the value is bigger than a thousand. For example:

```
AppLocalizations.of(context).currency(15.20), // $15.20
AppLocalizations.of(context).currency(115000.20)), // $115K
```

Numbers are internally manipulated with `NumberFormat`, from the `intl` package. This is a small cheat sheet with all the possible format values and the respective output (when the locale is set to `en_US`):

"format" value	Output for "118000.619"
compact	118K
compactCurrency (*)	USD118K
compactSimpleCurrency (*)	\$118K
compactLong	118 thousand
currency (*)	USD118,000.62
decimalPattern	118,000.619
decimalPercentPattern (*)	11,800,062%
percentPattern	11,800,062%

scientificPattern	1E5
simpleCurrency (*)	\$118,000.62

The (\*) next to the name indicates that you can supply an optional parameter to configure how the number is printed. For example, you can control how many digits should appear:

```
"value": {  
  "type": "double",  
  "format": "simpleCurrency",  
  "optionalParameters": {  
    "decimalDigits": 1  
  }  
}
```

In this way, `simpleCurrency` would for example print \$118,000.6 rather than \$118,000.62.

When you need to use tokens (such as = or {}) as normal characters, set the `use-escaping: true` property in the `l10n.yaml` file. In this way, the generator tool will ignore any string that is wrapped by a pair of single quotes. For example:

```
{  
  "message": "This is a curly bracket: '{'",  
}
```

The `l10n.yaml` file has a lot of options that allow you to configure how the `gen-l10n` tool behaves. Make sure to check out the official documentation<sup>96</sup> to see a table with the complete list of options and parameters.

---

<sup>96</sup> <https://docs.flutter.dev/development/accessibility-and-localization/internationalization#configuring-the-l10nyaml-file>

# 16 – Assets and images

---

## 16.1 Defining assets

An **asset** is a file that is deployed with your application and also is usable at runtime. Common types of assets are images, JSON files, font files (`.ttf`) or SQLite databases. The `pubspec.yaml` file is used to declare the assets that must be bundled with the application. For example:

```
flutter:  
  uses-material-design: true  
  
assets:  
  - assets/background.png  
  - assets/country_flags/  
  - assets/logo/my_logo.png
```

The `assets` subsection defines the path (relative to the project root) to specific images or folders. In this example, we have created the `assets/` folder in the project root and included a few PNG images. When building the application, Flutter places assets in a special archive called *asset bundle*, which can be read by your Dart code at runtime. Note that:

1. By convention, you should create a folder called `assets` in the project root and place all of your assets in there. This is not a requirement but we recommend to stay consistent with the guidelines and follow this convention.
2. The order in which directories and assets are specified does not matter. You can reference a directory (to bundle its contents) or specific files.

Once assets are declared in the `pubspec.yaml` file, they can be used with the `DefaultAssetBundle` inherited widget. For example, imagine you wanted to load a JSON file (or a generic text file). In this case, you should wait for the result of `loadString`:

```
Future<String> loadConfig() {  
  return DefaultAssetBundle.of(context).loadString('assets/cfg.json');  
}
```

The `loadString` method retrieves a string from the asset bundle and throws an exception if the resource doesn't exist. If you want to disable caching permanently, just set its parameter to `false`:

```
DefaultAssetBundle.of(context).loadString('assets/cfg.json', cache: false);
```

The `loadString` method is very clever because if the asset size is bigger than 50 KB, the decoding process is automatically deferred to a separated isolate to avoid jank on the main thread. Here is an example of how you could read a JSON file from assets:

```
class _AssetLoaderWidgetState extends State<AssetLoaderWidget> {
  late final configFuture = loadConfig();

  Future<String> loadConfig(){
    return DefaultAssetBundle.of(context).loadString('assets/cfgs.json');
  }

  @override
  Widget build(BuildContext context) {
    return FutureBuilder<String>(
      future: configFuture,
      builder: (context, snapshot) {
        if (snapshot.connectionState == ConnectionState.done &&
            snapshot.hasData) {
          return Text(snapshot.data!);
        }

        if (snapshot.hasError) {
          return const Text('The asset could NOT be loaded.');
        }

        return const Text('Loading...');
      },
    );
  }
}
```

A `FutureBuilder<T>` is used to determine the state of a future and render widgets accordingly. For more information about how it works, see *Deep dive – “Asynchronous widgets”*. To load generic binary assets rather than textual ones, the approach is the same. The only difference is that you use `load` rather than `loadString`. For example:

```
late final pdfFuture = loadPDF();

Future<ByteData> loadPDF() {
  return DefaultAssetBundle.of(context).load('assets/my_document.pdf');
}
```

The `ByteData` type is a fixed-length, random access sequence of bytes used to “pack” and “unpack” data from the disk. Whenever you don’t have access to a `BuildContext`, you can still load strings and binary assets with the `rootBundle` global variable. For example:

```
final txt = rootBundle.loadString('assets/cfg.json');
final pdf = rootBundle.load('assets/document.pdf');
```

The usage of `rootBundle` is not recommended<sup>97</sup>: you should prefer using `DefaultAssetBundle` instead.

### 16.1.1 Image assets and variants

When it comes to images, such as PNG or JPEG formats, Flutter is able to automatically choose (if possible) the asset that most closely matches the current device pixel ratio. For example, imagine you wanted to display the Dart logo at the center of your application. First of all, create the `assets` folder at the root and define the image in the `pubspec.yaml` file:

```
flutter:
  assets:
    - assets/dart_logo.png
```

The same image file might not look very well on any device because the screen dimension (and thus the resolution) changes. Flutter automatically chooses the variant that best fits the device pixel ratio if you follow a specific folder structure. For example:

```
flutter:
  assets:
    - assets/dart_logo.png # 32x32
    - assets/2.0x/dart_logo.png # 64x64
    - assets/3.0x/dart_logo.png # 128x128
```

If you created a series of sub-folders named `2.0x`, `3.0x`, and `1.5x` for example, you would map many versions of the same asset to different resolutions. For example, `2.0x` should contain the Dart logo image whose size is double the base asset. In our case:

- On devices with a pixel ratio of 2, Flutter chooses the variant in the `2.0x/` folder.
- On devices with a pixel ratio of 2.8, Flutter chooses the variant in the `3.0x/` folder.
- On devices with a pixel ratio of 1, Flutter chooses the base variant in `assets/`.
- On devices with a pixel ratio of 1.9, Flutter chooses the variant in the `2.0x/` folder.

When you don't specify `height` and `width` in the `Image` widget, the nominal resolution is used to scale the asset so that it takes the same amount of space as the main asset would have. In other

---

<sup>97</sup> <https://docs.flutter.dev/development/ui/assets-and-images#loading-assets>

words, if the pixel ratio was `3.0` and you didn't set `width` and `height`, Flutter would pick the base variant rather than the one in `3.0x/`. To make sure that variants work, all asset names must be the same. In fact, in our example, the file name is always called `dart_logo.png`.

### Note

To create asset variants, folders must be in the `Nx/` form, where `N` is a numeric identifier for the pixel ratio (such as `2.0` or `3.0`).

Try to always optimize images as much as you can. For example, before adding a PNG to your assets, make sure it was compressed (so that its size is minimized).

#### 16.1.2 Font assets

If you create an application using the `MaterialApp` widget for example, Flutter will use a series of pre-defined fonts for each platform:

- on Android and Linux, the default font is “Roboto”;
- on iOS and macOS, the default font is “San Francisco”;
- on Windows, the default font is “Segoe UI”;
- on the web, the default font is “Noto” (“Roboto” is used as fallback in a few rare cases).

To work with a custom font, you need to get one or more font files in one of these formats: `.ttc`, `.ttf` or `.otf`. The `.woff` and `.woff2` formats are only supported on specific platforms. Once you have one or more font files, create a top-level folder (generally called `fonts`) and define font files in the `pubspec.yaml` file like this:

```
flutter:  
  fonts:  
    - family: Roboto  
      fonts:  
        - asset: fonts/Roboto-Regular.ttf  
        - asset: fonts/Roboto-Italic.ttf  
          style: italic  
    - family: Lato  
      fonts:  
        - asset: fonts/Lato-Regular.ttf  
        - asset: fonts/Lato-Bold.ttf  
          weight: 700
```

The `font` attribute is used to specify custom fonts you want to use in your application. The example shows that we have downloaded inside the `fonts` folder some “*Roboto*” and “*Lato*” font files. There are a few more things to point out:

- The `family` attribute determines the font name which you use, for example, to style a `Text` widget. If you want to use a font as default for the entire application, set the `fontFamily` property as part of the theme. For example:

```
// Material
MaterialApp(
  theme: ThemeData(
    fontFamily: 'Lato', // same value as 'family' in the pubspec
  ),
  home: const HomePage(),
),

// Cupertino
const CupertinoApp(
  theme: CupertinoThemeData(
    textTheme: CupertinoTextThemeData(
      textStyle: TextStyle(
        fontFamily: 'Lato', // same value as 'family' in the pubspec
      )
    )
  ),
  home: HomePage(),
),
```

The value provided to `fontFamily` must match the `family` name in the `pubspec.yaml` file. If you want to use a font in a specific widget, use the `TextStyle` class. This is generally used by the `Text` widget:

```
const Text(
  'Hello!',
  style: TextStyle(
    fontFamily: 'Lato',
  ),
),
```

This code only applies the given font to the provided widget.

- The `style` attribute specifies whether font outlines are *italic* or *normal*. These values map to the `fontStyle` property of the `TextStyle` object.

- The `weight` attribute is generally used with bold fonts to control the overall thickness of the typeface's stroke.

If a `TextStyle` object is created with a weight or a style that does not match a font file, the Flutter engine uses a pre-built and more generic file (and it tries to extrapolate outlines for the requested weight and style).

### 16.1.3 Platform assets

You may need to handle platform-specific assets in a few special cases. For example, when you have to update your application's icon, there are various things to do (especially if you want to support all platforms). Here is a detailed list:

- Android. Inside the top-level `android/` folder, you need to navigate up to `res/`. Inside here, there are a series of other folders whose name starts with `mipmap`. They already contain a placeholder PNG file called `ic_launcher`. Replace the images to give your application the desired icon.
- iOS. Inside the top-level `ios/` folder, navigate to `AppIcon.appiconset` inside `Runner`. You will find a long series of placeholder images. Replace all of them with the images you want to use as icons.
- macOS. Repeat the same process we've described for the iOS platform.
- Windows: Inside the top-level `windows/` folder, you need to navigate to `runner/` and then `resources/`. Inside here, replace the `app_icon.ico` with your own but make sure to keep the same image size and file name.
- Web: Inside the top-level `web/` folder, update the `favicon.png` file and other images inside `icons/`. In this way, you can customize both the favicon (for browsers) and the application icon (for PWAs).
- Linux: There is no way to change the application icon from the Flutter project. The icon is set when you release the application on the official store. You will need to follow specific guides according to the deployment channel you will use.

Mobile devices also have another kind of asset that you can customize: the splash screen. Flutter already creates a default splash screen for any project. You have to change the image file:

- On Android, there is a drawable resource in `android/` called `launch_background.xml`. By default, Flutter adds a static image at the center of the screen so that you can easily replace it. If you know how layer list drawable XML files<sup>98</sup> work, you can customize the splash screen even more.
- On iOS, open the launch screen storyboard at `ios/Runner.xcworkspace` using XCode and inside `Runner/Runner/` you can change the `LaunchScreen` file. By default, the framework provides three launch image files showing the Flutter logo at the center of the screen.

This is a native mechanism that draws a transitional launch screen while the framework is loading. The splash screen appears until Flutter renders the first frame of your application. Even if Flutter start-ups very quickly, it may take a bit more than usual on slower devices (especially if you perform a lot of initializations before calling `runApp`). Defining a splash screen is always a good idea because it looks better than a black screen and is more pleasant for the users.

## 16.2 Images in Flutter

Images are particular kinds of assets in Flutter because they have a dedicated widget for rendering called `Image`. Once the image file is declared in the `pubspec.yaml` file, you can show it in this way:

```
class MyImage extends StatelessWidget {
  const MyImage({super.key});

  @override
  Widget build(BuildContext context) {
    return const Image(
      image: AssetImage('assets/background.png'),
      width: 65,
      height: 65,
    );
  }
}
```

The `AssetImage` class is used to load an image from the assets list with specific dimensions. Notice that we are allowed to use a constant constructor on the `Image` widget. You could also show images with the `Image.asset` named constructor, but we do not recommend using it because it can't be constant. For example:

<sup>98</sup> <https://developer.android.com/guide/topics/resources/drawable-resource#LayerList>

```

class MyImage extends StatelessWidget {
  const MyImage({super.key});

  @override
  Widget build(BuildContext context) {
    // Same as before but this constructor cannot be constant
    return Image.asset(
      'assets/background.png',
      width: 65,
      height: 65,
    );
  }
}

```

Either `width` and `height` should be specified, or the widget should be placed in a context that sets tight layout constraints. Other than performance reasons, passing specific constraints also avoids that image dimensions will suddenly change as the image is loaded. The `image` parameter is of `ImageProvider<T>` type, whose popular subclasses are:

- `AssetImage`: It is used to load an image from assets and automatically handles variants (if any) according to the device's pixel ratio. For example:

```

const Image(
  image: AssetImage('assets/background.png'),
),

```

The `Image.asset` named constructor internally uses `AssetImage` to handle image assets.

- `FileImage`: It is used to load an image from a `File` object. Note that `File` is a class from `dart:io` so it does not work on web platforms. This is generally used when loading an image from the filesystem at runtime. For example:

```

// 'File' is from 'dart:io' so this does NOT work on the web
Image(
  image: FileImage(File(pathToImage)),
),

```

The `Image.file` named constructor internally uses `FileImage` to handle files.

- `MemoryImage`: It decodes an `Uint8List` buffer and converts it into an image. This is a low-level way to manipulate images. For example, this code creates a gray square from a base-64 encoded string (the `base64Decode` method returns an `Uint8List` object):

```
Image(
  image: MemoryImage(
    base64Decode(
      'R0lGODlhAQABIAAAMLCwgAAACH5BAAAAAAALAAAAAABAAEAAAICRAEA0w==',
    ),
  ),
),
```

The `Image.memory` named constructor internally uses `MemoryImage` to handle in-memory assets.

- `NetworkImage`: It is used to fetch an image from an URL. The image will always be cached, regardless of cache-specific headers sent from the server. For example:

```
const Image(
  image: NetworkImage('https://website.com/image.png'),
),
```

The `Image.network` named constructor internally uses `NetworkImage` to handle files

The `Image.network` network constructor is particularly interesting. It is a convenient wrapper built around `NetworkImage` which also provides error handling and progress reporting. For example, this is how you could load an image from the web:

```
Image.network(
  'https://my-website.com/path/to/image/logo.png',
  loadingBuilder: (context, image, progress) {
    if (progress != null) {
      final int current = progress.cumulativeBytesLoaded;
      final int? total = progress.expectedTotalBytes;

      return total != null
        ? Text('${current / total * 100}%')
        : const Text('Loading...');
    }

    return image;
  },
  errorBuilder: (context, error, stackTrace) {
    return Text("Couldn't fetch the image: $error");
  },
  width: 250,
  weight: 90,
),
```

If you specify `width` and `height`, or at least one of them, the `Image` widget will not suddenly change size when the placeholder replaces the actual image. The `errorBuilder` function is called if an error occurs during image loading; if not provided, an exception is thrown. The `loadingBuilder` function is useful when you want to keep track of the loading progress. In fact:

```
loadingBuilder: (context, image, progress) {
    // When 'progress' is 'null', the image is not ready
    if (progress != null) {
        final int current = progress.cumulativeBytesLoaded;
        final int? total = progress.expectedTotalBytes;

        return total != null
            ? Text('${current / total * 100}%')
            : const Text('Loading...');
    }

    // The 'image' variable contains the Loaded image
    return image;
},
```

When `progress` is not `null`, we can calculate the progress status percentage. Note that `total` is a nullable variable because the total image size is not always available (because the server might not send the total file size to clients, for example). When `progress` is `null`, the image is completely loaded and the `image` parameter holds it. There is another similar callback you can use:

```
frameBuilder: (_, image, frame, __) {
    // When 'frame' is 'null', the image is not ready
    if (frame == null) {
        return const Text('Loading...');
    }

    // The 'image' variable contains the Loaded image
    return image;
},
```

A `frameBuilder` has the same purpose as a `loadingBuilder` with the only difference being that it does not give you the possibility to keep track of the loading progress. This is useful when you don't care about the progress status but you want to show a static placeholder. If `frame` is `null`, it means that the image cannot be rendered yet. If you define both a `loadingBuilder` and a `frameBuilder`, they will be chained together.

## Good practice

We recommend using builders because they can show the users a loading message or an animated indicator while they wait. The Flutter team recommends to:

- Use `loadingBuilder` to show the progress percentage of the fetching process.
- Use `frameBuilder` to show a waiting message (like “*Loading...*”) that does not depend on the progress status.

The `loadingBuilder` callback rebuilds its contents on every frame until the image is fully loaded. For this reason, you should only use it when you need information about the progress status.

If you don't need to know about the progress status, prefer `frameBuilder` because it rebuilds its contents only when the image is actually ready. This also means that a frame builder is less costly than a loading builder<sup>99</sup>.

When you want to change the image's opacity, use the `opacity`. If possible, try to apply the opacity directly in the image file so that the engine won't have extra work to do.

### 16.2.1 Good practices

As we have already seen, assets can be loaded in the `build` method with a constant constructor if you use `AssetImage`. It's a bit more verbose than using the `Image.asset` named constructor, but it is the approach we recommend. Whenever possible, try to also define `width` and/or `height`:

```
const Image(  
  image: AssetImage('assets/dart_logo.png'),  
  width: 32,  
  height: 32,  
)
```

If you have a big image asset (whose size may be a lot of megabytes), Flutter could take more than a few milliseconds to load it. As a result, the user would see an empty space and then the image would suddenly pop as soon as the engine loads it. To avoid this bad user experience, there are a few things you can do:

---

<sup>99</sup> <https://api.flutter.dev/flutter/widgets/Image/loadingBuilder.html>

1. Try to compress and optimize the image file as much as possible. Not only images but assets in general should take the least amount of space to not bloat the final executable size too much.
2. Find the right balance between image file sizes and the number of variants. It's ok to bundle multiple variants for the same asset. However, it's not ok to include dozens of variants for the same asset because they bloat the final executable size (and some of them might never be used by an user).
3. If you see that the image doesn't quickly load, you can take advantage of the `frameBuilder` callback, even on assets or in-memory images. It's handy to let the user know that the image is actually loading and your application didn't freeze:

```
Image(
  image: const AssetImage('assets/high_res_image_4k.png'),
  frameBuilder: (_, image, frame, __) {
    if (frame == null) {
      return const Text('Image is loading...');
    }
    return image;
  },
),
```

In the same way, you can also use the `loadingBuilder` callback. However, if you don't have to report the progress status, prefer `frameBuilder`.

4. The `FadeInImage` widget temporarily shows a placeholder image that fades out as soon as the asset is loaded. It is useful when you want to gracefully transition from a loading image to the actual one:

```
const FadeInImage(
  image: AssetImage('assets/high_res_image_4k.jpg'),
  placeholder: AssetImage('assets/loading_image.gif'),
  fadeInDuration: Duration(seconds: 1),
  fadeOutDuration: Duration(seconds: 1),
  width: 9800,
  height: 7000,
),
```

In this example, the animated gif is shown while `image` is being loaded. When the 4K asset is ready, the gif fades out and the actual asset fades in.

In addition to these techniques, you can use the `precacheImage` function to pre-fetch an image into the internal cache. For example, we could manage a huge image as follows:

```
class _MyImageState extends State<MyImage> {
  final bigImage = Image(
    image: const AssetImage('assets/high_res_image_4k.jpg'),
    width: 10300,
    height: 7000,
    frameBuilder: (_, image, frame, __) {
      if (frame == null) {
        return const Text('Image is loading...');
      }

      return image;
    },
  );

  @override
  void didChangeDependencies() {
    super.didChangeDependencies();

    // Called here because 'context' is not available in 'initState'
    precacheImage(bigImage.image, context);
  }

  @override
  Widget build(BuildContext context) {
    // Note that 'bigImage' was created in the state class and precached
    return Center(
      child: bigImage,
    );
  }
}
```

The cache may refuse to hold the image if it's disabled, if it is too large, or due to other details that are platform-specific. If you call `precacheImage` with an image that is later re-used, it will probably be loaded faster. An important note from the docs<sup>100</sup>:

---

<sup>100</sup> <https://api.flutter.dev/flutter/widgets/precacheImage.html>

Good practice from the Flutter docs.

“Callers should be cautious about pinning large images or a large number of images in memory, as this can result in running out of memory and being killed by the operating system.

The lower the available physical memory, the more susceptible callers will be to running into “Out Of Memory” issues. These issues lead to an immediate process death, with no other error messages in some cases.”

Smaller images are quickly loaded by the framework and they don’t need to be cached. Don’t use `precacheImage` with all images in your application because it would increase the chances of getting the aforementioned errors. We recommend to use it on larger images that take a notable amount of time to be rendered and would then benefit from caching.

### 16.2.2 Vectorial images

Flutter itself doesn’t have an `SvgImage` widget to parse and render vectorial images (also known as “SVG assets”). There is a very popular (and also Flutter favorite) package called `flutter_svg`<sup>101</sup>. It is a Dart-native rendering library for vectorial images. It’s easy to use:

```
@override  
Widget build(BuildContext context) {  
  return SvgPicture.asset(  
    'assets/background.svg',  
    width: 32,  
    height: 32,  
    placeholderBuilder: (context) => const Text('Svg is loading...'),  
  );  
}
```

This widget really needs to have tight constraints, coming either from `width` and `height` or from a parent (such as `SizedBox`). When the vectorial asset is complicated to parse, the engine might take more than a few milliseconds to decode it. The `placeholderBuilder` function is used to define a temporary “loading” widget that is displayed while the SVG is decoded. As it happens for the `Image` widget, there are some useful named constructors you can use:

---

<sup>101</sup> [https://pub.dev/packages/flutter\\_svg](https://pub.dev/packages/flutter_svg)

- The `SvgPicture.network` named constructor creates a widget that displays an SVG image fetched from the internet. All network images are cached regardless of HTTP headers. Here is an example:

```
SvgPicture.network(
  'https://website.com/path/to/image/background.svg',
  width: 32,
  height: 32,
  placeholderBuilder: (context) => const Text('Svg is loading...'),
),
```

This constructor also has a `headers` parameter to define extra headers for the HTTP call.

- The `SvgPicture.file` named constructor creates a widget that displays an SVG image that is loaded from the local filesystem. Some platforms, such as Android, might require special permissions to read contents from the disk. Here is an example:

```
SvgPicture.file(
  File('/path/to/file/on/disk/background.svg'),
  width: 32,
  height: 32,
  placeholderBuilder: (_) => const Text('Svg is loading...'),
),
```

Remember that `dart:io` (and thus the `File` type) is not available on web platforms.

- The `SvgPicture.string` named constructor creates a widget that displays an SVG image obtained from a Dart string. Here is an example:

```
Widget build(BuildContext context) {
  const svgString = '''<svg height="100" width="100">
    <circle cx="50" cy="50" r="40" fill="red" />
  </svg>''';

  return SvgPicture.string(
    svgString,
    width: 32,
    height: 32,
    placeholderBuilder: (_) => const Text('Svg is loading...'),
  );
}
```

If you use the `SvgPicture.asset` named constructor, specify the SVG assets in the `pubspec.yaml` file as usual. All vectorial images can always be minified and optimized. We recommend optimizing all of your vectorial assets, especially larger ones. In general:

- SVG files are great when you need images that scale a lot without losing quality. However, complex vectorial assets may take some time to load.
- Static images, such as PNG or JPG files, are great for complex images that don't need to be scaled. If you need to enlarge or reduce a PNG for example, the quality will drop.

You should determine which is the best based on the use case. In general, when the image is very detailed, the SVG representation may be hard to decode and parse. If you need to scale an image by significant factors, vectorial assets are great because they can be resized without losing quality.

## Deep dive: Loading images in CustomPainter



[https://github.com/albertodev01/flutter\\_book\\_examples/chapter\\_16/cp\\_images/](https://github.com/albertodev01/flutter_book_examples/chapter_16/cp_images/)

The `Image` widget can only be inserted in the widget tree. If you want to display an image asset on a `CustomPainter` for example, you need to use something else because widgets are not compatible with the `Canvas` object. Let's see an example:

1. To display a PNG in a `CustomPaint`, we first have to define the asset in the `pubspec.yaml` file as usual. For example:

```
flutter:  
  assets:  
    - assets/background.png
```

2. The canvas is only able to use the `Image` type that is defined in the `dart:ui` library. To avoid confusion with the `Image` type from the Flutter framework, we alias the import:

```
import 'dart:ui' as ui;
```

In this way, the `Image` object from `dart:ui` is declared as `ui.Image`.

3. Now we need a method that loads image assets and converts them into `ui.Image` objects. Since this process returns a future, a `FutureBuilder<T>` is required. Inside the state class, let's define this method:

```

late final uiImageFuture = decodeBackgroundImage();
Future<ui.Image> decodeBackgroundImage() async {
  final bundle = DefaultAssetBundle.of(context);
  final image = await bundle.load('assets/background.png');
  return decodeImageFromList(image.buffer.asUint8List());
}

```

The image is loaded as a binary resource using `load` because we need to convert it into an `Uint8List` type. The `decodeImageFromList` method is used to convert a `Uint8List` object into a `ui.Image` object.

4. The hardest part is done. Now that we have the `ui.Image` object, we can use the `Canvas` to paint the image with `drawImage`:

```

class MyPainter extends CustomPainter {
  final ui.Image image;
  const MyPainter({
    required this.image,
  });

  @override
  void paint(ui(Canvas canvas, ui.Size size) {
    canvas.drawImage(image, Offset.zero, Paint());
  }

  @override
  bool shouldRepaint(covariant MyPainter oldDelegate) =>
    image != oldDelegate.image;
}

```

5. Finally, the `FutureBuilder<ui.Image>` widget glues everything together. It waits until the image is decoded and then passes the result to the painter:

```

FutureBuilder<ui.Image>(
  future: uiImageFuture,
  builder: (context, snapshot) {
    if (snapshot.data != null) {
      return CustomPaint(
        size: const Size(90, 90),
        painter: MyPainter(image: snapshot.data!),
      );
    }
    return const CircularProgressIndicator();
  },
),

```

The procedure is complicated because you have to work with low-level `Uint8List` objects and wait for the decoding process to complete. There is no simpler way to do this because the `Image` type from the Flutter framework is not compatible with the Canvas.

# 17 – Animations

---

## 17.1 Implicit animations

Flutter comes with a series of pre-built widgets that automatically animate some of their properties. They are very easy to use and don't need to be maintained because they're part of the framework. Implicit animations are a good starting point if you're looking to animate something.

### Good practice

Implicit animations are easy to use and do not require maintenance at all. If possible, try to use them. Otherwise, if they don't cover your use case, you'll have to move to the next sections where we will show how to manually implement custom behaviors.

In this section we are going to list all of the implicitly animated widgets that are currently available in the framework. They always have a `curve` parameter, which defines how the animation motion changes over the time. The `Curves` class is a convenient collection of ready to use curve types. For example, some popular implementations are:

- `Curves.linear`: a curve that linearly increases over the time. This is the default value for all implicitly animated widgets.
- `Curves.ease`: a cubic curve that speeds up quickly and ends slowly.
- `Curves.easeInOut`: a cubic animation that starts slowly, speeds up and then ends slowly.
- `Curves.bounceIn`: an oscillating curve that grows in magnitude.
- `Curves.bounceOut`: an oscillating curve that grows and then shrinks in magnitude
- `Curves.slowMiddle`: a curve that starts quickly, slows down and then ends quickly.
- `Curves.fastOutSlowIn`: a curve that starts quickly and then eases into its final position.

Visit the official `Curves` class documentation<sup>102</sup> to see an animated preview of all kinds of curves. Each curve differs in how it distributes values over time. For example, this is a comparison between a `Curves.linear` and a `Curves.ease` where  $t$  represents the time and  $x$  the animation value:

---

<sup>102</sup> <https://api.flutter.dev/flutter/animation/Curves-class.html>

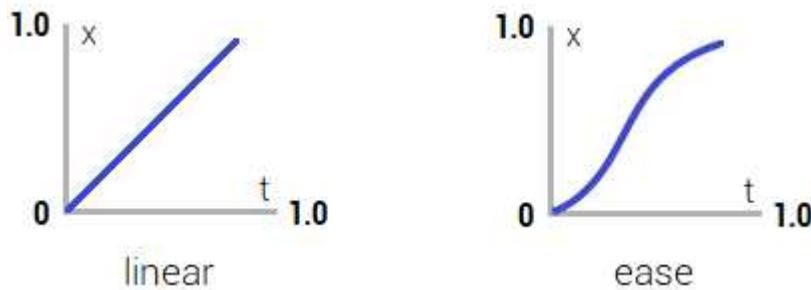


Figure 17.1: A linear curve on the left and an easing curve on the right

You can see that a linear curve moves a widget from one point to another at a constant speed. The easing curve instead starts moving the item quickly and then slowly decelerates. Curves are used to control the speed (or the “progression”) of your animation across the given `duration`. You can also reverse a curve with a `FlippedCurve` object (or the equivalent `flipped` getter):

`Curves.bounceOut`

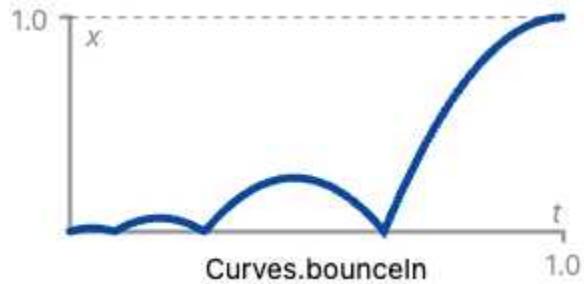


Figure 17.2: A bounce in curve type

`Curves.bounceOut.flipped`

or also

```
FlippedCurve(  
    Curves.bounceOut,  
)
```

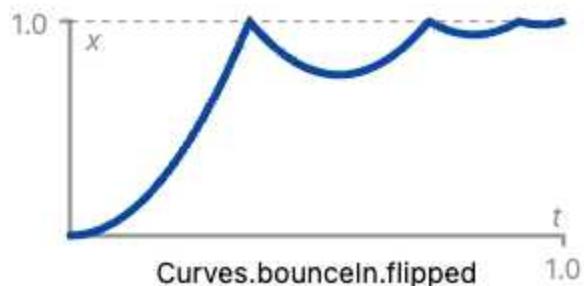


Figure 17.3: A flipped bounce in curve type.

You can also implement your curves using one of the `Curve` subclasses, such as `Cubic`. Make sure to check out the official `Curve` documentation to see all of the curve classes in the framework. Now that you know how to customize animations’ motion, let’s see a list of some popular “animated widgets”:

- **AnimatedAlign**: animated version of the `Align` widget that automatically transitions the child's position whenever its alignment changes. For example, this code moves the button from the bottom-left corner to the center (where `alignment` is a variable defined inside the state class):

```
AnimatedAlign(
  duration: const Duration(seconds: 1),
  alignment: alignment,
  child: ElevatedButton(
    onPressed: () {
      setState(() => alignment = Alignment.center);
    },
    child: const Text('Move!'),
  ),
);
```

- **AnimatedContainer**: animated version of the `Container` widget that gradually changes its values with a given duration. The widget animates many properties, such as `height`, `width`, `color`, or `alignment`. In this example, `selected` is a boolean variable in the state class:

```
AnimatedContainer(
  duration: const Duration(seconds: 1),
  width: selected ? 100.0 : 50.0,
  height: selected ? 100.0 : 50.0,
  color: selected ? Colors.green : Colors.grey,
  child: ElevatedButton(
    onPressed: () => setState(() => selected = !selected),
    child: const Text('Animate!'),
  ),
),
```

- **AnimatedPositioned**: animated version of the `Positioned` widget that transitions the child's position whenever it changes. This example moves the widget within a `Stack` in two seconds depending on the boolean `isRight` variable:

```
Stack(
  children: [
    AnimatedPositioned(
      right: isRight ? 0.0 : 200.0,
      duration: const Duration(seconds: 2),
      child: const SizedBox.shrink(),
    ),
  ],
),
```

- `AnimatedCrossFade`: this is used to gradually fade a widget into another. In other words, `AnimatedCrossFade` replaces a widget with another one at runtime with a fade animation. Note that the two child widgets should have the same sizes to avoid crops or clips:

```
AnimatedCrossFade(
  crossFadeState: firstVisible
    ? CrossFadeState.showFirst : CrossFadeState.showSecond,
  duration: const Duration(seconds: 1),
  firstChild: ElevatedButton(
    onPressed: () => setState(() => firstVisible = false),
    child: const Text('Switch to second'),
  ),
  secondChild: ElevatedButton(
    onPressed: () => setState(() => firstVisible = true),
    child: const Text('Switch to first'),
  ),
),
```

The `crossFadeState` property is used to determine which of the two children is currently visible. In our example, when `firstVisible` is changed using `setState` (or any other state management solution), one widget fades out and the other one fades in.

- `AnimatedSwitcher`: this is used to gradually animate the change from widget to another with a custom transition. An `AnimatedCrossFade` requires two widgets and always uses a fade animation. An `AnimatedSwitcher` instead allows you to use any kind of animation and it swaps two or more widgets:

```
Column(
  children: [
    AnimatedSwitcher(
      duration: const Duration(seconds: 1),
      transitionBuilder: (child, animation) { /* custom animation */ }
      child: child,
    ),
    ElevatedButton(
      onPressed: () => setState(() {
        child = child is Text
          ? const Icon(Icons.add) : const Text('Text');
      }),
      child: const Text('Change!'),
    )
  ],
),
```

In the example, the `child` parameter is of type `Widget` and whenever `setState` is called a cross fade transition is run. The `transitionBuilder` builder lets you configure a different transition effect.

- `AnimatedDefaultTextStyle`: animated version of `DefaultTextStyle` that transitions the text style from one configuration to another over a given duration. Any children of type `Text` will change style with an animation, like this:



Figure 17.4: A few frames of the `AnimatedDefaultTextStyle` widget in action.

In *Figure 17.4* you see a few frames of an `AnimatedDefaultTextStyle` widget that changes the color and the font size of a `Text`. To get that result, you can use this code where `color` and `size` are state variables that dynamically change:

```
AnimatedDefaultTextStyle(  
    style: TextStyle(color: color, fontSize: size),  
    duration: const Duration(seconds: 10),  
    child: const Text('Hello'),  
,
```

- `AnimatedOpacity`: animated version of the `Opacity` widget that transitions the opacity of the child with the given duration. This widget is relatively “expensive” because it requires painting the child on an intermediate buffer<sup>103</sup>. Here’s an example:

```
AnimatedOpacity(  
    opacity: value,  
    duration: const Duration(seconds: 2),  
    child: ElevatedButton(  
        onPressed: () => setState(() => value < 1 ? 1 : 0),  
        child: const Text('Animate opacity!'),  
,  
,
```

---

<sup>103</sup> <https://api.flutter.dev/flutter/widgets/AnimatedOpacity-class.html>

- **AnimatedPadding**: animated version of the **Padding** widget that transitions the padding between two widgets with the given duration. In this example, the **padding** variable is of type **EdgeInsets** and lives in the state class:

```
AnimatedPadding(
  padding: padding,
  duration: const Duration(seconds: 1),
  child: ElevatedButton(
    onPressed: () {
      if (padding == EdgeInsets.zero) {
        setState(() => padding = const EdgeInsets.all(20));
      } else {
        setState(() => padding = EdgeInsets.zero);
      }
    },
    child: const Text('Animate padding!'),
  ),
),
```

- **AnimatedRotation**: this widget animates the rotation of its child with the given duration. In this example, we rotate the **Text** by 90 degrees (a fourth of a turn) each time the button is pressed:

```
Column(
  mainAxisSize: MainAxisSize.min,
  children: [
    AnimatedRotation(
      turns: rotation,
      duration: const Duration(seconds: 1),
      child: const Text('Hi'),
    ),
    ElevatedButton(
      onPressed: () {
        setState(() {
          rotation += 1.0/4.0; // 90° are 1/4 of 360°
        });
      },
      child: const Text('Rotate!'),
    ),
  ],
),
```

- **AnimatedScale**: this widget animates the child's scale with the given duration. The example shows how to enlarge or resize the `FlutterLogo` widget when the button is pressed:

```
Column(  
  children: [  
    AnimatedScale(  
      scale: scale,  
      duration: const Duration(seconds: 1),  
      child: const FlutterLogo(),  
    ),  
    ElevatedButton(  
      onPressed: () => setState(() => scale = scale > 1 ? 1 : 3),  
      child: const Text('Scale!'),  
    ),  
  ],  
,),
```

Animated widgets are very helpful because they implement the animation internally and you just have to use them. Before working on your custom animations, try to see if an animated widget is good for your use case.

### 17.1.1 Hero animations

A “hero transition” is a UI pattern that keeps the user's focus on a particular item while the screen changes. For example, you can tap an image on route A, navigate to route B and keep the `Image` widget always visible on the screen. Look at this image:

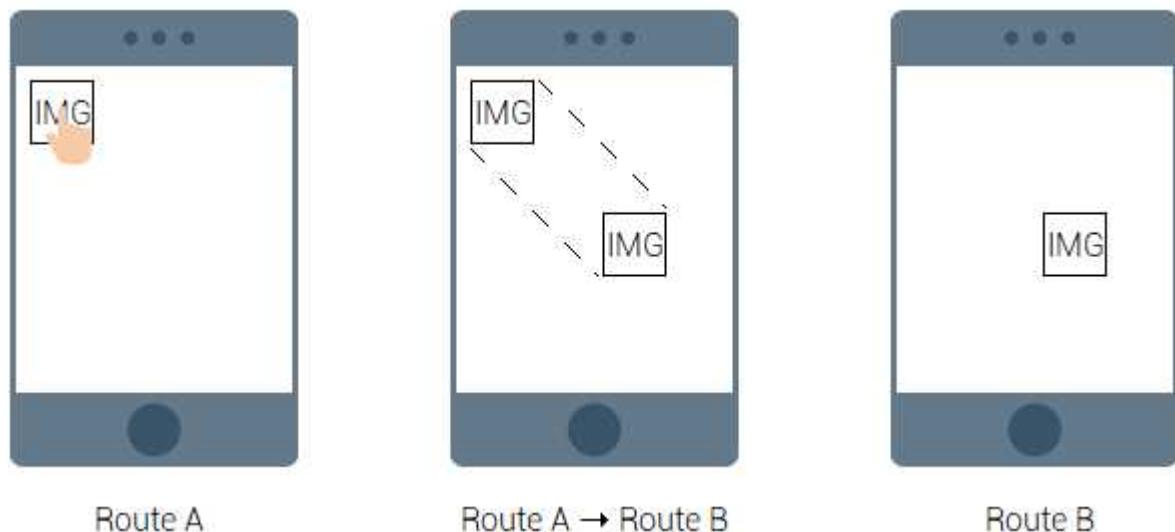


Figure 17.5: Navigating to a screen to another with a `Hero` animation on an `Image` widget.

In *Figure 17.5* you see an application with an `Image` widget in the top-left corner. When the user taps the image, the route changes. At the same time, the `Image` widget stays in the foreground and moves with a “slide” transition to the target position. Let’s see how it works:

1. Create a file named `hero_tags.dart` to contain a series of unique strings that identify the various hero animations. Make sure to use meaningful names like this:

```
const routeArouteBtagHero = 'routeA_to_routeB_hero_image';
```

2. On the initial page (called “*Route A*” in *Figure 17.5*), we need to wrap `Image` within a `Hero` widget. Note the `tag` property:

```
class RouteA extends StatelessWidget {  
    const RouteA({super.key});  
  
    @override  
    Widget build(BuildContext context) {  
        return Scaffold(  
            body: Center(  
                child: Column(  
                    mainAxisSize: MainAxisSize.min,  
                    children: [  
                        const Hero(  
                            tag: routeArouteBtagHero, // The 'tag' is string  
                            child: Image(  
                                image: AssetImage('assets/dart_logo.png'),  
                            ),  
                        ),  
                        ElevatedButton(  
                            onPressed: () async {  
                                await Navigator.of(context).pushNamed(routeB);  
                            },  
                            child: const Text('Other route'),  
                        ),  
                    ],  
                ),  
            );  
    }  
}
```

3. On the other page, we need to use another `Hero` widget and use the same tag. In this way, Flutter knows which is the “target” position for the animation in the other route. The child doesn’t necessarily need to be of the same type (`Image`), but it’s recommended:

```

class RouteB extends StatelessWidget {
  const RouteB({super.key});

  @override
  Widget build(BuildContext context) {
    return const Scaffold(
      body: Center(
        child: Hero(
          tag: routeArouteBtagHero, // Same 'tag' as in 'RouteA'
          child: Image(
            image: AssetImage('assets/dart_logo.png'),
          ),
        ),
      ),
    );
  }
}

```

When you navigate to `RouteB`, the `Image` widget stays visible and smoothly transitions to its new position. The `Hero` widgets should be located in different routes but must have the same `tag` to execute the animation correctly. Here are a few things to point out:

- While the tag must always match, the child doesn't have to be of the same type or the same size. If the child changes dimensions on the other route, Flutter will animate the dimension changes.
- The purpose of a hero is to keep the focus on the same widget while changing the route. As such, it's recommended to keep the same child in both routes for a smooth hero transition. You could surely have an `Image` on a route and a `Text` in the other, but you will notice that the animation won't be as smooth as if the widget type was the same.
- Under the hood, the `Hero` widget automatically interacts with the `Navigator` to stay visible while the route changes. As such, as long as you have a `Navigator` widget in the tree (which is always the case), hero animations will work out of the box.
- If the `Hero` subtree depends on an `InheritedWidget`, the animation may have discontinuity at the start or the end of the animation.
- If you don't use the same `tag` value on the `Hero` widgets in the two routes, the animation will not work. Routes must not contain more than one `Hero` for each `tag`.

If you want to see more complicated examples of hero animations<sup>104</sup>, visit the documentation on the Flutter website.

## 17.2 Explicit animations

When “animated widgets” don’t do what you want, it’s time to create the animation manually. The `Animation<T>` abstract class holds the animation status and its current value. It has an important subclass, called `AnimationController`, which is what you need to create and “drive” animations. For example:

```
class _HomeWidgetState extends State<HomeWidget>
  with SingleTickerProviderStateMixin {

  late final controller = AnimationController(
    vsync: this,
    duration: const Duration(seconds: 1),
  );

  @override
  void dispose() {
    controller.dispose();
    super.dispose();
  }

  @override
  Widget build(BuildContext context) {
    // The animated widget goes here...
  }
}
```

The most important feature of an `AnimationController` object is that it lets you run the animation forward or backward. It can also be listened to because it is a `Listenable` subclass. Before moving on with the example, let’s point out a few things:

- Any `AnimationController` must always be created in a stateful widget that is mixed with the `SingleTickerProviderStateMixin` mixin. It allows you to use `vsync: this`, which prevents offscreen animations from consuming unnecessary resources. In other words, the mixin makes sure that the animation won’t waste resources when not needed.

---

<sup>104</sup> <https://docs.flutter.dev/development/ui/animations/hero-animations>

- You should remember to always dispose the animation controller by calling `dispose()` in the `State.dispose` method.
- By default, an `AnimationController` linearly produces values from `0.0` to `1.0` with the given `duration`. In our example, the controller creates a series of values from `0.0` to `1.0` in a second. The number generation is tied to the screen refresh rate so (generally) 60 numbers per second are generated.

Now that you know what an animation controller is, let's move on with the example. This is the `build` method of the previous code fragment. There is a button that controls the animation status and a blue container that rotates clockwise or counterclockwise:

```
Widget build(BuildContext context) {
  return Column(
    children: [
      AnimatedBuilder( // Listens for animation status changes
        animation: controller,
        builder: (context, child) {
          return Transform.rotate(
            angle: controller.value * math.pi / 2,
            child: child,
          );
        },
        child: Container(
          width: 30,
          height: 30,
          color: Colors.blue,
        ),
      ),
      ElevatedButton(
        onPressed: () {
          if (controller.isCompleted) {
            // Rotates the box counterclockwise
            controller.reverse();
          } else {
            // Rotates the box clockwise
            controller.forward();
          }
        },
        child: const Text('Animate!'),
      ),
    ],
  );
}
```

The `AnimatedBuilder` widget, which is a subclass of `ListenableBuilder`, is essential to efficiently animate widgets. We've already seen in *chapter 12 – Section 3.2 “Performance considerations”* that the `child` parameter avoids rebuilding parts of the tree that do not change. In this example, the `Container` does not depend on the animation progress so we “cache” it using `child`:

```
AnimatedBuilder(  
    animation: controller,  
    builder: (context, child) { // When possible, use the 'child' parameter  
        return Transform.rotate(  
            angle: controller.value * math.pi / 2,  
            child: child, // <-- This references the container  
        );  
    },  
    child: Container( // This doesn't get rebuilt on every animation tick  
        width: 30,  
        height: 30,  
        color: Colors.blue,  
    ),  
,  
)
```

The `Transform` widget is used to apply various kinds of transformations (such as rotation, scaling or skewing) before the widget gets painted. In our example, the `AnimationController` produces a range of values from `0.0` to `1.0` in 60 seconds. For each value, the `AnimatedBuilder` rebuilds the `Transform.rotate` widget with a new angle and thus the container rotates at constant speed.

#### Note.

An `AnimationController` is an `Animation<double>` subclass, which is a `Listenable` itself. For this reason, you can listen for changes on a controller. This is the hierarchy:

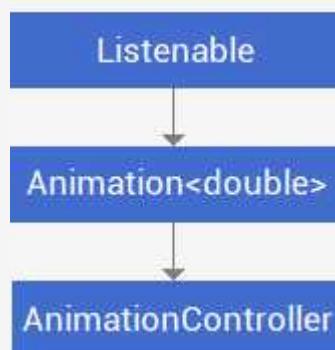


Figure 17.6: The `AnimationController` hierarchy.

Other named constructors are `Transform.translate`, which moves its child to a given offset, or `Transform.scale`, which animates the child scaling in the 2D plane. In general, these are the steps to follow to work with an `AnimationController`:

1. Create a stateful widget and mix it with `SingleTickerProviderStateMixin`, which allows the state class to be used as a `vsync`. This is very important because it prevents offscreen animations from consuming unnecessary resources.
2. Create the `AnimationController` and make sure to dispose in the state's `dispose()` method. Disposing the controller is needed to release the resources used by this object that would otherwise waste memory when not needed anymore.
3. Use an `AnimationBuilder` widget and assign the controller to its `animation` parameter to drive the animation. Make sure to include in the `builder` only those widgets that depend on the animation. If possible, use the `child` parameter to save rebuilds on children that do not depend on the animation.

The `AnimationController` class defines other interesting methods and properties to control the animation. For example, this is how to create an animation that immediately starts and repeats forever:

```
late final controller = AnimationController(  
    vsync: this,  
    duration: const Duration(seconds: 1),  
)..repeat();
```

As soon as the controller is initialized, we use the cascade operator to call the `repeat()` method (which, very intuitively, repeats the animation forever). The `stop()` method immediately stops the animation and preserves the current value, while `reset()` immediately stops the animation and brings it back to its initial value. You can even add listeners:

```
void initState() {  
    super.initState();  
  
    // You can add listeners because 'AnimationController' is a 'Listenable'  
    controller.addListener(() {  
        if (controller.isAnimating) {  
            debugPrint('rate of change = ${controller.velocity} (per second)');  
            debugPrint('${controller.status}');  
        }  
    });  
}
```

Remember that `AnimationController` is a subclass of `Listenable` and so it can be listened to. You could have also added a listener with the cascade notation in the declaration site, but we think it's not very readable:

```
late final AnimationController controller = AnimationController(  
    vsync: this,  
    duration: const Duration(seconds: 1),  
)..repeat()  
..addListener(() {  
    if (controller.isAnimating) {  
        debugPrint('rate of change = ${controller.velocity} (per second)');  
        debugPrint('${controller.status}');  
    }  
});
```

With more cascades, the code would become even less readable. We recommend initializing the controller with a `late final` declaration and adding listeners (or any other kind of setup) in the `initState` method. To add a curve to a controller, you need to use a `CurvedAnimation` object:

```
late final controller = AnimationController(  
    vsync: this,  
    duration: const Duration(seconds: 1),  
);  
  
late final curvedController = CurvedAnimation(  
    parent: controller,  
    curve: Curves.bounceIn,  
    reverseCurve: Curves.easeOut  
);
```

A `CurvedAnimation` object needs a controller because it is a sort of “wrapper” that applies a curve to an existing animation object. Another similar object is `ReverseAnimation`: it runs the animation in the reverse of another animation. For example:

```
late final controller = AnimationController(  
    vsync: this,  
    duration: const Duration(seconds: 1),  
);  
  
late final reversedController = ReverseAnimation(controller);
```

Both `CurvedAnimation` and `ReverseAnimation` do not need to be disposed but their associated controller does.

### 17.2.1 Tweens

By default, an `AnimationController` produces values from `0.0` to `1.0` with a given speed. You can also test this yourself by trying out this simple code, which prints values between zero and one on the console in two seconds:

```
late final controller = AnimationController(  
    vsync: this,  
    duration: const Duration(seconds: 2),  
);  
  
@override  
void initState() {  
    super.initState();  
  
    controller.addListener(() {  
        // Prints values in the [0.0, 1.0] range (0.0 and 1.0 are included).  
        debugPrint('${controller.value}');  
    });  
}
```

If you want to change the range of values a controller produces, use a `Tween<T>`. There is no way to change the value range produced by an `AnimationController`, but tweens come to the rescue. To produce values from `150.0` to `315.8` in four seconds, for example, you would need a controller and a tween:

```
late final controller = AnimationController(  
    vsync: this,  
    duration: const Duration(seconds: 4),  
);  
late final tween = controller.drive(  
    Tween<double>(  
        begin: 150,  
        end: 315.8,  
    ),  
);  
  
@override  
void initState() {  
    super.initState();  
  
    // Notice that we're listening on the tween rather than the controller  
    tween.addListener(() {  
        debugPrint('${tween.value}');  
    });  
}
```

Now the console prints values between `150.0` and `315.8` in four seconds. A tween is a sort of “value converter” that lets you define a custom range different from the controller’s default `[0.0, 1.0]`. In our example, `controller` always emits values from zero to one, but the tween converts all of them. In the `build` method, you can replace the controller with the tween:

```
AnimatedBuilder(  
  animation: tween,  
  builder: (context, _) {  
    return Text('${controller.value} | ${tween.value}');  
  },  
,
```

Here we’re printing both the controller and the tween values together. We would have obtained the same output if we used `animation: controller` as well. The tween is just a “wrapper” of a controller that totally depends on it and converts the values. To make another example, here’s how you’d rotate a widget with a tween. You first need to declare it:

```
late final controller = AnimationController(  
  vsync: this,  
  duration: const Duration(seconds: 1),  
,  
  
late final tween = controller.drive(  
  Tween<double>(  
    begin: 0,  
    end: math.pi,  
  ),  
,
```

For a complete rotation, we need `tween` to emit values from zero to double pi. With this setup, the tween can be used directly in the `AnimationBuilder` to drive the animation:

```
AnimatedBuilder(  
  animation: tween,  
  builder: (context, child) {  
    return Transform.rotate(  
      angle: tween.value,  
      child: child!,  
    );  
  },  
  child: const FlutterLogo(),  
,
```

We believe that this is more readable because we’re directly passing to `angle` the correct value, without making any computation. An equivalent version, without the tween, would look like this:

```

animation: controller,
builder: (context, child) {
  return Transform.rotate(
    angle: controller.value * math.pi * 2,
    child: child!,
  );
},

```

There are no performance differences, it's just a matter of readability because the tween does the calculation for us under the hood. You're not only tied to numbers, by the way:

```

late final colorTween = controller.drive(
  Tween<Color>(
    begin: Colors.red,
    end: Colors.blue,
  ),
);

late final offsetTween = controller.drive(
  Tween<Offset>(
    begin: const Offset(100, 100),
    end: const Offset(150, 210),
  ),
);

```

For example, these two tweens interpolate color and offset values within the given range. You can surely use an [AnimationController](#) and make calculations/conversions yourself to get the desired values but with tweens it is much easier. They automatically convert values for you with minimal effort. Tweens can be attached to controllers in two ways:

1. Using `drive` on the controller:

```

late final colorTween = controller.drive(
  Tween<double>(
    begin: -1.0,
    end: 1,
  ),
);

```

2. Using `animate` on the tween:

```

late final colorTween = Tween<double>(
  begin: -1.0,
  end: 1,
).animate(controller);

```

There are no differences: they are just two ways to do the same thing (the second option is maybe more readable). Tweens are mutable so you can change `begin` and `end` at runtime.

#### 17.2.1.1 Using a TweenAnimationBuilder

In the previous section, we rotated the `FlutterLogo` widget with a tween and a controller. For such a simple task, we have had quite a few things to do:

1. create an `AnimationController` to drive the animation;
2. dispose the controller;
3. create the tween;
4. create the `AnimationBuilder`.

Points 1, 2, and 4 can't be skipped because that is how an `AnimationController` is set up. In that example, we have never directly used the `controller` object. We have just used it to initialize the tween. When you need a tween but you don't need a controller, the `TweenAnimationBuilder<T>` comes to the rescue. For example:

```
double target = math.pi * 2;

@Override
Widget build(BuildContext context) {
  return Column(
    mainAxisSize: MainAxisSize.min,
    children: [
      TweenAnimationBuilder<double>(
        duration: const Duration(seconds: 1),
        tween: Tween<double>(begin: 0, end: target),
        builder: (context, value, _) {
          return Transform.rotate(
            angle: value,
            child: const FlutterLogo(),
          );
        },
      ),
      ElevatedButton(
        onPressed: () => setState(() {
          target = target > 0 ? 0 : math.pi * 2;
        }),
        child: const Text('Rotate'),
      ),
    ],
  );
}
```

Thanks to `TweenAnimationBuilder<T>`, you don't need a controller anymore. It's enough to use a state management solution to change the `tween` object and the widget will automatically animate its builder. As always, if the subtree cannot be constant, make sure to use the `child` parameter to avoid unnecessary rebuilds:

```
TweenAnimationBuilder<double>(
  duration: const Duration(seconds: 1),
  tween: Tween<double>(
    begin: 0,
    end: target,
  ),
  builder: (context, value, child) {
    return Transform.rotate(
      angle: value,
      child: child,
    );
  },
  child: Container(
    width: 10,
    height: 10,
    color: Colors.red,
  ),
),
```

A `TweenAnimationBuilder<T>` is only useful when you just need the tween itself. If you needed more control on the animation (such as the ability to stop or pause it), then you'd need to go for the complete `AnimationController` setup. It's a good practice<sup>105</sup> to create the tween directly in the `build` method to avoid interferences with the builder.

### 17.2.2 Widget transitions

Transitions are another group of widgets, driven by an `AnimationController`, that implement common behaviors for your application. For example, you can fade out a widget or scale it down with a given duration and curve. In all of the following examples, we assume that `controller` holds a reference to an `AnimationController` declared in the state class:

- `SizeTransition`: animated its own size and clips-aligns its child. You can see it as a sort of animated `ClipRect` that animates its height or width depending on the axis. For example, imagine having these widgets inside a `Column`:

---

<sup>105</sup> <https://api.flutter.dev/flutter/widgets/TweenAnimationBuilder-class.html>

```
Column(
  mainAxisSize: MainAxisSize.min,
  children: [
    SizeTransition(
      sizeFactor: controller,
      child: Center(
        child: Container(
          color: Colors.lime,
          height: 30,
          child: const Text('Hi!'),
        ),
      ),
    ),
    ElevatedButton(
      onPressed: () => controller.forward(),
      child: const Text('Animate!'),
    ),
  ],
),
```

The `controller` variable drives the animation because the `sizeFactor` parameter expects an object whose type is `Animation<double>` (which ranges from `0.0` to `1.0`). That is exactly what an `AnimationController` is. Here are some frames of the animation:



Figure 17.7: Some frames of a `ScaleTransition` animation running.

As soon as the animation controller starts the animation (by calling `forward()` for example) the container enlarges on the vertical axis until it fully appears. To animate its width rather than height, we'd simply need to set `axis` to `Axis.horizontal`.

- `ScaleTransition`: animates the scale of a widget. Using this widget is much easier than animating a `Transform.rotate` with an `AnimatedBuilder` (both approaches are efficient, but the `ScaleTransition` widget requires less code and maintenance):

```
    ScaleTransition(  
      scale: controller,  
      child: const FlutterLogo(),  
    ),
```

Similar widgets are `RotationTransition` (animated version of `Transform.rotate`) and `SlideTransition` (the animated version of `Transform.translate`).

- `FadeTransition`: animates the opacity of a widget based on the current controller state. For example:

```
FadeTransition(  
    opacity: controller,  
    child: const FlutterLogo(),  
,
```

- `AlignTransition`: animates the alignment property of an `Align` widget. It needs a tween that interpolates `AlignmentGeometry` values:

```
TweenAnimationBuilder<AlignmentGeometry>(  
    tween: Tween<AlignmentGeometry>(  
        begin: startAlign,  
        end: endAlign,  
    ),  
    duration: const Duration(seconds: 1),  
    builder: (context, value, child) {  
        return AlignTransition(  
            alignment: tween,  
            child: child!,  
        );  
    },  
    child: Container(  
        width: 90,  
        height: 50,  
        color: Colors.lightBlueAccent,  
        child: const Text('Hi!'),  
    ),  
,
```

We assume that `startAlign` and `endAlign` are two `AlignmentGeometry` variables defined in the state and they're updated with `setState` or any other state management solution.

You may have noticed that there are some similar animated widgets. For example, to animate the opacity of a widget you can either use `AnimatedOpacity` or `FadeTransition`.

The difference is that transitions allow you to explicitly control the animation with a controller, while animated widgets don't (because they manage a controller internally and you can't use it). In practice, you'd choose between them in this way:

- If you need to interact with the animation (such as having the ability to stop or reverse it), use `FadeTransition`. Transition widgets rely on `AnimationController` objects so you can drive the animation as you wish.
- If you need to play the animation with a given duration without interactions (you just need to let it run), use `AnimatedOpacity`. It's simpler to use and doesn't require a controller. You have no control on the animation controller because it's internally managed by the widget.

In other words, `AnimatedFoo` widgets are good when you don't need an animation controller to manually drive the animation. `FooTransition` widgets are good when you need to manage the controller yourself. Transitions are built-in explicitly animated widgets.

### 17.2.3 Routes transitions

Material and Cupertino libraries define standard behaviors when transitioning from one screen to another. If you don't like the default animation, you can of course customize it and implement your own using animations (and sometimes tweens). Here's how you can do it:

- **Imperative navigation.** Instead of returning the classic `MaterialPageRoute` widget, use the more generic `PageRouteBuilder` to define custom transitions. It has a callback for the route transition (`transitionBuilder`) and a callback for the route itself (`routeBuilder`):

```
PageRouteBuilder(
  pageBuilder: (context, _, _) => const MyAppPage(),
  transitionsBuilder: (context, animation, _, child)  {
    final tween = Tween(
      begin: const Offset(0, 1),
      end: Offset.zero,
    );
    final myAnimation = animation.drive(tween);

    return SlideTransition(
      position: myAnimation,
      child: child,
    );
  },
),
```

The `animation` parameter can be used as `AnimationController` so it can drive animations and tweens if needed. Transition widgets are the most preferred way of animating screens because they're easy to use.

- Declarative navigation. Instead of using a `MaterialPage` or a `CupertinoPage` class in the `Navigator`, you must define a custom page type. It basically is the same thing you'd do with the imperative approach:

```
class FadeTransitionPage<T> extends Page<T> {
  final Widget child;
  const FadeTransitionPage({
    required this.child,
    super.key,
    super.name,
    super.arguments,
    super.restorationId,
  });

  @override
  Route<T> createRoute(BuildContext context) {
    return PageRouteBuilder(
      settings: this,
      pageBuilder: (context, _, __) => child,
      transitionsBuilder: (context, animation, _, child) {
        return FadeTransition(
          opacity: animation,
          child: child,
        );
      },
    );
  }
}
```

The only difference is that here we need a dedicated class because the custom page object has to be used inside the `Navigator`:

```
Navigator(
  pages: [
    const FadeTransitionPage(
      child: HomePage(),
    ),
    // other pages here...
  ],
),
```

The `go_router` package also allows the implementation of custom transitions between screens. Instead of a `builder`, you must use a `pageBuilder` along with the `CustomTransitionPage<T>` type. Make sure to always remember to pass the `state.keyPage` value as key:

```

GoRoute(
  path: '/settings',
  pageBuilder: (context, state) {
    return CustomTransitionPage<void>(
      key: state.pageKey,
      child: const SettingsPage(),
      transitionsBuilder: (context, animation, secondaryAnimation, child) {
        return FadeTransition(
          opacity: animation,
          child: child,
        );
      });
  },
),

```

Transitions are very easy to use but of course, you're free to create any kind of animation you want. The `animation` and `secondaryAnimation` parameters are used to control the entrance and the leaving of a page.

## 17.3 Good practices

In this chapter, we've explored various techniques to create animations in Flutter. Some are easier, other are more complicated, but each address specific use cases. We believe that this summarizing table may be helpful for you to decide what to do when it comes to creating animations:

- Start by evaluating if you can use an implicit animation (*section 17.1 – Implicit animations*) or not. They are easy to use and don't require controllers. Implicit animations basically are animated versions of already existing widgets. For example, `AnimatedContainer` is the animated version of the `Container` widget.
- If no implicitly animated widgets are good for your use case, use an `AnimationController` (*17.2 – Explicit animations*) and drive the animation yourself. The animation controller helps control and change the animation status. For example, it runs the animation forward (or backward), listens for changes, or adds listeners.
- When you have an `AnimationController`, see if a transition widget may be good for your use case. The advantage of using a transition widget is that they handle everything for you: no need to set up builder widgets and manually create the animation. For example, the `FadeTransition` widget takes a controller and automatically fades in or out its child.

- By default, an `AnimationController` emits values from `0.0` to `1.0` with a certain duration and curve. When generating values on a different range, use a `Tween<T>` (*section 2.1 – Tweens*). A tween makes the math for you and converts values in the desired range. Rather than making complex calculations, let the tween handle them for you.
- If you only need a tween and you never use the associated `AnimationController`, then use a `TweenAnimationBuilder<T>`. The builder widget internally handles a controller. You just need to rebuild the widget with a new `Tween<T>` object to start the animation.

Remember that listening to an animation controller rebuilds the entire builder subtree on each tick. As such, you should maximize your effort to cache widgets as much as possible. For this purpose, use the `child` parameter of the builder function. For example:

```
AnimatedBuilder(
  animation: controller,
  builder: (context, child) {
    return Transform.rotate(
      angle: controller.value,
      child: child!,
    );
},
child: Container(
  width: 50,
  height: 50,
  color: Colors.white,
  child: const Text('Hi'),
),
),
```

As we have already seen, using the `child` parameter is very efficient. The same `child` instance is always passed back to the builder for each animation tick. In our case, the `Container` and its child will not be rebuilt whenever the controller emits a new value. Alternatively, if the child itself can have a `const` constructor, you can directly use it in the builder function:

```
AnimatedBuilder(
  animation: controller,
  builder: (context, child) {
    return Transform.rotate(
      angle: controller.value,
      child: const Text('hi'),
    );
},
),
```

Using the `child` parameter of the builder is not needed here because the `Text` widget already is a constant so it won't be rebuilt every time. You should always make sure that you're rebuilding only widgets that actually depend on the controller values.

### 17.3.1 Combining animations

The examples in this chapter are all about animating a single widget property. In some cases, you might want to apply multiple effects together. For instance, you could fade and scale together a container with a single animation. Here's how you can do it:

```
ScaleTransition(  
  scale: controller,  
  child: FadeTransition(  
    opacity: controller,  
    child: Container(  
      width: 40,  
      height: 40,  
      color: Colors.lime,  
    ),  
  ),  
,  
,
```

Even if they are two separate transition widgets, they share the same `AnimationController` so both animations will run together. This sequential overlapping of animation is defined in the Flutter documentation as "staggered" animation<sup>106</sup>. In the same way, you can animate any kind of widget. For example:

```
late final paddingAnimation = EdgeInsetsTween(  
  begin: const EdgeInsets.only(bottom: 16.0),  
  end: const EdgeInsets.only(bottom: 75.0),  
).animate(controller);
```

An `EdgeInsetsTween` is a specialized version of a tween that interpolates `EdgeInsets` values. This is an equivalent version:

```
late final paddingAnimation = Tween<EdgeInsets>(  
  begin: const EdgeInsets.only(bottom: 16.0),  
  end: const EdgeInsets.only(bottom: 75.0),  
).animate(controller);
```

---

<sup>106</sup> <https://docs.flutter.dev/development/ui/animations/staggered-animations>

Regardless of how you create the tween, this is an example of how to animate together a `Padding` and a `Container` widget within the same builder function:

```
@override
Widget build(BuildContext context) {
  return Column(
    children: [
      AnimatedBuilder(
        animation: controller,
        builder: (context, child) {
          return Padding(
            padding: paddingAnimation.value,
            child: Container(
              width: controller.value * 100,
              height: controller.value * 100,
              color: Colors.green,
            ),
          );
        },
      ),
      ElevatedButton(
        onPressed: controller.forward,
        child: const Text('Go'),
      )
    ],
  );
}
```

Even in this example, the same controller drives the padding tween and the container resizing logic. While running, the animation increments the padding and the container size at the same time. If you wanted to run these two animations separately but in sequence (the padding first and then the container resize), you'd need two separated controllers:

1. When you use two or more animation controllers, you have to change the mixin type from `SingleTickerProviderStateMixin` to `TickerProviderStateMixin`. The former should be preferred when there is a single controller because it's more efficient than the other. In any case their usage is the same, and they serve the same purpose (allowing the state to be passed as `vsync`):

```
class _StaggeredAnimationState extends State<RotatingWidget>
  with TickerProviderStateMixin {
  // state class code...
}
```

- Since we want the two animations to run separately and in sequence, we need two different animation controllers. We also need to manually chain them using a listener, which takes care of starting the size animation as soon as the padding one finishes:

```

late final paddingController = AnimationController(
  vsync: this,
  duration: const Duration(seconds: 1),
);
late final sizeController = AnimationController(
  vsync: this,
  duration: const Duration(seconds: 1),
);

@Override
void initState() {
  super.initState();

  paddingController.addListener(() {
    if (paddingController.isCompleted) {
      sizeController.forward();
    }
  });
}

@Override
void dispose() {
  paddingController.dispose();
  sizeController.dispose();

  super.dispose();
}

```

The `addListener` method is used to add a callback that, on each tick, checks whether the padding animation finished to initialize the size one as soon as possible.

- Rather than manually converting the controller values (which range from `0.0` to `1.0`) into `Padding` values, we use a tween (which does the conversion for us):

```

late final paddingAnimation = EdgeInsetsTween(
  begin: const EdgeInsets.all(0),
  end: const EdgeInsets.all(20),
).animate(paddingController);

```

- Everything is finally ready to run the animations. Since we have two controllers, we also need two `AnimatedBuilder` widgets:

```
@override
Widget build(BuildContext context) {
    return Column(
        children: [
            AnimatedBuilder(
                animation: paddingAnimation,
                builder: (context, child) {
                    return Padding(
                        padding: paddingAnimation.value,
                        child: child,
                    );
                },
            ),
            child: AnimatedBuilder(
                animation: sizeController,
                builder: (context, child) {
                    return Container(
                        width: sizeController.value * 100,
                        height: sizeController.value * 100,
                        color: Colors.green,
                    );
                },
            ),
        ],
    ),
},
```

We're using the `child` parameter on the outer `AnimatedBuilder` to avoid rebuilding the subtree while the padding animation is running. This is correct because the two animations are meant to run in sequence and separately, so while one is running the other isn't.

These examples show that combining animations with transition widgets is easy but a bit verbose. There is no other way to do it (unless you use an external package).

To run animations together, always use the same controller. To concatenate animations and run them separately, you'll need two or more controllers and the same number of `AnimatedBuilder` widgets in the `build` method.

### 17.3.2 Using AnimatedWidget

Sometimes, you may need to reuse an animation on a widget. For example, imagine you wanted to create a reusable widget that animates the thickness of a container border. You need a stateless widget with an [AnimatedBuilder](#) that listens on a controller:

```
class BorderAnimation extends StatelessWidget {
  final AnimationController controller;
  final Widget? child;

  const BorderAnimation({
    super.key,
    required this.controller,
    this.child,
  });

  @override
  Widget build(BuildContext context) {
    return AnimatedBuilder(
      animation: controller,
      builder: (context, child) {
        return Container(
          decoration: BoxDecoration(
            border: Border.all(
              color: Colors.indigo,
              width: controller.value * 1.5 + 1,
            ),
          ),
          child: child,
        );
      },
    );
  }
}
```

Note that we're also adding the usual `child` parameter to prevent unneeded subtree rebuilds. This implementation is perfectly fine and also very efficient. However, we can refactor this class to use less code and be more consistent with Flutter's animation library conventions.

When the only child of a widget is an [AnimatedBuilder](#) that listens on a controller, as it happens in our example, we can refactor the code with an [AnimatedWidget](#):

```

class BorderTransition extends AnimatedWidget {
  final Widget? child;
  const BorderTransition({
    super.key,
    required super.listenable,
    this.child,
  });

  Animation<double> get _border => listenable as Animation<double>;

  @override
  Widget build(BuildContext context) {
    return Container(
      decoration: BoxDecoration(
        color: Colors.lightGreen,
        border: Border.all(
          color: Colors.indigo,
          width: _border.value * 1.5 + 1,
        ),
      ),
      child: child,
    );
  }
}

```

The `AnimatedWidget` abstract class is a widget that automatically rebuilds whenever the value of its `listenable` object changes. In other words, it can be used to create “specialized” versions of the `AnimatedBuilder` widget. A few things to point out are:

- All transition widgets extend `AnimatedWidget` so that’s why we have called our example widget `BorderTransition`. You should keep consistency with Flutter’s animation library conventions so make sure to end the name of these classes with the “*Transition*” postfix.
- While `AnimatedBuilder` animates generic properties of a widget, our `BorderTransition` animates the border thickness of a `Container` widget. In other words, when subclassing `AnimatedWidget` you’re creating a specialized version of `AnimatedBuilder`.
- The constructor asks for a `Listenable` because `AnimatedWidget` takes care of rebuilding your widget when the controller ticks. The usual `child` parameter is used for performance reasons.

Now that we have created a custom transition widget, we can easily use it anywhere without having to always write the boilerplate code of an `AnimatedBuilder`:

```
BorderTransition(  
  listenable: controller,  
  child: const Text('Hi'),  
,
```

The controller is still required, but this is more consistent with Flutter's transition widgets pattern and also looks more familiar.

## Deep dive: Tickers and animations

An `AnimationController` object generates new values whenever the platform is ready to display new frames. In practice (if there are no “janky” frames), the controller could produce sixty or more values per second, according to the hardware on which the application is running. The controller internally uses a `Ticker` object to generate new values on each frame. Let's take a step back and see an example of a ticker. Imagine you had this code inside the state class of a stateful widget:

```
final ticker = Ticker((Duration elapsed) {  
  debugPrint('Now = ${DateTime.now()}');  
});  
  
@override  
void initState() {  
  super.initState();  
  
  // To start a ticker you must always call 'start()'  
  ticker.start();  
}
```

A `Ticker` object calls its callback once per animation frame. As such, this example prints the current date and time to the console every time a new frame is produced. If there is no jank, you will see that `DateTime.now` is printed to the console sixty (or more) times per second. Note that:

- When created, a ticker is in the “disabled” state. You must explicitly call `start` to enable it. The ticker will continue to execute its callback until you call `stop`. As such, a ticker could run for the entire application lifetime.
- A ticker can be muted if its `muted` property is set to `true` (by default, it is set to `false`). The clock of a muted ticker still continues to run but the callback is not invoked anymore.
- Don't confuse timers with tickers. A `Timer` is a countdown timer that runs at given intervals of time. A `Ticker` runs potentially forever and it executes on each new frame.

From this, you may understand why `AnimationController` internally uses a `Ticker`. Since a ticker runs its callback when a new frame is generated, the controller produces new animation values on every frame. For example, consider this state class:

```
class _MyWidgetState extends State<MyWidget>
  // this mixin implements the 'TickerProvider' interface
  with SingleTickerProviderStateMixin {

  // 'vsync' parameter is of 'TickerProvider' type
  late final controller = AnimationController(
    vsync: this,
    duration: const Duration(seconds: 1),
  );

  @override
  void dispose() {
    controller.dispose();
    super.dispose();
  }

  @override
  Widget build(BuildContext context) {
    return MyCustomAnimation(
      controller: controller,
    );
  }
}
```

A `TickerProvider` object is used to create a `Ticker` for you. Instead of manually creating a ticker every time, re-use the `SingleTickerProviderStateMixin` mixin which configures a ticker for you. This is a simplified version of the `AnimationController` class:

```
class AnimationController extends Animation<double> with A, B, C {
  Ticker? _ticker;

  AnimationController({
    required TickerProvider vsync,
    // many other parameters...
  }) {
    _ticker = vsync.createTicker(_tick);
  }

  // various members and methods such as 'forward', 'reverse', 'stop' etc...
}
```

We have removed many assertions, parameters and methods from the class to keep the focus on the ticker. The internal `_ticker` object of the controller is initialized by `createTicker`, a method of `TickerProvider` that creates a new ticker with the given callback. The `_tick` method is used to update the animation state and call `notifyListeners`.

### Note

To sum up, a `TickerProvider` mixin is needed by an `AnimationController` to create an internal `Ticker` object. The ticker updates the animation status on every new frame until it's stopped.

The `SchedulerBinding` mixin is the one that interacts with the engine to register the ticker callback to execute it for each new frame.

When you don't need the `AnimationController` anymore, you have to remember to dispose it to release the resources used by the internal `Ticker`. For example, this is the body of the controller's `dispose()` method:

```
void dispose() {
    assert(() { /* ... code ... */ });
    _ticker!.dispose();
    _ticker = null;
    clearStatusListeners();
    clearListeners();
    super.dispose();
}
```

As you can see, it takes care of disposing the ticker and it also cleans any pending listener. A `Ticker` is no more usable after it is disposed. If you try to use the controller after it was disposed, you will get runtime errors. The reason is that the internal `Ticker` is set to `null` in the `dispose()` method and thus the controller is not able to produce new values anymore.

# 18 – Forms and gestures

## 18.1 Form input widgets

The Flutter framework features a large set of input widgets. The most popular is `TextField`, which lets the user enter text in a field with either a hardware or a software keyboard. For example:

```
String text = '';  
  
@override  
Widget build(BuildContext context) {  
  return Column(  
    children: [  
      TextField(  
        onChanged: (value) {  
          setState(() => text = value);  
        },  
        ),  
      Text('Value = $text'),  
    ],  
  );  
}
```

The `onChanged` callback is called when the user changes the text of the `TextField`. In the example, we use it to retrieve the current text of the widget. By default, a `TextField` has no background and a grey line underneath. When pressed, the text field gets focused and the cursor starts blinking:



Figure 18.1: An unfocused `TextField` on the left and a focused one on the right.

The `decoration` parameter is used to control the design of the text field. For example, you could draw a rounded rectangle around the text field and add some more information (such as a hint):



Figure 18.2: A styled `TextField` when unfocused (on the left) and focused (on the right).

To obtain the result in *Figure 18.2*, you have to use the `InputDecoration` class. The values in the bottom-right corner are ruled by the `maxLength` property, which determines the maximum number of characters allowed in the text field:

```
const SizedBox(  
    width: 200,  
    child: TextField(  
        decoration: InputDecoration(  
            border: OutlineInputBorder(  
                borderRadius: BorderRadius.all(Radius.circular(10)),  
            ),  
            icon: const Icon(Icons.send),  
            helperText: 'Hello!',  
            hintText: 'Text here...',  
        ),  
        maxLength: 20,  
    ),  
,
```

A `TextField` always horizontally expands to be as big as possible so it's very common to put it into a widget that imposes tight constraints (such as a `SizedBox`). The borders can be customized with two classes that extend `InputBorder`:

1. `OutlineInputBorder`: draws a rounded rectangle around the widget with a particular gap.
2. `UnderlineInputBorder`: draws a horizontal line at the bottom of the widget. This is the default decoration.

Use `InputBorder.none` to specify that no borders should be drawn. If you wanted to decorate the text, you would need to use the `style` property of the `TextField` itself. For example:

```
const TextField(  
    style: TextStyle(  
        color: Colors.lime,  
    ),  
,
```

When a `TextField` is used to enter a password, you can obscure (or “mask”) the text. The default obscuring character is a dot (•) but it can be changed to a custom one. For example:



Figure 18.3: An obscured `TextField` that uses a hash to hide the password.

To obscure text, you need to work with the `obscureText` and `obscuringCharacter` properties of the text field. For example, this is how you obscure the text and use a custom obscuring character:

```
const TextField(  
  decoration: InputDecoration(  
    hintText: 'Password',  
,  
  obscureText: true,  
  obscuringCharacter: '#',  
)
```

Another important property we want to cover is the `restorationId` one (more information about state restorations can be found in *Deep dive – “State restoration”*). A text field allows defining its own restoration id property to restore the current text offset. For example:

```
const TextField(  
  restorationId: 'text-field-id',  
)
```

Unless the widget has a controller (more on it in the next section), the restoration id is also used to restore the text value.

### 18.1.1 Using a `TextEditingController`

In the previous section, we have seen that we can combine the `onChanged` callback with `setState` to retrieve the current text. This approach has a few intrinsic limitations:

- calling `setState` for each text change means rebuilding many widgets very often. This may be a problem if you rebuild a large portion of the subtree (especially if most widgets do not care about the text field value);
- if the user selected a portion of text, there would be no way to retrieve selection data.
- If multiple widgets across the tree depended on the text field value, `setState` might not be a good choice. Creating an inherited widget with a listenable object would be easier. In this way, the `onChanged` callback would update the listenable object using the context.

Using `onChanged` and `setState` is not practical and generally not a good idea. Instead, you should use a `TextEditingController`. It is a listenable object that notifies listeners when the text field value changes. A text controller is a `ValueNotifier<String>` that calls `notifyListeners` when the text changes. This is how to use it:

```

class _TextInputWidgetState extends State<TextInputWidget> {
  final TextEditingController = TextEditingController();

  @override
  void dispose() {
    // Do NOT forget to dispose controllers!
    textController.dispose();

    super.dispose();
  }

  @override
  Widget build(BuildContext context) {
    return Column(
      mainAxisAlignment: MainAxisAlignment.min,
      children: [
        SizedBox(
          width: 150,
          child: TextField(
            controller: textController,
          ),
        ),
        ListenableBuilder(
          listenable: textController,
          builder: (_, __) => Text(textController.text),
        ),
      ],
    );
  }
}

```

All controllers should always be disposed and `TextEditingController` is no exception. Since the controller is a listenable class, we can attach it to a text field and use a `ListenableBuilder` to listen for changes. In our example, the controller notifies its listeners when the `TextField` value changes. This approach is better than using `setState` because it only rebuilds those widgets that depend on the text field state. The controller can be initialized with a value:

```

// This string will be the initial value of the text field.
final TextEditingController = TextEditingController(
  text: 'Initial value',
);

```

We also recommend to prefer using a `TextEditingController` because of its text selection API. The `TextSelection` class gathers information about the text that is currently selected in the text field, if any. For example:



Figure 18.4: Text selection on desktop and mobile.

Thanks to the `selection` getter, you can easily determine which portion of text the user selected, either by using a physical or a software keyboard. The `start` and `end` properties indicate the initial and final selection range index. For example:

```
textController.addListener(() {
  final start = textController.selection.start;
  final end = textController.selection.end;
  debugPrint('Selected = ${textController.text.substring(start, end)}');
});
```

This code prints to the console the text that the user selected. If nothing was selected, an empty string is returned by `substring`. You can also check whether the selection is on multiple lines or not and manually increase or decrease the selection area. Make sure to visit the official [TextSelection](#) documentation<sup>107</sup> for a complete API overview.

### 18.1.2 Other form input widgets

Text fields are often fundamental in forms, but users might also need to insert other kinds of data. For example, they may need a switch to toggle a configuration or a checkbox to select a value. The framework comes with a rich series of pre-built form widgets:

- [Checkbox](#). The checkbox itself doesn't maintain any state so you need to keep track of it in a stateful widget. In this example, `isChecked` is a boolean variable defined in the state class:

```
Checkbox(
  value: isChecked,
  onChanged: (bool? newValue) {
    setState(() => isChecked = newValue!);
  },
),
```

---

<sup>107</sup> <https://api.flutter.dev/flutter/services/TextSelection-class.html>

There is a property called `tristate` that, when `true`, allows the checkbox state to be in one of those three values: selected, deselected, or indeterminate. This is what they look like:



Figure 18.5: All the possible states of the checkbox component.

Since the “indeterminate” state is represented by `null`, the `onChanged` callback provides a nullable boolean parameter:

```
Checkbox(  
  value: isChecked,  
  tristate: true, // 'false' by default  
  onChanged: (bool? newValue) {  
    //           ^ this is nullable!  
    setState(() => isChecked = newValue);  
  },  
,
```

In this example, `isChecked` has the `bool?` type. With this configuration, if you tapped the checkbox it would go from “deselected” to “indeterminate” to “selected”. When `tristate` is `false`, the checkbox flow just goes from “deselected” to “selected”. When `onChanged` is `null`, the widget is disabled.

- **CheckboxListTile**: This widget consists of a `Checkbox` and a label next to it. For example, it is very useful when you have a list of settings and you need a descriptive label next to the checkbox. For example:

```
CheckboxListTile(  
  value: isChecked,  
  title: const Text('Dark theme?'),  
  subtitle: const Text('Uses dark colors.'),  
  secondary: const Icon(Icons.cached),  
  onChanged: (bool? newValue) {  
    setState(() => isChecked = newValue!);  
  },  
,
```

It works in the same way as a `Checkbox`. It has a `title` property to define text next to the checkbox. This code produces the following result:



Figure 18.6: A `CheckboxListTile` with an icon and a subtitle.

This widget also has the `tristate` property for the “indeterminate” state, represented by `null`. When `onChanged` is `null`, the widget is disabled.

- `Radio<T>`: A radio button is used to select between a series of mutually exclusive options. This widget is generally never alone because it’s meant to be part of a group where only a single radio button can be selected at time. For example:

```
class _ExampleState extends State<Example> {
    // The state of the radio buttons (= the currently selected value)
    int option = 1;

    @override
    Widget build(BuildContext context) {
        return Column(
            mainAxisAlignment: MainAxisAlignment.min,
            children: [
                Radio<int>(
                    value: 1,
                    groupValue: option,
                    onChanged: (value) {
                        setState(() => option = value!);
                    },
                ),
                Radio<int>(
                    value: 2,
                    groupValue: option,
                    onChanged: (value) {
                        setState(() => option = value!);
                    },
                ),
            ],
        );
    }
}
```

The `groupValue` property indicates the currently selected value of a group, while `value` is the value associated with a single button. When `value` matches the `groupValue`, the radio button is selected. When the `onChanged` callback is `null`, the widget is disabled.

- **RadioListTile**: This widget consists of a **Radio** and a label to it. For example, this is very useful when you have a list of settings and you need a descriptive label next to the button. For example:

```
RadioListTile<int>(
    value: 1,
    groupValue: option,
    title: const Text('Option 1'),
    onChanged: (value) => setState(() => option = value!),
),
RadioListTile<int>(
    value: 2,
    groupValue: option,
    title: const Text('Option 2'),
    onChanged: (value) => setState(() => option = value!),
),
```

It works in the same way as a **Radio** and it has the **title** property to define text next to the button. This code produces the following result:



Figure 18.7: Two **RadioListTile** widgets.

- **Switch** and **CupertinoSwitch**: A switch is a toggle that quickly visualizes the on or off state of a single configuration. Both have the usual **onChanged** callback to maintain the widget state. When **onChanged** is **null**, the widget is disabled. For example:

<pre><code>Switch(     value: isOn,     onChanged: (state) {         setState(() {             isOn = state         });     }, ),</code></pre>	<pre><code>CupertinoSwitch(     value: isOn,     onChanged: (state) {         setState(() {             isOn = state         });     }, ),</code></pre>
--	---

The only difference between these two switches is how they look:



Figure 18.8: A `Switch` on the left and a `CupertinoSwitch` on the right

- `SwitchListTile`: This widget has some text on the left and of a `Radio` widget on the right. It is particularly useful when you have a list of settings and you need a descriptive text next to a switch. For example:

```
SwitchListTile(  
    value: isOn,  
    title: const Text('Enable feature'),  
    subtitle: const Text('Some more text here'),  
    onChanged: (newValue) => setState(() => isOn = newValue),  
) ,
```

It works in the same way as a `Switch` and it has a `title` property to define text next to the switch. This code produces the following result:



Figure 18.9: A `SwitchListTile` widget in the “enabled” state.

- `Slider` and `CupertinoSlider`: a slider is used to select from a range of values. The default behavior allows you to use a continuous range of values. For example, this is how to create a slider that selects values between `0` and `1` (edges included):

```
double value = 0;  
  
{@override  
Widget build(BuildContext context) {  
    return Slider(  
        value: value,  
        min: 0,  
        max: 1,  
        onChanged: (newValue) {  
            setState((() => value = newValue));  
        },  
    );  
}}
```

You can also configure the slider to range between `0` and `1` but with a discrete set of values using the `divisions` property. For example, if we set divisions to `5`, the slider would only take the following values: `0, 0.2, 0.4, 0.6, 0.8`, and `1`. Here's the same example but with a discrete range:

```
Slider(  
  value: value,  
  min: 0,  
  max: 1,  
  divisions: 5, // Add this  
  onChanged: (newValue) {  
    setState(() => value = newValue);  
  },  
) ,
```

If `onChanged` is not defined or `min` is equal to `max`, the slider is disabled. A discrete range looks different from a continuous range:



Figure 18.10: A continuous and a discrete slider.

A discrete slider has small dots on the slider track to indicate the divisions. Furthermore, on a continuous slider, you can freely move the thumb. On a discrete slider, the thumb jumps from one sector to another. While dragging a discrete slider for example, you can use the `label` property to add a box above the thumb:



Figure 18.11: A discrete slider with a label.

By convention, the label shows the current value but you're free to set any value you want. All of these considerations also apply to the `CupertinoSlider` widget. The only difference with `Slider` is that, when `division` is set, no dots appear in the track:



Figure 18.12: A `CupertinoSlider` widget.

All of these widgets are either part of the Material or Cupertino libraries. If you can't find the one that fits your needs, you can create your own using `Rows`, `Columns`, `Containers` and all the other foundation widgets provided by Flutter.

## 18.2 Handling forms



[https://github.com/albertodev01/flutter\\_book\\_examples/chapter\\_18/18.2/](https://github.com/albertodev01/flutter_book_examples/tree/master/chapter_18/18.2/)

Forms allow users to enter data for processing. We will build a basic login page to see how to create forms in Flutter. Since the focus of the example is on the functionalities and not on the design, we keep the UI minimal:

The form consists of three main components: a text input field labeled "Email" with a light gray placeholder, a text input field labeled "Password" with a light gray placeholder, and a blue rectangular button labeled "Login".

Figure 18.13: The login form we are going to build.

When working with forms, rather than a `TextField` you should use a `TextFormField`. Both widgets mostly do the same things but there is a slight difference:

- A `TextField` is used to let the user enter text using a keyboard.
- A `TextFormField` is a wrapper of a `TextField` that adds the possibility to validate the input and interact with the `Form` widget.

Both `TextFormField` and `TextField` do the same things so you can interchange them. However, we recommend using `TextFormField` when working with the `Form` widget and using the classic `TextField` in all the other cases. Let's start with the form setup:

```

class _LoginFormState extends State<LoginForm> {
  final GlobalKey<FormState> formKey = GlobalKey<FormState>();

  final TextEditingController emailController = TextEditingController();
  final TextEditingController passwordController = TextEditingController();

  @override
  void dispose() {
    emailController.dispose();
    passwordController.dispose();

    super.dispose();
  }

  @override
  Widget build(BuildContext context) {
    return SizedBox(
      width: 200,
      child: Form(
        key: formKey, // Use the form key here!
        child: Column(
          mainAxisSize: MainAxisSize.min,
          children: [ /* ... children ... */ ],
        ),
      ),
    );
  }
}

```

Since the user is asked to enter both email and password, we need two text controllers. It is also very important to wrap everything in a `Form` widget, which acts as a “container” for grouping and validating multiple form fields.

### Good practice

The  `GlobalKey` is used to uniquely identify the form and allows validating fields. This is the recommended<sup>108</sup> way to access a form. However, if your form is complex and you need to split it into multiple widgets, you can use the `Form.of` method. It is an inherited widget that gives you access to the form state from nested widgets below `Form`.

---

<sup>108</sup> <https://docs.flutter.dev/cookbook/forms/validation>

We can validate the email and the password using the `validator` callback. If the input is not valid, the function returns a `String` with the error message. In case of no errors, the validator must return `null`. For example:

```
children: [
  TextFormField(
    controller: emailController,
    validator: emailValidator,
    decoration: const InputDecoration(
      hintText: 'Email',
    ),
  ),
  TextFormField(
    controller: passwordController,
    validator: passwordValidator,
    obscureText: true,
    decoration: const InputDecoration(
      hintText: 'Password',
    ),
  ),
  ElevatedButton(
    onPressed: formSubmit,
    child: const Text('Login'),
  ),
],
```

As a good practice, you should define the validation logic in a separate class. We have defined the function directly inside the state as follows for simplicity. The implementation is quite naive, but it gives you the idea:

```
String? emailValidator(String? value) {
  if (value == null || !value.contains('@gmail.com')) {
    return 'Invalid email!';
  }
}

String? passwordValidator(String? value) {
  if (value == null || value.isEmpty) {
    return 'Enter some text!';
  }
}
```

In this example, we want to ensure that the email contains the Gmail domain and that the password field is not empty. The `formSubmit` method takes care of validating the contents of the text fields and it decides whether the information can be submitted or not:

```
void formSubmit() {  
    if (formKey.currentState?.validate() ?? false) { /* do something */ }  
}
```

The `formKey` object gives access to the current `Form` state (represented by the `FormState` type). The `validate()` method calls the `validator` callback of all the `TextField` widgets within the form. If everything is valid, it returns `true`. Otherwise, text fields are rebuilt with an error message. For example:

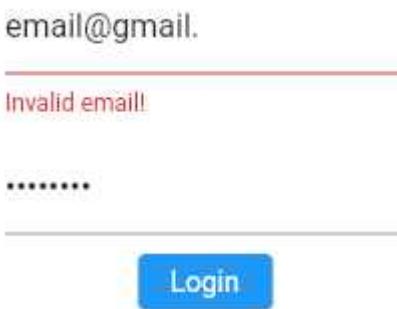


Figure 18.14: A form with an invalid text field.

A  `GlobalKey<FormKey>` gives access to the form state and thus gives you the possibility to validate data. A `TextField` is internally configured to communicate with the parent `Form` widget and expose its validation logic, if any. A normal `TextField` instead does not interact with `Form` widgets.

### 18.2.1 Focus overview

Focus management is a very important UX factor for creating forms with an intuitive flow. When a text field is selected and it's accepting inputs, it's said to have focus. You can see that a text field is focused because it has a blue outline and the cursor blinks.

#### Example

Imagine that your application had a search bar to filter a set of data. It would be nice if the text field was focused as soon as the screen is opened. In this way, the user could start writing right away without having to select the text field. This is an example of how the focus system can be helpful to enhance the user experience.

This section's considerations and examples are also valid for `TextField`, not only `TextField`. Use the `autofocus` property to automatically give focus to a text field as soon as it's created. For example:

```
TextField(  
    autofocus: true, // this is 'false' by default  
)
```

If two or more text fields have `autofocus` enabled, the focus is assigned according to the currently selected traversal algorithm (more on it in *section 2.3 – “Controlling focus traversal”*). When you have to programmatically set the focus on a specific text field, create a new `FocusNode` object. For example:

```
final focusNode = FocusNode();  
  
@override  
void dispose() {  
    focusNode.dispose();  
    super.dispose();  
}
```

A `FocusNode` is like any other controller class: it has to be disposed when not needed anymore and is a listenable (it extends `ChangeNotifier`). Once created, the `FocusNode` can be assigned to the `focusNode` property of a text field. For example:

```
Column(  
    mainAxisSize: MainAxisSize.min,  
    children: [  
        TextField(  
            focusNode: focusNode,  
        ),  
        ElevatedButton(  
            onPressed: () => focusNode.requestFocus(),  
            child: const Text('Focus it!'),  
        ),  
    ],  
)
```

Thanks to the `requestFocus` method, you can programmatically focus any text field you want. Note that only a single text field can have focus. You can remove focus from a text field with the `unfocus` method. For example:

```
onPressed: () => focusNode.unfocus(),
```

Any `FocusNode` object can be assigned to a `ListenableBuilder` to listen for changes and rebuild the subtree accordingly. For example:

```
ListenableBuilder(  
  listenable: focusNode,  
  builder: (context, _) {  
    if (focusNode.hasFocus) {  
      return const Text('The widget has focus!');  
    }  
    return const Text('The widget does NOT have focus.');
```

},  
)

One or more `FocusNode` objects create the “focus tree”, representing widgets interested in being focused. Nodes of the focus tree are linked to each other so that the active focus is passed around (by pressing the *Tab* keyboard key, for example). A few more considerations:

- Imagine you had two text fields. If you called the `focusNode.nextFocus` method while the first text field is focused, the focus would move to the second text field. In the same way, if you called `focusNode.previousFocus`, the focus would move back to the previous node.
- You cannot truly “unfocus” all nodes. The `focusNode.unfocus()` method in reality removes the focus from the associated widget and moves it somewhere else in the tree. Even if you don’t see it, there always is a widget that is focused. In practice, `unfocus()` moves the focus away from your widget to another one that is not visible.
- Using the same `FocusNode` on multiple widgets is dangerous because it mostly messes the traversal algorithm logic. For safety, do not assign the same focus node to different widgets.
- It is very important to dispose focus nodes inside the stateful widget’s `dispose` method. If you forget to do it, the `FocusNode` object will continue to live in the focus tree and it may also receive the focus ownership at a certain point.
- Make sure to never create a `FocusNode` inside the `build` method. Creating a focus node in the `build` method causes memory leaks and messes up the focus tree.

All widgets can be focused, not only text fields. Using a `FocusNode` is convenient but it is not the only way to programmatically handle the focus ownership. You generally don’t care about focus on mobile devices but it really matters when it comes to web and desktop support, mostly because of keyboard interactions.

## 18.2.2 The Focus widget

We have seen that you can directly pass a `FocusNode` to a text field to control its focus. However, you can use `Focus` (which internally manages a `FocusNode`) to focus any widget you want without creating focus nodes. For example, this is how you can make a `Text` widget focusable:

```
Focus(  
  onFocusChange: (isFocused) {  
    setState(() => isFocused ? text = 'Focused' : 'Unfocused');  
  },  
  child: Text(text),  
,
```

If you're using a keyboard, for example, you can press the `Tab` key until you see that the `Text` widget gains focus. The `Focus` widget internally inserts a new `FocusNode` in the focus tree so that it can be traversed along with the others. You could also supply your node:

```
Focus(  
  focusNode: focusNode,  
  onFocusChange: (isFocused) {  
    setState(() => isFocused ? text = 'Focused' : 'Unfocused');  
  },  
  child: Text(text),  
,
```

In this way, the `Focus` widget will use your `FocusNode` rather than internally creating a new one. This is generally useful when you need to call `requestFocus` or `unfocus`. You can even use `Focus` with text field widgets as well:

Good	Also good
<pre>Focus(   focusNode: focusNode,   child: TextFormField(), ,</pre>	<pre>TextFormField(   focusNode: focusNode, ,</pre>

In general, we recommend using the `focusNode` property of a widget (if possible). `Focus` should be used on widgets that don't expose a `focusNode` property. Some more examples are:

- `Focus` may be useful when you want to automatically focus a `Container` widget as soon as it's created. In this case, set `autofocus: true`.

- Sometimes you may need to obtain a reference to the internal `FocusNode` object of a widget to read its attributes. A descendant of `Focus` can retrieve a reference to the internal focus node using the context. Make sure to use a `Builder` widget if you're using the `of` method from the same build context. For example:

```
Focus(
    child: Builder(
        builder: (context) {
            debugPrint('Has focus = ${Focus.of(context).hasFocus}');

            return MyWidget();
        }
    ),
),
),
```

- The `skipTraversal` attribute is `false` by default but, when `true`, it excludes the focus node from the traversal algorithm. For example, if you pressed the *Tab* key on your keyboard, all those widgets with `skipTraversal: true` will not get focused.
- When `canRequestFocus` is set to `false`, calling `requestFocus` has no effect. It implies that the node is ignored by the focus traversal algorithm because focus cannot be requested.

Both `skipTraversal` and `canRequestFocus` can be configured in a `FocusNode` object as well. Let's go deeper and see how to control the focus nodes traversal.

### 18.2.3 Controlling focus traversal

As we've already seen, Flutter internally creates a focus tree whose entries are `FocusNode` items. When you press the *Tab* key on the keyboard or when you call the `focusNode.nextFocus` method for example, you are moving from a focus node to another following the rules of a default algorithm. In other words, you are traversing the focus tree using Flutter's default rules.

#### Note

The default algorithm (`ReadingOrderTraversalPolicy`) Flutter uses to traverse the focus tree is pretty good in most of the cases. It follows the “reading order”, meaning that the traversal order goes first in the reading direction and then down. If you have a specific use case and you don't like how the focus is passed from a node to another, you can customize the behavior.

Let's make a simple example with only three text fields. If you run this code on a desktop platform for example, you could press the *Tab* keyboard key and see that text fields are focused in order:

```
Column(  
  mainAxisSize: MainAxisSize.min,  
  children: const [  
    TextField(), // 1.  
    TextField(), // 2.  
    TextField(), // 3.  
,  
) ,
```

Comments on the left indicate the order. The example uses Flutter's default traversing algorithm, which starts from the top and then moves to the next node following the reading order. With a bit more code, we can change this behavior:

```
FocusTraversalGroup(  
  policy: OrderedTraversalPolicy(),  
  child: const Column(  
    mainAxisSize: MainAxisSize.min,  
    children: [  
      FocusTraversalOrder(  
        order: NumericFocusOrder(1.0),  
        child: TextField(),  
      ), // 1.  
      FocusTraversalOrder(  
        order: NumericFocusOrder(3.0),  
        child: TextField(),  
      ), // 3.  
      FocusTraversalOrder(  
        order: NumericFocusOrder(2.0),  
        child: TextField(),  
      ), // 2.  
,  
,  
) ,
```

The `FocusTraversalGroup` widget allows you to "override" the default algorithm and traverse the focus node tree differently. In particular, our example visits the text fields in the following way:

1. the topmost text field is first focused;
2. the third text field (the one at the bottom) is then focused;
3. the second text field (the middle one) is focused last.

Thanks to `OrderedTraversalPolicy` we can use `FocusTraversalOrder` widgets to determine the traversal order using `double` values. Instead of `NumericFocusOrder` we could also have used the `LexicalFocusOrder` type, which does the same but uses strings:

```
FocusTraversalGroup(  
  policy: OrderedTraversalPolicy(),  
  child: const Column(  
    mainAxisSize: MainAxisSize.min,  
    children: [  
      FocusTraversalOrder(  
        order: LexicalFocusOrder('a'),  
        child: TextField(),  
      ), // 1.  
      FocusTraversalOrder(  
        order: LexicalFocusOrder('c'),  
        child: TextField(),  
      ), // 3.  
      FocusTraversalOrder(  
        order: LexicalFocusOrder('b'),  
        child: TextField(),  
      ), // 2.  
    ],  
  ),  
,
```

The order here follows the “lexical order” of the strings so `a` is first, `b` is second, and `c` is third. When you set the `policy` parameter, you basically change the rules that determine which widget comes “next” or “previous” when traversing the focus tree. There are three kinds of policies overall:

1. `ReadingOrderTraversalPolicy`, the default one, traverses the focus tree in the reading direction and then down.
2. `OrderedTraversalPolicy` traverses the focus tree by following the order determined by `FocusTraversalOrder` widgets.
3. `WidgetOrderTraversalPolicy` traverses the focus tree by following the widget tree hierarchy order. This policy is used when the desired order is the one in which widgets are created in the widget tree.

If you don’t define a `policy`, then it defaults to `ReadingOrderTraversalPolicy`. In most of the cases, when a custom traversal is needed, the `OrderedTraversalPolicy` policy is chosen. You can even implement your own policies by extending the `FocusTraversalPolicy` abstract type.

## 18.3 Gestures

The term **gesture** refers to a group of “semantic actions” such as tapping, dragging, or panning for example. Any widget can be tapped or dragged, not only buttons. For example, you already know that when you tap an `ElevatedButton`, the `onPressed` callback is triggered:

```
ElevatedButton(  
    onPressed: () => debugPrint('Button callback'),  
    child: const Text('Button'),  
)
```

Some widgets, such as a `Container` for example, don’t provide the `onPressed` callback by default. However, you can make a container “tappable” thanks to the `GestureDetector` widget. Despite the name, the `onTap` callback responds to both finger taps (on mobile platforms) and mouse clicks (on desktop platforms). For example:

```
GestureDetector(  
    onTap: () => debugPrint('Container tapped(clicked!)'),  
    child: Container(  
        width: 30,  
        height: 30,  
        color: Colors.blue,  
)  
)
```

The `GestureDetector` widget recognizes various kinds of gestures and triggers the associated non-null callbacks. For example, if `GestureDetector` recognizes a double tap but `onDoubleTap` is `null`, nothing happens. For example:

```
GestureDetector(  
    onTap: () => debugPrint('Container tapped(clicked!)'),  
    onTapDown: (_) => debugPrint('Tap down'),  
    onTapUp: (_) => debugPrint('Tap up'),  
    child: Container(  
        width: 30,  
        height: 30,  
        color: Colors.blue,  
)  
)
```

In this case, there `onDoubleTap` property is not set and thus double taps will be recognized but ignored. The `onTapDown` callback is called when a tap has contacted the screen on a particular area; `onTapUp` is called when the previously tapped area has stopped making contact. Finally, `onTap` is

fired when a full tap or click happens (so `onTapDown` and `onTapUp` have been fired in sequence). Let's see what happens when you click with the mouse on the container in the example:

1. When you press the left mouse button, `onTapDown` is fired (only once). If you keep pressing, nothing happens.
2. When you release the left button, `onTapUp` is fired (only once). At the same time, `onTap` is also fired.

As you can see, the `GestureDetector` widget is able to fire multiple events if needed. It has more than fifty callbacks<sup>109</sup> to recognize gestures. Visit the official documentation to see the complete coverage of the `GestureDetector` API.

### Note

Each gesture (tap, double tap, drag, pan...) has its own recognizer. The framework has a gesture arena where recognizers could join to determine the winner. For example, this is (a simplification of) what happens when you drag your finger or your mouse cursor on the screen:

1. A dragging gesture is detected by the framework, so both the horizontal and vertical dragging recognizers join the gesture arena.
2. The recognizers look at the pointer move events. If the user moved the pointer more than a certain number of logical pixels in the horizontal axis, then the horizontal recognizer would win. Winners fire the callback while losers leave the arena and don't fire the callback.

This is a case where the arena is beneficial because there only are two kinds of drag recognizers. If one wins, the other is discarded and no further gesture disambiguation is needed.

In case of taps, for example, there are various kinds of recognizers (`onTap`, `onTapDown`, `onTapUp`, `onDoubleTap`...) so the arena is quite "crowded". There might be multiple winners but the algorithm is able to efficiently recognize all of them.

---

<sup>109</sup> <https://api.flutter.dev/flutter/widgets/GestureDetector-class.html>

A `GestureDetector` defers to its child sizing and behavior. When the child is invisible, touches are ignored by default. However, you can use the `behavior` property to override this behavior:

- `HitTestBehavior.deferToChild`: the default behavior when `child` isn't `null`. The child can receive gesture events within its bounds. This is good in most of the use cases.
- `HitTestBehavior.translucent`: the default behavior when `child` is `null`. The child can receive gesture events within its bounds and also allows interaction with targets visually behind it. For example, in a `Stack` a translucent behavior makes `GestureDetector` and other children behind it interactive.
- `HitTestBehavior.opaque`: has the opposite behavior of `translucent`. The child is able to receive gesture events within its bounds but no interactions are allowed with targets visually behind it.

The `IgnorePointer` widget is used to make a widget invisible during hit testing. In other words, it's the `GestureDetector` counterpart because it removes any kind of interaction in the subtree.

### 18.3.1 Drag and drop



[https://github.com/albertodev01/flutter\\_book\\_examples/tree/master/chapter\\_18/18.3.1](https://github.com/albertodev01/flutter_book_examples/tree/master/chapter_18/18.3.1)

In this section, we're going to build a simple application where the user is asked to drag and drop a number in a box. The number at the center is randomly generated and it has to be dropped in the correct box:

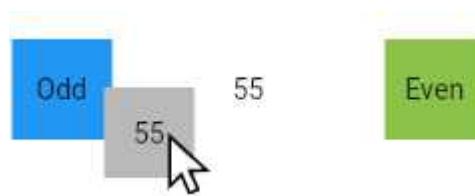


Figure 18.15: The number at the center is dragged to the left or right and dropped in a box.

Since `55` is odd, we need to select it, drag the gray rectangle to the left and drop it in the blue box. You could start from this example and create a game with a timer and maybe some more advanced features such as leaderboards or multi-player mode.

As you can see in *Figure 18.15*, the UI is mainly divided into three parts: two containers at the edges and a number at the center. As such, it's a good idea to create three separate widgets:

```
return const Row( // Note that we can make the entire subtree 'const'
  mainAxisAlignment: MainAxisAlignment.min,
  children: [
    OddContainer(),
    Padding(
      padding: EdgeInsets.symmetric(horizontal: 60),
      child: CentralNumber(),
    ),
    EvenContainer(),
  ],
);
```

Since `OddContainer` and `EvenContainer` are very similar, we only analyze one. Both of them have a `DragTarget<int>` widget, which is used as a “landing area” for objects being dragged. The `int` type in the diamonds indicates the expected data type:

```
class EvenContainer extends StatelessWidget {
  const EvenContainer({super.key});

  @override
  Widget build(BuildContext context) {
    return DragTarget<int>(
      onAccept: (data) {
        ScaffoldMessenger.of(context).showSnackBar(
          SnackBar(
            content: Text('Even number accepted: $data'),
          ),
        );
      },
      onWillAccept: (data) => data?.isEven ?? false,
      builder: (context, _, __) {
        return Container(
          width: 50,
          height: 50,
          color: Colors.LightGreen,
          child: const Center(
            child: Text('Even'),
          ),
        );
      },
    );
  }
}
```

The `onWillAccept` function is used by the widget to determine whether accepting or not the data being dropped. In our case, we only want to take even numbers, hence the condition:

```
onWillAccept: (data) => data?.isEven ?? false,
```

The `onAccept` function is called when an acceptable piece of data is dropped on the widget. In our example, we know that this callback is called when an even number is dropped. The `OddContainer` widget is very similar to this one; it only differs for the target conditions:

```
onAccept: (data) {
  ScaffoldMessenger.of(context).showSnackBar(
    SnackBar(
      content: Text('Odd number accepted: $data'),
    ),
  );
},
onWillAccept: (data) => data?.isOdd ?? false,
```

Let's see how the widget at the center of the two boxes is made. For simplicity, the random number is stored in a `static` variable. In a more advanced scenario, the number could be generated by a stream in a separate state object. To freely drag the number over the screen and release it on a `DragTarget<int>`, we need to use a `Draggable<int>` widget:

```
class CentralNumber extends StatelessWidget {
  static final _value = Random().nextInt(100) + 1;
  const CentralNumber({super.key});

  @override
  Widget build(BuildContext context) {
    return Draggable<int>(
      data: _value,
      feedback: Material(
        child: Container(
          width: 45,
          height: 45,
          color: Colors.black26,
          child: Center(
            child: Text('${_value}'),
          ),
        ),
      ),
      child: Text('${_value}'),
    );
  }
}
```

The `data` parameter holds the value that will be dragged (in our case, an `int` value). The `feedback` widget instead is only shown while the user is dragging data on the screen. In the example, we can clearly see that the gray container represents the `feedback` widget while the `child` just holds the number:

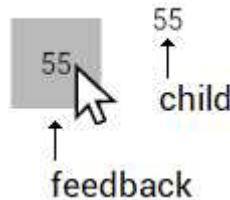


Figure 18.16: The feedback and the child of a `Draggable<T>` widget.

Defining `feedback` is considered a good practice because it gives the user a “dragging feedback”.

### 18.3.2 The MouseRegion widget

The `MouseRegion` widget is used to keep track of mouse movements and thus it's mainly useful on web and desktop applications. For example, this is how you can keep track of the mouse cursor position when it hovers on a widget:

```
class _MouseDetectorState extends State<MouseDetector> {
  var position = Offset.zero;

  @override
  Widget build(BuildContext context) {
    return MouseRegion(
      onHover: (mouseData) {
        setState(() => position = mouseData.position);
      },
      child: Padding(
        padding: const EdgeInsets.all(20),
        child: Text('$position'),
      ),
    );
  }
}
```

The `onHover` callback is only invoked when the cursor is above `Padding` and no mouse buttons are pressed. There also are two other callbacks that are fired when the cursor enters or exits the area of the child. They are called `onEnter` and `onExit`:

```

class _MouseDetectorState extends State< MouseDetector > {
  var position = Offset.zero;
  var enters = 0;
  var exits = 0;

  @override
  Widget build(BuildContext context) {
    return MouseRegion(
      onHover: (mouseData) {
        setState(() => position = mouseData.position);
      },
      onEnter: (mouseData) {
        setState(() => ++enters);
      },
      onExit: (mouseData) {
        setState(() => ++exits);
      },
      child: Padding(
        padding: const EdgeInsets.all(20),
        child: Text('($enters | $exits) $position'),
      ),
    );
  }
}

```

In this example, whenever the mouse enters and exits the `MouseRegion`'s child area, counters are incremented (regardless of whether buttons are pressed or not). If you use the `cursor` property, you can change the cursor appearance:

```

const MouseRegion(
  cursor: SystemMouseCursors.forbidden,
  child: ElevatedButton(
    onPressed: null,
    child: Text('Disabled'),
  ),
),

```

In this case, if you hover the child of `MouseRegion`, the cursor will indicate that a certain operation will not be carried out. For example, this is how the cursor looks like on Windows (it may be different on other operating systems):



Figure 18.17: How the cursor looks when defining `SystemMouseCursors.forbidden` on a child.

## Deep dive: Actions, intents, and shortcuts



[https://github.com/albertodev01/flutter\\_book\\_examples/chapter\\_18/actions\\_intents/](https://github.com/albertodev01/flutter_book_examples/chapter_18/actions_intents/)

For a fully-fledged user experience on your Flutter applications, keyboard integration is something you really need to consider if you're also targeting web and desktop. Some widgets are already well integrated with keyboard shortcuts, such as `TextField` and `TextFormField`:

```
TextField(  
  onSubmitted: (text) {  
    // 'Enter' key was pressed.  
  },  
,  
TextFormField(  
  onSubmitted: (text) {  
    // 'Enter' key was pressed.  
  },  
,
```

For example, when `TextField` or `TextFormField` is focused, you can use `CTRL+A` and then `CTRL+C` to select the entire text and copy it to the clipboard. The `onSubmitted` callback is generally invoked whenever you press *Enter* on your physical keyboard or *Search* on a software keyboard. Some other widgets may not be keyboard-friendly by default. For example:

```
class CenteredText extends StatelessWidget {  
  final String text;  
  const CenteredText({  
    super.key,  
    required this.text,  
  });  
  
  @override  
  Widget build(BuildContext context) {  
    return Center(  
      child: Text(text),  
    );  
  }  
}
```

Here we don't have a text field, so you cannot select the text and copy it to the clipboard using the keyboard. In these cases, you have to manually make the widget compatible with keyboard events. For example, let's try to add to `CenteredText` the possibility of copying the text to the clipboard using the `CTRL+C` keys combinations. There are a few steps to follow:

1. Flutter uses the `Intent` class to represent a generic action that the user wants to perform. Since we want to “select and copy” the text contained in `CenteredText`, a good name for the class may be `SelectAllIntent`:

```
class SelectAllIntent extends Intent {  
  const SelectAllIntent();  
}
```

An intent represents a generic action you want to make, but it doesn’t contain any logic or implementation detail.

2. The `Action` type defines the actual implementation of an `Intent`. In other words, an `Action` implements the logic that an `Intent` is expected to have. For example, we want to copy a string to the clipboard and report it to the user. The only abstract method you must override is `invoke` (see *Deep dive: Asynchronous widgets* to learn about “asynchronous gaps”):

```
class SelectAllAction extends Action<SelectAllIntent> {  
  final BuildContext context;  
  final String text;  
  SelectAllAction(this.context, this.text);  
  
  @override  
  bool isEnabled(covariant SelectAllIntent intent) => text.isNotEmpty;  
  
  @override  
  Future<Object?> invoke(covariant SelectAllIntent intent) async {  
    await Clipboard.setData(ClipboardData(text: text));  
  
    // the 'context.mounted' check is to avoid 'asynchronous gaps'  
    if (context.mounted) {  
      ScaffoldMessenger.of(context).showSnackBar(const SnackBar(  
        content: Text('Data copied to the clipboard!'),  
      ));  
    }  
  
    return null;  
  }  
}
```

The `Clipboard` utility class is used to copy and/or paste data programmatically so `setData` copies data to the system clipboard. If the action made some calculations (which is not our case), it could also return data. We also have overridden `isEnabled` because we want this action to be invoked only if there is text to copy.

- Actions and keyboard commands only work if the child has focus. Since a `Container` itself cannot be focused, we need to wrap it using `Focus`:

```
Actions(
  actions: <Type, Action<Intent>>{
    SelectAllIntent: SelectAllAction(context, widget.text),
  },
  child: Focus(
    onFocusChange: (hasFocus) {
      setState(() => hasFocus ? bgColor = Colors.black12 : null);
    },
    child: Container(
      width: 50,
      height: 50,
      color: bgColor,
      child: Center(
        child: Text(widget.text),
      ),
    ),
  ),
),
```

To see that the widget has focus, we change its background color. Actions are inserted in the widget tree using the `Actions` widget, where we can pass in a map the intent we have created.

- The last step is associating the CTRL+C keyboard commands with `SelectAllAction`. To do so, we have to wrap the `Actions` widget below a `Shortcuts` widget. It creates key bindings to specific actions for its descendants:

```
Shortcuts(
  shortcuts: <LogicalKeySet, Intent>{
    LogicalKeySet(
      LogicalKeyboardKey.control,
      LogicalKeyboardKey.keyC,
    ): const SelectAllIntent(),
  },
  child: Actions(
    actions: <Type, Action<Intent>>{
      SelectAllIntent: SelectAllAction(context, widget.text),
    },
    child: Focus( /* ... code ... */),
  ),
);
```

We use the `LogicakLeySet` type to associate `CTRL` and `C` keys (pressed together) to the `SelectAllIntent` intent. In this way, when the user presses `CTRL+C`, the action is called. You could attach more shortcuts to the same widget:

```
shortcuts: <LogicalKeySet, Intent>{
  LogicalKeySet(
    LogicalKeyboardKey.control,
    LogicalKeyboardKey.alt,
    LogicalKeyboardKey.arrowUp,
  ): const AnotherIntent(),
  LogicalKeySet(
    LogicalKeyboardKey.digit4,
  ): const YetAnotherIntent(),
},
```

The important thing is to make sure that `Actions` is a `Shortcut` child and not vice versa.

Remember that `Shortcuts` and `Actions` widgets only work if the child is focused. This is the main reason why we've wrapped the container with a `Focus` widget (other than changing the background color).

### The FocusableActionDetector widget

We have just seen that adding keyboard support to a widget is not very intuitive and requires a lot of code. You have to use a `Focus` widget (if needed) below `Actions` and the `Shortcuts` widget has to be above everything. The order in which you nest widgets does matter. For example:



Figure 18.18: The correct hierarchy is on the left. On the right, `Focus` and `Shortcuts` are swapped.

In *Figure 18.18*, the hierarchy on the right is wrong because `Actions` and `Shortcuts` only work if a child has focus. If you place those widgets in the wrong order, no errors appear but the keyboard interaction won't work.

Other than making sure of placing widgets in the correct order, you need to take care of focus. This is what our example would look like if we also added mouse events with `MouseRegion`:

```
@override
Widget build(BuildContext context) {
    return MouseRegion(
        onHover: mouseHoverCallback,
        child: Shortcuts(
            shortcuts: <LogicalKeySet, Intent>{
                LogicalKeySet(
                    LogicalKeyboardKey.control,
                    LogicalKeyboardKey.keyC,
                ): const SelectAllIntent(),
            },
            child: Actions(
                actions: <Type, Action<Intent>>{
                    SelectAllIntent: SelectAllAction(context, widget.text),
                },
                child: Focus(
                    onFocusChange: (hasFocus) {
                        setState(() => hasFocus ? bgColor = Colors.black12 : null);
                    },
                    child: Container(
                        width: 50,
                        height: 50,
                        color: bgColor,
                        child: Center(
                            child: Text(widget.text),
                        ),
                    ),
                ),
            ),
        );
}
```

This is way too much code for binding a keyboard shortcut to a widget. When you have to define actions and keyboard bindings, the `FocusableActionDetector` widget is generally a better option. It combines the functionalities of `Shortcuts`, `Actions`, `Focus`, and `MouseRegion`. For example, this is how the above code would look like if we used the `FocusableActionDetector` widget instead:

```
@override
Widget build(BuildContext context) {
  return FocusableActionDetector(
    onShowHoverHighlight: mouseHoverCallback,
    shortcuts: <LogicalKeySet, Intent>{
      LogicalKeySet(
        LogicalKeyboardKey.control,
        LogicalKeyboardKey.keyC,
      ): const SelectAllIntent(),
    },
    actions: <Type, Action<Intent>>{
      SelectAllIntent: SelectAllAction(context, widget.text),
    },
    onFocusChange: (hasFocus) {
      setState(() => hasFocus ? bgColor = Colors.black12 : null);
    },
    child: Container(
      width: 50,
      height: 50,
      color: bgColor,
      child: Center(
        child: Text(widget.text),
      ),
    ),
  );
}
```

This code is shorter and much more readable. You also do not have to remember the order in which **Shortcuts**, **Actions** and **Focus** have to be nested. Everything is internally handled by the widget. This dramatically reduces the complexity of your code and the maintenance cost.

# 19 – Testing Flutter applications

---

## 19.1 Introduction to widget testing

In *chapter 9 – Section 2 “Testing Dart Code”* we have seen how to test a “pure” Dart application. To test widget classes, you’ll need to import the `flutter_test` package (which is built on top of Dart’s `test` package). It allows writing both unit and widget tests with a single dependency:

```
dev_dependencies:  
  flutter_test:  
    sdk: flutter
```

The goal of a widget test is to make sure that a certain widget is rendered as expected. For example, you can make sure that a widget contains a certain subtree or it can be scrolled in a certain direction without causing errors. To make a practical example, imagine you wanted to test `MyApp`:

```
class MyApp extends StatelessWidget {  
  const MyApp({super.key});  
  
  @override  
  Widget build(BuildContext context) {  
    return const MaterialApp(  
      home: Scaffold(  
        body: Center(  
          child: Text('My app!'),  
        ),  
      ),  
    );  
  }  
}
```

As it happens for any other Dart project, you need to go inside the `test/` folder and create a test file. We could call it `myapp_test.dart`, making sure to add the usual `_test` suffix at the end of the file name. For example:

```
// 'myapp_test.dart' file  
import 'package:flutter_test/flutter_test.dart';  
  
void main() {  
  group('Widget tests', () {  
    // test code here...  
  });  
}
```

If we wrote a unit test for `MyApp`, we would only be able to validate some class properties. To test how the widget is rendered we would need a `BuildContext`, but we cannot manually create one. For example:

```
test('Smoke test', () {
  expect(const MyApp().key, isNull);

  // How do we get a 'context'?
  expect(const MyApp().build(context), isA<Widget>());
});
```

As you can see, we have no way to obtain a reference to the context so we cannot test the widget behavior itself. The `flutter_test` package exactly solves this issue: it gives you a way to mount your widgets in a special environment with a context. Instead of the `test` method, for widget tests you have to use `testWidgets`. For example:

```
group('Widget tests', () {
  testWidgets('Smoke test', (WidgetTester tester) async {
    await tester.pumpWidget(const MyApp());
  });
})
```

The `tester` variable is what puts your widget in a “special” environment that lets you drive the test as if `MyApp` was mounted on a real Flutter application. The `tester.pumpWidget` method internally builds your widget on the widget tree and allows you inspections using `finders`. For example:

```
testWidgets('Smoke test', (WidgetTester tester) async {
  // builds a widget tree for testing purposes
  await tester.pumpWidget(const MyApp());

  expect(find.byType(MaterialApp), findsOneWidget);
  expect(find.byType(Scaffold), findsOneWidget);
  expect(find.text('My app!'), findsOneWidget);
});
```

The top-level `find` property, from the `flutter_test` package, allows you to ensure that a widget is located in the tree as you would expect. In our case:

- `find.byType` looks for a widget of the given type in the widget tree;
- `find.text` looks for `Text` widgets in the tree that contain the given string.

Finders are covered more in detail in *Section 1.1 “Using finders”*. The `findsOneWidget` constant is a matcher used to assert that the finder locates exactly one widget in the tree. There are various matchers to use:

- `findsOneWidget`, ensures that only one widget is found in the tree;
- `findsNWidgets`, ensures that a specific number of widgets are found in the tree;
- `findsNothing`, ensures that no widgets are found in the tree.

Most of those finders are just short hands that use `findsNWidgets` under the hood. For example, in this test `findsNothing` is the equivalent of `findsNWidgets(0)`:

```
testWidgets('Smoke test', (WidgetTester tester) async {
  await tester.pumpWidget(const MyApp());

  // For example, 'findsNothing' is a shorter version of 'findsNWidgets(0)'
  expect(find.byType(MaterialApp), findsOneWidget);
  expect(find.text('My app'), findsNothing);
  expect(find.byType(Builder), findsNWidgets(3));

  // The above tests are the same as:
  expect(find.byType(MaterialApp), findsNWidgets(1));
  expect(find.text('My app'), findsNWidgets(0));
  expect(find.byType(Builder), findsNWidgets(3));
});
```

Tests are run with the `flutter test` command. When writing tests, we recommend grouping them in logical blocks. This can result in a great developer experience because when tests fail, the logger will report the full path that follows group names. For example:

```
group('MyWidget tests', () {
  group('Scrolling tests', () {
    testWidgets('Scroll test 1', (tester) async { /* test */ });
    testWidgets('Scroll test 2', (tester) async { /* test */ });
  });

  group('Gesture tests', () {
    group('Taps', () {
      testWidgets('Tap test 1', (tester) async { /* test */ });
      testWidgets('Tap test 2', (tester) async { /* test */ });
    });
    group('Long taps', () {
      testWidgets('long tap test 1', (tester) async { /* test */ });
      testWidgets('Long tap test 2', (tester) async { /* test */ });
    });
  });
});
```

Imagine you had a failing test on *Tap test 2*. In this case, the `flutter test` command would report a failure at “*MyWidget tests Gesture tests Taps Tap test 2*”, which is the “path” to the failing test. If

you ran the test using Android Studio or Visual Studio Code for example, it would be even better because you would have the possibility to directly navigate to the failing test.

### Note

From the terminal, all Flutter tests have to always be executed with the `flutter test` command. If you use `dart test`, it won't work.

The reason is that `flutter test` is a wrapper of `dart test` which adds some special configurations to setup a sort of "fake" Flutter application to render widgets. This is something that the `dart test` command doesn't do.

The main goal of a widget test is to ensure that widgets are rendered as you would expect. The testing library gives you utilities to find widgets, check how many times they occur when rendered and much more. In other words, widget tests are handy to check the behavior of individual widgets.

#### 19.1.1 Using finders

The main goal of a widget test is to locate one or more widgets in the tree (and generally check how many times it occurs). So far, we have seen how to count the occurrences of a widget by looking at its type. For example:

```
testWidgets('Widget test', (tester) async {
    // The widget tree we want to test
    await tester.pumpWidget(
        MaterialApp(
            home: Center(
                child: Row(
                    children: [
                        Container(height: 40, color: Colors.red),
                        Container(width: 60, color: Colors.green),
                    ],
                ),
            ),
        ),
    );
}

// Using finders to search for widgets by type
expect(find.byType(MaterialApp), findsNWidgets(2));
expect(find.byType(Container), findsNWidgets(2));
});
```

In some cases, you might want to check for the presence of a specific widget. The following example has two `Containers`, but we would like to only check for the presence of the green one. To find a specific widget in the tree, give it a key and use the `byKey` finder:

```
testWidgets('Widget test', (tester) async {
  await tester.pumpWidget(
    MaterialApp(
      home: Center(
        child: Row(
          children: [
            Container(
              height: 40,
              color: Colors.red,
            ),
            Container(
              key: const Key('Green-Container-Key'),
              width: 60,
              color: Colors.green,
            ),
          ],
        ),
      ),
    ),
  );
}

expect(find.byKey(const Key('Green-Container-Key')));
});
```

When the tree has multiple widgets of the same type and you want to find a specific one, keys are very useful. Of course, you need to make sure that the tree has a unique key with the given id. Using keys to easily find widgets is a common (and recommended) testing technique.<sup>110</sup> Alternatively, you could also look for a widget by its instance:

```
final widgetToTest = Container(color: Colors.red);

await tester.pumpWidget(
  MaterialApp(
    home: widgetToTest,
  ),
);

expect(find.widgetWithText(widgetToTest), findsOneWidget);
```

---

<sup>110</sup> <https://docs.flutter.dev/cookbook/testing/widget/finders#2-find-a-widget-with-a-specific-key>

The `byWidget` finder finds widgets whose current widget is the given instance. As such, if you pass the same widget but with a different instance, the test will fail. In other words, the above test will fail in this case:

```
expect(find.byWidget(Container(color: Colors.red)), findsOneWidget);
```

The reference of the widget in the tree differs from the reference of the widget you've created, so the test fails. There also are other interesting finders:

- `find.text`: This finder matches all `Text` or `RichText` widgets whose text contains equals to the given string.
- `find.widgetWithText`: Looks for widgets that contain a `Text` descendant with the given string. For example, if you had a widget that looked like this...

```
ElevatedButton(  
    onPressed: () {},  
    child: const Text('Press me'),  
) ,
```

... you could use `find.widgetWithText(ElevatedButton, 'Update')` to look for a button that contained the *Press me* string.

- `find.byIcon`: This is a dedicated finder that looks for all of those `Icon` widgets having the given `IconData` value.
- `find.byPredicate`: This is used to create sort of “queries” to find widgets that match the given predicate. For example:

```
expect(  
    find.byWidgetPredicate((widget) {  
        return widget is Container && widget.color == Colors.red;  
    }),  
    findsOneWidget,  
) ;
```

This code looks for all widgets in the tree whose type is `Container` and the `color` property is `Colors.red`. You can see this as a “filtered” version of `find.byType` because you can look for a widget by type and add more constraints.

In general, `byType` or `byKey` are the most used finders (especially the keyed one) but there is not a rule of thumb to follow. Use the finders that are more appropriate for your use case.

### 19.1.2 Pumping methods and rebuilds

We have seen that you need an initial `tester.pumpWidget` call to render the widget tree in a test environment and find widgets. This method internally calls `runApp`, triggers a new frame and makes the first widget tree build. In various cases, you will need to rebuild a widget more than once. For example, imagine you had to test this:

```
class Counter extends StatefulWidget {
  const Counter({super.key});

  @override
  State<Counter> createState() => _CounterState();
}

class _CounterState extends State<Counter> {
  var count = 0;

  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: () => setState(() => ++count),
      child: const Text('Increase'),
    );
  }
}
```

The `setState` call schedules a rebuild for `Counter` in the next frame. The testing environment has to be driven by you, the developer, so you also need to manually “refresh” the tree after a `setState` call (or any other event that would cause a rebuild). This is where the `pump()` method comes to the rescue:

```
testWidgets('Widget test', (tester) async {
  await tester.pumpWidget(const MaterialApp(home: Counter()));
  expect(find.text('0'), findsOneWidget); // The counter is '0'

  await tester.tap(find.text('0'));
  await tester.pump(); // <- This is important

  expect(find.text('1'), findsOneWidget);
  expect(find.text('0'), findsNothing);
});
```

We first call `pumpWidget` to create the testing environment and to build the widget tree for the first time. After that, we test the `Counter` widget with a series of actions:

1. ensure that the counter is zero when the widget is created;
2. thanks to `tester.tap`, we can simulate the button tap action that triggers the `onPressed` callback;
3. in a test environment, Flutter does not automatically rebuild widgets. Since `onPressed` calls `setState`, we must manually use `tester.pump()` to rebuild the widget tree. After the `pump` call, the tree is rebuilt and the widget is updated with the most recent counter value.
4. We make sure that the counter is now set to `1` rather than `0`.

The `pump()` method is essential because it rebuilds the widget tree whenever you need it. If we had not called `pump()` after the button had tapped, the counter would have remained at `0` (because the widget tree doesn't automatically rebuild). For example, look at this (wrong) example:

```
await tester.tap(find.text('0'));
// await tester.pump(); <- remove this line to break the test

expect(find.text('1'), findsOneWidget); // Test fails
expect(find.text('0'), findsNothing); // Test fails
```

Now that we have removed the `pump()` method call, the button is tapped and `setState` is called, but the widget tree never gets rebuilt (and the counter never updates). When it comes to animations, you also need to manually rebuild the tree until the animation finishes. For example, consider this case where `controller` is an `AnimationController` object:

```
@override
Widget build(BuildContext context) {
  return RotationTransition(
    turns: controller,
    child: ElevatedButton(
      onPressed: () => controller.forward(),
      child: AnimatedBuilder(
        animation: controller,
        builder: (_, __) => Text('${controller.value}'),
      ),
    ),
  );
}
```

When the button is pressed with `pump()`, the `forward()` method is correctly called, but the widget never rotates. To correctly run the animation, we need to use the `pumpAndSettle()` method, which internally calls `pump()` until there are no more frames. In other words, it waits for all animations to complete:

```
testWidgets('Widget test', (tester) async {
  await tester.pumpWidget(const MaterialApp(home: Test()));

  // The animation hasn't started so the controller is at 0.
  expect(find.text('0.0'), findsOneWidget);

  await tester.tap(find.text('0.0'));
  await tester.pumpAndSettle(); // With animations, use this instead of 'pump'

  // The animation finished so the controller value is 1.
  expect(find.text('1.0'), findsOneWidget);
});
```

The `pumpAndSettle()` method calls `pump()` repeatedly to rebuild the widget until the animation finishes. The recommendation<sup>111</sup> is:

- Use `pump()` when you need to rebuild a widget once. This is useful after a `setState` call for example or when a listenable class notifies its listeners.
- Use `pumpAndSettle()` when you need to wait for one or more animations to complete.

Always remember that the test environment is very “naive” and so you need to always instruct him on what to do (like it happens with rebuilds).

## 19.2 Widget testing strategies

Writing widget tests does not only mean using finders to ensure that all UI elements are in place. You should also exercise the widget (if that’s the case) to guarantee that it works as expected. In general, we recommend following these steps for testing a widget:

1. **Smoke tests.** A smoke test aims to ensure that one or more fundamental widgets are in the correct place. Don’t underestimate the value of smoke tests: they are a “preliminary” check to ensure that a widget has solid foundations. For example:

---

<sup>111</sup> <https://docs.flutter.dev/cookbook/testing/widget/introduction#notes-about-the-pump-methods>

```

testWidgets('Smoke test', (tester) async {
  await tester.pumpWidget(const LoginForm());

  // Login button
  expect(find.byType(ElevatedButton), findsOneWidget);

  // Username and password fields
  expect(find.byType(TextFormField), findsNWidgets(2));
});

```

This is a *smoke test* of a login form. Since we expect to always have at least two input fields and a button, we “smoke test” the login form for those three components. This test covers the basic case where all core components (two text fields and a button) are there.

2. Exercise tests. These tests are required if your widget defines behaviors. For example, there might be a button with an `onPressed` callback or some events triggered by a controller that notifies its listeners.
3. Golden tests. They are “screenshots” of a widget that are used as reference to rule how a widget should look like. We will cover them in detail in *Section 2.3 – Golden tests*.

There are various techniques to smoke test a widget and exercise it. In the next sections, we are going to cover the most common testing strategies you should be aware of. We recommend to also check out the official documentation<sup>112</sup> to read about widget testing.

### 19.2.1 Accessing widgets and state instances

Finders are great for locating widgets in the tree, but they don’t give you a way to inspect the object they have found. For example, imagine you wanted to test this widget:

```

Container(
  padding: const EdgeInsets.all(10),
  alignment: Alignment.center,
  color: Colors.lime,
  child: const Center(
    child: Text('Hello'),
  ),
),

```

---

<sup>112</sup> <https://docs.flutter.dev/testing#widget-tests>

With finders, you can ensure that the `Container` and the `Text` are rendered, but you cannot also check the `padding` and `alignment` properties, for example. The `WidgetTester` object is helpful in these situations:

```
testWidgets('Test widget properties using WidgetTester', (tester) async {
  await tester.pumpWidget(
    MaterialApp(
      home: Container(
        padding: EdgeInsets.all(10),
        alignment: Alignment.center,
        color: Colors.lime,
        child: const Center(child: Text('Hello')),
      ),
    ),
  );
}

final containerFinder = find.byType(Container);

// 'tester.widget' returns a reference to the 'Container' object
final container = tester.widget<Container>(containerFinder);

expect(container.padding, equals(const EdgeInsets.all(10)));
expect(container.alignment, equals(Alignment.center));
});
```

The `tester.widget<T>` method returns the reference of the widget matched by the finder. In this way, we can inspect its internal properties. If the widget had had a list of containers rather than a single one, we could have matched all of them and iterated over the result:

```
final finder = find.byType(Container);

tester.widgetList<Container>(finder).forEach((container) {
  expect(container.padding, equals(const EdgeInsets.all(10)));
  expect(container.alignment, equals(Alignment.center));
});
```

The `widgetList<T>` method returns an `Iterable<T>` that contains all the widgets matched by the finder. You could also easily get a reference to all the widgets mounted in the tree with this simple getter:

```
final Iterable<Widget> widgets = tester.allWidgets;
```

In a similar way, you can also access the `State<T>` class of a stateful widget in your tests, but it has to be public. For example, if we created a counter widget, we would need to make its state class public and add the `@visibleForTesting` annotation (which is useful to the analyzer). For example:

```
// The annotation is useful for the IDE and the analyzer tool
@visibleForTesting
class CounterState extends State<Counter> {
    var count = 0;

    @override
    Widget build(BuildContext context) {
        return ElevatedButton(
            onPressed: () => setState(() => ++count),
            child: Text('$count'),
        );
    }
}
```

Since the state class is now visible from the outside (it's not library-private anymore), you can get a reference to `CounterState` from the test file. It's very similar to what we did when we looked for a specific widget instance:

```
testWidgets('Widget test', (tester) async {
    await tester.pumpWidget(const MaterialApp(home: Counter()));

    // Verify the counter using a finder
    expect(find.text('0'), findsOneWidget);

    // Increase the counter
    await tester.tap(find.byType(ElevatedButton));

    // Trigger a frame to make the setState call effective
    await tester.pump();

    // Verify the new counter value directly in the state class
    final state = tester.state<CounterState>(find.byType(Counter()));
    expect(state.count, equals(1));
});
```

Thanks to `tester.state<T>` we can get a reference to the state object of the `Counter` widget and check its properties. We don't recommend this approach because the state class is designed to be private and should stay so. There almost always is a way to test properties from "the outside" using finders so try to keep state classes private.

## 19.2.2 Testing forms and gestures

The `WidgetTester` class has a rich set of methods that make forms and gestures testing very easy. For example, imagine you had to test a login form with two `TextField` widgets and a button

at the bottom. In this case, you could enter some text in the `TextField` widgets and press the login button to see what happens. For example:

```
testWidgets('Widget test', (tester) async {
  await tester.pumpWidget(const MaterialApp(home: Counter()));

  // Finding all text fields and entering values
  final textFields = find.byType(TextFormField);
  await tester.enterText(textFields.at(0), 'my@email.com');
  await tester.enterText(textFields.at(1), 'password123');

  // Login!
  await tester.tap(find.byType(ElevatedButton));
  await tester.pump();

  // And then verify that something happened...
});
```

We first use a finder to get a reference to all text fields in the `Counter` widget. Since there are two `TextFields` and we know the order in which they appear, we can access them by index. Then, we use `enterText` to focus the given text field and enter the text we want. This method specifically expects the finder to be either a `TextField` or a `TextFormField`.

### Note

In this example, we knew the order of the text fields so we could easily access them by index... but this is not always the case! When you cannot assume the exact position of one or more widgets, use the `find.byKey` finder because:

1. it does not force you to maintain the tests when the widgets' positions change;
2. it is an easy and direct way to access a widget without losing the benefit of being `const` (if it's the case) because since `Key` has a constant constructor.

The `tester` reference is also handy for scrolling use cases. For example, imagine you had to test a scrollable page and you wanted to tap the *Register* button at the bottom of a `ListView`. To ensure the button is visible, you can scroll the list before tapping it. For example:

```
final finder = find.text('Register');

await tester.scrollUntilVisible(finder, 100);
await tester.tap(finder);
```

If the button is not visible, the `tester.tap` call will fail. As such, the `scrollUntilVisible` is used to automatically scroll down until the target is visible. This method repeatedly scrolls a scrollable widget by the given amount (in our case, `100`) until the widget we need is visible. The number `100` is also called `delta`. Note that a positive delta (`100`) scrolls the list down while a negative delta value (`-100`) scrolls the list up.

### Note

If you have more than a single scrollable widget in your test, make sure to pass the right one in the `scrollable` property:

```
await tester.scrollUntilVisible(  
    find.byType(SomeWidget),  
    50,  
    scrollable: find.byKey(const Key('ListView-Scroll-Test')),  
,
```

In this way, the test environment scrolls a specific `Scrollable` widget.

Note that `scrollUntilVisible` internally calls `pump()` already, so you don't need to `pump()` again. Other than scrolling, you can also emulate many more gestures using the `tester` object. The most common methods are:

- `tester.press`: This is used to dispatch a “pointer down” event at the center of the given widget. Imagine you had to test a button. The `press` method presses the button without triggering its callback. The `tap` method presses and releases the button (which triggers the callback).
- `tester.drag`: This method attempts to drag the widget by a given offset. For example, it is useful when you need to scroll a `ListView` (or any other scrollable widget) by a certain amount:

```
// Horizontally scrolls a list (Offset has the Y axis set)  
await tester.drag(find.byType(ListView), const Offset(0, 100));  
  
// Vertically scrolls a list (Offset has the X axis set)  
await tester.drag(find.byType(ListView), const Offset(100, 0));
```

`Offset` is used to determine the direction in which we want to scroll. For example, in the first case only the Y value is set and so the tester scrolls along the horizontal axis. If the value was negative (`-100`), the tester would have scrolled in the opposite direction.

- `tester.restartAndRestore`. This method simulates the state restoration of a widget tree after the application is restarted. This is used to test your state restoration implementation, if any. For example:

```
testWidgets('State restoration test', (tester) async {
  await tester.pumpWidget(const RestorableWidget());

  // 1. Modify the state here...

  // 2. Simulate state restoration
  await tester.restartAndRestore();

  // 3. Check if the state was correctly restored...
});
```

Thanks to `tester.getRestorationData()` you can even get the current restoration data, wrapped by a `TestRestorationData` object.

In *chapter 18 – Section 3.1 “Drag and drop”* we have seen how to implement drag and drop using the `Draggable` and `DragTarget` widgets. This is an example of how you can widget test the drag and drop behavior:

```
// 1. Finders for 'Draggable<T>' and 'DragTarget<T>'.
final draggable = find.byType(Draggable<String>);
final target = find.byType(DragTarget<String>);

// 2. Create a 'gesture' object to drive the drag-and-drop gesture.
final gesture = await tester.createGesture(kind: PointerDeviceKind.touch);

// 3. Press on the widget.
await gesture.down(tester.getCenter(draggable));

// 4. While pressing, drag above the 'DragTarget' widget.
await gesture.moveTo(tester.getCenter(target));

// 5. Release the widget in the target.
await gesture.up();

// 6. Refresh the UI to complete the 'drag and drop' gesture.
await tester.pump();
```

The `createGesture` method returns an object that lets us create and drive any kind of gesture. In this case, we're using it to test our widget's “drag-and-drop” behavior. In particular:

1. `gesture.down`: simulates the finger tap on a widget;
2. `gesture.moveTo`: simulates the finger moving from one point to another and preserves any other active gesture. In our example, it simulates the finger that moves on the screen while staying pressed.
3. `gesture.up`: simulates the finger not touching the screen anymore. In our example, this is used to “drop” the `Draggable` widget on its target.

The `getCenter` calls are used to get an `Offset` object that represents the center of the widget. The coordinates of the center make sure that the tester is able to correctly locate the start and the finish points for the `down` and `moveTo` methods.

### Note

The `tester` object has many other useful methods to get the coordinates of a widget in the tree. They all return an `Offset` object. For example:

- `getCenter`
- `getTopLeft`
- `getTopRight`
- `getBottomLeft`
- `getBottomRight`

In our example, the pointer is represented by the user’s finger but we could have also changed it to be a trackpad or a mouse cursor. For example:

```
final gesture = await tester.createGesture(  
    kind: PointerDeviceKind.mouse, // mouse  
>);  
  
await gesture.down(tester.getCenter(draggable));  
await gesture.moveTo(tester.getCenter(target));  
await gesture.up();  
  
expect( /* something here */ , /* finder here */ );  
await gesture.removePointer(); // this is VERY important
```

If you use the mouse, always remember to call the `removePointer` method at the end to correctly dispose the mouse tracker instance from the testing suite. If you forget to remove it, the test will fail with an error message (which will remind you to call `removePointer`).

### 19.2.3 Golden tests

A golden test is a sort of “screenshot” of your widgets, stored in a PNG file, that is used to detect rendering changes. It creates a master image of what a widget has to look like, and subsequent test runs will try to match it. For example, imagine you had a custom painter like this:

```
class MyPainter extends CustomPainter {
  const MyPainter();

  @override
  void paint(Canvas canvas, Size size) {
    final linePaint = Paint()..color = Colors.indigo;
    final rectPaint = Paint()..color = Colors.grey;

    canvas
      ..drawLine(Offset.zero, const Offset(100, 100), linePaint)
      ..drawRect(const Rect.fromLTWH(20, 20, 45, 45), rectPaint);
  }

  @override
  bool shouldRepaint(covariant MyPainter oldDelegate) => false;
}
```

It would be very hard to write a widget test to ensure that a line and a rectangle are drawn with the correct colors and positions. The best we can do is look for the presence of the painter and check some of its properties:

```
await tester.pumpWidget(
  const MaterialApp(
    home: CustomPaint(
      key: Key('My-Painter'),
      painter: MyPainter(),
    ),
  ),
);

final customPaint = tester.widget<CustomPaint>(
  find.byKey(const Key('My-Painter')),
);
expect(customPaint.painter, isA<MyPainter>());
expect(customPaint.painter!.shouldRepaint(const MyPainter()), isFalse);
```

This is more of a smoke test because we check various properties without looking at the painter's internals. In this case, we could make a "widget screenshot" to create a master image and use it as a reference for how the painter should look. For example:

```
testWidgets('Golden test', (tester) async {
  await tester.pumpWidget(
    const MaterialApp(
      home: CustomPaint(
        key: Key('My-Painter'),
        painter: MyPainter(),
      ),
    ),
  );
}

final customPaint = tester.widget<CustomPaint>(
  find.byKey(const Key('My-Painter')),
);

expect(customPaint.painter, isA<MyPainter>());
expect(customPaint.painter!.shouldRepaint(const MyPainter()), isFalse);

await expectLater(
  find.byKey(const Key('My-Painter')),
  matchesGoldenFile('my_painter.png'), // this is the important part
);
});
```

If we run the `flutter test --update-goldens` command, when this test is evaluated, an image called `my_painter.png` is created in the same path as the test file. This PNG file is also called **golden** image and it contains a render of the widget (in our case, the painter):

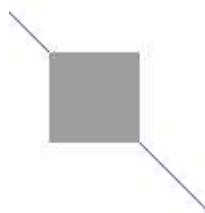


Figure 18.1: The golden image generated by a golden test.

This image is a "preview" of how the `CustomPaint` widget will look like once rendered. Other than being useful for the developer to quickly see how the painter paints the canvas, it's also helpful to spot any changes. Now that we have generated this golden image, it will be used as a reference when running tests. For example, consider this workflow:

1. Generate the golden image with `flutter test --update-goldens`.
2. Go back to the painter and change the color from `Colors.grey` to `Colors.orange`.
3. Run `flutter test` again (without the `--update-goldens` flag) and notice the failure.

The test fails because Flutter notices that the golden image (`my_painter.png`) was created with a grey rectangle but the widget tree in the test environment has an orange rectangle. The solution is to either revert the color back to `Colors.grey` or run tests again with the `--update-goldens` flag to recreate the golden image.

## Note

In general, this is how you should work with golden images:

1. A golden test is a test that creates a golden image. To create a golden test, add this method call after you've pumped the widget:

```
await expectLater(  
    /* any useful finder here */,  
    matchesGoldenFile('goldens/my_golden_image.png'),  
);
```

It is a good practice to group together all golden images in the `goldens/` folder. As such, we prefix the golden name with the relative `goldens/` path.

2. Run the `flutter test --update-goldens` command to create or regenerate all golden images.
3. Now that you have generated golden images, run tests using the `flutter test` command (without `--update-goldens`). In this way, Flutter will compare your widget tree with the golden image to spot any change.

The `--update-goldens` command should only be used to create new goldens or update old ones.

Golden images are used by the test framework as a “master image” to indicate how a widget must look when rendered. They are extremely useful for various reasons:

- they give the developer a preview of how a widget will be rendered;
- they easily test intrinsic widget properties all together (such as spacings, sizes, and colors);

- they can detect the most subtle changes when the rendered widget doesn't perfectly match the given golden image.

It's worth to point out that goldens generated on a Windows machine might fail on macOS or Linux for example. The reason is that each operating system has slight differences in how PNG files are generated. If you pixel-by-pixel compare the same golden image generated by different operating systems, you may see some tiny differences. Failures may not always happen, but you should be aware of the fact that some golden tests might not pass if run them on different operating systems.

### Note

If you used two different versions of the same operating systems, golden tests might still fail because of the aforementioned reasons.

Run tests in the same environment to avoid rendering issues when you generate golden images.

#### 19.2.3.1 Fonts in golden tests

Golden testing is not limited to custom painters only. You can create golden images for any widget. For example, imagine you wanted to create the golden image of this login form:

```
// This is inside the 'LoginForm' widget
Column(
  children: [
    Flexible(
      child: TextFormField(
        controller: usernameController,
      ),
    ),
    Flexible(
      child: TextFormField(
        controller: passwordController,
      ),
    ),
    ElevatedButton(
      onPressed: () {},
      child: const Text('Login'),
    ),
  ],
),
```

We want to generate a golden image with some text inside the text fields. To do so, we fill the fields before calling `matchesGoldenFile`. The `pumpAndSettle` call is essential because the focus change of a text field is animated (and so we have to wait for all scheduled frames to be processed):

```
testWidgets('Golden test', (tester) async {
  await tester.pumpWidget(
    const MaterialApp(
      home: Scaffold(
        body: LoginForm(),
      ),
    ),
  );
}

// Enter text before generating the golden
final finder = find.byType(TextFormField);

await tester.enterText(finder.first, 'my@email.com');
await tester.enterText(finder.last, 'password');
await tester.pumpAndSettle();

await expectLater(
  find.byType(LoginForm),
  matchesGoldenFile('goldens/login_form.png'),
);
});
```

The generated golden file looks a bit weird because there are black and white squares instead of actual text, but this behavior is expected. By default, the Flutter test framework uses a particular font called `FlutterTest` which replaces all characters with squares:



Figure 18.2: A golden image with the default text font “FlutterTest”.

The reason behind this decision is to make golden file generation more consistent across multiple operating systems. The `FlutterFont` font has particular line heights and font sizes that should often be rendered in the same way across different operating systems. This font is used to make golden

file generation as much platform-independent as possible. However, the squares make the golden image less valuable. If you want to override the default font and replace it with a readable one, you must manually load a local font file. For example, we could download for free the Roboto font from [fonts.google.com](https://fonts.google.com) and add it to the `pubspec.yaml` file. In this way, the font file can be loaded from the test environment:

```
testWidgets('Golden test', (tester) async {
    // Retrieve the font file and call 'Load' to override Ahem
    final fontLoader = FontLoader('Roboto')
        ..addFont(rootBundle.load('font/Roboto-Regular.ttf'));
    await fontLoader.load();

    await tester.pumpWidget(
        const MaterialApp(
            home: Scaffold(
                body: LoginForm(),
            ),
        ),
    );
};

final finder = find.byType(TextFormField);
await tester.enterText(finder.first, 'my@email.com');
await tester.enterText(finder.last, 'password');
await tester.pumpAndSettle();

await expectLater(
    find.byType(LoginForm),
    matchesGoldenFile('goldens/login_form.png'),
);
});
```

Before calling `pumpWidget`, we use the `FontLoader` class to override `FlutterTest` and use `Roboto` instead. Of course, you could have used any other font file. This is the result we get with `Roboto`, which is more valuable than before:

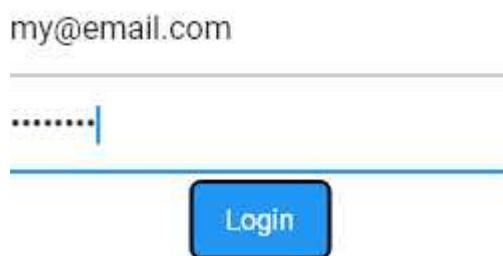


Figure 18.3: The golden image rendered with Roboto instead of FlutterTest.

The problem with this approach is that we always need to manually call the font loading logic at the beginning of each golden test. This is tedious and also implies copy-pasting code. The solution is to move the loading logic in the global testing file.

## Note

Before a test file is run, the framework looks for the `flutter_test_config.dart` file. If it's found, the file is assumed to contain a method with the following signature:

```
Future<void> testExecutable(FutureOr<void> Function() testMain)
```

Flutter executes this method and passes in the `main` method of the test. In other words, if this file exists, it is always executed before evaluating a test file. This is very helpful for cases where you need to run some custom code before performing any test.

In our case, we will always need to load a custom font file to make the golden images more valuable and avoid code repetition. As such, we can create the `flutter_test_config.dart` file and use the `FontLoader` there. For example:

```
Future<void> testExecutable(FutureOr<void> Function() testMain) async {
  setUpAll(() async {
    final fontLoader = FontLoader('SomeFont')
      ..addFont(rootBundle.load('font/Roboto-Regular.ttf'));
    await fontLoader.load();
  });

  await testMain();
}
```

The `testMain()` callback represents the `main()` function of a test file. Since the font file must be loaded before generating the golden, we place the `load()` call before `testMain()`. You can only load a single font file per `FontLoader` instance so your golden images cannot be rendered with multiple fonts.

### 19.2.4 Good practices

The Flutter team does not give advices on how to structure tests. The following recommendations come from our experience on large projects where the testing process is crucial. When writing a widget test, we suggest doing the following:

1. Create the test file and start with a smoke test. It ensures that all fundamental pieces and configurations are correct.
2. Then proceed with testing the widget as much as possible. If the widget has behaviors (such as callbacks or gestures) make sure to exercise them all. We generally refer to this phase as “behavior testing”.
3. Finally, create at least one golden image. If your widget has statuses (such as “pressed” or “hover”) or has color variants, create multiple golden images to cover as many combinations as possible.

You could also generate a code coverage report by running `flutter test --coverage`, which creates a LCOV file in the `coverage/` directory. Coverage reports are discussed in *chapter 9 – Section 2.4 “Code coverage”* while *chapter 21 – Section 2 “Maintaining a Flutter project”* covers how to work with tests and coverage reports in a Flutter project.

## 19.3 Integration tests

Integration tests are used to test how a group of widgets work together. They also capture the performance of an application that runs on a real device. In other words, integration tests allow you to automate some processes that would otherwise need the manual work of an engineer. To run integration tests in a Flutter project, use the `integration_test` package:

```
dev_dependencies:
  integration_test:
    sdk: flutter
  flutter_test:
    sdk: flutter
```

It is already included in the SDK so you need to import it with the  `sdk` property in the `pubspec.yaml` file. This section shows how to write an integration test for a “counter app” (a single route with a button that increases a counter). To get started, we create a new Flutter project and add a new top-level folder called `integration_test` with a `counter_app_test.dart` file inside. The project structure looks like this:

```
counter_app/
  integration_test/
    counter_app_test.dart
  lib/
    main.dart
```

The `main.dart` file contains the simple counter application we want to test:

```

void main() {
  runApp(const MaterialApp(
    home: Scaffold(
      body: CounterApp(),
    ),
  )));
}

class CounterApp extends StatefulWidget {
  const CounterApp({super.key});

  @override
  State<CounterApp> createState() => _CounterAppState();
}

class _CounterAppState extends State<CounterApp> {
  var counter = 0;

  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: () => setState(() => ++counter),
      child: Text('$counter'),
    );
  }
}

```

Integration tests are located in the `integration_test/` directory. In practice, an integration test in Flutter is a “special variant” of a widget test that invokes the application’s `main()` entry point. There is nothing new to learn here because this basically is a widget test that runs on a real platform (desktop, mobile, or web) rather than on a test environment:

```

import 'package:counter_app/main.dart' as counter_app;

void main() {
  IntegrationTestWidgetsFlutterBinding.ensureInitialized(); // required

  testWidgets('The counter can be increased', (tester) async {
    counter_app.main(); // The application's main entry
    await tester.pumpAndSettle();
    expect(find.text('0'), findsOneWidget);
    await tester.tap(find.byType(ElevatedButton));
    await tester.pumpAndSettle();
    expect(find.text('1'), findsOneWidget);
  });
}

```

The `IntegrationTestWidgetsFlutterBinding` is the “heart” of a Flutter integration test because it manages a service that executes tests on a simulator or (even better) a physical device. Note that integration tests can be run on any platform (mobile, desktop and web).

### Note

While desktop platforms require no setup, for mobile devices you need an emulator or a real device. For the web, you’ll need to use the “`ChromeDriver`”<sup>113</sup> tool.

Use the `flutter test integration_test` command to run the tests. Once the application is built and run, the integration test framework will automatically press the button to increase the counter. You will see a “robot” that interacts in real-time with the running application. Let’s highlight some differences between a widget test and an integration test:

Widget test	Integration test
Used to test individual widgets in a special test environment.	Used to test widgets many all together in a real environment.
Widget dependencies are generally mocked. Native method channels calls, for example, are not handled by the test environment so we need to replace them with “ <i>fake</i> ” (or “ <i>mocked</i> ”) calls.	Widget dependencies are not mocked. Native method channel calls, for example, are executed without mocks to test their “health” on a real platform.
Widget test execute quickly because the test environment is very light.	Integration tests are slower because they build the application and run it on a target platform.

The biggest difference is that integration tests run on real devices. As such, you shouldn’t mock HTTP calls or native integrations for example. Integration tests have the highest confidence level because they test the code on a real device. Widgets tests instead test individual widgets on a “fake” Flutter environment and HTTP calls or native integrations (for example) should be mocked.

---

<sup>113</sup> <https://docs.flutter.dev/cookbook/testing/integration/introduction#5b-web>

## Deep dive: Performance profiling with integration tests

Integration tests are also great for measuring and reporting multiple task performance timelines. For example, imagine your application had a `FutureBuilder<T>` to fetch data over HTTP and a long lazily-loaded list view. To constantly measure performance and obtain a detailed report, you could do the following:

1. write an integration test that triggers the HTTP call and scrolls the list;
2. run the test in one (or more) platforms and track the runtime performance;
3. save the results in a human-readable form.

This is extremely useful on larger applications where manual testing is time-consuming. You can write the integration test once, run it on various platforms, and automatically collect the reports. For simplicity, let's see how we can get a performance report of a classic counter application:

```
import 'package:counter_app/main.dart' as counter_app;

void main() {
  final binding = IntegrationTestWidgetsFlutterBinding.ensureInitialized();

  testWidgets('The counter can be increased', (tester) async {
    counter_app.main();

    await tester.pumpAndSettle();
    expect(find.text('0'), findsOneWidget);

    // This is used to record the runtime performance of a series of actions
    await binding.traceAction(
      () async {
        await tester.tap(find.byType(ElevatedButton));
        await tester.pumpAndSettle();
      },
      reportKey: 'press_widget',
    );
    expect(find.text('1'), findsOneWidget);
  });
}
```

The only special addition we made is the `traceAction` method call, provided by the test bindings instance. Thanks to `traceAction`, we can generate a detailed performance report of the actions within its callback. Specifying a `reportKey` is essential when calling `traceAction` more than once in your integration tests because it allows splitting reports into multiple categories (and thus files).

We also need to create a new top-level directory called `test_driver/` and create a new file called `integration_test.dart` with the following contents:

```
// Contents of '/test_driver/integration_test.dart'
import 'package:flutter_driver/flutter_driver.dart' as driver;
import 'package:integration_test/integration_test_driver.dart';

Future<void> main() {
  return integrationDriver(
    responseDataCallback: (data) async {
      if (data != null) {
        final timeline = driver.Timeline.fromJson(data['press_widget']);
        final summary = driver.TimelineSummary.summarize(timeline);

        await summary.writeTimelineToFile(
          'press_widget',
          pretty: true, // optional
          includeSummary: true, // optional
          destinationDirectory: 'my/custom/path/', // optional
        );
      }
    },
  );
}
```

Once the test passes, we must save the performance report on the disk. The `integrationDriver` function is used to “capture” the performance report and convert it into a human-readable form. In particular:

- The output of the `traceAction` method can be “captured” by the `integrationDriver` callback and converted into a human-readable JSON file using a `Timeline` object.
- The output of a `Timeline` object is extremely hard to read because it contains a lot of event information and VM-specific data. A `TimelineSummary` object generates another JSON file with an easy-to-read performance report.
- Here you can see why we set a `reportKey` in the `traceAction` method. The `data` parameter of `responseDataCallback` contains the data of all reports that were created while running the integration tests. Thanks to that key, we can only extract the information we want.

If you have multiple `traceAction` calls, you might want to collect all performance reports. In this case, modify the callback a bit to iterate over all reports and save them to the disk:

```

responseDataCallback: (data) async {
  if (data != null) {
    for (var entry in data.entries) {
      final timeline = driver.Timeline.fromJson(data[entry.key]);
      final summary = driver.TimelineSummary.summarize(timeline);

      await summary.writeTimelineToFile(
        entry.key,
        pretty: true,
        includeSummary: true,
      );
    }
  }
},

```

Instead of referencing reports with a string, you can iterate over all data and use `entry.key`, which holds the `reportKey` key of each report. Regardless of the strategy you want to use, make sure to run integration tests with this command:

- If the integration test runs on a real device, add the `--profile` option to get a benchmark whose performance is closest to a release build:

```
flutter drive --driver=test_driver/integration_test.dart --no-dds
--target=integration_test/counter_app_test.dart --profile
```

- If the integration test runs on an emulator, don't add the `--profile`. Because of this, the test will run in debug mode (and you will inevitably see lower performance):

```
flutter drive --driver=test_driver/integration_test.dart --no-dds
--target=integration_test/counter_app_test.dart
```

You cannot run tests on emulators in profile mode. Measuring performance of integration tests that run in debug mode produces misleading results (because a debug build is slower than a profile or release build).

The `--no-dds` flag is only required when running on mobile devices or emulators. After the test is finished, you will find the performance reports inside the `build/` folder as JSON files. In our case, we will find all the details inside the `press_widget.timeline_summary.json` file. To summarize, here's what is needed to track your app's performance in an integration test:

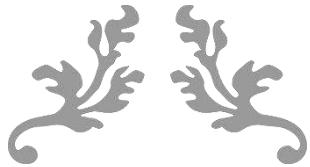
1. Write the integration test and wrap inside `binding.traceAction` those calls for which you want to generate a performance report.

2. Define the `integrationDriver` function and use its callback to save the report as JSON file on your computer.
3. Run the integration test(s), preferably on a real device and in profile mode. For desktop and web platform, the “device” is the host platform itself.
4. Read the report (which is generated in the `build/` folder).

Remember that integration tests, without performance reporting, are run with this command:

```
flutter test integration_test
```

To generate performance reports, you must use the `flutter drive` command.



## Part 3: Dart and Flutter ecosystems

---

20 – Creating and maintaining a package

21 – Creating and maintaining a Flutter app

22 – HTTP servers and low-level HTML

23 – Platform interactions

Appendix – Performance and profiling



# 20 – Creating and maintaining a package

---

## 20.1 Package creation



<https://github.com/albertodev01/equations>

In the Dart world, packages are used to share libraries and tools to other developers, mainly via the pub repository<sup>114</sup>. Regardless, it is also possible to use a package that is located in the filesystem or in a GitHub repository. Dart and Flutter use a package manager system (called pub) that handles dependencies, versioning, updates and much more for you.

### Note

A Flutter package is a regular Dart package with some more linter rules and the `flutter` dependency in the `pubspec.yaml` file. Even if this chapter focuses on Dart packages, all the considerations and best practices also apply to Flutter packages.

To create a Dart or a Flutter package that follows the recommended structure, you should use the command line tool. It will create a minimal project from which you can start working right away:

- `dart create -t package pkg_name`: This command creates the skeleton of a Dart package that follows the recommended guidelines.
- `flutter create -t package pkg_name`: This command creates the skeleton of a Flutter package that follows the recommended guidelines.

In this chapter we're taking the `equations` package as example, which is used to solve equations. The `dart create -t package equations` command generates a series of folders and Dart files for us. The most relevant parts, which also identify what's needed for a minimal package, are:

1. The `lib/` folder, which contains the Dart source code of the package. You can structure the project as you wish but we strongly recommend to follow the official Dart guidelines<sup>115</sup> on how to organize a package. The contents of this folder are public to other packages.

---

<sup>114</sup> <https://pub.dev>

<sup>115</sup> <https://dart.dev/guides/libraries/create-library-packages>

2. The `pubspec.yaml` file, which is required by any Dart (and Flutter) project. It defines project metadata and lists all dependencies (if any).

Of course, there are more folders and files other than these two but we will see them throughout the chapter. Let's get started and see how the Dart team recommends to structure your package code within the `lib/` folder.

### 20.1.1 Package organization

The package organization <sup>116</sup> is not mandatory but the Dart team strongly recommends it. It aims to minimize maintenance, extension, and testing efforts. The `lib` folder of the `equations` package rigorously follows these rules for example:

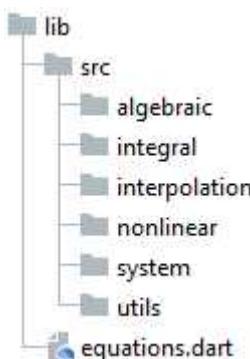


Figure 20.1: The `lib/` folder contents of the `equations` package.

Inside `lib` there is a single Dart file having, by convention, the same name as the package. This is used to export the public APIs and allows users to use your package with a single `import` statement. All the Dart code goes in the `src` folder. Here is the `equations.dart` file content:

```

/// This package is used to solve numerical analysis problems.
library equations;

export 'package:fraction/fraction.dart';

export 'src/algebraic/algebraic.dart';
export 'src/algebraic/types/constant.dart';
export 'src/algebraic/types/cubic.dart';
export 'src/algebraic/types/durand_kerner.dart';
// and many more 'export' statements...
  
```

<sup>116</sup> <https://dart.dev/guides/libraries/create-library-packages#organizing-a-library-package>

All the Dart code is inside `lib/src` and the public API only is exported by the `equations.dart` file. This allows you to not expose internal utility classes to the outside and keep them only visible within the package itself.

### Note

When the `library` directive is not defined, a unique tag is automatically generated by the compiler. In *Section 1.2 “Documentation”* we will see a case where `library` may be helpful for a package.

Inside `lib/src`, you can organize the code as you wish. If you support multiple platforms, in some cases, you might need to import or export files conditionally. For example, some core libraries (such as `dart:io`) are not available when running Dart on the web. In this case, you should create three different implementations. For example:

```
export 'src/io_none.dart'
if (dart.library.io) 'src/hw_io.dart'
if (dart.library.html) 'src/web_io.dart';
```

This code (which uses a “conditional export”) does the following:

1. If your application can use `dart:io` (desktop, mobile and embedded platforms), then the `hw_io.dart` file is exported.
2. If your application can use `dart:html` (on the web) then the `web_io.dart` file is exported.
3. If both `dart:io` and `dart:html` aren’t available, return the default `io_none.dart`.

The same example would also work with the `import` statement. This conditional statement checks whether the library is available but does not guarantee platform compatibility. All conditionally exported libraries must have the same API. For example, with the previous conditional `export` statement, we would need to create a few files. First of all:

```
// io_none.dart
class CustomIO {
  const CustomIO();

  void write(String data) => throw UnsupportedError('CustomIO write');
  void read() => throw UnsupportedError('CustomIO read');
}
```

This is the default implementation, which stubs method calls by throwing `UnsupportedError`. This file is used as “fallback” in those cases where both `dart:io` and `dart:html` are not available. It’s a good practice to throw `UnsupportedError` rather than another error.

Now that the fallback implementation is ready, let’s create the actual ones. Since there are two conditional statements, one for `dart:io` and one for `dart:html`, we need two files. Each of them must use the same type names and have the same API:

```
// 'hw_io.dart' file
import 'dart:io';

class CustomIO {
  const CustomIO();
  void write(String data) { /* write data using 'dart:io' */ }
  void read() { /* read data using 'dart:io' */ }
}

// 'web_io.dart' file
import 'dart:html';

class CustomIO {
  const CustomIO();
  void write(String data) { /* write data using 'dart:html' */ }
  void read() { /* read data using 'dart:html' */ }
}
```

You will only need a single `import` statement when using this library because Dart automatically picks the correct implementation based on your platform. For example:

```
import 'package:my_package/my_package.dart';

void main() {
  const customIO = CustomIO();

  // Dart automatically uses the correct version:
  // - 'web_io.dart' on web
  // - 'hw_io.dart' on mobile, desktop, or embedded
  customIO.read();
}
```

If you were running this in a Dart CLI application for example, you’d be using the `CustomIO` class from `hw_io.dart`. On a web browser instead, Dart would automatically use `web_io.dart`. This is extremely helpful when creating platform-specific packages.

## 20.1.2 Documentation

Inline documentation strings are extremely useful for giving information about public members. All public members should always be documented with at least one “docstring comment”. Types and members can be documented using triple slashes. For example, this is how we have documented the `Complex` class in the `equations` package:

```
/// A Dart representation of a complex number in the form `a + bi` where `a`  
/// is the real part and `bi` is the imaginary (or complex) part.  
///  
/// ... and much more info ...  
class Complex implements Comparable<Complex> {
```

The analyzer ignores all text unless it's enclosed in brackets. Names in brackets are resolved in the lexical scope of the document and allow you to reference classes, methods, fields, functions, top-level variables or parameters.

### Note

Dart's documentation strings use markdown to decorate text in the generated HTML website. As such, double asterisks make the text bold for example.

Once you've documented the code, you can run the `dart doc` command from the package root to automatically generate an HTML reference documentation. For example, the documentation of the `equations` package for the `Complex` type generates this HTML content (there actually is much more content, but it's not shown in the image below):

## Complex class Null safety

A Dart representation of a complex number in the form `a + bi`  
where `a` is the real part and `bi` is the imaginary (or complex) part.

New instances of `Complex` can be created either by:

- using one of the constructors
- the extension method on `num`

A `Complex` object is **immutable**.

Figure 20.2: The generated HTML page for the `Complex` type.

The `dart doc` tool converts your docstring comments into an HTML website (which is generated in the `doc/api` directory). With the `--output` flag you can specify a different path and generate files where you want. Here are a few suggestions<sup>117</sup> for writing great docstring comments:

- Document all of your public API with triple slashes. For legacy reasons, Dart also supports the “JavaDoc” documentation style (which uses `/** ... */` instead of `///`), but it’s not recommended:

```
/// This is a docstring comment.  
/// This is the recommended way.  
void myMethod() {}  
  
/**  
 * This is a docstring comment.  
 * This is NOT recommended.  
 */  
void myMethod() {}
```

There is a dedicated linter rule<sup>118</sup> that emits a warning when you don’t use triple slashes for docstrings.

- The `library` keyword is used when you want to write library-level docstring comments. For example, it might contain a summary of what the library is for, explanations of terminology and some small code snippets. This comes from the `equations` package:

```
/// An equation solving library you can use to:  
///  
/// - solve polynomial equations;  
/// - solve nonlinear equations;  
/// - solve linear systems of equations;  
/// - evaluate integrals;  
/// - interpolate data points.  
///  
/// This library also has utility classes to work with complex  
/// numbers, fractions and real/complex matrices.  
library equations;
```

---

<sup>117</sup> <https://dart.dev/guides/language/effective-dart/documentation>

<sup>118</sup> [https://dart-lang.github.io/linter/lints/slash\\_for\\_doc\\_comments.html](https://dart-lang.github.io/linter/lints/slash_for_doc_comments.html)

We have briefly introduced the package and listed its capabilities. You don't need a massive introduction with many details but it's a good practice to always leave (at least) a description of what a package does.

- Use square brackets to refer to in-scope identifiers. This also allows hyperlink generation to other types when using the `dart doc` tool. For example:

```
/// Concrete implementation of [Algebraic] that represents a second
/// degree polynomial equation in the form _ax^2 + bx + c = 0_.
///
/// This equation has exactly 2 roots, both real or both complex,
/// depending on the value of the discriminant.
class Quadratic extends Algebraic {
```

This docstring describes very well what the `Quadratic` class does. Since we have put the `Algebraic` type between square brackets, the `dart doc` tool will produce a hyperlink that redirects to the `Algebraic` page:

## Quadratic class

Null safety

Concrete implementation of [Algebraic](#) that represents a second  
degree polynomial equation in the form  $ax^2 + bx + c = 0$ .

This equation has exactly 2 roots, both real or both complex,  
depending on the value of the discriminant.

Figure 20.3: A hyperlink generated by the `dart doc` tool.

This is very useful when you're reading the documentation. Other than classes, you can also create links to functions, getters or named constructors<sup>119</sup>. In case of methods, just use `[Class.methodName]`, without specifying the function signature. To reference the default constructor, you need to use `[Class.new]`.

---

<sup>119</sup> <https://dart.dev/guides/language/effective-dart/documentation#do-use-square-brackets-in-doc-comments-to-refer-to-in-scope-identifiers>

- If you use three backticks, you can highlight a code snippet in docstring comments. This is the recommended way of including code examples in the documentation. For example:

```
/// This method is used to sum two numbers. For example:
///
/// ``dart
/// final result = sum(18.45, -3.1);
/// ``
double sum(double a, double b) => a + b;
```

The triple backticks at the top also need the language name right after their definition, in this case `dart`. Code blocks are highlighted and formatted in the generated HTML page.

- In some cases, it might be useful to reuse some sentences in more than a single place. In such cases, prefer using templates rather than copy-pasting content. For example, this is what we've done in the `Complex` class of the `equations` package:

```
/// {@template natural_ordering_complex_numbers}
/// There is no natural linear ordering for complex numbers. In fact,
/// any square in an ordered field is  $\geq 0$  but in the complex field we
/// have that  $i^2 = -1$ .
///
/// A possible comparison strategy involves comparing the
/// modulus/magnitude [abs] of the two complex number.
///
/// In this implementation, we compare two [Complex] instances by
/// looking at their modulus/magnitude.
/// {@endtemplate}
bool operator >(Complex other) => abs() > other.abs();

/// {@macro natural_ordering_complex_numbers}
bool operator >=(Complex other) => abs() >= other.abs();

/// {@macro natural_ordering_complex_numbers}
bool operator <(Complex other) => abs() < other.abs();

/// {@macro natural_ordering_complex_numbers}
bool operator <=(Complex other) => abs() <= other.abs();
```

We want to use the same text for all four operator overloads. Thanks to templates, we can assign a unique id and then use macros to “copy” the text in other places automatically. In particular:

- 1 The `{@template id}` syntax defines a reusable template, which must always have a terminating `{@endtemplate}` token at the end. To avoid errors, make sure to use unique ids for templates.
- 2 The `{@macro id}` token tells the `dart doc` tool to copy the template contents and apply them to the current string.

In this way, we can reuse various pieces of documentation without manually copy-pasting strings. Templates can be defined anywhere, and macros must always reference valid ids. Of course, templates and macros can be integrated with other regular documentation. For example:

```
/// Text
/// {@macro id}
/// Other text
```

Markdown is a simple text-to-HTML formatting syntax used in the most popular websites.

### 20.1.3 Static analysis and linter rules

Static analysis allows you to find problems in the Dart code before running it. It prevents common bugs and ensures that your code conforms to the official style guidelines. All warnings and errors can be reported in two ways:

1. if you run the `dart analyze` tool (or `flutter analyze` in case of Flutter projects)
2. if you use an IDE such as Android Studio or Visual Studio Code with the official Dart and/or Flutter extension(s).

There is no difference between the two approaches because IDEs use the `analyze` tool under the hood, so the result is the same. When you create a new Dart or Flutter project, an important file called `analysis_options.yaml` is generated at the root. It has two main purposes:

- It contains a series of rules, called lints, that are used to configure the `dart analyze` tool. For example, you can add or remove rules to make the analyzer more or less severe.
- When a new package or project is created, this file is configured to use a series of linter rules recommended by the Dart team. This is considered the “basic” setup for your project, and you should add more rules on top of these.

Let's make some examples to clarify the usage. When you create a new Dart or Flutter project, a basic `analysis_options.yaml` file is created so that `dart analyze` has the recommended setup. For example:

some_file.dart	analysis_options.yaml
<pre>Future&lt;int&gt; getContents() async {   try {     return await compute();   } on Exception {}   return -1; }</pre>	<pre># Default configuration. # Only a single 'include' statement. include: package:lints/recommended.yaml</pre>

Notice that there is an empty catch block. Even if it's not a compile-time error, you shouldn't use empty catch blocks because it is a bad practice. Because of `analysis_options.yaml`, when we run the `dart analyze` tool, we get this result on the console:

```
info • lib\src\example.dart:36:20 • Avoid empty catch blocks. • empty_catches
```

This is just a warning, so the code will still compile. However, you shouldn't ignore these warnings and try to resolve them. In our case, it is enough to add a body to the catch block:

some_file.dart	analysis_options.yaml
<pre>Future&lt;int&gt; getContents() async {   try {     return await compute();   } on Exception {     return -1;   } }</pre>	<pre># Default configuration. # Only a single 'include' statement. include: package:lints/recommended.yaml</pre>

If you try to run the analyzer again, the warning no longer appears:

```
No issues found!
```

Even if we don't recommend it, you could also disable this rule to remove the warning. In general, all rules can be disabled or enabled. However, you shouldn't disable recommended rules because they guarantee that your code meets the minimum code quality standards. To be more precise, you could do the following:

## some\_file.dart

```
Future<int> getContents() async {
  try {
    return await compute();
  } on Exception {}
  return -1;
}
```

## analysis\_options.yaml

```
# Default configuration.
# Only a single 'include' statement.
include: package:lints/recommended.yaml

linter:
  rules:
    empty_catches: false # Add this
```

You can disable individual rules so that the analyzer will ignore them. We suggest to NOT ignore rules coming from the recommended lint set. A very good practice instead is to add stricter rules and checks. For example:

```
include: package:lints/recommended.yaml
```

```
analyzer:
  language:
    strict-casts: true
    strict-inference: true
    strict-raw-types: true

linter:
  rules:
    - avoid_slow_async_io
    - avoid_type_to_string
    - recursive_getters
```

The first block is about severe type checks for language features. Those three rules are the only ones (currently) supported:

- **strict-casts**: When enabled, it ensures that a `dynamic` type is never automatically cast to another type. In other words, any `dynamic` type in your code will always need an explicit cast to another type (the compiler will not automatically make the cast for you).
- **strict-inference**: When enabled, the type inference engine never uses `dynamic` when the type cannot be deduced. For example:

```
final map = {};
```

This code would normally evaluate to `Map<dynamic, dynamic>` but when strict inference is enabled, the analyzer complains (but the code still successfully compiles). The reason is

that `dynamic` is not considered a safe option when the inference engine has no clue on the actual type. If you really need a generic type, prefer `Object?` instead of `dynamic`.

- `strict-raw-types`: When enabled, the type inference engine never uses `dynamic` when type arguments are not provided. This is commonly encountered when you use a collection without specifying the type. For example:

```
List values = const ['a', 'b'];
```

To solve the problem, just use `List<String>` with the explicit type in the diamonds.

To enable individual rules, list them all under the `rules` tag. Notice the slight syntax difference between adding and removing rules:

Adding linter rules	Removing linter rules
<pre>include: package:lints/recommended.yaml  linter:   rules:     - avoid_slow_async_io     - avoid_type_to_string     - recursive_getters</pre>	<pre>include: package:lints/recommended.yaml  linter:   rules:     avoid_slow_async_io: false     avoid_type_to_string: false     recursive_getters: false</pre>

To add rules, just lists them without using boolean flags. To remove rules, you need to pass `false` explicitly. This is the recommended<sup>120</sup> way of adding and removing rules form your analysis file.

## Note

The `analysis_options.yaml` file of our `equations` package contains almost all rules. There Dart website has a useful page with all<sup>121</sup> available lint rules. What we did is copy-pasting (almost) all of those rules into our project.

We have removed some conflicting ones, such as preferring single quotes and preferring double quotes (of course, only one of them can be enabled).

<sup>120</sup> <https://dart.dev/guides/language/analysis-options#disabling-individual-rules>

<sup>121</sup> <https://dart-lang.github.io/linter/lints/options/options.html>

The more rules you add to your `analysis_options.yaml` file, the healthier your project will be. Of course, using all rules doesn't guarantee that your code is perfect, but it surely follows Dart's best coding practices. If you think that some rules are essential, you can also change their severity. For example:

```
analyzer:  
  errors:  
    empty_catches: error  
    invalid_assignment: warning
```

For example, by default `empty_catches` is a warning, but with this change it becomes an error. This also means that the IDE will underline your code in red if it's the case. There are four severity levels:

- `error`, which makes the analysis health check fail;
- `warning`, which doesn't make the analysis health check fail (unless it is configured to treat warnings as errors) but logs errors in the console;
- `info`, which shows an informal message and never makes the analysis health check fail.
- `ignore`, which tells the analyzer to skip a rule.

Each rule has a default severity, and this configuration is the best way to change it. We recommend to not use `ignore` on rules unless there is a valid motivation.

## 20.2 Package maintenance

Dart's guidelines and best practices help developers to create projects that are easy to maintain. If you followed what we discussed in the previous section, you've already done a great job. There are a few more recommendations we want to give. They aren't strictly required but you should consider them:

- Avoid using `part` and `part of` directives when creating packages<sup>122</sup>; prefer the creation of small libraries instead. As we have already seen in *chapter 4 – Section 1.2.1 “part and part of directives”*, these directives allow you to split a library into multiple Dart files with access to private members. Prefer creating mini, internal libraries instead.
- Try to always use the `dart format` command in your projects to apply code formatting rules that follow the official Dart guidelines. This command changes your Dart files to apply

---

<sup>122</sup> <https://dart.dev/guides/libraries/create-library-packages#organizing-a-library-package>

the recommended formatting, such as removing white spaces when not needed or always add a new line after a comma.

- When creating Dart or Flutter packages, spend some time on writing a good `README.md` file. It serves as an overview of your package and thus it should catch the reader's attention with images and helpful information. It shouldn't contain a complete coverage of all features; just a "general purpose" introduction. The Dart online guide has a useful page<sup>123</sup> on how to write great README documents.
- Whenever your package version increases, add an entry in the `CHANGELOG` file. The purpose of this file is to list changes and new features you've introduced on each release. In general, it has a header with the version number followed by the list of changes.

The complete layout conventions for Dart and Flutter packages describe how you should organize files, directories, and files. These are all the contents that a package might have:

- `pubspec.yaml` file: every package has a pubspec file to define dependencies and project-level metadata.
- `README` file: it is used to describe the project using Markdown.
- `CHANGELOG` file: it is used for logging changes or new features of each package version. A changelog is especially useful when it comes to breaking changes, because you can include a summary or (if needed) a migration guide. Here's an example:

```
## 1.1.0
- BREAKING: Renamed C to D.
- Click here for the migration guide

## 1.0.1
- Fixed an issue in B.
- Improved performance on A.

## 1.0.0
- Initial version.
```

---

<sup>123</sup> <https://dart.dev/guides/libraries/writing-package-pages>

Each version has its own heading (generally of level 2) and then a bullet list with changes. In general, breaking changes are prefixed with the “*BREAKING*” term in bold.

- **LICENSE** file: if you are publishing the package to pub, you should also include a license file. The recommended one is the BSD-3-Clause so that other developers can reuse your code.
- **lib** and **test** folders: you already know these very well. The library code goes inside **lib** and tests are written inside **test**. Note that **lib** is public to other packages, meaning that all contents of the directory are visible by other packages.
- **bin** folder: This folder contains programs (written in Dart or any other scripting language) that can be run directly from the command line. For example, inside **bin/** you could place utility programs that set up the development environment or manipulate configuration files. You can run scripts in the **bin/** folder from anywhere using the **dart pub global** command.
- **benchmark** folder: if your package has performance-critical code and you want to measure the speed of a feature, this is the right place to create your benchmarks. There is an official package from the Dart team (**benchmark\_harness**<sup>124</sup>), which is the recommended starting point for building benchmarks. For example, imagine you had to test the performance of this **compute** method:

```
class Factorial {  
    const Factorial();  
  
    int compute(int n) {  
        if (n == 1) {  
            return 1;  
        }  
  
        return n * compute(n - 1);  
    }  
}
```

After having added the **benchmark\_harness** package as **dev\_dependency**, file create a new file inside the **benchmark** folder. It could look like this:

---

<sup>124</sup> [https://pub.dev/packages/benchmark\\_harness](https://pub.dev/packages/benchmark_harness)

```

import 'package:benchmark_harness/benchmark_harness.dart';

class FactorialBenchmark extends BenchmarkBase {
  const FactorialBenchmark() : super('Factorial');

  @override
  void run() {
    const Factorial().compute(20);
  }
}

void main() {
  const FactorialBenchmark().report();
}

```

The `run()` method runs ten times (for legacy reasons) and reports the average execution time in the console. If you don't like this, you can override the `exercise()` method to run benchmarks only once:

```

@Override
void run() {
  const Factorial().compute(20);
}

@Override
void exercise() => run();

```

In either case, call `dart run` on the benchmark file and read the output time (expressed in microseconds). The output is automatically generated when calling the `report()` method from the `main()` entry point:

```

PS C:\Users\Alberto\equations> dart run .\benchmark\factorial_benchmark.dart
Factorial(RunTime): 0.02049330545901339 us.

```

Figure 20.4: The benchmark output for the `Factorial.compute` method.

To benchmark asynchronous operations, extend the `AsyncBenchmarkBase` class instead of `BenchmarkBase`.

- `web` folder: for web packages, the `main()` Dart entry-point goes here along with other static files such as HTML pages, CSS styles or images. Flutter for example uses this folder to gather all JavaScript, HTML and CSS code of your application.

- `tool` folder: if your package needs Dart scripts to automate some processes (such as code generation tools), this folder is the best place for them. Unlike `bin`, the `tool` folder is not for external users of the package.
- `example` folder: this folder is used to contain a demo Dart (or Flutter) application that uses your package to show its features. For example, you could create a small Flutter application inside `example` to demonstrate the package features. The Dart team recommends to always add a `README` file as well to describe the content of the `example/` folder.

In Flutter, you might also want to add an `assets` folder for your static assets and `font` for custom fonts loading.

### 20.2.1 GitHub repository setup

Our `equations` package is hosted on GitHub as an open-source project. Besides following Dart's best practices and guidelines for package creation, we also have set up a CI pipeline. Inside the `.github/workflows` folder<sup>125</sup>, we created an action that checks the package health every time you push a commit on an open PR. Here's the most relevant part:

- `name`: Installing the dependencies  
`run`: `dart pub get`
- `name`: Making sure the package is formatted  
`run`: `dart format --set-exit-if-changed lib test`
- `name`: Making sure that there are no analysis warnings or errors  
`run`:  
`|`  
`dart analyze --fatal-infos --fatal-warnings lib test`  
`dart run dart_code_metrics:metrics analyze lib test`
- `name`: Running tests with coverage  
`run`:  
`|`  
`dart pub global activate coverage`  
`dart pub global run coverage:test_with_coverage`
- `name`: Making sure that code coverage is 100  
`uses`: `VeryGoodOpenSource/very_good_coverage@v2.1.0`  
`with`:  
`min_coverage: 100`

<sup>125</sup> <https://github.com/albertodev01/equations/tree/master/.github/workflows>

Make sure to check the `equations` GitHub repository to see the entire configuration. Thanks to this workflow, GitHub automatically checks our package health:

1. if we forgot to call `dart format`, the CI workflow will fail;
2. if the `dart analyze` reports warnings or errors, the CI workflow will fail;
3. if the `dart_code_metrics` package reports warnings or errors, the CI workflow will fail;
4. if one or more tests fail, the CI workflow will fail;
5. if the coverage is less than 100%, the CI workflow will fail.

Rather than manually running all of these commands in your machine, GitHub can do everything for you. This is a systematic and automated process that checks the code health whenever someone pushes changes. We strongly recommend that you setup a CI workflow for your package (even if you're using another platform such as GitLab or BitBucket).

### Note

To create a GitHub action, you just need to create a YAML file in `.github/workflows`. For advanced configurations, check the official GitHub documentation <sup>126</sup> about actions.

The `equations` package also has a demo Flutter application in the `example` folder that showcases its features (such as equation and system-solving algorithms). This folder has a CI workflow that checks the Flutter code health. Some other nice additions are:

- Badges in the `README` file to report full code coverage and successful CI checks. These only are aesthetic additions but they capture the user's attention and give important information about the package quality:



Figure 20.5: Badges of the `equations` package.

---

<sup>126</sup> <https://docs.github.com/en/actions>

The pub badge indicates that the package is published at <https://pub.dev> and its current version.

- We include a pull request and an issues template, still in the `.github` folder, to guide new users when they interact with our repository.
- Consider adding a `CODE_OF_CONDUCT.md` file to your repository. This is a GitHub file that defines standards for how to engage in a community and it's considered a good practice adding it. Since it's pretty standard, you can copy-paste existing ones and use them as templates to craft your own.

An interesting GitHub feature called *GitHub pages* allows you to host web projects within your repository. In our case, the `example` folder contains a Flutter application that uses the `equations` package. We are using this CI workflow to build the application for the web and publish it to the repository's GitHub pages space:

- `name`: Installing the dependencies  
`run`: flutter pub get
- `name`: Building for the web  
`run`: flutter build web --csp --release
- `name`: Setting the git username  
`run`: git config user.name github-actions
- `name`: Setting the git email  
`run`: git config user.email github-actions@github.com
- `name`: Adding web source  
`run`: git --work-tree build/web add --all
- `name`: Adding a commit message  
`run`: git commit -m "Automatic web deployment by github-actions"
- `name`: Automatic web deployment  
`run`: git push origin HEAD:equations\_web -force

When we release a new package version, we merge everything into `master`. At this point, GitHub automatically builds the application and publishes it at <https://albertodev01.github.io/equations/>. This URL is automatically generated and is associated with the repository. We think this is awesome because it is an automated way of building the Flutter demo application without efforts and users can use it from anywhere.

## Note

Users don't have to clone your repository, build the project and run it. They just need to click the GitHub pages link and play with the Flutter web build.

You don't even need to maintain the web app deployment since GitHub automatically deletes the old build and replaces it with the new one every time you push to `master` (or anywhere else you want).

When possible, we recommend to create a Flutter demo in the `example` folder so that it can be published to GitHub pages. This is not possible when, for example, you're building a command line Dart application or a web server because GitHub pages only process static websites.

### 20.2.2 Dependencies and pub package manager

When you add a dependency to your package, you can either specify a specific version or declare a range of allowed versions. Package versions are commonly prefixed with a caret to support multiple releases. For example:

```
dependencies:  
  equations: ^4.1.0
```

The `^4.1.0` value means that your project is able to work with versions from `4.1.0` to `5.0.0`, but `5.0.0` is not included. This syntax also has some equivalents, which we don't recommend using. For example:

```
dependencies:  
  equations: '>=4.0.1 <5.0.0'
```

Alternatively, if you didn't use the caret, you'd lock your dependency to `4.1.0` and no new versions will be fetched if available. The Dart team recommends<sup>127</sup> to always use the caret syntax. However, this is not the only way you have to depend on a package. There are several variants:

1. Hosted packages. They are fetched from `pub.dev` and require the version:

```
dependencies:  
  my_package: ^2.4.1
```

---

<sup>127</sup> <https://dart.dev/tools/pub/dependencies#best-practices>

2. Git packages. They are fetched from a git repository, either using an HTTP or an SSH uri. The optional `ref` key is used in case you want to depend on a specific commit or branch:

```
dependencies:  
  my_package:  
    git: https://github.com/user-name/my_package.git  
    ref: branch-name  
  
  my_other_package:  
    git: git@github.com:user-name/my_other_package.git
```

If the package is not located in the root of the git repository, you must specify its `path` using the `path` key.

3. Path packages. They are loaded from your local filesystem and use the `path` key:

```
dependencies:  
  my_package:  
    path: /path/to/package/my_package
```

Relative paths are allowed and are relative to the directory that contains the pubspec file. Path dependencies are helpful for development, but they do not work when you're sharing the package with the outside world.

4. SDK packages: They are located in the SDK and are retrieved from there. For example, Flutter is shipped with the `flutter_test` package, so you can directly load it from there using the `sdk` key:

```
dev_dependencies:  
  flutter_test:  
    sdk: flutter
```

Currently, `flutter` is the only supported SDK.

To install and use the dependencies you declared, use the `dart pub get` command. On a Flutter project, use `flutter pub get` instead, which is a `dart pub` wrapper with some additions. Pub is Dart's package manager tool with a series of useful commands. Here are the most popular ones:

- `dart pub get`: gets all the dependencies listed in the pubspec file. It fails in case of version conflicts. It maintains a cache of packages for offline use and better fetching performance. By default, `get` always fetches data from the internet but if you are offline, you can pass the

--offline flag to try to fetch packages from the cache. When calling get, a lockfile is always created and maintained to make sure that future fetches will use the same dependencies versions.

- **dart pub outdated**: when you have a lot of dependencies, and some of them are very old, you might **get** errors when calling get due to version conflicts. Thanks to **outdated**, you can see which packages need to be updated and which is the highest compatible version. For example:

```
PS C:\Users\Alberto\example> dart pub outdated
Showing outdated packages.
[*] indicates versions that are not the latest available.

  Package Name      Current  Upgradable  Resolvable  Latest
  direct dependencies:
    flutter_svg        *1.1.6    *1.1.6      2.0.5       2.0.5
    go_router           *4.5.1    *4.5.1      6.5.7       6.5.7
    google_fonts        *3.0.1    *3.0.1      4.0.3       4.0.3
    intl                 0.18.0    *0.18.0     *0.18.0     0.18.1

  dev dependencies:
    dart_code_metrics   *4.21.3    *4.21.3      5.7.2       5.7.2

  4 dependencies are constrained to versions that are older
  than a resolvable version.
  To update these dependencies, edit pubspec.yaml, or run
  `dart pub upgrade --major-versions`.
```

Figure 20.6: Output of **dart pub outdated** on a Flutter application.

This command shows a complete overview of the dependencies versions statuses. After you ran this command, you might need to manually update versions in the pubspec files. In other cases, the tool might tell you to run **dart pub upgrade** to upgrade locked dependencies. Make sure to visit the official documentation for a detailed overview of this command.

- **dart pub publish**: publishes your package to the *pub.dev* website. We will see how to use this command in the next section.

- `dart pub global`: this command allows you to run Dart command line applications when you are outside of that package. After having activated the package, you can run its `bin` directory content from anywhere. For example, you could create a new Dart CLI application called `demo` and run it this way:

- 1 Activate it with `dart pub global activate demo`.
- 2 Run the application from anywhere using `dart pub global run demo`.

The `run` command looks for the `main()` entry point in your application's `bin` folder.

- Instead of manually adding and removing dependencies from the `pubspec.yaml` file, you can use `dart pub add <pkg-name>` and `dart pub remove <pkg-name>` from the console. When adding a package, `add` automatically downloads the latest version. To add a package as dev dependency, use `dart pub add --dev <pkg-name>`.

When using Flutter, replace `dart` with `flutter`. Note that pub ignores all dev dependencies and doesn't include them into versioning checks. For this reason, test packages (such as `flutter_test`, `test` and `coverage`) are generally added as dev dependencies so that everyone can use specific versions without external constraints.

## Deep dive: Publishing packages at pub.dev

Once your package is ready and well tested, you could decide to publish it at `pub.dev`. Keep in mind that publishing is forever<sup>128</sup>: once uploaded, a package can only be discontinued but it won't be deleted. A discontinued package won't appear in pub searches but it will still remain uploaded.

### Note

When you publish a package, other people might depend on it. If you will later decide to remove it, you will end up breaking other people's work as well. For this reason, pub discontinues a package rather than deleting it. In this way, any user can move away from your package at its own pace.

---

<sup>128</sup> <https://dart.dev/tools/pub/publishing#publishing-is-forever>

When publishing a package at `pub.dev`, you should follow Dart's guidelines and best practices as much as possible. Before publishing or uploading a new version, run this command:

```
dart pub publish --dry-run
```

This is not required, but it makes sure that your package has a valid `pubspec` file and follows Dart's layout conventions. The console will also output potential warnings or errors to address. If it's all ok, then you can proceed with the actual publishing:

```
dart pub publish
```

If it is a new package, the console will output a validation link you must open. It will register a Google account of your choice to pub and upload the package. After a few minutes, you will see the package online. For example, this is the header of <https://pub.dev/packages/equations>:

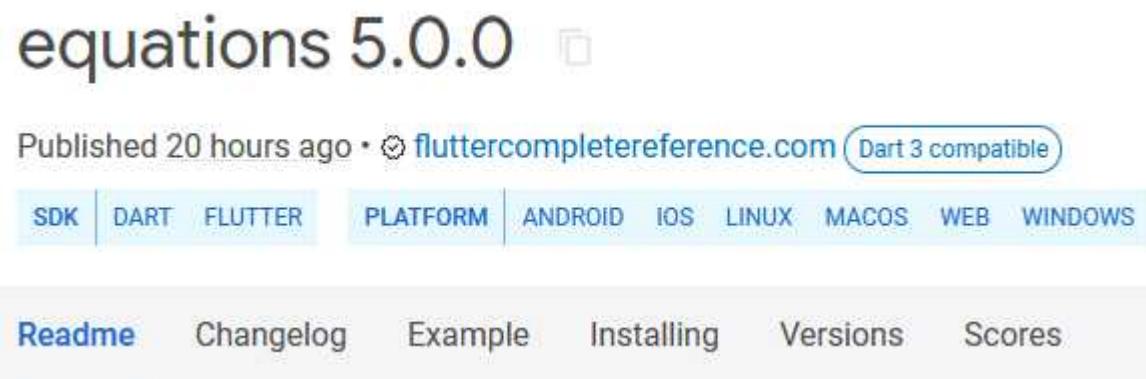


Figure 20.7: The header of the `equations` package at `pub.dev`.

Below the package name and its version, you can see the supported platforms. In the next section, *Verified publishers*, we will see how to add the validation mark to your pub packages. The six tabs below serve different purposes:

1. **Readme.** This tab shows the `README.md` file at the root of the package. In other words, this is your package's home page and thus you should write a very good README. For example, try to include a colored logo and a short description at the top. Then, use lists or sections to include important information, write examples and use visual contents when needed.
2. **Changelog.** This tab shows the `CHANGELOG.md` file that keeps track of changes of each new version. It should contain a series of bullet lists, sorted by release date, of all changes that occurred in your package.

3. Example. This tab is linked to the `example` folder of your package. If you have created it, it will appear here. In our case, we have just placed a README file to indicate the presence of a Dart and a Flutter example projects in two separated subfolders. Users can navigate to the GitHub repository to see them.
4. Installing. A quick guide on how to depend on the package.
5. Versions. A list with all the versions your package was updated to. This page also allows you to download a ZIP file with the package contents of a specific version.
6. Scores. This page shows some metrics and assigns a score based on package quality and health. For example, this is the score page of the `equations` package:

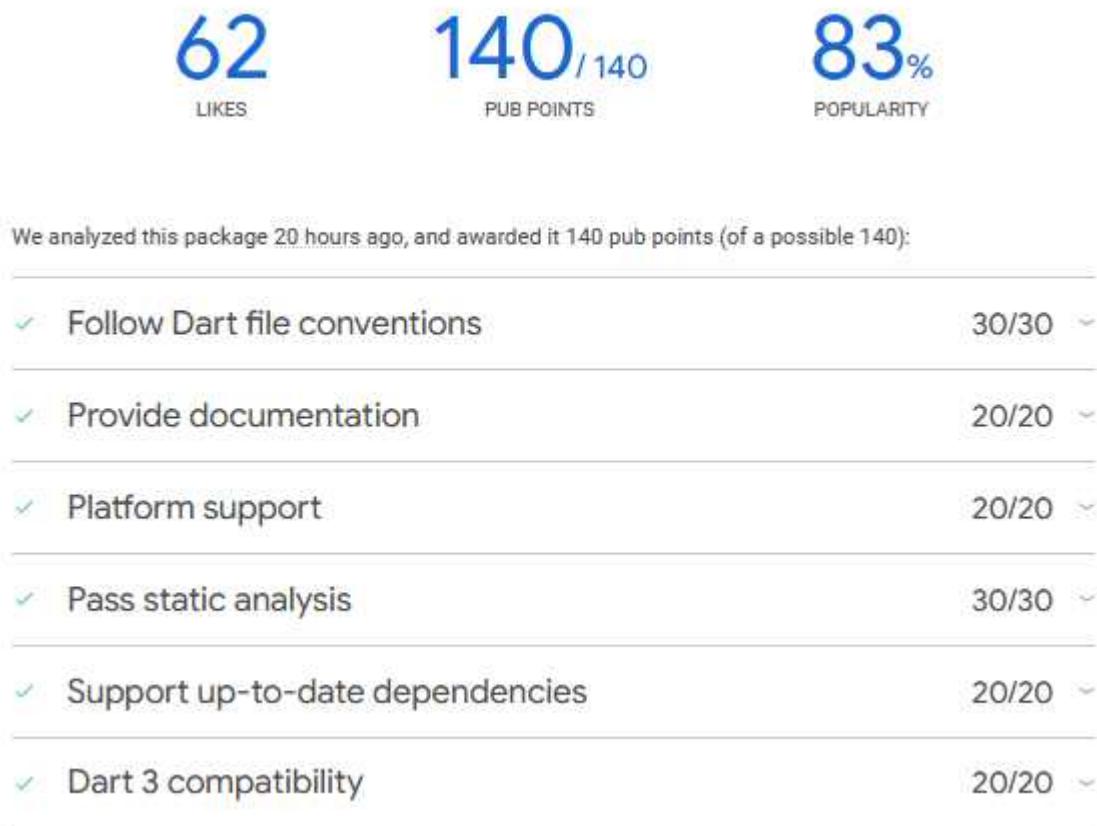


Figure 20.8: The scores of the `equations` package.

[Likes](#) indicates how many people clicked the *Thumbs up* button (in the upper-right corner of the package home page). [Popularity](#) measures the number of applications that depend on the package on the past 60 days. [Pub points](#) measures your package quality and it's the sum of all metrics you see listed in *Figure 20.8*.

You should always keep your quality bar very high for published packages. We recommend to try to always have the highest score possible to get more trust from developers and demonstrate your effort to create a great content.

## Verified publishers

While visiting package pages at *pub.dev*, you might notice that certain publishers have a check icon next to their names. This symbol doesn't give any information about the quality of the code, but it's all about the publisher's authenticity. For example:



Figure 20.9: The verified publisher icon next to the domain name.

The official Dart documentation<sup>129</sup> also recommends becoming a verified publisher. Imagine you already had a Google account registered at *pub.dev* and a few published packages. To become a verified publisher, you need to take a few steps:

1. Any verified publisher must have a domain to associate to his pub account. In our case, the domain is *fluttercompletereference.com* (which also points to our website). You just need a domain for this purpose, it doesn't also have to be associated to an active website.
2. Using the same Google account you used to upload your packages, open the Google Search Console<sup>130</sup> website. From here, click on the dropdown menu in the top-left corner and choose “Add property”. Insert your domain and click “Continue”. Follow the on-screen steps to add a new TXT entry in your domain’s DNS configuration. Note that DNS changes may take a few hours to apply.

---

<sup>129</sup> <https://dart.dev/tools/pub/publishing#verified-publisher>

<sup>130</sup> <https://search.google.com/search-console>

3. Once DNS changes propagated, go back to `pub.dev` and login with your Google account. In the top-right corner, hover the dropdown and click the “*Create publisher*” option. Enter the domain name you verified in the previous step and confirm your changes.

Go in the Admin area of your non-verified packages, select your domain from the publisher list and click the red “*Transfer to publisher*” button. In this way, your package will be associated to the verified publisher you created.

## Flutter favorites

Being a verified publisher is not only a matter of pride but it also gives you a chance to become eligible for the Flutter Favorite<sup>131</sup> program. It aims to identify packages that you should consider first when building your applications. Note that a Flutter Favorite package may not always be the best choice for your use case but, if it is, you have guarantees on its quality and stability.



Figure 20.10: The Flutter Favorite logo.

Each Flutter Favorite package has the logo in *Figure 20.10* on its pub page. Becoming part of this program is not easy because your package must pass a series of high-quality standards that follow these metrics:

- great overall package score,
- a permissive license,
- all declared features must be implemented,

---

<sup>131</sup> <https://docs.flutter.dev/development/packages-and-plugins/favorites>

- the package must be owned by a verified publisher,
- high-quality dependencies,
- great runtime behavior,
- great code quality, good documentation, and a good number of examples.

Flutter team members and community managers gather together to decide when a package is good to become a Flutter Favorite. It is unlikely that a newly published package immediately joins the program because community approval and widespread usage are very important. If you follow all Dart and Flutter best practices and guidelines, your packages already are on a good track to possibly become “Flutter Favorites” in the future.

# 21 – Creating and maintaining a Flutter app

---

## 21.1 Flutter project creation



[https://github.com/albertodev01/equations/tree/master/example/flutter\\_example](https://github.com/albertodev01/equations/tree/master/example/flutter_example)

A Flutter application can be seen as a particular variant of a Dart package. Instead of using the `dart` tool, you need to use its wrapper called `flutter`. To create a new Flutter application that follows the recommended project structure, you can either:

- create a new project with your favorite IDE, such as Android Studio or VS code;
- use the `flutter create project_name` command, which is also used under the hood by your IDE.

In this chapter, we will use as a reference the Flutter demo application located in the `example` folder of the `equations` package. Most of the best practices we have seen in *chapter 20 – Creating and maintaining a package* also apply here. For example, you already know that the application logic goes into `lib` and unit/widget/golden tests go into the `test` folder. In addition:

- A basic `analysis_options.yaml` file is automatically generated for new projects but it only has a few rules that are considered the minimum for a good quality application. Make sure to add the highest number of linter rules possible. You don't necessarily need to add them all but the more the better.
- A Flutter application is not meant to be published at pub.dev. As such, the `pubspec` file has the `publish_to: none` property set to ensure that you don't accidentally try to upload the project at pub.
- If you remove a supported platform from the project, you will be able to add it later. For example, imagine you deleted the `web` folder from your application and some days after you wanted to add it back. In this case, run the `flutter create .` command to re-generate the `web` folder with all of its contents.

Most of `dart` commands have their respective `flutter` counterpart. For example, to run tests, you'd use `flutter test` instead of `dart test`. An interesting command, only available on the `flutter` tool, is the `channel` one. The term “Flutter channel” indicates the source from where you

download and install Flutter. Different channels have different Flutter releases and also different stability statuses. There are three channels:

- **master**: this channel contains the very latest Flutter build. “**master**” is mainly used by the Flutter team to introduce new features and has the lowest stability level. You should only use this channel to play with in-progress features but do not use it for production because there might be breaking bugs.
- **beta**: this channel contains a more stable and reliable version of Flutter. Once a month, the Flutter team branches **beta** from **master** and starts accepting cherry-pick requests for high impact issues.
- **stable**: this is the recommended channel for building production applications because it contains the most stable Flutter build.

When you install Flutter, the **stable** channel is the default one. The only reason why you should change the channel is to play with some upcoming features. For example:

```
flutter channel dev
```

This command changes your Flutter version to the one that is currently in the **beta** channel. Always remember to use **stable** when you’re working on production applications. To see which channel you’re using, just run **flutter channel** without arguments.

### 21.1.1 Project organization

The Flutter team doesn’t recommend any specific folder structure, so you have freedom of choice in this area. There are some additional folders for which you should keep the naming conventions choices:

- **assets**: this top-level folder gathers all static assets, such as images or multimedia files. You should always remember to compress assets so that they take the least amount of space. If you have various types of assets, you should consider splitting them into dedicated folders.  
For example:



Figure 21.1: A possible structure of the assets folder.

In this case, we have a single JSON file but multiple vectorial and audio files. As such, we kept the single file inside `assets` but others have been grouped together in folders having the same name as the extension type.

- `google_fonts` or `fonts`: Flutter uses some platform-specific default font styles but you can of course customize them. We have already seen it in *chapter 16 – Section 1.2 “Font assets”*. If those fonts interact with the `google_fonts` package, create the `google_fonts` folder; otherwise, you use the generic `font` name.
- `packages`: in some cases, you might need to create reusable packages for your application. For example, imagine you had to implement a REST client with a complicated JSON parser. You could create a top-level folder called `packages` and create mini-libraries in there that are shared across your application. For example:

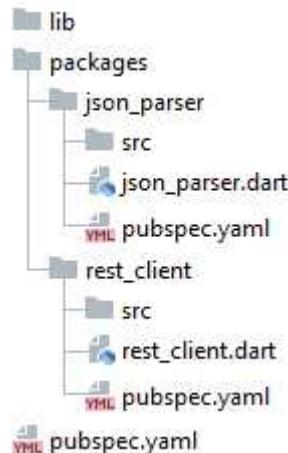


Figure 21.2: A series of mini-libraries inside the `packages` folder.

This structure makes maintenance easier. If other applications could also use those libraries, you could move them to a separate project. Note that each package has its own `pubspec` file, for easier dependencies management. We recommend creating the `package` folder for consistency with Dart’s package conventions. We think it is better than having a huge folder structure inside `lib`.

- `integration_test`: this is the conventional folder name for integration tests and should not be changed. If you have one test per feature, you can just put single files directly inside `integration_test`. Consider grouping test files in the same folder if you have multiple tests

for the same feature. When it comes to writing tests, we recommend to use in `test` the same folders and file structure you use inside `lib`.

### 21.1.2 Good practices and adaptive apps

If you are extremely sure that your application doesn't need to be localized and strings don't need translations, then you can hard-code all values. In general, we would still prefer to internationalize our application (even if it's for a single language) for a few reasons:

- The ARB file system moves all strings into dedicated files that can easily be maintained. You won't have hard-coded strings spread across the application: they will all be in a single place, and Flutter will take care of them. This is very useful even for just a single language.
- If you wanted to add support for other languages, you would simply have to add more ARB files. No need to refactor the Dart code because Flutter will automatically handle everything.
- Internationalizing a mature application is certainly a time-consuming operation, especially if you hard-coded all values before. If you start the internationalization process from the first day, adding new locales will be smooth.

We recommend setting up internationalization in your projects from the beginning, following the recommendations of *chapter 15 – Section 1 “Internationalization”*. If your application has to run on different platforms, it should also be adaptive from the beginning.

#### Note

Making an adaptive application means that, when it runs on different platforms, it deals with different mechanics. If a Flutter application runs on mobile and desktop platforms for example, it has to support touch inputs and keyboard events. The same application could also use cascading menus on desktop and bottom sheets on mobile. In practice, the application automatically “adapts to the context” and changes itself to appeal the user according with the platform.

For example, if you know you're creating a mobile-only application, you can ignore a wide series of use cases related to keyboard events, mouse gestures, or cascading menus. If your application is meant to run on any platform, the complexity and the maintenance efforts are inevitably higher. For example, imagine you're working on a custom button for your application:

```

GestureDetector(
  onTap: tapCallback,
  onTapCancel: tapCancelCallback,
  child: Container(
    width: 250,
    height: 60,
    color: Colors.blue,
    child: const Text('OK'),
  ),
),
),

```

This implementation is good for mobile devices since the input only comes from a tap on the screen. However, this widget is not adaptive because if it was used on a desktop platform, it wouldn't be able to interact with the keyboard or the mouse cursor. To make it adaptive, we would have more work to do. For example:

```

GestureDetector(
  onTap: tapCallback,
  onTapCancel: tapCallback,
  child: FocusableActionDetector( // handles keyboard events
    actions: actions,
    shortcuts: shortcuts,
    onShowHoverHighlight: (isCursorHover) {
      // handle cursor 'hover' state here
    },
    child: Container(
      width: 250,
      height: 60,
      color: Colors.blue,
      child: const Text('OK'),
    ),
  ),
),

```

Now this widget can handle finger taps, detect the mouse cursor position and listen for keyboard events. We have covered this topic in *chapter 18 – Section 3 “Gestures”*. Supporting more than a single platform also means dealing with various screen sizes, from small ones on mobile phones to larger ones on ultra-wide 4k monitors. There is a lot to do to create a dynamic UI:

- Consider setting some breakpoints and use `LayoutBuilder` to change the UI according with the screen size. For example, on a mobile phone a `BottomNavigationBar` is a good idea but on a desktop, especially with a very wide monitor, you should prefer something else such as a `NavigationRail`.

- The official Flutter documentation recommends<sup>132</sup> the creation of constant values, inside dedicated files to control your widgets styling. For example, rather than using hard-coded inset values, prefer gathering all values in a dedicated file and reuse them:

```
// file: 'insets_values.dart'  
const xSmallInset = 3.0;  
const smallInset = 4.0;  
const mediumInset = 6.0;  
const largeInset = 8.0;  
const xLargeInset = 12.0;
```

Having a single place where all values are defined makes maintenance and testing easier. Of course, your widgets should use these constants rather than other hard-coded values:

```
const Padding(  
  padding: EdgeInsets.all(mediumInset),  
  child: MyWidget(),  
,
```

In this way, the change of a value automatically propagates to any widget that depends on it. Other than insets, you can also create constant values for strokes, animation timings, font families, font sizes, corner radiiuses and much more.

- Both desktop and mobile users expect scrollbars, but they behave differently. Generally, the scrollbar is thinner on mobile and only appears while the list is scrolled. On desktop, the scrollbar is thicker and always visible. For this reason, you should use the `Scrollbar` widget (which supports adaptive colors and sizes):

```
Scrollbar(  
  thumbVisibility: isDesktop,  
  controller: scrollController,  
  child: ListView.builder(  
    controller: scrollController,  
    children: children,  
,  
,
```

---

<sup>132</sup> <https://docs.flutter.dev/development/ui/layout/building-adaptive-apps#single-source-of-truth-for-styling>

The `thumbVisibility` property, when `true`, keeps the scroll bar indicator always visible. It should only be `true` on desktop devices because web ones might also run on mobile devices (or inside webviews). You could implement the `isDesktop` getter in this way:

```
bool get isDesktop => TargetPlatform.macOS || TargetPlatform.windows  
|| TargetPlatform.linux);
```

The `TargetPlatform` enum is part of Flutter and is the recommended way for determining the host platform. It's compatible with mobile, desktop, web, and embedded platforms.

- Avoid hard-coding sizes and layouts. Prefer using `Row`, `Column`, `Wrap`, `Stack`, and all those widgets that shrink or expand as much as possible.
- Desktop users generally expect to be able to select the visible text with the mouse cursor. The regular `Text` and `RichText` widgets cannot be selected with the mouse cursor. Instead, the text of a `SelectableText` widget can be selected:

```
Column(  
  children: const [  
    // Use 'SelectableText' instead of 'Text'  
    SelectableText('Hi!'),  
    // Use 'SelectableText.rich' instead of 'RichText'  
    SelectableText.rich(  
      TextSpan(  
        children: [  
          TextSpan(text: 'Hello'),  
          TextSpan(  
            text: 'World',  
          ),  
        ],  
      ),  
    ),  
  ],  
,
```

Text selection also supports the right-click action to copy the text.

As you can see, there is quite a lot of work to do to create an adaptive application. Flutter comes with a lot of ready-to-use widgets for a wide variety of use cases. If your application is designed to run on a specific platform, then you can ignore some functionalities and focus on others.

## 21.2 Maintaining a Flutter project

The same recommendations we gave in *chapter 20 – Section 2 “Package maintenance”* are also valid for Flutter projects. The only difference is that you need to use the `flutter` tool rather than the `dart` one. For example, to fetch packages you have to use `flutter pub get` and not `dart pub get`.

### 21.2.1 Tests and code coverage

Rather than just “*good*”, we believe that writing tests is a “*best*” practice. Your Flutter applications should always have (at least) unit and widget tests. It would be even better if you also wrote golden and integration tests. In general, we recommend to:

- write unit tests for pure Dart objects;
- write widget tests and at least one golden test for each widget of your application;
- when possible, include integration tests (at least for critical use cases).

Golden tests should be created inside folders called `goldens/` (which is a popular convention). We believe that golden tests play a crucial role in the overall widget testing experience. They allow you to test in a single shot many details such as borders, colors, spaces, or any other property that would be tedious to test manually. For example:

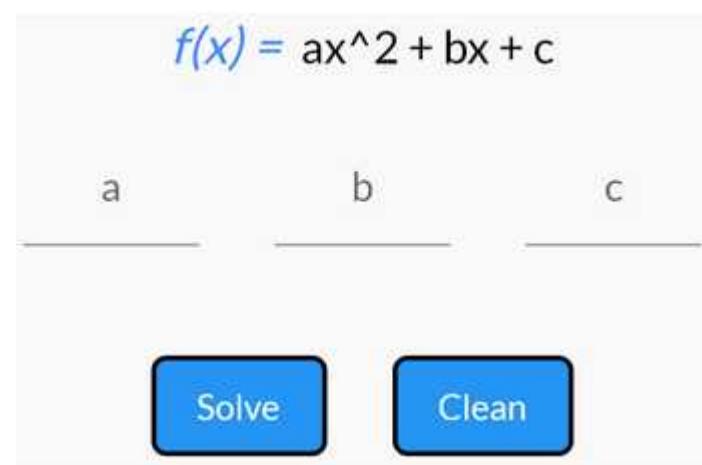


Figure 21.3: The golden test of the quadratic equation input form.

In *Figure 21.3* you see the golden image of the form we use to input quadratic equations. At first sight, it already gives you a rough idea of what the widget looks like, even if you've never seen the

application before. More importantly, it's advantageous to ensure that all widgets are spaced as expected, colors match the design we want to achieve, and font sizes are correct.

### Note

We could have tested colors, spaces and paddings in regular widget tests as well but it would have been much more difficult. After having used `tester.widget<T>` to get the reference to the widget, we would have had to check properties one by one. Of course, all of this code has to be maintained.

Golden images instead are automatically generated by Flutter and a golden test is able to detect all of those changes for us.

In the Flutter application of the `equations` package, for example, we tested widgets following this approach:

1. Start with a smoke test to ensure that all properties are set and internal widgets are in place. If there were assertions in the constructor, we would create multiple smoke tests to cover all cases.
2. Proceed by exercising the widgets as much as possible. For example, we always ensure that all callbacks are invoked, verify that animations are correctly run, and check that all inherited widgets are located in the right place. Listenable classes are unit tested.
3. Create at least one golden test per widget. If a widget has more than a single state, create multiple golden images (one per state).

Finally, for each application page, we created an integration test. We can assume that this is a solid testing strategy for a Flutter project.

Flutter also has built-in support for code coverage, which works in the same way as the `dart` tool. You need to run a single command, and it will generate the coverage report in the `coverage` folder:

```
flutter test --coverage
```

This command outputs the classic LCOV file, which can be parsed by dedicated tools to generate coverage reports. In our case, we have linked the GitHub repository to `codecov.io`. It's a popular website that generates HTML coverage reports and integrates with your GitHub repository.

### 21.2.1 GitHub repository setup

We have already seen in *chapter 20 – Section 2.1 “GitHub repository setup”* how to use GitHub actions to build and publish a Flutter web application to GitHub pages. It is a convenient way to share the latest version of your application without asking users (or testers) to install it. This is an extract of the workflow setup we use for the example application of the [equations](#) package:

```
- name: Checkout
  uses: actions/checkout@v3
- uses: subosito/flutter-action@v2
  with:
    channel: 'stable'

- name: Installing dependencies
  run: flutter pub get

- name: Making sure the package is formatted
  run: dart format --set-exit-if-changed .

- name: Making sure that there are no analysis warnings or errors
  run: |
    flutter analyze --fatal-infos --fatal-warnings lib test
    flutter pub run dart_code_metrics:metrics analyze lib test

- name: Running unit, widget and golden tests
  run: flutter test
```

We strongly recommend using a well-configured `analysis_options.yaml` file, with a lot of rules and strict type checks. It's also a good practice to use `dart format .` on your project root to format all Dart files with the `format` tool. If you have integration tests, as it happens for our project, you can also run them using this configuration:

```
runs-on: macos-latest
steps:
- name: Checkout
  uses: actions/checkout@v3
- uses: subosito/flutter-action@v2
  with:
    channel: 'stable'

- name: Installing dependencies
  run: flutter pub get

- name: Running integration tests on macOS
  run: flutter test integration_test -d macOS
```

This script runs integration tests on macOS, but you could also run them on iOS, Android, Windows, Linux, or web browsers. Setting up a workflow for integration tests on desktop platforms is easier because no emulators are required (which are tedious to install and maintain).

### Note

We have noticed that integration tests on mobile devices aren't easy to set up on CI workflows and take a lot of time. We prefer running integration tests on real mobile devices and using desktop platforms on GitHub actions.

We believe that an automated testing suite is one of the most important configurations to take care of. We also believe that writing a suitable README file is also fundamental. For example:

## Equations solver

An equation-solving application created with Flutter that can run on Android, iOS, Windows, macOS, Linux, and the web. This project shows how the `equations` package is used to solve equations, systems, integrals, and much more. The application currently supports the following languages:

- English
- Italian
- French

If you wish to add more languages, please create a new `app_xx.arb` file (replace `xx` with the language code) inside `lib/localization` and submit a PR! 



[Equation Solver - Flutter web demo](#)

Figure 21.4: A piece of the README file of the `equation`'s demo Flutter application.

As you can see in *Figure 21.4*, we have added a link to the online Flutter web demo and added a small introduction to the application. We also recommend including templates for new issues and pull requests as described in *chapter 20 – Section 2.1 “GitHub repository setup”*.

## 21.3 Creating release builds

This section describes the most important steps to take to create a production build, for different platforms, of your Flutter application. We are not covering how to distribute the executable in the various stores because the processes require working with external tools (whose coverage is not in the scope of this book). The Deployment section of the official Flutter documentation<sup>133</sup> contains updated information on how to publish your applications in the various stores.

### 21.3.1 Android

We recommend doing a few things before creating the final release build of an Android application. Some are optional but, for the sake of quality, you should try to address all the points we’re listing:

- When you create a new Flutter project, a default application icon for Android is created. Inside `android/app/src/main/res/` you’ll find a long series of folders with the *mipmap* prefix. Each folder has a single image file containing the same logo in different sizes. Don’t change the file name: update the various images and keep the original size. Android will automatically pick the best asset according to the device screen.
- If you have a large application or use many plugins, you may get Android’s dex limit of 64k method when targeting API 20 or lower. This may also happen when running the application in debug mode and code shrinking is not enabled. Flutter is smart enough to recognize this issue and asks you to fix the error. Unless you want to fix the problem manually, say “Yes” to the automatic fix request.
- If you need internet access or read/write permissions, for example, you should declare them in the `AndroidManifest.xml` file, located at `android/app/src/main`. There are a lot of permissions labels, which can be found in the official documentation<sup>134</sup>. Note that when the application runs in debug mode, the internet permission is enabled by default. However, in

---

<sup>133</sup> <https://docs.flutter.dev/>

<sup>134</sup> <https://developer.android.com/guide/topics/manifest/manifest-intro#perms>

release mode, there is no default internet permission (you must manually enable it). Make sure to only add permissions that your application needs.

- In the manifest file, you can change the application name that appears on the device's home screen (below the logo). You need to give a value to the `label` attribute:

```
    android:label="My application"
```

In the Flutter demo application of the equations package<sup>135</sup>, we localized the name. To do so, we've passed a reference to `label` rather than hard-coding a value:

```
    android:label="@string/app_title"
```

After this change, we created a `strings.xml` file inside the `res/values/` folder with the English name (the default value). For each supported language, we created a folder called `values-xx/`, where `xx` is the two-letter country code identifier:

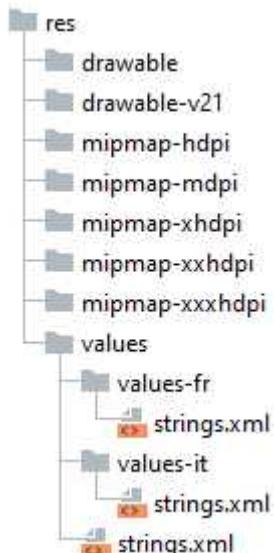


Figure 21.5: The structure of the `values/` folder to localize the app name.

---

<sup>135</sup> [https://github.com/albertodev01/equations/tree/master/example/flutter\\_example/android](https://github.com/albertodev01/equations/tree/master/example/flutter_example/android)

In Figure 21.5, you can see that we have translated the title into Italian (`values-it/`) and French (`values-fr/`). We have copy-pasted the `strings.xml` file in each folder and we have only changed the `app_title` value:

```
<!-- values/values.xml -->
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_title">Equation solver</string>
</resources>

<!-- values/values-it/values.xml -->
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_title">Risolutore di equazioni</string>
</resources>

<!-- values/values-fr/values.xml -->
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_title">Équation solver</string>
</resources>
```

Android automatically picks the correct file according to the device's locale settings. In this case, English is the default language.

1. All Android applications don't start immediately: they require some initialization time while the operating system initializes the process. Because of this, Android provides the concept of *launch screen*. It is an image that immediately appears on the screen and goes away when the application is ready. This is what we did for our Flutter demo:

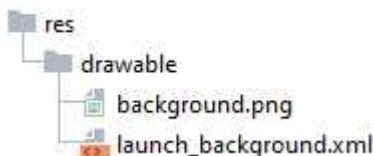


Figure 21.6: A custom Android splash screen setup.

Flutter creates a default launch screen at `res/drawable/launch_background.xml` with its logo at the center. We have added or own logo (`background.png`) and referenced it in the XML configuration file. This creates a static launch screen that automatically fades out when the app is fully loaded.

- By default, the R8 code shrinker is enabled for Android release builds. You could decide to disable with the `--no-shrink` flag in the build command, but we do not recommend doing so. Code shrinking makes the application as small as possible by removing unused resources and code.
- If you want to publish your application to the Play Store, you must give it a digital signature. To do so, make sure to follow the official Flutter documentation<sup>136</sup>. It has a detailed guide on how to create, install and manage a keystore file using the `keytool` Java tool.

Once you have reviewed everything, it's finally time to create the Android release executable. You can create an "app bundle" or an "apk". In any case, should remember to obfuscate the Dart code:

- **app bundle.** The Google Play Store prefers this format when publishing applications. An app bundle contains your Dart code and the Flutter runtime compiled for ARM 32-bit, ARM 64-bit, or x86 64-bit. Use this command to generate the bundle:

```
flutter build appbundle --obfuscate --split-debug-info=/path/to/folder
```

This command will generate an `.aab` file.

- **apk.** Although app bundles are preferred, you can also create an APK file. It is a large file that contains your code compiled for all target ABIs (ARM v7, ARM v8, and x86). For this reason, we recommend using the `--split-per-abi` flag to differentiate builds:

```
flutter build apk --obfuscate --split-debug-info=/path/to/folder  
--split-per-abi
```

This command will generate three different `.apk` files, one per architecture. In this way, you can distribute a specific APK to your users. Without splitting, you'll give your users a single, big APK file that also contains unused compiled code for each platform.

We recommend using `.apk` files for development because they're easy to distribute and install. To publish the application, use an `.aab` file instead.

---

<sup>136</sup> <https://docs.flutter.dev/deployment/android#signing-the-app>

### 21.3.2 macOS and iOS

Both macOS and iOS are very similar, so we can cover them together. When customizing the project configuration, open the `ios` or `macos` folders using XCode and make changes from there. Do not manually change values yourself on individual configuration files. Here's what we recommend to do before publishing the application:

- Changing the application icon is the only action you can safely do without XCode. For both iOS and macOS, navigate to `Runner/Assets.xcassets/AppIcon.appiconset` and replace all images with your custom ones. Make sure to keep the same file name and image size.
- All iOS applications submitted to the App Store must provide a *launch screen*. This page is temporarily visible while the application is loading and automatically fades out as soon as Flutter is ready. There already is a default launch screen, but you can customize it:

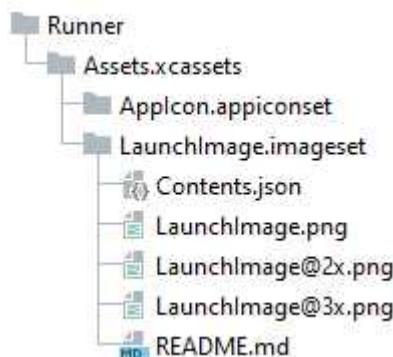


Figure 21.7: A custom iOS launch screen setup.

Again, just replace the PNG file contents without changing the file name and the image size. To further customize the launch screen configuration, open the `LaunchImage.imageset` folder with XCode. There is no launch screen for macOS projects.

- You can localize your application name for macOS and iOS builds with the same procedure. Use XCode to either open the project in the `macos` or `ios` folder and follow these steps:
  - 1 Select *Runner* from the left menu and then *Runner* again under the Project section.
  - 2 Scroll down until you see the *Localizations* group and add the languages you want. For example:

Localizations	
Localization	Resources
Base	2 Files Localized
English — Development Language	1 File Localized
Italian	3 Files Localized
+	-

Figure 21.8: Localizing iOS and macOS application name.

For example, we have added English and Italian to the Flutter demo application of the [equations](#) package.

- 3 Select *Runner* from the left menu and then *Runner* again in the subfolder. When you expand *Info*, you will see all the supported languages. In our case, we will have two entries: one for English and one for Italian:

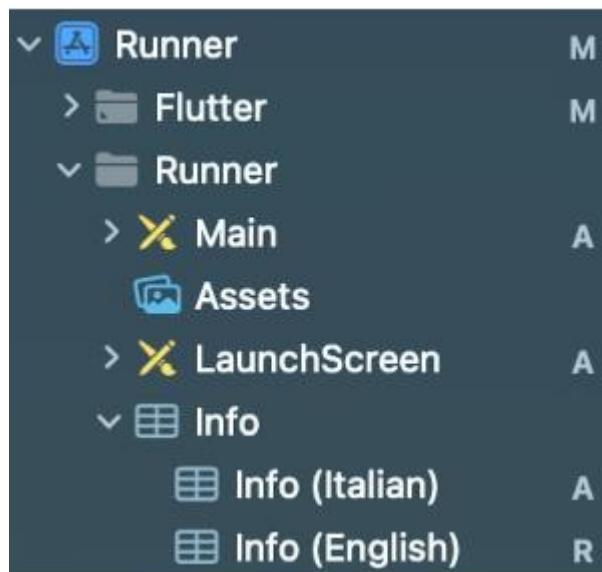


Figure 21.9: The two different localization strings configurations.

Click on each language variant and change the *Bundle display name* value.

The screenshot of *Figure 21.9* was taken from the iOS project but the procedure for macOS is the same.

- If you want to publish the application to the App Store, you must register it to the *App Store Connect* service. To do so, make sure to follow the official Flutter documentation for both iOS<sup>137</sup> and macOS<sup>138</sup>. It has a detailed guide on how to register a Bundle ID and create a new entry in the store.

Once you have reviewed everything, it's finally time to create the iOS or macOS release executables. In both cases, we recommend reducing the final code size and obfuscating the code to make it more difficult to reverse engineer:

- iOS: If you want to create a release build that can be published to the App Store, run this command:

```
flutter build ipa --obfuscate --split-debug-info=path/to/folder
```

It will generate an XCode build archive (`.xarchive` file) and an App Store bundle (`.ipa` file). If you just want to create a portable file for development purposes, you should also add the `--export-method` flag:

```
flutter build ipa --obfuscate --split-debug-info=path/to/folder  
--export-method development
```

Other available export methods<sup>139</sup> flags are `ad-hoc` and `enterprise`.

- macOS: If you want to create a release build that can be published to the App Store, run this command:

```
flutter build macos --obfuscate --split-debug-info=path/to/folder
```

It will generate a single executable file that you can double click and run immediately.

---

<sup>137</sup> <https://docs.flutter.dev/deployment/ios#register-your-app-on-app-store-connect>

<sup>138</sup> <https://docs.flutter.dev/deployment/macos#register-your-app-on-app-store-connect>

<sup>139</sup> <https://help.apple.com/xcode/mac/current/#/dev31de635e5>

### 21.3.3 Windows and Linux

There isn't much work to do before creating the final release build of a Windows application. The icon is stored as an `.ico` file and is located at `windows/runner/resources/app_icon`. Change the image as you wish but make sure to keep it 256x256. If you want to change the application title, open the `main.cpp` file and replace this string:

```
if (!window.CreateAndShow(L"My original application title", origin, size)) {  
    return EXIT_FAILURE;  
}
```

The `windows/runner/Runner.rc` file defines your application's metadata, such as copyright info, version, file description, and more. The generated Windows executable filename can be changed by modifying the binary name in `windows/CMakeLists.txt`. For example:

```
# Change this to change the on-disk name of your application.  
set(BINARY_NAME "equations_solver")
```

This is the value for the demo application of the `equations` package. With this change, Flutter will create a binary file called `equations_solver.exe`. To create the release build for a Windows platform, run this command:

```
flutter build windows --obfuscate --split-debug-info=path/to/folder
```

As always, we recommend obfuscating the Dart code to reduce the total file size and make reverse engineering harder. This command generates the executable file (in `.exe` format) and some other files inside `build/windows/runner/release`.

#### Note

The `.exe` file must be in the same location as the `flutter_windows.dll` library and the `data` folder. It is not possible to use the executable alone because it depends on the other resources to work.

Make sure to follow the official documentation to see how to prepare a Windows application for distribution<sup>140</sup> and how to distribute it to the Microsoft Store<sup>141</sup>. When it comes to Linux, there is

---

<sup>140</sup> <https://docs.flutter.dev/development/platform-integration/windows/building#distributing-windows-apps>

<sup>141</sup> <https://docs.flutter.dev/deployment/windows>

even less work to do on the Flutter side of the project. To change the application title, just open `linux/my_application.cc` and replace the default values:

```
if (use_header_bar) {
    GtkWidget* header_bar = GTK_HEADER_BAR(gtk_header_bar_new());
    gtk_widget_show(GTK_WIDGET(header_bar));
    gtk_header_bar_set_title(header_bar, "My application title"); // <- Here
    gtk_header_bar_set_show_close_button(header_bar, TRUE);
    gtk_window_set_titlebar(window, GTK_WIDGET(header_bar));
} else {
    gtk_window_set_title(window, "My application title"); // <- Here
}
```

As it happens on Windows, you can also change the binary name in `linux/CMakeLists.txt`. For example:

```
set(BINARY_NAME "equations_solver")
```

This is the value for the demo applications of the `equations` package. On a Linux project, there is no way to directly set the icon. It's configured when you publish the executable to the Snap Store. Make sure to follow the official documentation<sup>142</sup> to see how to release a Linux build to the Snap Store.

#### 21.3.4 Web

When you create a release build for a web application, the Dart code is compiled to JavaScript, minified, and tree-shaken. By default, Flutter already prepares a default project to run the project on a web browser, but there are two main things we recommend doing:

1. Update the icons inside `web/icons` with your own, making sure to not change the file name and the size. In the Flutter demo application of the `equations` package, we have added more icon variants for better compatibility and declared them in `manifest.json`. This JSON file also contains other important metadata about your project that should be updated.
2. A Flutter web application takes a few seconds to load the first time it's run. The application will load very quickly from the second time onward because the browser caches most of the contents. The problem is that, regardless of the startup time, the user will see a white page

---

<sup>142</sup> <https://docs.flutter.dev/deployment/linux>

until the application is ready. In the demo application of the `equations` package, we have added this loading placeholder inside the `index.html` file<sup>143</sup>:



Figure 21.10: The loading indicator that appears while the application is loading.

The animated blue arc rotates at a constant speed. This loading placeholder is immediately shown when the web page is opened and goes away when the application is loaded. To do this, we have added some HTML before Flutter's initialization code:

```
<!-- This custom HTML goes BEFORE the <script> tag -->

<div id="loading" class="status"></div>
<div class="loading_spinner"></div>

<script>
  window.addEventListener('load', function(ev) {
    <!-- Flutter initialization code -->
  });
</script>
```

Check our GitHub repository<sup>144</sup> to see how we animated the blue arc using CSS. All of this is optional, but we believe it's nice to have because it shows the user that the application is loading. A static, white page may give the idea that something crashed or the application is just slow. To enhance this experience even more, we have also added textual descriptions for each Flutter initialization phase:

---

<sup>143</sup> [https://github.com/albertodev01/equations/blob/master/example/flutter\\_example/web/index.html](https://github.com/albertodev01/equations/blob/master/example/flutter_example/web/index.html)

<sup>144</sup> [https://github.com/albertodev01/equations/tree/master/example/flutter\\_example/web](https://github.com/albertodev01/equations/tree/master/example/flutter_example/web)



Figure 21.11: The initialization flow of the Flutter web app.

Besides showing the colored logo and the rotating arc, we also update the text at the center with the current initialization status. This is very easy to do because you just need to add a few lines to the existing Flutter initialization code, which is automatically created when you generate the project:

```
<script>
window.addEventListener('load', function(ev) {
  var loading = document.querySelector('#loading'); // Add this
  loading.textContent = 'Loading entrypoint...'; // Add this

  _flutter.loader.loadEntrypoint({
    serviceWorker: {
      serviceWorkerVersion: serviceWorkerVersion,
    },
    onEntrypointLoaded: async function(engineInitializer) { // Add this
      loading.textContent = 'Initializing Flutter...';
      let appRunner = await engineInitializer.initializeEngine();

      loading.textContent = "We're almost there...";
      await appRunner runApp();
    }
  });
});
</script>
```

All of this code is still inside the `index.html` file. The text in the `<div>` element dynamically changes to reflect the current engine initialization status. This is a simple example of how you can manage the initialization of a Flutter web application. If you are a JavaScript ninja, you can invent more complex startup animations and effects.

Once you've set up the loading screen and updated the icons, it's time to create the release build for deployment. To do so, run this command:

```
flutter build web
```

It will create an [index.html](#), some JavaScript files (generated from the Dart code), and a series of static assets in the [build/web](#) folder. You can publish these contents to your server or rely on ready-to-use solutions like Firebase. Some environments such as "GitHub Pages" require the application to disable dynamic code generation for safety reasons. In such cases, build the application with the `--csp` flag:

```
flutter build web --csp
```

In short, CSP (Content Security Policy) is a safety measure to lock down the application in various ways, mitigating the risk of content injection vulnerabilities such as cross-site scripting.

### Note

Flutter web applications can be embedded in an [`<iframe>`](#) HTML tag, as you would do with any other content. This wasn't possible in Flutter 1.x versions and lower.

Starting from release 1.20 onwards, Flutter web project support core features for installable and offline-capable PWA applications. A Progressive Web App (PWA) is an application that runs on the browser but it's downloaded as if it was a desktop executable. Flutter automatically creates a [manifest.json](#) file that makes your web project also a PWA.

#### 21.3.4.1 Web renderers

When running applications for the web, you can use two different renderers:

1. HTML renderer. It uses HTML, CSS, JavaScript and SVG elements to run the application on a web browser. This renderer has a small download size.
2. CanvasKit renderer. It's a WASM module that uses Skia to draw contents on the canvas. This renderer is entirely consistent with mobile and desktop versions, has faster performance and higher widget density. However, this renderer has a large download size.

The [flutter build web](#) command automatically uses both renderers, according to the platform. In particular, the HTML renderer is used in mobile browsers and the CanvasKit renderer is used on

desktop browsers. This choice is made for a matter of performance, but you can decide to always use a single renderer for your application. For example:

- `flutter build web --web-renderer html`: always uses the HTML renderer. Use this option if you optimize download size over performance on desktop and mobile browsers.
- `flutter build web --web-renderer canvaskit`: always uses the CanvasKit renderer. Use this option if you are prioritizing performance and pixel-perfect consistency on both desktop and mobile browsers.

In most cases, the default option is the best choice because it dynamically chooses the best renderer based on the platform. It also reduces the download size on mobile and optimizes performance on desktop browsers.

#### 21.3.4.2 Image limitations on the web

The `Image` widget is well-integrated into Flutter web applications, but it has some limitations when compared to mobile and desktop platforms. The main reason is that web browsers are built to run untrusted code safely. The HTML renderer, which shows images using the `<img>` element, has the following limitations:

- No support for `ImageShader`.
- A tiny set of shaders can be applied to images.
- No way to control the image memory (the browser separately manages it).
- Limited support for byte conversions.
- No access to frame data on animated images.

The CanvasKit renderer instead requires access to image pixels to fully implement the "Flutter image API. This is subject to CORS policies and might cause problems.

#### Note

CORS is a mechanism used by browsers to control how a website accesses resources coming from other websites. It prevents scripts on another site from acting on behalf of the user and from gaining access to another site's resources without permission.

When the `<img>`, `<canvas>`, or `<picture>` elements are used, the browser automatically blocks access to pixels when it knows that the image is coming from an external website and the CORS

policy disallows fetching those data. WebGL requires access to the image data, and so there must be a compliant CORS policy. Here are a few solutions:

- If you're loading images from your assets (defined in the pubspec file), you will never have CORS issues. The reason is that the images and the application are located in the same place. You can use all `Image` constructors and they work seamlessly with both CanvasKit and HTML renderers.
- If your application is hosted on the same server as the images, you can load them without CORS problems. This is because the Flutter application and the image files are located in the same place (in technical terms, they have the "*same-origin*").
- If you're trying to load an image that is hosted on another server, then you need to either use a CDN (popular ones are Firebase and CloudFlare) or setup a CORS proxy (popular ones are Firebase Functions or CloudFlare Workers).

When possible, try to only load images from your assets or from the same server so that you won't have external tools or services to maintain.

## Deep dive: Embedding Flutter in HTML pages



[https://github.com/albertodev01/flutter\\_book\\_examples/tree/chapter\\_21/html\\_embedding](https://github.com/albertodev01/flutter_book_examples/tree/chapter_21/html_embedding)

By default, a Flutter application that runs on a web browser occupies the entire page. From version 3.10 onwards, you can integrate your Flutter content like any other CSS element on the page. All of this is possible because Dart code can be compiled into JavaScript.

### Note

This section assumes that you have basic HTML, CSS and JavaScript knowledges.

We are going to build a counter application that is hosted in a custom HTML element and interacts with external JavaScript code. We encourage you to clone the project from our GitHub repository and try to add more interactions between Flutter and the web browser.

1. Create a new application and put this code inside the `main.dart` file.

```

void main() => runApp(const CounterApp());

class CounterApp extends StatelessWidget {
  const CounterApp({super.key});

  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      home: Scaffold(
        body: Center(
          child: CounterBody(),
        ),
      ),
    );
  }
}

```

We will see how the `CounterBody` widget is made later. For now, it's enough to know that it only has a button that increases a counter.

2. Open the `index.html` file in the `web/` folder and add a `<div>` element inside the body. You must give it a unique id. For example:

```
<div id="app_container"></div>
```

The Flutter application will be run inside this `<div>` element rather than covering the entire screen. Now add some CSS rules inside the `<head>` element to size the container and place it in the middle of the screen:

```

<style>
  body {
    display: flex;
    align-items: center;
    justify-content: center;
    /* other properties... */
  }

  #app_container {
    width: 500px;
    height: 250px;
    margin: 20px;
    border: 3px solid #0099CC;
    box-shadow: 0px 8px 24px rgba(0, 0, 0, .25);
  }
</style>

```

We are putting everything inside the `index.html` file to keep the example simple.

3. Inside the `index.html` file, you will find some JavaScript code at the bottom of the `<body>` element. Change the existing entry-point loading code with this new version:

```
onEntryPointLoaded: async function (engineInitializer) {  
  let appRunner = await engineInitializer.initializeEngine({  
    hostElement: document.querySelector("#app_container"),  
  });  
  await appRunner.runApp();  
},
```

The crucial part is where we set `hostElement` to point to the element we previously created. In this way, the application will be displayed inside the `<div>` rather than covering the whole screen.

Now that everything is set, you can run the Flutter application on a web browser. The application works as usual but it's located in `<div>` element, which also has some CSS rules. This is the result on Google Chrome:

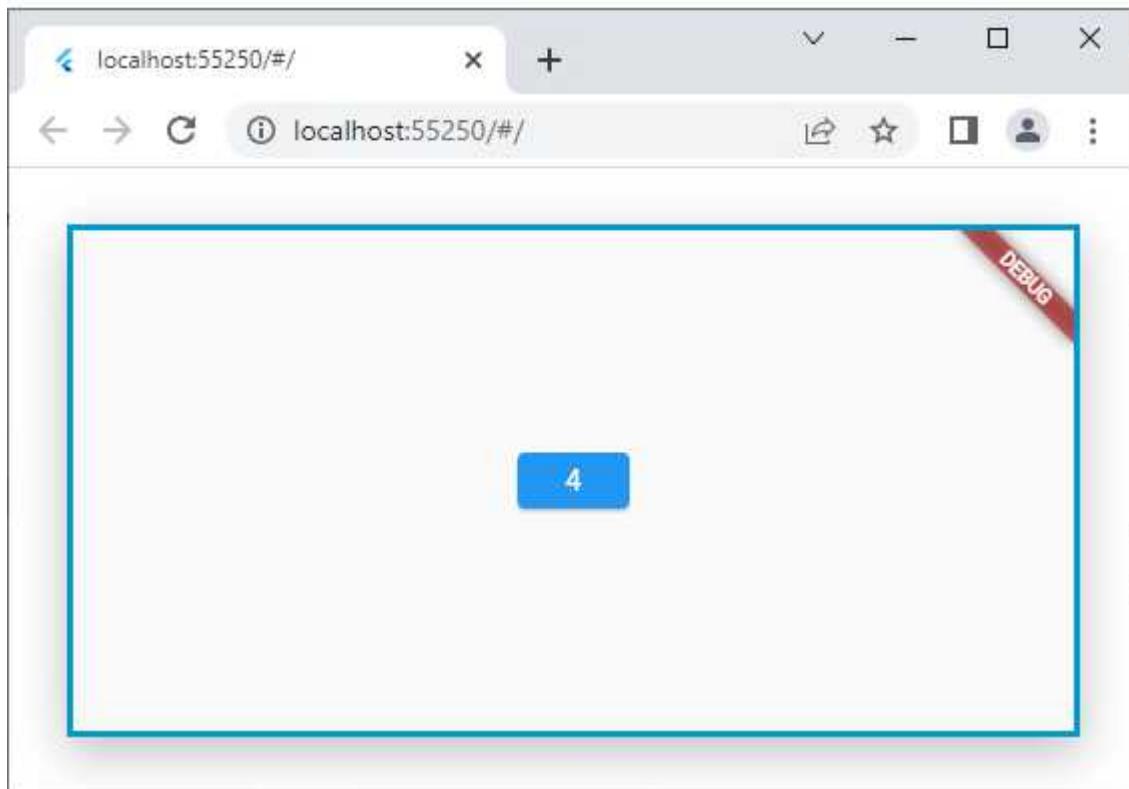


Figure 21.12: A Flutter application that is rendered inside a custom HTML element.

There also is the possibility to increase the counter from outside the Flutter application. To do this, start by adding a dependency to the `js` package in the pubspec file. Then, modify the state class of the `CounterBody` stateful widget to look like this:

```
@JSExport() // Notice this
class _CounterBodyState extends State<CounterBody> {
    int _counter = 0;

    @JSExport() // Notice this
    void increment() {
        setState(() {
            _counter++;
        });
    }

    @override
    void initState() {
        super.initState();
        final export = js_util.createDartExport(this);

        js_util.setProperty(js_util.globalThis, '_counterState', export);
        js_util.callMethod<void>(js_util.globalThis, '_countChanger', []);
    }

    @override
    Widget build(BuildContext context) {
        return ElevatedButton(
            onPressed: increment,
            child: Text('${_counter}'),
        );
    }
}
```

The `JSExport` annotation allows a Dart class to interact with JavaScript code. We have to add this annotation to the state class and the increment function to make them “exportable”. There are some more things to configure inside the `initState` method:

- The `createDartExport` function is used to link Dart and JavaScript objects. In simple terms, this function looks for exportable Dart objects (those with the `JSExport` annotation) and creates associated JavaScript objects. In this way, you can use a Dart object from JavaScript.
- The `setProperty` method creates a JavaScript property that interacts with the state class of the `CounterBody` widget. The `callMethod` method creates a JavaScript function that we can use to invoke the `increment()` function.

Create an HTML button inside the `<body>` element, which will be used to increase the counter in the Flutter application. Make sure to give it a unique id:

```
<input id="increment" value="Increment" type="button" />
```

Add the following code inside the a `<script>` element to consume Dart objects from JavaScript:

```
<script>
(function () {
  window._countChanger = function () {
    let incrementButton = document.querySelector("#increment");
    incrementButton.addEventListener("click", (event) => {
      window._counterState.increment();
    });
  };
}());
</script>
```

The `_countChanger` function was defined on the Dart side using `callMethod`. The `_counterState` property represents the state class of the `CounterBody` widget and has the `increment()` function, which calls the exported `increment()` method from Dart. This is the final result:



Figure 21.13: An HTML button that uses JavaScript to increase the counter on the Flutter application.

If you press the HTML button, you will see that the counter in the Flutter application increases. The most exciting part of element embedding is that your Flutter application can directly interact with

JavaScript and the DOM. We have even applied CSS styles around the application itself. You don't need an `<iframe>` element to integrate the application with the rest of the website.

### Note

We are not covering the `js` package and JavaScript interoperability in detail because this topic is out of the scope of this book. We want to keep the focus on the Dart code and the Flutter framework.

If you want to know more about the JavaScript interoperability, check the official documentation<sup>145</sup> of the Dart website.

---

<sup>145</sup> <https://dart.dev/web/js-interop>

# 22 – HTTP servers and low-level HTML

---

## 22.1 Creating HTTP servers with dart:io

The `HttpServer` and `HttpClient` classes, defined in the `dart:io` library, are used for low-level HTTP functionalities. However, instead of using these two classes, the Dart team recommends to use other more developer-friendly and composable packages. In particular:

- To create HTTP clients, prefer using Dart's official `http` package instead of `HttpClient`. We have covered the `http` package in *Deep Dive: The http package*.
- To create HTTP servers, prefer using Dart's official `shelf` package instead of `HttpServer`. We will cover the `shelf` package later in *Section 2 – “Creating HTTP servers with the shelf package”* of this chapter.
- The `shelf` package is built on top of `HttpServer` and the `http` package is built on top of `HttpClient`. For this reason, don't think that one is better than the other. These packages basically wrap low-level classes from the I/O library with more developer-friendly APIs.

The `HttpServer` class is used to deliver responses, such as static web pages or dynamic content, using the HTTP protocol. The `HttpServer` class is a `Stream<HttpRequest>` that can be listened to with the classic techniques. For example, this is the most basic example we can think of:

```
Future<void> main() async {
  // servers should run in a guarded zone to handle async errors
  runZonedGuarded(
    () {
      final server = await HttpServer.bind('localhost', 80);
      server.listen((HttpRequest request) async {
        request.response.write('Hello client!');
        await request.response.close(); // required to send the response
      });
    },
    (error, stack) => debugPrint('Error = $error'),
  );
}
```

Any client interacting with the server creates an `HttpRequest` event, which holds data about the incoming request. Request objects always carry an `HttpReponse` object, which is used by the server to send a response to the client. The `HttpServer` object should live inside a guarded zone to avoid

that unexpected errors shut down the server. See *Deep dive – “Zones”* for more information on the zone API. Instead of giving the server a hard-coded address (as we did in the example), you should use the `InternetAddress` helper class. It has a variety of static members that represent different kinds of addresses:

- `loopbackIPv4`: assigns an IP version 4 loopback address to a server. This object maps to the `127.0.0.1` (`/localhost`) address:

```
await HttpServer.bind(InternetAddress.loopbackIPv4, 80) // 127.0.0.1
```

- `loopbackIPv6`: assigns an IP version 6 loopback address to a server. This object maps to the `::1` (`/localhost`) address:

```
await HttpServer.bind(InternetAddress.loopbackIPv6, 80) // ::1
```

- `anyIPv4` and `anyIPv6`: both assign the “any IP” address to a server, either with IPv4 or IPv6 respectively:

```
await HttpServer.bind(InternetAddress.loopbackIPv4, 80) // 0.0.0.0
await HttpServer.bind(InternetAddress.loopbackIPv6, 80) // ::
```

If the port is already in use, you’ll get a `SocketException` exception, and the server will not start. To create an HTTPS server, use `bindSecure` instead of `bind` and provide a `SecurityContext` with key and chain file paths. Remember to run this code with `runZoneGuarded`:

```
var chain = Platform.script.resolve('path/to/chain.pem').toFilePath();
var key = Platform.script.resolve('path/to/key.pem').toFilePath();

final context = SecurityContext()
  ..useCertificateChain(chain)
  ..usePrivateKey(key);

final server = await HttpServer.bindSecure(
  InternetAddress.anyIPv4,
  443,
  context,
);
server.listen((HttpRequest request) async {
  request.response.write('Hello client!');
  await request.response.close();
});
```

The `Platform.script` getter is used to obtain the URI of the current Dart program. Alternatively, you could have hard-coded the file paths, but we recommend avoiding it. The `SecurityContext` object holds the certificates to trust when making a secure connection.

### Note

Certificates and keys for HTTPS connections can be consumed by `SecurityContext` from either “PEM” or “PKCS12” containers. These files can be created, for example, with tools from the OpenSSL project.

You may also want to redirect all HTTP requests to HTTPS. You can directly do this in Dart by creating an HTTP server that permanently redirects all requests to the HTTPS one. This is an example of how you could do it (all of this is executed with `runZoneGuarded`):

```
final context = SecurityContext()
  ..useCertificateChain(path)
  ..usePrivateKey(path);

// This HTTPS server handles all incoming requests
final https = await HttpServer.bindSecure(address, 443, securityContext);
https.listen((request) async {
  request.response.write('Hello secure client!');
  await request.response.close();
});

print('HTTPS server started...');

// This HTTP server always redirects ALL requests to the HTTPS instance
final http = await HttpServer.bind(address, 80);
http.listen((request) async {
  await request.response.redirect(
    request.uri.replace(scheme: 'https'),
    status: HttpStatus.movedPermanently;
);
});

print('HTTP server started...');
```

The `uri.replace` method creates a copy of the URI object and only changes some parts (if any). In our case, we just want to change the scheme from HTTP to HTTPS and keep everything else as it is. The HTTP server permanently redirects all requests to the HTTPS server. The `redirect` method calls

`close()` internally. Let's move on to the next section to learn more about the `HttpRequest` and `HttpResponse` classes.

### 22.1.1 Handling requests and responses

Each request received by an `HttpServer` generates a new `HttpRequest` object and pushes it into the stream. An `HttpRequest` object is used to carry much information about the request itself, such as headers, body, method, and URI. For example:

```
final server = await HttpServer.bind(InternetAddress.LoopbackIPv4, 80);

server.listen((request) async {
  if (request.method == 'GET') {
    await doSomethingWithGet(request);
  } else {
    request.response.write("Only 'GET' requests are supported.");
    await request.response.close();
  }
});
```

In this example, we're using the `method` property to only handle GET requests and reject the others. Request objects carry an `HttpResponse` object, which is used to send a response back to the client. Unless the documentation says otherwise, you always have to call `close()` on a response object.

#### Note

For example, the documentation of the `response.redirect(Uri)` method says that it internally calls `close()`. At the time of writing this book, the `redirect` case is the only one where you don't need to manually close the response.

An `HttpRequest` object holds the request headers and provides the request body (if it exists). All incomplete requests (where the header is partially or fully missing) are ignored, and no exceptions are thrown. Headers are wrapped in an `HttpHeaders` object:

```
final HttpHeaders headers = request.headers;
print(headers.host);
print(headers.contentType);
print(headers.contentLength);
print(headers.date);
print(headers.expires);
print(headers.value('some-header-key'));
```

If you had to create custom headers for a response object, you would have to do it before any `write` call. In other words, headers can only be generated while the body is still empty. For example:

```
server.listen((request) async {
  request.response.headers.set(
    HttpHeaders.contentTypeHeader, // 'content-type'
    '${ContentType.html}', // 'text/html; charset=utf-8'
  );

  switch (request.uri.path) {
    case '/':
      final index = await File('source/static/index.html').readAsString();
      request.response.write(index);
      break;
    case '/info':
      final info = await File('source/static/info.html').readAsString();
      request.response.write(info);
      break;
    default:
      final error = await File('source/static/error.html').readAsString();
      request.response.write(error);
  }

  await request.response.close();
});
```

This code reads HTML files from the local filesystem and serves them with the proper content-type header. The `ContentType` class has convenient static properties, such as `html`, that automatically configure the header values. We could have manually added header values in other ways, such as:

1. With a hard-coded string:

```
request.response.headers.set(
  HttpHeaders.contentTypeHeader,
  'text/html; charset=utf-8',
);
```

2. With a `ContentType` object:

```
final contentType = ContentType("text", "html", charset: "utf-8");

request.response.headers.set(
  HttpHeaders.contentTypeHeader,
  contentType,
);
```

In our example, it is better to use `ContentType.html` because it avoids hard-coding content type values. When `ContentType` does not have a static object for the type you are looking for, create a new one. In this case, remember to always specify the charset (which defaults to Latin-1 if you forgot to configure it).

As we have already said before, headers cannot be modified after the body is written. For example, this code would throw an exception:

```
switch (request.uri.path) {
  case '/':
    final index = await File('index.html').readAsString();
    request.response.write(index);
  case '/info':
    final info = await File('info.html').readAsString();
    request.response.write(info);
  default:
    final error = await File('error.html').readAsString();
    request.response.write(error);
}

// Wrong!! Headers cannot be set after 'write'
request.response.headers.set(
  HttpHeaders.contentTypeHeader,
  '${ContentType.html}',
);
```

The code throws an exception because there is a `write` call before `headers.set`. A general good practice (that also applies beyond the scope of Dart servers) is to set headers as soon as possible, before any content is written to the response body. Before moving on, here are a few suggestions:

- When working with query parameters or the address itself, prefer using the `Uri` class rather than manipulating the string yourself. For example:

```
const testUri = 'https://website.com?key=value&key2=value2';

final uri = Uri.parse(testUri);
print(uri.scheme); // https
print(uri.host); // website.com
print(uri.queryParameters); // {'key': 'value', 'key2': 'value2'}
```

The `Uri` object takes care of parsing the address and gives you easy access to all parts. This is much easier than manually extracting data from the string.

- You may need to create flavors of an application for each production stage. For example, the same application should start with different server configurations in development and production modes. This is a possible approach to the problem:

```
Future<void> main(List<String> args) async {
  if (args.first == 'dev') {
    // start an HTTP server for development
  } else if (args.first == 'prod') {
    // start an HTTPS server for production
  } else {
    // No recognized flavors
    print('Invalid command!');
  }
}
```

This example is extremely simplified, but you get the idea. Using command line arguments, you can pass a flavor to the executable when it's run:

```
dart run my_server.dart dev
dart run my_server.dart prod
```

Alternatively, you could define the flavor using environment variables (more info in *chapter 2 – Section 2.6 “Dart’s environment variables”*).

- Prefer using [ContentType](#) and [HttpHeader](#) utility classes rather than hard-coding values.

The `bin` folder should only contain the server initialization part of your application. The routing and all the other backend logic should live in the `lib` folder.

### 22.1.2 Unit testing HttpServer using HttpClient

To test an `HttpServer` implementation, we recommend to use the `HttpClient` class to simulate actual clients that send requests to the server. Mocking a server instance is generally not easy and often time-consuming.

#### Note

To make testing even easier, you could use the `http` package instead of `HttpClient`. However, in this section we don't want to consume external packages. We want to show how to use Dart's core API objects to create and test server-side projects.

The server implementation is often split across multiple files and managed by a class that handles initialization, routing, session management (if any), and much more. However, to keep the example easy, we place everything in a single function:

```
Future<HttpServer> createServer() async {
    final server = await HttpServer.bind(InternetAddress.loopbackIPv4, 80);

    server.listen((HttpRequest request) async {
        request.response.write('Hello client!');
        await request.response.close();
    });

    return server;
}
```

We need the server instance to stay “online” for the whole test duration to exercise this code. For this purpose, we can use `setUpAll` to start the server before running all tests and `tearDownAll` to stop the server after all tests are completed. For example:

```
group('HttpServer tests', () {
    late HttpServer server;
    final endpoint = InternetAddress.loopbackIPv4;

    setUpAll(() async {
        server = await createServer();
    });

    tearDownAll(() async {
        await server.close();
    });

    test('Testing the base address', () async {
        final client = HttpClient();

        try {
            final request = await client.get(endpoint.host, 80, '/');
            final response = await request.close();
            expect(response.statusCode, equals(200));
        } finally {
            client.close();
        }
    });
});
```

With this approach, the server instance is created only once and closed when all tests are run. All the logic is wrapped by a `try-finally` block because an `HttpClient` must always be closed when

not needed anymore, even if an exception occurred. Requests complete when you call `close()`, which returns the `HttpResponse` object generated from the server. Of course, a client can make any HTTP request, such as:

- `client.get`, for HTTP GET requests;
- `client.post`, for HTTP POST requests;
- `client.put`, for HTTP PUT requests;
- `client.patch`, for HTTP PATCH requests;
- `client.delete`, for HTTP DELETE requests;
- `client.head`, for HTTP HEAD requests.

If your tests are all independent and don't modify the configuration of the server object, you could create a single client. It will have the same lifetime as the server, so it will be created in `setUpAll` and closed in `tearDownAll` as well. For example:

```
group('HttpServer tests', () {
    late HttpServer server;
    late HttpClient client;
    final endpoint = InternetAddress.loopbackIPv4;

    setUpAll(() async {
        server = await createServer();
        client = HttpClient();
    });

    tearDownAll(() async {
        client.close();
        await server.close();
    });

    test('Testing the base address', () async {
        final request = await client.get(endpoint.host, 80, '/');
        final response = await request.close();

        expect(response.statusCode, equals(200));
    });
});
```

This is efficient because it creates the server and the client only once, but it's not always the right choice. For example, if you made a POST or a DELETE request that changed the server configuration, other tests will also run against that change (which may not always be desirable). In other words, pay attention to side effects!

## 22.2 Creating HTTP servers with the `shelf` package

The `shelf` package is created and maintained by the Dart team to give developers a composable and easy-to-use package for backend development. There even is a dedicated template for the `dart create` command to generate a starter application that uses `shelf`:

```
dart create -t server-shelf my_shelf_project_name
```

Since `shelf` uses `HttpServer` internally, you may already be familiar with some APIs. For example, when you start a new server, the `serve` method returns the usual `HttpServer` instance from the `dart:io` library. For example:

```
import 'dart:io';
import 'package:shelf/shelf.dart';
import 'package:shelf/shelf_io.dart' as shelf_io;

void main(List<String> args) async {
    // The 'HttpServer' class is from 'dart:io'
    final HttpServer server = await shelf_io.serve(
        // 'Request' and 'Response' are from 'shelf'
        (Request request) async => Response.ok('Hello client'),
        InternetAddress.LoopbackIPv4,
        80,
    );
    print('Server started at ${server.address}');
}
```

The `serve` method already uses `runZoneGuarded`, so you don't have to remember to use it. The `serve` method asks for a callback function (which defines the server behavior), the address, and the port. To use the HTTPS protocol, create a `SecurityContext` object and pass it to `serve`:

```
final securityContext = SecurityContext()
    ..useCertificateChain('path/to/chain.pem')
    ..usePrivateKey('path/to/key.pem');

final HttpServer server = await shelf_io.serve(
    (Request request) async => Response.ok('Hello client'),
    InternetAddress.LoopbackIPv4,
    80,
    securityContext: securityContext,
);
```

This is the same process we had seen in *section 1 – “Creating HTTP servers with `dart:io`”* when we created a secure server using `bindSecure`. So far, it may seem that `shelf` is a mere wrapper of the

[HttpServer](#) class but in the next sections you'll see its strength points: **handlers**, **middlewares**, and the **router**.

### 22.2.1 Handlers and middleware

A **handler** is a function that takes a [Request](#) parameter and returns a [Response](#) object. A handler can either manage the request and return a response or forward the processing to another handler. When a new server is created with [serve](#), the first parameter is a handler:

```
await shelf_io.serve(  
    (request) async => Response.ok('Hello client'), // Handler  
    InternetAddress.LoopbackIPv4, 80,  
);
```

A handler that does some processing and then forwards the request to another handler without returning a response is called **middleware**. The Shelf package generally comprises many layers of middleware with one or multiple handlers at the core. Let's see two examples of how handlers and middleware can be used:

- In the simplest example, we can directly pass a handler to the server as we've seen in the previous example. The [Request](#) object contains various data about the incoming request, such as the HTTP method, the URI, headers, the protocol version, and more. For example:

```
// Remember that 'serve' executes its code in a guarded zone so you are  
// "safe" from asynchronous errors  
await shelf_io.serve((request) async {  
    if (request.method == 'GET') {  
        return Response.ok('Hello!');  
    } else {  
        return Response.badRequest('Only "GET" requests are allowed.');  
    }  
}, InternetAddress.LoopbackIPv4, 80);
```

All handlers **always** return a [Response](#) object. If you looked at the package internals, you'd see that a handler is defined as such:

```
typedef Handler = FutureOr<Response> Function(Request request);
```

The [Response](#) object has various named constructors (such as [ok](#) or [badRequest](#)) that build an HTTP response for the client. For example, the [Response.ok](#) constructor creates a "200 OK" response to indicate that the request has succeeded. Of course, you can use the default [Response](#) constructor to build your own response.

- Let's see a more complicated case where we use middleware to process a request before sending a response. For each request, we log some data into the console before serving the response and then send a usage report to an external service. This is a possible way of doing it:

```
// This function is a middleware
Handler logMessages(Handler innerHandler) {
    // This is a handler because it takes a request and returns a response
    return (request) async {
        final response = await innerHandler(request);

        print('request method = ${request.method}');
        print('response code = ${response.statusCode}');
        print('response headers = ${response.headers}');

        return response;
    };
}

// This function is a middleware
Handler sendUsageAnalytics(Handler innerHandler) {
    // This is a handler because it takes a request and returns a response
    return (request) async {
        const analytics = MyAnalyticsManager();
        await analytics.sendUsage(
            method: request.method,
            uri: request.url,
        );

        // In this example, 'innerHandler' refers to 'LogMessages'
        return innerHandler(request);
    };
}

void main(List<String> args) async {
    final handler = const Pipeline()
        .addMiddleware(logMessages)
        .addMiddleware(sendUsageAnalytics)
        .addHandler((request) async => Response.ok('Hello'));

    await shelf_io.serve(handler, InternetAddress.LoopbackIPv4, 80);
}
```

A middleware is a function that takes a handler, makes some processing, and then returns a response or another handler. The `Pipeline` object makes it easy to chain middleware in

order and thus creates a composable set of layers. In our example, this is what happens each time a request is sent to the server:

- a. The `logMessages` middleware is called, meaning that some information is printed to the console.
- b. The pipeline moves to the `sendUsageAnalytics` middleware, which is executed as soon as the previous one terminated.
- c. At the end of the chain there must always be an `addHandler` call, whose purpose is to send to the client a response.

As the name suggests, the `Pipeline` object creates an ordered sequence of actions that the server has to perform for each incoming request. You can add as many middleware you want but there must always be a final `addHandler` call that returns a response to the client.

- In the previous example, we created two middleware functions. Even if the code works, the implementation is not solid (errors are not handled, and the logic should be wrapped within a `Future.sync()` object). This is a more robust way to create a middleware, which also is considered good practice:

```
final messageLogger = createMiddleware(  
    requestHandler: (request) {  
        print('request method = ${request.method}');  
    },  
    responseHandler: (response) {  
        print('response code = ${response.statusCode}');  
        print('response headers = ${response.headers}');  
        return response;  
    },  
    errorHandler: (error, stackTrace) {  
        print('Error: $error\nStack trace = $stackTrace');  
    },  
);  
  
final analytics = createMiddleware(  
    requestHandler: (request) async {  
        const analytics = SomeAnalyticsManager();  
        await analytics.sendUsage(  
            method: request.method,  
            uri: request.url,  
        );  
    }  
);
```

The `shelf` package defines the `createMiddleware` function that, as the name suggests, helps you create `Middleware` functions. It handles errors in the right way, and it efficiently passes data from one handler to another:

```
final messageLogger = createMiddleware(  
    // code...  
);  
  
final analytics = createMiddleware(  
    // code...  
);  
  
final serverHandler = const Pipeline()  
    .addMiddleware(messageLogger)  
    .addMiddleware(analytics)  
    .addHandler((request) async => Response.ok('Hello'));
```

We recommend to use the `createMiddleware` function instead of creating middleware by hand, which is more error-prone and makes the code more verbose.

Note that you could use a `Pipeline` object with no middleware and only add a handler. However, in this case it would be better if you directly passed the handler inside `serve`. A pipeline is only valuable when you have one or more middleware.

## 22.2.2 The `shelf_router` and `shelf_static` packages

The Dart team created two interesting packages to help you route requests and serve static content such as HTML pages. The `Router` class is used to route requests to handlers based on the HTTP verb and pattern. For example:

```
void main(List<String> args) async {  
    final router = Router()  
        ..get('/', (request) => Response.ok('Home page'))  
        ..post('/info', (request) {  
            return Response.ok(  
                "{'content': 'Info page'}",  
                headers: {  
                    HttpHeaders.contentTypeHeader: ContentType.json,  
                },  
            );  
        });  
  
    await shelf_io.serve(router, InternetAddress.loopbackIPv4, 80);  
}
```

The router we created in the example handles a GET and a POST request. If multiple routes match the same request, the handler of the first route is called. In case no route matches a request, the router returns a `Response.notFound` by default. Here is how to customize the not found response:

```
final Router router = Router()
notFoundHandler: (request) {
    // Custom '404 not found' response
    return Response.notFound('Whoops! 404 :(');
},  
);
```

A very interesting feature of `Router` is the possibility of parsing patterns from routes and matching them in the handler. For example, imagine you wanted to parse an integer from the URL and use it to generate a random value. Thanks to the diamond syntax, you can define parameters that can be passed to the handler:

```
final Router router = Router()
..get('/random/<maxRandomValue>', (Request request) async {
    // The diamonds ('<' and '>') define parameters that can be parsed by
    // the handler.
    final maxRandom = int.tryParse(request.params['maxRandomValue'] ?? '');
    if (maxRandom != null) {
        final value = Random().nextInt(maxRandom) + 1;
        return Response.ok('Random between 1 and $maxRandom: $value');
    } else {
        return Response.badRequest(
            body: "Couldn't parse the number",
        );
    }
});
```

In this example, `<maxRandomValue>` is a “route parameter” because it’s enclosed by diamonds. The parameter is used to generate a random number and, in case the value wasn’t valid, the server would return a *400 Bad Request* error:



Figure 23.1: Using parameters in a `Router` route.

A route can have one or more parameters and even regular expressions. For example, these are two equivalent requests that accept numbers, but the second one uses a special syntax to allow the evaluation of regular expressions:

```
final router = Router()
..get('/users/<userId>/questions/<id>', (Request request) async {
    final result = await myDataSource.getQuestion(
        request.params['userId'],
        request.params['id'],
    );
    return Response.ok('$result');
})
..get(r'/users/<userId>/answers/<id|\d+>', (Request request) async {
    final result = await myDataSource.getAnswer(
        request.params['userId'],
        request.params['id'],
    );
    return Response.ok('$result');
});
```

In the second route definition, notice `<id|\d+>` syntax: it has the parameter name on the left, a pipe (`|`) as a separator, and then the regular expression (without `^` and `$`). To be more precise:

- `id`: it's the parameter name, which is referenced in the handler with `params['id']`;
- `|`: it separates the name from the regular expression;
- `\d+`: it's a regular expression that matches one or more digits in a string.

You have seen that `Router` is great for dynamic content generation but can also serve static files. For example, if you had a static website that only used HTML and CSS, you could load files from the local filesystem. While it's certainly doable, it's a bit complicated because you have to take care of a few things altogether:

1. correct MIME types handling;
2. cross-platform file paths manipulation, which is usually done with the `path` package;
3. create a robust and secure implementation that doesn't give access to undesired locations on the filesystem to your users;
4. manage symbolic links;
5. provide proper response headers, such as `content-length` or `content-type`.

These things, plus many more, must be considered when serving static files from your filesystem. It is quite hard and time-consuming to implement everything on your own. To easily serve static files from your filesystem, use the `shelf_static` package:

```

void main() async {
  final htmlWebsite = createStaticHandler(
    'my_website/',
    defaultDocument: 'index.html',
  );

  await shelf_io.serve(htmlWebsite, InternetAddress.LoopbackIPv4, 80);
}

```

The Dart team created the `shelf_static` package to provide a robust and secure handler for static files. In the example, we have created a simple server that serves all files inside the `my_website/` folder. You can even combine a static handler with a `Router`:

```

void main() async {
  final pdfStaticHandler = createStaticHandler(
    'documents/path',
  );
  final router = Router()
    ..get('/', (_) => Response.ok('Hello!'))
    ..get('/pdf', pdfStaticHandler);

  await shelf_io.serve(router, InternetAddress.LoopbackIPv4, 80);
}

```

With this implementation, if you had a file located at `documents/path/other/file.pdf` in your filesystem, it will be reachable from the server at `/pdf/other/file.pdf`. This package is excellent because it handles everything for you and has a minimal configuration.

### 22.2.3 Good practices

We have seen that the `dart:io` library provides a powerful but low-level solution to create HTTP servers with `HttpServer`. However, for backend solutions, the Dart team recommends using the `shelf` package family. Here are a few advice:

- Follow Dart's team recommendation: use `shelf` to create HTTP servers. Doing everything on your own is challenging, error-prone, and requires more maintenance time. If you use `shelf` instead, most of the complexity is internally handled and you don't have to reinvent the wheel. Just use its composable and flexible API without worrying about low-level details of the HTTP protocol.
- Using HTTPS is basically a non-written requirement for any website. You can easily route all HTTP requests to HTTPS by spawning two server instances: one that listens on port 80 and

the other on 443. The HTTP server will forward all requests to the secure address, while the HTTPS one will serve contents to the clients. For example:

```
final context = SecurityContext()
    ..useCertificateChain(path)
    ..usePrivateKey(path);

// Redirects all requests to https
Future<Response> httpServer(Request request) async {
    return Response.movedPermanently(
        request.url.replace(scheme: 'https'),
    );
}

// The actual server
final httpsServer = Router()
    ..get('/', (_) => Response.ok('Hello secure world!'));

await shelf_io.serve(httpServer, InternetAddress.LoopbackIPv4, 80);
await shelf_io.serve(httpsServer, InternetAddress.LoopbackIPv4, 443);
```

To generate private key and certificate files, you have to use an external tool such as *Let's Encrypt* or any other certificate authority.

- Using the `Cascade` class, you can serve static and dynamic contents from the same server. For example, imagine you wanted to create a server with an HTML home page and a JSON API that generates dynamic content. Here's how you can do it:

```
void main() async {
    final staticContents = createStaticHandler(
        'var/www/html/',
        defaultDocument: 'index.html',
    );

    final dynamicContents = Router()
        ..post('api/json/get-version', infoHandler)
        ..post('api/json/get-data', infoHandler);

    final server = Cascade()
        .add(staticContents)
        .add(dynamicContents)
        .handler;

    await shelf_io.serve(server, InternetAddress.LoopbackIPv4, 80);
}
```

A [Cascade](#) is used to call several handlers, in sequence, and return the first acceptable response. In our example, the handler first tries to serve the HTML website. If the request doesn't match the static handler, then it moves to the next one (defined by the [Router](#)). If no matches are found, a *404 Not found* response is sent.

- Even if [shelf](#) uses [HttpServer](#) under the hood, don't mix the two. If you're using [shelf](#), then make sure to only use its API and companion packages.

Pay attention to not confuse the [Cascade](#) and [Pipeline](#) objects. They have similar usage but two completely different purposes:

- A [Pipeline](#) object is used to chain middleware and executes them all in sequence.
- A [Cascade](#) object is used to chain handlers and only executes the first acceptable match.

A pipeline is generally used when you have to perform some actions before returning each response to the clients. A cascade instead is usually used when you want to create a server that handles both static and dynamic contents.

## 22.3 Creating and maintaining an HTTP server



[https://github.com/albertodev01/flutter\\_book\\_website](https://github.com/albertodev01/flutter_book_website)

In this section we're using Dart and the [shelf](#) package to build and deploy the official website of this book. You can see the result at <https://fluttercompletereference.com> and the complete source code on our GitHub repository.

### Note

Without going too much into the details of the server environment (which also goes out of the scope of this book) here's our configuration:

- We're running the server on the "Ubuntu Server 22.04.1 LTS" distribution. We've decided to go for Linux just for a matter of preference but of course we could have compiled and run the application on another platform.
- We're using *Let's Encrypt* to enable secure HTTPS connections to our server.

Since the primary scope of the server is to serve static contents, such as images and [html](#) files, we use `shelf_static`. As we have seen at the beginning of the chapter, to create a new server-side project with Shelf, we can directly use the `dart create` command:

```
dart create -t server-shelf flutter-book-website
```

Once the project is created, we update all dependencies to the latest versions and we configure the `analysis_options.yaml` file. We have added a `public` folder to contain all static assets to serve (mostly HTML and CSS files):

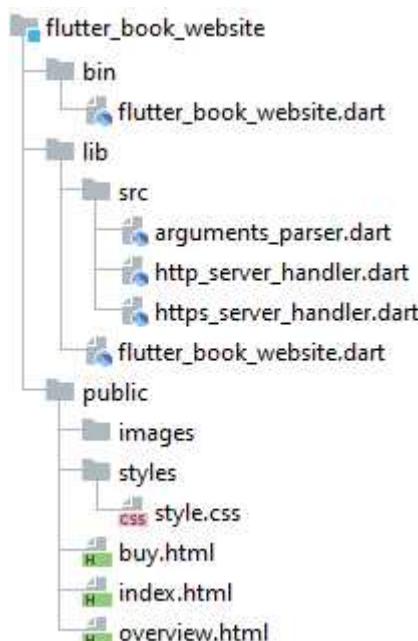


Figure 23.2: The project structure of the server app.

Command-line applications have a `bin` folder that contains scripts written in various languages. As such, this is the right place to write a Dart script that starts our servers:

```
// Contents of 'bin/flutter_book_website.dart'
Future<void> main(List<String> arguments) async {
    await shelf_io.serve(httpServerHandler, InternetAddress.LoopbackIPv4, 80);

    await shelf_io.serve(
        httpsServerHandler,
        InternetAddress.LoopbackIPv4,
        443,
        securityContext: buildSecurityContextFromArgs(arguments),
    );
}
```

The first server uses the HTTP protocol to redirect all requests to the HTTPS address permanently. The second server instead uses the HTTPS protocol to serve static files to the clients. Handlers and utility functions are defined in the `lib` folder:

- The HTTP server handler sends a *301 Moved permanently* response to redirect all incoming requests to HTTPS:

```
// 'Lib/src/http_server_handler.dart'  
Future<Response> httpServerHandler(Request request) async =>  
    Response.movedPermanently(request.url.replace(scheme: 'https'))
```

- The HTTPS server handler uses `shelf_static` to serve files from `public/`, which contains the HTML, CSS, and image files of the website:

```
// 'Lib/src/https_server_handler.dart'  
final httpsServerHandler = createStaticHandler(  
    'public',  
    defaultDocument: 'index.html',  
)
```

It's important to notice that the `bin` folder only contains code that interacts with the console. We only start the servers there; all the logic, even if simple, is defined in the `lib` folder. This is because we follow Dart's package layout conventions<sup>146</sup> and guidelines<sup>147</sup> (which we've already covered in *chapter 20 – “Creating and maintaining a package”*). If our server also served dynamic contents for example, we would still create the implementation inside `lib`:

```
// 'Router' is from the 'shelf_router' package  
final _dynamicContents = Router()  
    ..get('/api', welcomeHandler)  
    ..get('/api/data', dataHandler);  
  
final httpsServerHandler = Cascade()  
    .add(createStaticHandler('public', defaultDocument: 'index.html'))  
    .add(_dynamicContents)  
    .handler;
```

When a handler requires more than a single line of code, we prefer to move it into a dedicated Dart file.

---

<sup>146</sup> <https://dart.dev/tools/pub/package-layout>

<sup>147</sup> <https://dart.dev/guides/libraries/create-library-packages>

### 22.3.1 Setup and deployment

Before running the application, we need to generate the TLS encryption certificates (using *Let's Encrypt*, for example) to enable the HTTPS protocol in our server. Once they're ready, we can run a quick test on the server to make sure that everything works as expected. From the project root, run this command:

```
dart run bin/flutter_book_website.dart --chain-path=path/to/cert/chain.pem  
--key-path=path/to/key.pem
```

The server is up and running if you don't see error messages. The path to certificates is passed from the outside rather than being hard-coded in the application. Let's see again how the HTTPS server is setup:

```
Future<void> main(List<String> arguments) async {  
    // other code here...  
    await shelf_io.serve(  
        httpsServerHandler,  
        serverAddress,  
        443,  
        securityContext: buildSecurityContextFromArgs(arguments), // <- this  
    );  
}
```

Inside the `lib` folder, we have created the `buildSecurityContextFromArgs` function to parse command line arguments and extract the paths to chain and key files:

```
SecurityContext buildSecurityContextFromArgs(List<String> arguments) {  
    final argParser = ArgParser()  
        ..addOption('chain-path', mandatory: true)  
        ..addOption('key-path', mandatory: true);  
  
    final values = argParser.parse(arguments);  
  
    return SecurityContext()  
        ..useCertificateChain(values['chain-path'] as String)  
        ..usePrivateKey(values['key-path'] as String);  
}
```

The `ArgParser` object comes from `args`,<sup>148</sup> a package maintained by the Dart team that is used to parse raw command-line arguments. Thanks to `addOption`, we can define required command-line

---

<sup>148</sup> <https://pub.dev/packages/args>

options that must be passed as arguments when running the executable. A `FormatException` is thrown if the application isn't run without the two options. These arguments can also be passed to the compiled binary file of course, which is the recommended way of running the application.

## Note

We've used `dart compile` to produce a native, self-contained executable file within the `bin` folder of the project:

```
dart compile exe bin/flutter_book_website.dart
```

Then, we moved the `flutter_book_website` executable in a dedicated folder. At this point, we could have run the server with this command:

```
./flutter_book_website --chain-path=path/to/cert/chain.pem  
--key-path=path/to/key.pem
```

The problem is that as soon as we end the SSH session, the process ends. Since we want to start the application and keep it running even after we've closed the terminal, we configured `systemd`. It's a great Linux software suite that makes it easy to manage long running and background tasks.

If we were on Windows Server instead, we would run the program as a service so that it would keep running even after we logged out.

When you're ready to use the application in production, prefer building an executable file with `dart create` rather than using `dart run`. Besides being faster, the executable can keep your application running in the background and stay alive even if you close the session. Dart cannot automatically keep running itself in the background, so you have to use platform-specific tools to do that.

### 22.3.2 Unit tests

In this section, we're covering how we tested the various components of the application and how we made some benchmarks. We've generated development-only certificates with the `openssl` tool to be able to test the `SecurityContext` object. We have run the following command to generate `server.key` and `server.pem`:

```
openssl req -new -x509 -keyout server.key -out server.pem -days 3650 -nodes
```

We moved the two certificates inside `test/test_certificates` so that we can easily reference them when writing unit tests. Again, use these files only for unit tests; in production, use a tool like *Let's Encrypt* or another certificate authority to generate the keys. This is how we have tested the `buildSecurityContextFromArgs` method:

```
import 'package:path/path.dart' as path;

void main() {
    final chainPath = path.join('test', 'test_certificates', 'server.pem');
    final keyPath = path.join('test', 'test_certificates', 'server.key');

    group("Testing the 'buildSecurityContextFromArgs' method", () {
        test('Valid arguments', () async {
            expect(
                buildSecurityContextFromArgs([
                    '--chain-path=$chainPath',
                    '--key-path=$keyPath',
                ]),
                isA<SecurityContext>(),
            );
        });

        test('Missing arguments', () async {
            expect(() => buildSecurityContextFromArgs([]), throwsFormatException);
        });

        test('Missing chain path', () async {
            expect(
                () => buildSecurityContextFromArgs(['--key-path=b']),
                throwsFormatException,
            );
        });

        test('Missing key path', () async {
            expect(
                () => buildSecurityContextFromArgs(['--chain-path=a']),
                throwsFormatException,
            );
        });
    });
}
```

The `path` package, maintained by the Dart team, is used for cross-platform path manipulations on the filesystem. The `join` method returns a path to a file using the correct separator (in our case, it would use `\` on Windows and `/` on macOS). Since the HTTPS handler only serves static contents, we

can use `httpsServerHandler` to start a server locally and make requests using an `HttpClient` object. An HTTP server is easier to set up than an HTTPS, but the protocol is irrelevant in this case. We have to test the handler, not the server protocol:

```
void main() {
    late HttpServer server;
    final ip = InternetAddress.loopbackIPv4;
    const port = 80;

    setUpAll(() async {
        server = await serve(httpsServerHandler, ip, port);
    });

    tearDownAll(() async {
        await server.close();
    });

    Future<void> sendRequest(String path, {bool invalidPath = false}) async {
        final client = HttpClient();
        try {
            final request = await client.get(ip.address, port, path);
            final response = await request.close();

            if (invalidPath) {
                expect(response.statusCode, equals(404));
                expect(response.headers.contentType, isNotNull);
                expect(response.headers.contentType!.mimeType, equals('text/plain'));
            } else {
                expect(response.statusCode, equals(200));
                expect(response.contentLength, greaterThan(0));
                expect(response.headers.contentType, isNotNull);
                expect(response.headers.contentType!.mimeType, equals('text/html'));
            }
        } finally {
            client.close();
        }
    }

    group("Testing the 'httpsServerHandler' handler", () {
        test('Home page', () async => sendRequest('/'));
        test('Overview page', () async => sendRequest('/overview.html'));
        test('Buy page', () async => sendRequest('/buy.html'));
        test('Invalid route', () async => sendRequest('/x', invalidPath: true));
    });
}
```

We use `httpsServerHandler` in production because we need the HTTPS protocol. However, we do not care about HTTPS in this test because we need to only exercise the handler.

### Note

The `sendRequest` method was created to avoid code repetition and make the test cases more readable. It also makes sure that the client is always correctly closed, thanks to the `finally` block.

Last but not least, the test for the HTTP handler is easy since it only makes a permanent redirect for all incoming requests:

```
test('Permanent redirect test', () async {
  final response = await httpServerHandler(
    Request('GET', Uri.http('example.com', '/')),
  );
  expect(response.statusCode, equals(301));
  expect(response.headers['location'], contains('https:'));
});
```

The HTTP error code for a *Moved permanently* redirect is `301` so that's why we test against it. We could have setup the HTTP and the HTTPS servers to test the redirect with a client but that is not trivial. We have decided to manually check the properties of the returned `Response` object to keep the test fast and easy to maintain.

## 22.4 Low-level HTML manipulation with `dart:html`

### Note

For this section, we assume that you have a basic understanding of HTML, CSS and the DOM (Document Object Model).

A Dart application can interact with the DOM and modify it programmatically. In other words, you can “connect” Dart and HTML together to create dynamic web pages. All of this is done with the classes of the core `dart:html` library. In this section, we will create a simple Dart web application that shows the current date and time. Run the `dart create -t web demo` command to create a

new Dart web application that uses the default template. The project has the same structure of a Dart package with a few exceptions:

- There is a `web/` folder where the `main.dart` file is located, along with support HTML or CSS files, for example. You are free to organize the contents as you wish<sup>149</sup>, but we recommend splitting `.html` and `.css` files into separate folders. For example:



Figure 22.1: The recommended structure of the `web` folder.

The documentation recommends to create the `lib/` folder in case the `main.dart` file used any library code (such as helper functions or classes). This is the same thing you would do if you had to create a package or a console application.

- There is no `bin/` folder because this is not a console application.
- In general, assets (such as CSS, HTML or PNG files) should go in the `web/` folder. However, if you want them to be public and thus re-usable by other packages, put them inside `lib/`.

We want this example to be simple, so we will keep everything inside `web/`. To get started, open the `index.html` file and create a `<div>` element with a meaningful id:

```
<body>
  <div id="time"></div>
</body>
```

This `<div>` will be dynamically updated by Dart to show the current date and time. Note that you could have used any other HTML element instead of a `<div>` (such as `<p>` or `<h1>` for example). To change the color and the font family, open the `styles.css` file and use these rules:

---

<sup>149</sup> <https://dart.dev/tools/pub/package-layout#web-files>

```
#time {  
  color: blue;  
  font-family: Arial, Helvetica, sans-serif;  
}
```

From the `main.dart` file, we can use the `dart:html` library to find the `<div>` element and change its contents. For example, we could create a periodic timer that constantly updates the text with the current date and time:

```
import 'dart:async';  
import 'dart:html';  
  
void main() {  
  Timer.periodic(const Duration(seconds: 1), (_){  
    final now = DateTime.now();  
    final date = '${now.year}/${now.month}/${now.day}';  
    final time = '${now.hour}:${now.minute}.${now.second}';  
  
    querySelector('#time')?.text = '$date $time'; // Updates the <div> text  
  });  
}
```

The top-level `querySelector` function (from the `dart:html` library) is used to select and retrieve a single DOM element of an HTML page. It uses CSS selectors to find the first element in the DOM that matches the selector. If no element is found, it returns `null`. Let's give a closer look:

```
final Element? div = querySelector('#time');
```

This code looks for any element in the DOM whose id is `time` and either returns `Element` or `null`. In other words, `querySelector` is used to find and “convert” a DOM element into a Dart object:

```
<body>  
  <div id="time"></div> → querySelector('#time')?.text = '$date $time';  
</body>
```

The `text` property allows you to dynamically change the text to show the date and the time within the `<div>`. To run the web application, you need to compile the Dart code into JavaScript using the `webdev` tool. It has two commands:

- The `serve` command is used to run a local development server that serves your Dart web application. From the root directory of your project, run these commands (in order):

```
dart pub global activate webdev  
webdev serve
```

This will start the development server and the web application at [127.0.0.1:8080](http://127.0.0.1:8080). As you update the code, the development server will automatically reload the application so that you can refresh the page and see your changes in real time. For example:



Figure 22.2: The Dart web application running on Google Chrome.

This is how our application looks like on Google Chrome. If we changed the font color on the CSS file, as soon as we save the `.css` file the server would reload the application. We would then press F5 or the “refresh” button to see the changes on the browser.

- The `build` command is used to build your Dart web application into static files that can be deployed to a web server. From the root directory of your project, run these commands (in order):

```
dart pub global activate webdev  
webdev build
```

This will build your application and output the static files in the `build/` directory (generated at the root). To deploy the application, copy and paste the contents of `build/` in your web server.

Note that the `webdev build` command will optimize the application for production by minifying and tree-shaking your code. This means that the resulting build will be smaller and faster than your development version. We recommend to only use the `webdev serve` command to develop and debug the application.

#### 22.4.1 DOM elements interactions

The `Node` class represents a generic node in the DOM tree. The `Element` class is a subtype of `Node` that is “specialized” for a given page node. For example, it knows position, width, and height of an element on the page. You can manipulate the DOM by adding and removing nodes. For example, imagine that you had this list in an HTML file:

```
<ul id="list">
  <li>1</li>
  <li>2</li>
</ul>
```

We have already seen that the `querySelector` function finds nodes in the DOM and converts them into `Element` objects. Note that `Element` has dozens of subclasses, many of which correspond to various HTML tags. For example:

```
void main() {
  // Finds the List and casts the type
  final list = querySelector('#list')! as ULListElement;
}
```

The `ULListElement` class represents the `<ul>` element in the DOM. To add more entries to the list, we have to create a new `LIElement` object, which is the equivalent of a `<li>` HTML tag. This is how you could add more items to the list:

```
void main() {
  // Finds the List and casts the type
  final list = querySelector('#list')! as ULListElement;

  for (int i = 2; i <= 10; ++i) {
    // Create a new '<li>' element
    final li = LIElement()..text = '$i';
    // Insert the '<li>' element in the '<ul>'
    list.children.add(li);
  }
}
```

An `Element` object has a single parent (whose type is still `Element`) and many children. The parent object is `final` and cannot be changed. As such, you cannot move elements by changing the `parent` property. The `children` property instead is a `List<Element>` that can be changed. In the example, we have added more children, but we could have also removed any:

```
final list = querySelector('#list')! as ULListElement;

// Removes all '<li>' elements
list.children.clear();

// Inserts new '<li>' elements
for (int i = 2; i <= 10; ++i) {
  final li = LIElement()..text = '$i';
  list.children.add(li);
}
```

An element is removed from the DOM when it is removed from the parent's `children` list. As such, if we call `clear()`, we remove all the `<li>` elements in the list. You should use the `remove` method to remove specific elements from the DOM. For example:

```
// Fill the list
final list = querySelector('#list')! as ULListElement;
for (int i = 2; i <= 10; ++i) {
  list.children.add(Element.li()..text = '$i'); // Adds entry
}

// Remove the penultimate item on the list
list.children[9].remove();

// After 3 seconds, remove the entire list
await Future.delayed(
  const Duration(seconds: 3),
  () => list.remove(),
);
```

This code fills the list with some `<li>` elements and removes the penultimate from the `children` list. After three seconds, the entire `<ul>` list element is removed from the DOM. The `Element.li` factory constructor has the same purpose as `LIElement`, but it's a bit more readable. The `Element` class can also add and remove CSS rules or classes from an object. For example:

```
<p id="paragraph">-</p>
```

This `<p>` element currently has no CSS classes, but we could use the `classes` property to add one or more. For example, this code assumes that your CSS files have a “`blue-text`” class and adds it to the element whose id is `paragraph`:

```
// Adds the '.red' CSS class to the '<p>' element
querySelector('#paragraph')?.classes.add('blue-text');
```

The `remove` method is used to remove a CSS class from the element. The `Element`<sup>150</sup> API is massive and we recommend visiting the official documentation for complete coverage of all properties.

When a user enters text in an `<input>` element, for example, a “change event” is fired to indicate that the input field changed. Generally, elements on the DOM can react to various events. For example, consider this configuration:

---

<sup>150</sup> <https://api.dart.dev/stable/dart-html/Element-class.html>

```

<body>
  <p id="click">-</p>
  <p id="text">-</p>
  <input id="input_element" />
</body>

```

In Dart, the `<input>` element is represented by the `InputElement` class. We could configure some callbacks for the “clicked” and “changed” events like this:

```

void main() {
  final input = querySelector('#input_element')! as InputElement;

  input.onClick.listen((_) => querySelector('#click')?.text = 'Click!');
  input.onChange.listen((_) => querySelector('#text')?.text = input.value);
}

```

The `onClick` stream fires an event when the element is clicked on, while the `onChange` stream fires an event when the text is changed. The `listen` method accepts a callback whose parameter type is `Event`. The `Event` object carries information about the event that occurred. For example:

```

querySelector('#paragraph')?.onClick.listen((MouseEvent event) {
  final Point offset = event.offset; // Mouse pointer coordinates
  final bool ctrl = event.ctrlKey; // Whether 'CTRL' was pressed or not
  final int btn = event.button; // Which button triggered the event
});

```

In this example, `MouseEvent` is a subclass of `Event` and carries information about the mouse cursor that triggered the event. Not all callbacks provide the same event, so check out the documentation for more details.

## 22.4.2 Deferred imports

Imagine that you created a `utils.dart` file inside the `lib/` folder of your Dart web application. For example, it may contain some utility functions like this:

```

/// Returns the sum of [a] and [b].
double sumValues(double a, double b) => a + b;

/// Returns the difference between [a] and [b].
double subtractValues(double a, double b) => a - b;

```

Any other location would have been ok, but library code should always live inside `lib`. As usual, the `import` statement allows you to use the code within this library. However, when you compile Dart for the web you could use the `deferred as` keyword to defer the loading of a library at a later time.

When a library is marked as deferred, it is not loaded immediately as soon as the program starts. It is loaded on-demand when it's actually needed. For example:

```
import 'dart:html';
import 'package:demo/utils.dart';

void main() {
  final div = querySelector('#container')!; // <div id="container"></div>
  div.onClick.listen(_ => div.text = '${sumValues(10, 20)}');
}
```

In this program, the `sumValues` function from the `utils` library is only used if an element is clicked. If you never click on the `<div>` element, the `sumValue` function is never called, and thus the `utils` library was unnecessarily loaded (because none of its members were ever used). You could improve this code with the “deferred loading” feature:

```
import 'dart:html';
import 'package:demo/utils.dart' deferred as utils; // deferred Loading

void main() {
  final div = querySelector('#container')!; // <div id="container"></div>

  div.onClick.listen(_ async {
    await utils.loadLibrary();
    div.text = '${utils.sumValues(10, 20)}';
  });
}
```

When the program starts, `utils.dart` is not loaded. Furthermore, the library is never loaded if you never click the `<div>` element. The library is loaded when you call the asynchronous `loadLibrary` function. From that moment onwards, you can use all of its public members just like you would if you had imported it normally.

### Note

You must always call `loadLibrary()` before using a deferred library. If you forget about it, a runtime error will occur. Calling `loadLibrary()` multiple times is safe because the library is only loaded once (and subsequent calls just are ignored).

Deferred imports can help reduce the application’s startup time and memory usage, mainly if the library contains a lot of rarely used code.

## Deep dive: The http package

The Dart team maintains a package called `http`<sup>151</sup> which, very intuitively, is used for making HTTP requests. It has a set of high-level, multi-platform classes that run on desktop, mobile, and web. The simplest way to use this library is via top-level functions. For example, this is how you would make a GET request:

```
import 'package:http/http.dart' as http;

Future<void> main() async {
  final response = await http.get(
    Uri.http('website.com', 'data.json'),
    headers: {
      'header1': 'value1',
      'header2': 'value2',
    },
  );

  print(' > Body: ${response.body}'); // contents of the 'data.json' file
  print(' > Status: ${response.statusCode}'); // 200
  print(' > Headers: ${response.headers}'); // a Map with all headers
}
```

In this example, we await the `get` function to get a `Response` object with the body, the status code, and other information about the response from the server. By convention, the library is aliased with `http`, but that is not a requirement. Note that you need to use a `Uri` object to define an endpoint, which can be created in various ways:

- Default (factory) constructor: it lets you build the entire URL from scratch. When you don't set a port, it uses default ones (`80` for HTTP and `443` for HTTPS):

```
Uri(
  scheme: 'http',
  host: 'website.com',
  path: 'ads.txt',
  port: 80,
),
```

You can also define query parameters or path fragments. This constructor is particularly useful when you want to make a request with non-HTTP schemes. For example, you could

---

<sup>151</sup> <https://pub.dev/packages/http>

create a `mailto` URI and use the `queryParameters` property to append one or more query parameters at the end:

```
Uri(  
  scheme: 'mailto',  
  path: 'my@email.com',  
  queryParameters: {'key': 'value'},  
, // mailto:my@email.com?key=value
```

This URI maps to `mailto:my@email.com?key=value`.

- `http` (factory) named constructor is a shorthand for the HTTP schema:

```
Uri.http(  
  'website.com',  
  'path/to/resource/file.json',  
,
```

That URI maps to `http://website.com/path/to/resource/file.json`.

- `https` (factory) named constructor is a shorthand for the HTTPS schema:

```
Uri.https(  
  'website.com',  
  'path/to/resource/file.json',  
,
```

That URI maps to `https://website.com/path/to/resource/file.json`.

- The `tryParse` static method tries to create a valid `Uri` object from the given string. If the conversion fails, the method returns `null`:

```
Uri.tryParse('https://website.com/path/to/file.json');  
Uri.tryParse('https://website.com/path/to/file.json');
```

The first call returns a `Uri` object that maps to `https://website.com/path/to/file.json` but the second one returns `null` because `https` is not a valid schema.

Note that `Uri` is part of the core Dart SDK: it does not come from the `http` package. Of course, you can make other types of requests, such as POST, PUT, PATCH, HEAD, or DELETE. For example, this is how you would make a POST request to a server:

```

final response = await http.post(
  Uri.http(
    'website.com',
    'some/path',
  ),
  body: {
    'key1': 'value1',
    'key2': 'value2',
  },
  headers: {
    'header1': 'value1',
  },
);

```

You will always have to provide a valid `Uri` object for any HTTP request type. The structure is the same for other top-level functions (`delete`, `head`, `patch`, and `put`). These functions should be used when you need to make a single request. When you have to send multiple requests to the same server, use a `Client` instead:

```

const endpoint = 'website.com';
final client = http.Client();

try {
  final fileName = await client.get(Uri.http(endpoint, 'fileName.txt'));
  final response = await client.get(Uri.http(endpoint, fileName.body));
  final value = doSomething(response.body);

  await client.put(Uri.http(endpoint, 'something'), body: value);
} finally {
  client.close();
}

```

A `Client` object keeps an open and persistent connection with the server, which is more efficient than sending individual requests with top-level functions. Calling `client.close` is very important when you're done with all requests. To ensure that `close()` is always called, even if an exception occurred, place it in a `finally` block. Note that we could have also written the code in this way:

```

const endpoint = 'website.com';

final fileName = await http.get(Uri.http(endpoint, 'file.json'));
final response = await http.get(Uri.http(endpoint, fileName.body));
final value = processResponse(response.body);

await http.put(Uri.http(endpoint, 'something'), body: value);

```

This is less efficient than using a client because top-level `get` and `put` methods (along with all the others) always close the connection when the request is completed. This would mean that each call would open and close the connection to the same server multiple times (rather than keeping it alive). To summarize:

- A `http.Client` object keeps the connection open until you manually close it. This is good when you have to make multiple requests to the same server (because there is no need to close and re-open the same connection multiple times).
- Top-level functions (`http.get`, `http.post`, `http.put` and the others) always automatically close the connection once the server sends a response. This is good for one-time requests because the client is internally managed for you.

The package has a top-level function called `read` which performs an HTTP GET request and returns its body as a `String`. It's the same as awaiting the `get` method and reading its body, but it's more compact. To be clear, these two examples are identical, but the first one returns a `Response` object while the second one directly returns the `body` content:

get	read
<pre>final resp = await http.get(     Uri.http('website.com'), );  print(resp.runtimeType); // Response print(resp.body);</pre>	<pre>final resp = await http.read(     Uri.http('website.com'), );  print(resp.runtimeType); // String print(resp);</pre>

We recommend to use `read` when you only need to retrieve the `body` of a `Response` object. In all the other cases, use `get`.

#### Note

When HTTP requests fail, a `ClientException` object is thrown (with message and URI).

The `http` package is designed to be composable. For example, if you had to send multiple requests to a server with the same content type, you could extend `http.BaseClient` and create a reusable client like this:

```

class ContentTypeClient extends http.BaseClient {
    final String contentType;
    final http.Client _client;
    ContentTypeClient(this.contentType) : _client = http.Client();

    @override
    Future<http.StreamedResponse> send(http.BaseRequest request) {
        request.headers['content-type'] = contentType;
        request.maxRedirects = 4;

        // other request-specific configurations here...

        return _client.send(request);
    }

    @override
    void close() {
        _client.close();
        super.close();
    }
}

```

This class is basically a wrapper of `http.Client` that sets a custom header for you. As always, clients must be closed when not needed anymore, so override `close()` as we did. We want to keep the internal client private so that it cannot be accidentally closed from the outside. Custom clients are consumed in the usual way:

```

final client = ContentTypeClient('application/json');

try {
    final resp = await client.get(Uri.http('website.com', 'file.txt'));

    print(resp.request!.headers); // contains 'content-type: application/json'
} finally {
    client.close();
}

```

The request has the `application/json` header as we specified. Creating custom clients is useful when reusing specific request configurations and avoiding code duplication.

The `RetryClient` class is used to retry failing requests a certain number of times automatically. It is a `BaseClient` sub-type that implements a retry logic:

```

/// An HTTP client wrapper that automatically retries failing requests.
class RetryClient extends BaseClient {

```

As it happens with any other HTTP client, all `RetryClient` objects must be closed when not needed anymore. We have already seen this pattern multiple times up to now: wrap everything after the object creation in a `try` block and make sure to call `close()` inside `finally`:

```
final client = RetryClient(  
    http.Client(),  
    retries: 2,  
);  
  
try {  
    final response = await client.post(  
        Uri.https('website.com', 'path/to/resource'),  
        body: {'key': 'value'},  
    );  
  
    if (response.statusCode == 200) {  
        print('> OK: ${response.body}');  
    } else {  
        print('> Error!');  
    }  
} finally {  
    client.close();  
}
```

In this example, the client will retry the POST request two more times in case the first one fails. If there were more HTTP calls, they would still be subject to the same retry policy.

### Note

By default, a `RetryClient` retries any request (whose response has status code 503) up to three times. It waits 500 milliseconds before the first retry, and increases the delay by 1.5x each time.

You can use the `RetryClient` constructor to customize the retry policy. For example, you can set the number of times the client should retry the request, the delay between retries or add a callback for each retry.

# 23 – Platform interactions

---

## 23.1 Platform-specific features

In some cases, the Flutter framework automatically adopts specific behaviors to mimic as much as possible the platform's ones. There are slight differences in how some features are presented to the users, but the final result doesn't change. Here are a few examples:

- When you call `Navigator.push` the application navigates to a new route, but the default animation may change. For example, on iOS, there is a slide transition; on Android, the new route slides upwards and fades. Use the theme to override the default animation type on specific platforms. For example:

```
MaterialApp(  
    theme: ThemeData(  
        pageTransitionsTheme: const PageTransitionsTheme(  
            builders: <TargetPlatform, PageTransitionsBuilder>{  
                TargetPlatform.iOS: FadeUpwardsPageTransitionsBuilder(),  
                TargetPlatform.macOS: CupertinoPageTransitionsBuilder(),  
                TargetPlatform.windows: ZoomPageTransitionsBuilder(),  
            },  
        ),  
    ),  
,  
,
```

You can of course do the same with `CupertinoApp` using `CupertinoThemeData` rather than `ThemeData`.

- When you want to go back to the previous route, on Android you can use the back button in the bottom bar while on iOS you can edge-swipe to the left. If you try to edge-swipe to the left on Android for example, you will see that you won't go back to the previous page because the gesture is specific to iOS devices.
- There are different scrolling physics in some platforms. Android scrolls “more” on stronger flings and stops less abruptly while iOS has “less” scroll on stronger flings and stops quicker for example. In *chapter 14 – Section 3.2.1 “Scrollbar”* we saw how to override the default scroll behavior on different platforms. On iOS, if you tap on the OS status bar, the primary scrollable widget animates to the top.

- The default overscroll behavior is also different across platforms. On desktop platforms, for example, there is no overscroll because you either use the mouse wheel or the scroll bar on the right (which doesn't go past the edge of a scrollable widget). On Android and iOS instead, the overscroll is visible:

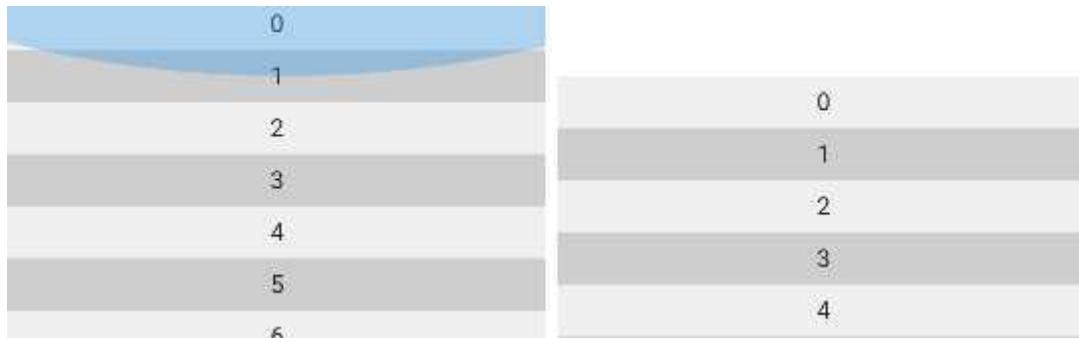


Figure 24.1: Example of overscroll in Android (on the left) and iOS (on the right).

On Android, there is a sort of blue semi-circle shadow whose center projection follows the finger. On iOS the list acts as if a spring pulled the list upwards.

- Flutter uses different font families that are appropriate for the platform. For example, on Android the font is Roboto while on iOS and macOS it's San Francisco. The same also goes for icons. For example, the back button on the `AppBar` widget shows `←` on Android and `<` on iOS.
- Haptic feedback is a mobile-only feature that makes the device vibrate. It has a platform-specific behavior<sup>152</sup>, so the vibration's intensity and duration on Android and iOS changes. For example:

```
ElevatedButton(
  onPressed: () async {
    await HapticFeedback.selectionClick();
    doSomething();
  },
  child: const Text('Press me'),
),
```

---

<sup>152</sup> <https://api.flutter.dev/flutter/services/HapticFeedback-class.html>

When the user presses the button, the device vibrates for a variable amount of time and a given intensity, which depends on the platform. The possible feedbacks are:

- `HapticFeedback.heavyImpact()`, used to give the feedback of a collision with a heavy mass;
- `HapticFeedback.lightImpact()`, used to give the feedback of a collision with a light mass;
- `HapticFeedback.mediumImpact()`, used to give the feedback of a collision with a medium mass;
- `HapticFeedback.vibrate()`, used to vibrate the device for a short duration;
- `HapticFeedback.selectionClick()`, used to vibrate in response to an action such as a tap.

Since this class totally depends on the platform, it's not suitable for precise control of the system's haptic feedback module.

- Text selection and keyboard interactions also follow platform-specific behaviors.

Some widgets could work on any platform but not have consistent behaviors. The `MouseRegion` widget, for example, is always mounted on the widget tree but all of its listeners are not triggered on mobile devices. To enable or disable some features on your application according to the host platform, we remind you to use the `BuildContext` as follows:

```
switch (Theme.of(context).platform) {  
  case TargetPlatform.android:  
    return const Text('Android');  
  case TargetPlatform.fuchsia:  
    return const Text('Fuchsia');  
  case TargetPlatform.iOS:  
    return const Text('iOS');  
  case TargetPlatform.linux:  
    return const Text('Linux');  
  case TargetPlatform.macOS:  
    return const Text('macOS');  
  case TargetPlatform.windows:  
    return const Text('Windows');  
}
```

In the following two sections, we will see how to create Flutter packages where the Dart code can communicate with other programming languages supported by the host platform.

### 23.1.1 Platform-specific packages for Flutter

The Flutter team maintains platform-specific packages on its verified publisher account<sup>153</sup>. Some packages use native APIs, so the implementation might not be consistent across different operating systems. Other packages only work on specific platforms. Here is a list of the most popular ones:

- `url_launcher`: this package (which works on mobile, desktop, and web) launches a URL with a particular schema. For example:

```
ElevatedButton(  
  onPressed: () async {  
    // Returns 'true' if the URL was Launched successfully, otherwise  
    // either returns 'false' or throws a PlatformException depending  
    // on the failure.  
    final result = await launchUrl(Uri.https('flutter.dev'));  
  
    if (!result) {  
      setState(() {  
        error = "Couldn't launch the URL";  
      });  
    }  
  },  
  child: const Text('Open'),  
,
```

In this example, when you press the button, Flutter tries to open the browser at the given URI. The method can either return `false` or throw an exception if something goes wrong. This package parses various schemas, but some may only be available on specific platforms. For example:

```
// Mobile-only feature  
final result = await launchUrl(Uri.parse('tel:+1-222-333-444'));  
  
if (!result) {  
  setState(() {  
    error = "Couldn't dial the number";  
  });  
}
```

---

<sup>153</sup> <https://pub.dev/publishers/flutter.dev/packages>

The `tel` schema only works on mobile devices because it uses the default phone application to dial the number. The package currently supports the following schemas: `http` or `https` to open the URL on the default browser, `mailto` to send emails using the email application, `sms` to send SMS messages, `file` to open a file or a folder using the default application, and `tel` to dial a number.

- `camera`: a Flutter plugin that gives access to the device camera(s). To use the package, you need a `CameraController` (which is a `ValueNotifier`) and manage its lifetime. In the state class of a stateful widget, for example, you could use this setup:

```
late final CameraFuture = initializeCameras();
CameraController? controller;

Future<void> initializeCameras() async {
  final cameras = await availableCameras();
  if (available.isNotEmpty) {
    controller = CameraController(cameras.first, ResolutionPreset.max);
    await controller!.initialize();
  }
}

@Override
void dispose() {
  controller?.dispose();
  super.dispose();
}
```

The `availableCameras` method returns a list of all available cameras (if any) that can be attached to a controller. You must call the `initialize()` method on the controller before using it. You can use the `CameraPreview` widget to see a live preview of the camera output. For example:

```
FutureBuilder<void>(
  future: cameraFuture,
  builder: (context, snapshot) {
    if (snapshot.connectionState == ConnectionState.done) {
      if (controller != null && controller!.value.isInitialized) {
        return CameraPreview(controller!);
      }
      return const Text('No camera available');
    }
    return const CircularProgressIndicator();
  },
);
```

To take a picture use the `controller.takePicture` method, which captures an image and returns an `XFile` object. You can also record a video and save it to the local filesystem using `controller.startVideoRecording` first and then `controller.stopVideoRecording` to stop the recording and save the file (`stopVideoRecording` returns an `XFile` object).

- `video_player`: this package is used to play various video file formats. The `pub.dev` page of the package has platform-specific instructions and considerations. The usage is very easy:

```
class _CameraWidgetState extends State<CameraWidget> {
    late final controller = VideoPlayerController.asset(
        'assets/video.mp4',
    );

    @override
    void dispose() {
        controller.dispose();
        super.dispose();
    }

    @override
    Widget build(BuildContext context) {
        return Scaffold(
            body: controller.value.initialized
                ? AspectRatio(
                    aspectRatio: controller.value.aspectRatio,
                    child: VideoPlayer(controller),
                )
                : const Text('No video to play'),
            floatingActionButton: FloatingActionButton(
                onPressed: () =>
                    controller.value.isPlaying
                        ? controller.pause()
                        : controller.play(),
                child: const Text('Play/Pause'),
            ),
        );
    }
}
```

The `VideoPlayer` widget is used to play the video on the screen, while `AspectRatio` makes sure that the widget is constrained correctly according to the video's aspect ratio. There are other named constructors for the controller, such as `network` (to play a video obtained from the internet) and `file` (to play a video from a `File` object).

- **file\_selector**: this package manages files and platform-specific interactions with dialogs. For example, this is how you can open multiple files at once with different extension types:

```
const types = XTypeGroup(label: 'images', extensions: ['jpg', 'png']);

final XFile? file = await openFile(
  acceptedTypeGroups: <XTypeGroup>[types],
  confirmButtonText: 'Open files',
);
```

The `openFile` method returns `null` when the user cancels the operation. You can also add multiple groups with unrelated file types.

- **shared\_preferences**: this package is used to save small key-value pairs at platform-specific locations. You can see `SharedPreferences` as a `Map<String, Object?>` that persists data on the disk. For example:

```
Future<void> writeData() async {
  final prefs = await SharedPreferences.getInstance();

  await prefs.setBool('boolean', false);
  await prefs.setInt('integer', 10);
  await prefs.setDouble('double', -0.5);
  await prefs.setString('string', 'ello bois');
  await prefs.setStringList('list-string', const ['a', 'c']);
}
```

You can only store numbers, booleans, strings and list of strings. Any other data type is not supported by `SharedPreferences`. To retrieve values, make sure to use the same key and use one of the “get” functions. For example:

```
// Getters return nullable values
final bool? something = prefs.getBool('something');

// Use 'containsKey' to check if the value is there
if (prefs.containsKey('number')) {
  final int number = prefs.getInt('number')!;
}
```

Note that “get” functions return `null` if the key doesn’t match with a value. To remove a value from the `SharedPreferences` storage, use the `remove` method. To remove all stored values, use the `clear` method.

In section 3 – *Creating plugin and FFI plugin packages* we will see an example of how to create these packages that use platform-specific APIs.

## 23.2 Writing platform-specific code

This section covers how Dart and other platform-specific languages can communicate together with method calls. This can be useful when you need to extract information about the platform in which the application is running or interact with hardware components. For example, here are some use cases where platform-specific code is required:

- knowing the platform’s battery level;
- using Bluetooth to discover nearby devices;
- receiving data from sensors, such as the gyroscope or the accelerometer (if available);
- retrieving system information such as the device version, time zone preferences, user locale, etc.

Dart alone is insufficient for those use cases since it does not have direct access to the operating system. We have already seen all of this in *chapter 1 – Section 3 “The Flutter framework”*. Flutter is a layered system where each level is independent by design, so Dart is “isolated” from the platform and has no hooks to the native APIs.

### Note

Flutter can only know the operating system in which the application is running. This info is available to you using `Theme.of(context).platform`, which returns an `enum` value. Use the `Platform` class to retrieve information about the underlying operating system on “pure” Dart programs.

The only way Flutter has to communicate with native code is via platform channels. Flutter’s flexible system allows Dart and the other platform-specific language to communicate. In particular, these are the languages that Dart can bind to:

- on Android, Dart can communicate with Kotlin or Java;
- on iOS, Dart can communicate with Swift or Objective-C;
- on Windows, Dart can communicate with C++;
- on macOS, Dart can communicate with Swift;
- on Linux, Dart can communicate with C.

On Dart, you use a [MethodChannel](#) object to send a message to the platform to tell that you want to invoke a native method. The operating system always listens for message requests and uses the native programming language to send a response back to Dart. For example, this image represents how to communicate with the operating system to get the battery level (if available):

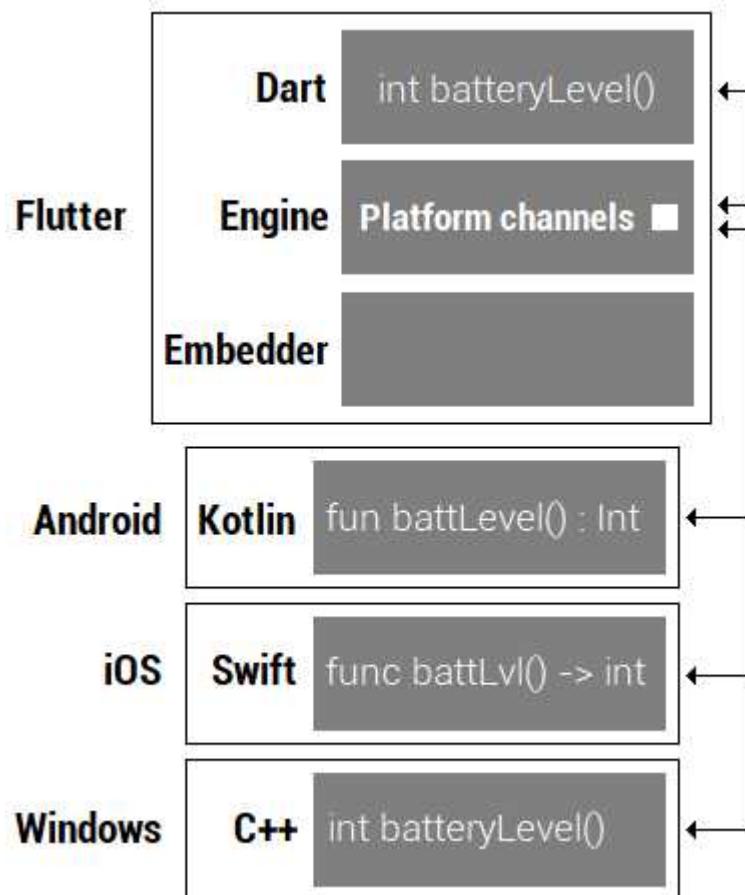


Figure 24.2: Dart communicating with the underlying platform using platform channels.

Flutter layers and the operating system are all independent. However, the engine layer has platform channels: a messaging system that sends information back and forth between Dart and another language. The image above shows how you'd get the current Android, iOS, and Windows battery levels. Here are a few things to add to the example:

- the image shows that there are no implementations for macOS and Linux. As such, you will not be able to retrieve the current battery level (if available) on those platforms;

- the engine asynchronously exchanges messages between Dart and the native language. All of this is internally handled by Flutter;
- on the Dart side, you will use a `MethodChannel` object to send a message to the operating system to notify that you want to call a native method. On the platform side, you will write native code (using Kotlin, Java, Swift, C++, or any other supported language) to write code that communicates with the Flutter engine.
- Dart types have their counterpart in other programming languages, which can be found in the documentation<sup>154</sup>. For example, the Dart `String` type maps to `std::string` in C++ and `NSString` in Objective-C.
- Data is serialized from a Dart type like `Map` into a standard format and then deserialized into an equivalent representation (like a `HashMap` in Kotlin, for example).

You may have noticed that we haven't said anything about the web platform. The reason is that Flutter web applications don't have the engine layer and thus platform channels don't exist.

### 23.2.1 Platform channels



[https://github.com/albertodev01/flutter\\_book\\_examples/tree/chapter\\_23/23.2.1](https://github.com/albertodev01/flutter_book_examples/tree/chapter_23/23.2.1)

This section shows how to call a platform-specific API to retrieve date and time. Note that you could avoid doing all of this and use Dart's `DateTime` class. However, we want to keep the example simple, so pretend that `DateTime` didn't exist and we wanted to create it using native method calls.

#### Note

This example adds the platform-specific code to the Flutter project itself. If you wanted to reuse the platform-specific code and create a package, the project creation and setup would be different. We will cover “plugin packages” in *Section 3 – Creating Flutter plugin packages*.

---

<sup>154</sup> <https://docs.flutter.dev/development/platform-integration/platform-channels?tab=type-mappings-c-plus-plus-tab#codec>

Let's start by creating a new Flutter application with `flutter create` from the terminal or using your favorite IDE. By default, Flutter uses Kotlin on Android and Swift on iOS, but you can change one or both with a specific flag. For example:

```
flutter create -i objc -a java platform_datetime
```

There isn't much to do on Dart because most of the implementation is done with native code. In this example, we want to show the current date and time at the center of the screen without using Dart's `DateTime` class. Setup these two files:

```
// 'channel_names.dart' file
const methodChannelDatetime = 'fluttercompletereference.com/date-time';
const methodGetDateTime = 'getDateTime';

// 'main.dart' file
void main() {
  runApp(
    const MaterialApp(
      home: Scaffold(
        body: Center(
          // the implementation of 'PlatformDatetime' is on the next page
          child: PlatformDatetime(),
        ),
      ),
    ),
  );
}
```

The first file contains some constant strings we're about to use. The second file contains the Flutter application itself. The `PlatformDateTime` widget uses a `MethodChannel` object to send a message to the platform asking for the current date and time. Note that:

- The platform channel must use the same name on both sides to connect Dart with the native language. In practice, the value of `methodChannelDatetime` must also be the same on the platform side to open the channel and start the communication. Channel names in the same application must be unique.
- The convention used to create unique channel names is to use a domain prefix and separate with a slash the method description (`fluttercompletereference.com/date-time`).

Since the platform channel system is asynchronous, we will use the usual `FutureBuilder` pattern to wait for a future in the `build` method. This is what the state of the `PlatformDatetime` widget looks like:

```

class _PlatformDatetimeState extends State<PlatformDatetime> {
  static const channel = MethodChannel(methodChannelDatetime);
  late final Future<String> future = platformDate.getTime();

  Future<String> platformDate.getTime() async {
    const error = "Couldn't get date and time";

    try {
      return await channel.invokeMethod<String>(methodGetDateTime) ?? error;
    } on Exception {
      return error;
    }
  }

  @override
  Widget build(BuildContext context) {
    return Column(
      mainAxisSize: MainAxisSize.min,
      children: [
        const Text('Date and time on the platform:'),
        const SizedBox(
          height: 20,
        ),
        FutureBuilder<String>(
          future: future,
          builder: (context, snapshot) {
            if (snapshot.hasData) {
              return Text(snapshot.data!);
            }

            return const Text('Loading...');
          },
        ),
      ],
    );
  }
}

```

You first need to create a `MethodChannel` object and make sure to give it a unique name across the entire application. The `invokeMethod<T>` function is used to send a message to the host platform to request calling a native method. The diamonds hold the expected return type (`String`) of the native method. To make a similitude, you can see the channel structure as follows:

- A `MethodChannel` object creates a “tunnel” that connects Flutter with the operating system and allows Dart to talk with Kotlin, Swift, C++, or any other language spoken by the platform;

- the `invokeMethod` function is a mailman that travels through the tunnel to send and receive requests. In our example, we send the `getDateAndTime` message to the platform which will execute some native code and then send a response back.
- it is essential that the channel on the Dart side and the channel on the native side have the same name; otherwise, the mailman (`invokeMethod`) doesn't know where messages have to be delivered.

It's a good practice wrapping `invokeMethod` in a try-catch because it might throw an exception if the platform couldn't send a response back. In the following sections, we will see how to receive the message on each platform and send a reply back to Dart.

### 23.2.2 Android implementation

Inside the `android/` folder of your Flutter project, open `MainActivity.kt` in `app/src/main/`. Using Android Studio and opening the `android/` folder as a separate project would be even better. Create the `configureFlutterEngine` override inside `MainActivity` and listen on the channel:

```
private val CHANNEL = "fluttercomplettereference.com/date-time"

override fun configureFlutterEngine(@NonNull flutterEngine:FlutterEngine){
    super.configureFlutterEngine(flutterEngine)

    MethodChannel(flutterEngine.dartExecutor.binaryMessenger, CHANNEL)
        .setMethodCallHandler {call, result ->
            if (call.method == "getDateAndTime") {
                result.success(LocalDateTime.now().toString());
            } else {
                result.notImplemented()
            }
        }
}
```

Notice that the `CHANNEL` variable holds the same channel name we have on the Dart side, which is fundamental for linking the channels. The `call.method` property contains the value of the string we passed to the `invokeMethod` on the Dart side. Here is a closer comparison of the two sides of the code:

- `channel = MethodChannel('fluttercomplettereference.com/date-time') // Dart`
- `CHANNEL = "fluttercomplettereference.com/date-time" // Kotlin`

- *// Dart*  
`await channel.invokeMethod<String>('getDateAndTime')`
  
- // Kotlin*  
`if (call.method == "getDateAndTime") {  
 result.success(LocalDateTime.now().toString());  
} else {  
 result.notImplemented()  
}`

The `invokeMethod` message (sent from Dart) is received in the Kotlin handler, and the response is sent using the `result.success` method. We send a `notImplemented` response in case Dart sends an invalid message. For example:

```
try {
  return await channel.invokeMethod<String>('getDate') ?? error;
} on Exception {
  return error;
}
```

The `getDate` message is correctly sent to the platform but it has no implementations so Kotlin sends a `result.notImplemented()` response. This is why we recommend always protecting the code in a `try` block. This pattern is also the same for the other platforms.

### 23.2.3 iOS implementation

Inside the `ios/` folder of your Flutter project, open `AppDelegate.swift` inside `Runner/`. Using XCode and opening the `ios/` folder as a separate project would be even better. In the `application` function, before `GeneratedPluginRegistrant`, create a new channel and initiate the listener. This is the same thing we previously did for Android (the only difference is the language):

```
let channel = FlutterMethodChannel(
  name: "fluttercompletereference.com/date-time",
  binaryMessenger: controller.binaryMessenger)
channel.setMethodCallHandler({
  (call: FlutterMethodCall, result: FlutterResult) -> Void in
  guard call.method == "getDateAndTime" else {
    result(FlutterMethodNotImplemented)
    return
  }
  let formatter = DateFormatter()
  formatter.dateStyle = .short
  result(formatter.string(from: Date.now()))
})
```

This is the same code you'd write in a native Swift application. Notice that, the channel name and the message (`getDateAndTime`) are the same as the ones we defined in Dart. On macOS, you still need to open the `AppDelegate.swift` file (in the `macos/` folder) and use the same API to create a channel listen for messages.

### 23.2.4 Windows implementation

Inside the `windows/` folder of your Flutter project, open `flutter_windows.cpp` inside the `runner/` folder. To open the project with Visual Studio, first run the `flutter build windows` command in the root to generate the `.sln` file and then open it. The channel and the listener are implemented in the `flutter_window.cpp` file:

```
flutter::MethodChannel<> channel(
    flutter_controller_->engine()->messenger(),
    "fluttercompletereference.com/date-time",
    &flutter::StandardMethodCodec::GetInstance());  
  
channel.SetMethodCallHandler(
    [](const flutter::MethodCall<>& call,
       std::unique_ptr<flutter::MethodResult<>> result) {
        if (call.method_name() == "getDateAndTime") {
            auto now = std::chrono::system_clock::now();
            auto time_t = std::chrono::system_clock::to_time_t(now);
            std::tm tm; gmtime_s(&tm, &time_t);
            std::stringstream ss;
            ss << std::put_time( &tm, "UTC: %Y-%m-%d %H:%M:%S");
            result->Success(ss.str());
        } else {
            result->NotImplemented();
        }
    });
);
```

You should be familiar with this pattern. This code is located inside the `OnCreate()` method after the `RegisterPlugins` call. Even if it's C++, we're always doing the same things: initializing the channel, adding a listener to process incoming messages, and returning a valid response (a `std::string` with date and time) or a *not implemented* one.

### 23.2.5 Linux implementation

Inside the `linux/` folder of your Flutter project, open `my_application.cc` and put your code inside the `my_application_activate` function after the `fl_register_plugins` method call. The code should look like this:

```

FlEngine *eng = fl_view_get_engine(view);
g_auto_ptr(FlStandardMethodCodec) codec = fl_standard_method_codec_new();
g_auto_ptr(FlBinaryMessenger) msg = fl_engine_get_binary_messenger(eng);

// Channel creation
g_auto_ptr(FlMethodChannel) channel = fl_method_channel_new(
    msg, "fluttercompletereference.com/date-time", FL_METHOD_CODEC(codec));

// Listener
fl_method_channel_set_method_call_handler(
    channel, method_dall, g_object_ref(view), g_object_unref);

```

After this, we have to create a new static function called `method_call` that listens for messages and sends back responses. For simplicity, here we're just showing the most relevant parts. Check out our GitHub repository for the complete source code:

```

static void method_call_cb(FlMethodChannel *channel,
                           FlMethodCall *method_call,
                           gpointer user_data)
{
    const gchar *method = fl_method_call_get_name(method_call);

    if (strcmp(method, "getDateTime") == 0) {
        // The actual response goes here
        FlValue *res = fl_value_new_string(dateTimeAsString());
        g_auto_ptr(FlMethodResponse) response =
            FL_METHOD_RESPONSE(fl_method_success_response_new(res));

        g_auto_ptr(GError) error = nullptr;
        fl_method_call_respond(method_call, response, &error);
    } else {
        // The 'not implemented' error
        g_auto_ptr(FlMethodResponse) response =
            FL_METHOD_RESPONSE(fl_method_not_implemented_response_new());

        g_auto_ptr(GError) error = nullptr;
        fl_method_call_respond(method_call, response, &error);
    }
}

```

It's a bit more complicated to read, but we're still doing the same thing. We check if the message is `getDateTime` and then either return a response or an unimplemented error.

The Flutter team could add support for more platforms or languages in the future. They will always use method channels to communicate with Dart. As you have seen, the strategy is always the same. The only difference is in the native language used by the platform, but Flutter has a consistent API.

### 23.2.6 macOS implementation

Inside the `macos/` folder of your Flutter project, open `MainFlutterWindow.swift` inside `Runner/`. Using XCode and opening the `macos/` folder as a separate project would be even better. Locate the `awakeFromNib` function and, before `RegisterGeneratedPlugins`, create the channel listener. The code is almost identical to iOS:

```
// Create the channel
let registrar = flutterViewController.registrar(forPlugin: "DateAndTime")
let channel = FlutterMethodChannel(
    name: "fluttercompletereference.com/date-time",
    binaryMessenger: registrar.messenger)

// Add the listener
channel.setMethodCallHandler({
    (call: FlutterMethodCall, result: FlutterResult) -> Void in
    guard call.method == "getDateTime" else {
        result(FlutterMethodNotImplemented)
        return
    }
    let formatter = DateFormatter()
    formatter.dateStyle = .short
    result(formatter.string(from: Date()))
})
```

Note that iOS and macOS use Swift, but the API is a bit different. There are two major configuration differences:

1. On iOS, the code is located in the `AppDelegate.swift` file. On macOS, the code goes in the `MainFlutterWindow.swift` file.
2. On iOS, the `FlutterViewController` object has a property called `binaryMessenger`:

```
let vc = window?.rootViewController as! FlutterViewController
let channel = FlutterMethodChannel(
    name: "fluttercompletereference.com/date-time",
    binaryMessenger: vc.binaryMessenger)
```

On macOS, the controller has the `messenger` property (instead of `binaryMessenger`):

```
let reg = flutterViewController.registrar(forPlugin: "DateAndTime")
let channel = FlutterMethodChannel(
    name: "fluttercompletereference.com/date-time",
    binaryMessenger: reg.messenger)
```

## 23.3 Creating Flutter plugins and FFI plugin packages

In section 2 – “*Writing platform-specific code*” we have seen how to write platform-specific code directly inside a Flutter application. This section teaches you how to create a reusable package with platform-specific code. Before moving on, let’s take a moment to review the terminology:

- **Dart package:** a package written only with Dart (*chapter 20 – Creating and maintaining a package*).
- **Flutter package:** a package written only with Dart that depends on the Flutter framework (*chapter 20 – Creating and maintaining a package*).
- **Plugin package:** a package written using Dart and other platform-specific languages such as Kotlin, C++, or Swift (*section 3.1 – Developing a federated plugin package*).
- **FFI plugin package:** a package whose API is written in Dart and combines different languages using `dart:ffi` (*section 3.2 – Flutter FFI plugin package*).

In practice, plugin and FFI plugin packages are “specialized” versions of Dart package that also use other platform-specific languages (such as Kotlin on Android). To create plugin packages, the Flutter team recommends to follow the federated plugin package structure. Don’t get confused by the terminologies: the federated adjective refers to a particular way of organizing files and directories in a plugin package.

### Note

A federated plugin package is a plugin package that follows Flutter’s recommended<sup>155</sup> architecture and design. We strongly advise you to follow these instructions, although they’re not required. In fact, you could also create a plugin package (a non-federated one) that follows your own file/folder structure and architecture.

The most significant advantage of following the federated plugin package structure is that it allows users to depend on a single package on all platforms. For example, users could just import your package once for all...

```
dependencies:  
  flutter:  
    sdk: flutter  
  my_plugin: ^1.0.0
```

---

<sup>155</sup> <https://docs.flutter.dev/development/packages-and-plugins/developing-packages#federated-plugins>

... rather than depending on every single platform-specific implementation:

```
dependencies:  
  flutter:  
    sdk: flutter  
  
  my_plugin: ^1.0.0  
  my_plugin_windows: ^1.0.0  
  my_plugin_macos: ^1.0.0  
  my_plugin_android: ^1.0.0  
  my_plugin_web: ^1.0.0  
  my_plugin_ios: ^1.0.0
```

Both examples are good, but the first one (an endorsed, federated plugin package) is better because it has a single dependency, and Flutter resolves everything automatically. It's less effort on the user side and also less maintenance. Let's move on to the next section to see an example of how a federated plugin package is created.

### 23.3.1 Developing a federated plugin package

In this section, we will create a federated plugin package to get the date from the host platform. You could avoid all of this and use Dart's `DateTime` class. However, we want to keep the example simple to focus on the package structure. As such, pretend that `DateTime` did not exist. To get started, create a directory called `date_plugin` and run the following commands inside it:

```
flutter create --template=plugin date_plugin  
flutter create --template=plugin --platforms=android date_plugin_android  
flutter create --template=plugin --platforms=ios date_plugin_ios  
flutter create --template=plugin --platforms=linux date_plugin_linux  
flutter create --template=plugin --platforms=macos date_plugin_macos  
flutter create --template=plugin --platforms=web date_plugin_web  
flutter create --template=plugin --platforms=windows date_plugin_windows  
flutter create --template=package date_plugin_platform_interface
```

Alternatively, you can create a new project using Android Studio or VS Code and choose the “plugin” template option. We have created many packages inside the `date_plugin` directory using different templates. This is how a federated plugin package has to be structured. You can identify three main package categories:

1. The “app facing” package (`date_plugin`). This is the package that users will depend on in their pubspec file and contains the public API of the package.

2. The “platform-specific” packages (`date_plugin_android`, `date_plugin_macos` and so on). These plugin packages contain the platform-specific implementation code that is called by the *app-facing* package.
3. The “platform-interface” package (`date_plugin_platform_interface`). This package is the glue between the *app-facing* and the *platform-specific* packages. This package defines a common interface so that platform-specific packages can implement the same APIs.

Now that the setup is ready, let’s move on and see how to configure each single package and how to link them all together. If you want to see a concrete and always up-to-date example of a plugin package, check out the official `url_launcher`<sup>156</sup> project from the Flutter team.

#### 23.3.1.1 The app-facing package

This package is the one that your users will import as a dependency in their pubspec file. Flutter will automatically detect all platform-specific implementations and resolve dependencies on its own. To do this, we need to configure the pubspec file of the `date_plugin` folder in this way:

```
flutter:  
  plugin:  
    platforms:  
      android:  
        default_package: date_plugin_android  
      ios:  
        default_package: date_plugin_ios  
      linux:  
        default_package: date_plugin_linux  
      macos:  
        default_package: date_plugin_macos  
      web:  
        default_package: date_plugin_web  
      windows:  
        default_package: date_plugin_windows
```

As you can guess from the syntax, the `plugin` property specifies the platforms supported by the plugin. Along with this, we also need to add each platform-specific package as a dependency, still in the same pubspec file:

---

<sup>156</sup> [https://github.com/flutter/packages/tree/main/packages/url\\_launcher](https://github.com/flutter/packages/tree/main/packages/url_launcher)

```

dependencies:
  date_plugin_android:
    path: ../../date_plugin_android
  date_plugin_ios:
    path: ../../date_plugin_ios
  date_plugin_linux:
    path: ../../date_plugin_linux
  date_plugin_macos:
    path: ../../date_plugin_macos
  date_plugin_platform_interface:
    path: ../../date_plugin_platform_interface
  date_plugin_web:
    path: ../../date_plugin_web
  date_plugin_windows:
    path: ../../date_plugin_windows

dev_dependencies:
  flutter_test:
    sdk: flutter

  plugin_platform_interface: ^2.1.4

```

Since we're working on a local package, we need to use `path` to specify its location. If the package will be published, we will have to publish all platform-specific packages as well and use the classic versioning system:

```

date_plugin_android: ^1.0.0
date_plugin_ios: ^1.0.0
date_plugin_linux: ^1.0.0
date_plugin_macos: ^1.0.0
date_plugin_platform_interface: ^1.0.0
date_plugin_web: ^1.0.0
date_plugin_windows: ^1.0.0

```

Inside `lib`, the package implementation is minimal. There only are two files:

- `lib/date_plugin.dart`. The classic file that exports all the files within the `src/` folder:

```
// Contents of the 'lib/date_plugin' file
export 'src/get_date.dart';
```

- `lib/src/get_date.dart`. The only method we need calls the platform plugin API:

```
// Contents of the 'lib/src/get_date.dart' file
Future<String> getDate() => DatePluginPlatform.instance.getDate();
```

Let's move to the next section to see how the `DatePluginPlatform` class is implemented.

### 23.3.1.2 The platform-interface package

The `date_plugin_platform_interface` package is the “glue” between platform-specific and Dart packages. No special settings in the pubspec file are required. Create two files:

- `date_plugin_platform_interface`: this base class defines the methods of the platform interface following the federated plugin structure. You should implement the singleton and the `_token` object in this exact way for consistency with Flutter’s best practices:

```
abstract class DatePluginPlatform extends PlatformInterface {
    DatePluginPlatform() : super(token: _token);
    static final Object _token = Object();
    static DatePluginPlatform get instance => _instance;
    static set instance(DatePluginPlatform instance) {
        PlatformInterface.verify(instance, _token);
        _instance = instance;
    }
    static DatePluginPlatform _instance = MethodChannelDatePlugin();
    // Put here all the methods with platform-specific implementations
    Future<String> getDate();
}
```

We have used the singleton exposed by this class in the previous section to give a concrete implementation of the `getDate` method. The class holds a default method channel object which can be changed (if needed) by platform-specific packages.

- `method_channel_date_plugin.dart`: Defines a default `MethodChannel` object that acts as a default option if the platform-specific package doesn’t provide one:

```
const _channel = MethodChannel('plugins.example.com/date_plugin');
const _error = "Couldn't get the date";

class MethodChannelDatePlugin extends DatePluginPlatform {
    @override
    Future<String> getDate() => _channel
        .invokeMethod<String>('getDate')
        .then((value) => value ?? _error);
}
```

When platform-specific packages don't use the native language (and thus a `MethodChannel` is not needed), `MethodChannelDatePlugin` is the default one.

### 23.3.1.3 The platform-specific packages

Now it's time to implement all platform-specific packages. Since the strategy is always the same for any platform, we will only cover two native implementations (Android and Windows) to show the similarities. In the `date_plugin_android` package, for example, the pubspec file has the following dependencies:

```
dependencies:
  flutter:
    sdk: flutter

  date_plugin_platform_interface:
    path: ../../date_plugin_platform_interface

flutter:
  plugin:
    platforms:
      android:
        package: com.example.date_plugin_android
        pluginClass: DatePluginAndroidPlugin
```

Create a single file inside `lib/` called `date_plugin_andorid`. It defines a new method channel and overrides the singleton from the plugin platform package with the new one:

```
const _channel = MethodChannel('plugins.example.com/date_plugin');

class DatePluginAndroid extends DatePluginPlatform {
  static void registerWith() {
    DatePluginPlatform.instance = DatePluginAndroid();
  }

  @override
  Future<String> getDate() => _channel
    .invokeMethod<String>('getDate')
    .then((value) => value ?? "Couldn't get the date");
}
```

Now we must move to the native side and retrieve the date from Kotlin. To do this, add a listener on the method channel in `android/src/main/kotlin` and open the `.kt` file. Find `onMethodCall` and implement it as follows:

```

if (call.method == "getDate") {
    res.success(LocalDateTime.now().toString())
} else {
    res.notImplemented()
}

```

Repeat this procedure for the remaining platforms. For example, for the Windows implementation, we need to configure the pubspec file in the `date_plugin_windows` package as follows:

```

dependencies:
  flutter:
    sdk: flutter

  date_plugin_platform_interface:
    path: ../../date_plugin_platform_interface

flutter:
  plugin:
    implements: date_plugin
    platforms:
      windows:
        pluginClass: DatePluginWindowsPluginCApi
        dartClass: DatePluginWindows

```

Similarly to what we did for Android, create a new `date_plugin_windows.dart` file inside `lib/`. It defines a new method channel object and attaches it to the singleton instance of the platform plugin:

```

const _channel = MethodChannel('plugins.example.com/date_plugin');

class DatePluginWindows extends DatePluginPlatform {
    static void registerWith() {
        DatePluginPlatform.instance = DatePluginWindows();
    }

    @override
    Future<String> getDate() => _channel
        .invokeMethod<String>('getDate')
        .then((value) => value ?? _error);
}

```

Now we must move to the native side and retrieve the date from C++. Open the `windows/` folder and modify the `date_plugin_windows_plugin.cpp` file and use the channel to send the response back to Dart. As you can see, the steps for both platforms are the same:

1. in the pubspec file, configure `plugin` directives and depend on the platform interface plugin package;
2. create a `DatePluginX` class that extends the plugin platform class and update the `instance` value of the singleton;
3. handle the message sent by `invokeMethod` by writing native code.

Each package has an `example/` folder (automatically generated when you created the project) with a Flutter application that can be run on the target platform to test the implementation. Of course, if you're working on the `date_plugin_macos` package for example, you have to run the application on a macOS machine to test the code.

### 23.3.2 Flutter FFI plugin package

In *chapter 9 – section 1.4 “Native interoperability with FFI”* we saw how the `dart:ffi` library can be used to call C functions from Dart applications. Instead, this section will see how to create an FFI plugin package to make the code reusable. Let's start using the `plugin_ffi` template. It generates a basic plugin package project that sums two numbers using a C function:

```
flutter create --template=plugin_ffi --platforms=windows,android demo
```

If you want to create the project with Android Studio or VS Code, choose the “FFI plugin” option. In this example, we've only added support for Windows and Android (of course, you could create an FFI plugin package for macOS, iOS, and Linux too). The newly generated `demo` project is a starting point for a Flutter FFI plugin with the following core folders:

- `lib/`: contains the Dart code that defines the plugin's API and uses `dart:ffi` to call native functions. As usual, all of your logic goes here.
- `src/`: contains the native source code and a `CMakeLists.txt` file for building the code into a dynamic library.
- platform folders (`windows/`, `android/`, `ios/`, etc..): contain configuration files for building and bundling the native code library with the platform application.

If you change (or add new) method signatures to the C code in the `src/` folder, you must update the Dart bindings accordingly. Rather than doing it manually, it's easier if you use Flutter's `ffigen` package to generate bindings for you:

```
flutter pub run ffigen --config ffigen.yaml
```

This command generates a very convenient Dart class that handles low-level FFI calls and types conversions for you. The `ffigen.yaml` file is already bundled with the generated project and is configured to output the class in the `lib` folder. For example, in the `demo` project we created, there are the following C functions defined:

```
// 'demo.h'  
FFI_PLUGIN_EXPORT intptr_t sum(intptr_t a, intptr_t b);  
FFI_PLUGIN_EXPORT intptr_t sum_long_running(intptr_t a, intptr_t b);
```

Dart bindings are generated in the `demo_bindings_generated.dart` file, which handles all the FFI type conversions and lookups for you. Rather than manually writing it, Flutter can generate this file for you:

```
class DemoBindings {  
  final ffi.Pointer<T> Function<T extends ffi.NativeType>(String symbolName)  
    _lookup;  
  DemosBindings(ffi.DynamicLibrary dynamicLibrary)  
    : _lookup = dynamicLibrary.lookup;  
  
  int sum(int a, int b) => _sum(a, b);  
  
  late final _sumPtr = _lookup<ffi.NativeFunction<ffi.IntPtr  
    Function(ffi.IntPtr, ffi.IntPtr)>>('sum');  
  late final _sum = _sumPtr.asFunction<int Function(int, int)>();  
  
  int sum_long_running(int a, int b) => _sum_long_running(a, b);  
  
  late final _sum_long_runningPtr = _lookup<ffi.NativeFunction<ffi.IntPtr  
    Function(ffi.IntPtr, ffi.IntPtr)>>('sum_long_running');  
  late final _sum_long_running =  
    _sum_long_runningPtr.asFunction<int Function(int, int)>();  
}
```

If you changed anything on the C code, just run the `ffigen` tool, and the above `DemoBindings` class will be re-generated to match the new signature. This class exposes a convenient API you can access anywhere in your library:

```
final DemosBindings _bindings = DemosBindings(_dylib);  
final DynamicLibrary _dylib = (){  
  // code here is generated by Flutter....  
}();  
  
/// A very short-lived native function.  
int sum(int a, int b) => _bindings.sum(a, b);
```

The `_bindings` and `_dylib` variables are also generated by Flutter when you create the project, so the effort on your side is minimal. The `sum` function is public, but it internally uses the binding class to call the native C function. Here are our recommendations:

1. Don't create an FFI plugin package from scratch because it's difficult to configure. Use the `flutter create` command, which prepares the project for you and takes care of platform-specific configurations.
2. Maintaining bindings on your own can be very difficult and error-prone. We recommend using the `ffigen` package. The code generation tool does all Dart-to-C conversions for you and provides a simple, type-safe API.
3. In our example, the sum of two values is a very trivial operation that takes a very small amount of time to execute. However, for longer tasks, run them on a separate isolate.

The `ffigen.yaml` file is used to configure the `ffigen` tool. The pubspec file contains the `plugin` key, which specifies the supported platforms. Our example only supports Android and Windows so the configuration looks like this:

```
flutter:  
  plugin:  
    platforms:  
      android:  
        ffiPlugin: true  
      windows:  
        ffiPlugin: true
```

The `flutter create` tool prepares everything so that you can start coding immediately without worrying about configurations or project management. The native build systems that are invoked by FFI plugins are:

- Gradle for Android (which uses the Android NDK for native builds);
- Xcode for iOS and macOS (which uses CocoaPods);
- Cmake for Windows and Linux.

Each platform folder contains build instructions as inline comments in their build configuration file.

## Deep dive: Type-safe native communication with pigeon

In this chapter, we have always configured a `MethodChannel` object to communicate between the host and the client. This approach requires a lot of configurations and is not type-safe. Calling and

receiving data depends on the host and client declaring the same types for messages to work. The Flutter team created the [pigeon](#) package, a type-safe alternative to using `MethodChannel` objects.

### Note

The [pigeon](#) package eliminates the need to match strings between host and client for the names and datatypes of messages. It also generates channels and messaging setups for you on both ends (Dart and the native platform).

Let's see an example of how to show the current date using platform-specific code and the [pigeon](#) package. Start by creating a new Flutter project called `demo_pigeon` and add the package as a dev dependency in the pubspec file:

```
dev_dependencies:  
  pigeon: ^10.0.0 # replace this with the latest version
```

Create a new Dart file outside of the `lib/` folder. We recommend you to follow the convention and create a top-level `pigeons/` folder that contains pigeon-related files. For example, we could create a `date_pigeon.dart` file in there and give it the following content:

```
// 'pigeon/date_pigeon.dart' file  
import 'package:pigeon/pigeon.dart';  
  
{@HostApi()  
abstract class NativeDateAPI {  
  @async  
  String currentDate();  
}}
```

The `HostApi` annotation is fundamental because it allows [pigeon](#) to automatically generate native code that communicates with Dart. Since we want to retrieve the current date, we have only added a single method called `currentDate`. There are a few things to point out here:

1. The file should not contain function bodies (only declarations). The class that is annotated with `HostApi` should always be `abstract`.
2. By default, the package will generate synchronous handlers. However, it generally is a good idea to create asynchronous handlers because they can be awaited by Dart. For this reason, we have added the `@async` annotation.

3. You should check the official documentation<sup>157</sup> to see which return data types are allowed.

In our case, we want to obtain the current date and return it as a `String`. Now that the pigeon file is ready, we can generate the platform-specific code. At the moment of writing this book, `pigeon` supports generating: Kotlin and Java for Android, Swift and Objective-C for iOS, Swift for macOS, and C++ code for Windows. For example, we could run this command to support native integration with Android and iOS:

```
flutter pub run pigeon \
--input pigeons/date_pigeon.dart \
--dart_out lib/date_pigeon.dart \
--swift_out ios/Runner/DatePigeon.swift \
--java_out ./android/app/src/main/java/io/flutter/plugins/DatePigeon.java \
--java_package "io.flutter.plugins"
```

The command generates a new file inside the `lib/` folder called `date_pigeon.dart`. This file has a user-friendly API that internally communicates with the host platform. For example, in our case, it is enough on the Dart side to use a `FutureBuilder` to wait for a future to complete:

```
final future = NativeDateAPI().currentDate();

@Override
Widget build(BuildContext context) {
    return FutureBuilder<String>(
        future: future,
        builder: (context, snapshot) {
            if (snapshot.hasData) {
                return Text(snapshot.data!);
            }
            return const CircularProgressIndicator();
        },
    );
}
```

The `NativeDateAPI` class used here is located in the `lib/` folder and is automatically generated by `pigeon`. So far, you have only had to configure a `.dart` file and run a code generation tool. You do not need to manually configure channels or native code

Let's move to the platform side:

---

<sup>157</sup> <https://docs.flutter.dev/platform-integration/platform-channels?tab=type-mappings-java-tab#codec>

- Android. Navigate to `android/app/src/main/kotlin` and open the `MainActivity.kt` file. You don't have to manually configure a `MethodChannel` object because the `pigeon` package already configured everything. Just import the `DatePigeon` class and use it:

```

import io.flutter.plugins.DatePigeon // <-- Generated by 'pigeon'
import io.flutter.embedding.android.FlutterActivity
import io.flutter.embedding.engine.FlutterEngine
import io.flutter.plugins.DatePigeon.Result
import java.time.LocalDate

class MainActivity: FlutterActivity() {
    private class DateTimeClass: DatePigeon.NativeDateAPI {
        override fun currentDate(result: Result<String>) {
            result.success(LocalDate.now().toString())
        }
    }

    override fun configureFlutterEngine/flutterEngine: FlutterEngine) {
        super.configureFlutterEngine(flutterEngine)

        // Add the following lines
        val messenger = flutterEngine.dartExecutor.binaryMessenger
        DatePigeon.NativeDateAPI.setup(messenger, DateTimeClass())
    }
}

```

This is similar to what we did in *Section 2.2 – Android implementation*. However, here we're working with a type-safe API and don't have to configure the communication channels.

- iOS: Navigate to `ios/Runner` and open the `AppDelegate.swift` file. As it happened with Android, you don't have to configure the channel manually. The `pigeon` package configured everything for you, so just use the `NativeDateAPI` object. To keep the code clean, make a separate (and private) class:

```

private class NativeDateImplementation: NativeDateAPI {
    func currentDate(completion: @escaping (Result<String, Error>) ->
Void) {
        let formatter = DateFormatter()
        formatter.dateStyle = .short
        completion(.success(formatter.string(from: Date.now())))
    }
}

```

Use this class inside the `AppDelegate.application` class to set up the channel:

```
GeneratedPluginRegistrant.register(with: self)

let fvc = window?.rootViewController as! FlutterViewController
let api = NativeDateImplementation()
NativeDateAPISetup.setUp(binaryMessenger: fvc.binaryMessenger, api: api)

return super.application(...)
```

This is similar to what we did in *Section 2.3 – iOS implementation*. However, here we are working with a type-safe API and don't have to configure the communication channels.

The code differs on Android and iOS, but the implementation follows the same structure. In the same way, you have to use C++ on Windows, but the architecture and the coding pattern are the same. In summary, do the following to make the communication between Flutter and the host platform type-safe:

1. Add the [pigeon](#) package as a [dev\\_dependency](#).
2. Create a top-level [pigeons/](#) folder and put the “pigeon templates” there. Remember to use the [@HostApi](#) annotation on abstract classes and the [@async](#) annotation on methods.
3. Run the code generation tool and specify the platforms you want to support.
4. For each platform you support, add the native code to the platform-specific folder of your Flutter project.

We recommend using [pigeon](#) where possible because it is a type-safe way of communicating with the native platform. It also automatically configures the Dart and the native code so you won't have to deal with platform-specific configurations manually.

# Appendix – Performance and profiling

## A.1 Working with DevTools

DevTools is a debugging and performance profiling tool suite for Dart and Flutter applications. It is designed to help developers identify and fix issues with their code, optimize performance, and improve the overall quality of their applications. Once the application is running in debug mode, you can launch DevTools in various ways:

- On Android Studio, it is enough to have installed the Flutter plugin. When the application is run, the Dart logo will appear in the “Run” window: click on it and the Android Studio will start the DevTools instance for you.

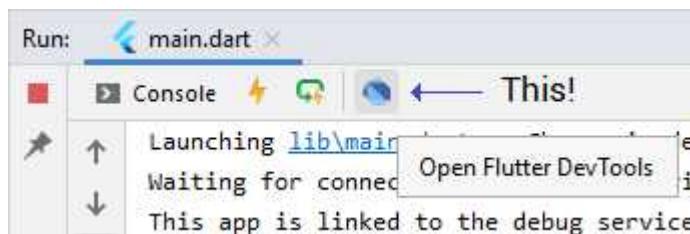


Figure A.1: The Flutter DevTools button on Android Studio.

- On VS Code, it is enough to have installed the Flutter extension. When the application is run, launch *Dart: Open DevTools* from the command palette and click “Always open” if asked.
- If you prefer working with the terminal, you just need to execute `flutter run` and after a few moments you'll see a message that looks like this:

```
An Observatory debugger and profiler on XXX is available  
at: http://127.0.0.1:XXXX/XXXXXXXXXXXX/
```

Open the URL and you will be redirected to the DevTools instance.

DevTools runs on a web browser (it is entirely written with Dart and Flutter!). As soon as you stop the execution of the application, the DevTools instance also terminates. For web applications, some features (such as “app size inspector” and “memory inspector”) are not available. DevTools basically is a “container” for a large set of independent tools, which are grouped into tabs. Let's briefly see all of them.

### A.1.1 Flutter inspector

The “Flutter inspector” is a powerful tool for exploring the widget tree. This tab updates in real-time when the running application changes the widget tree. For example, this is the inspector view of the classic counter application:

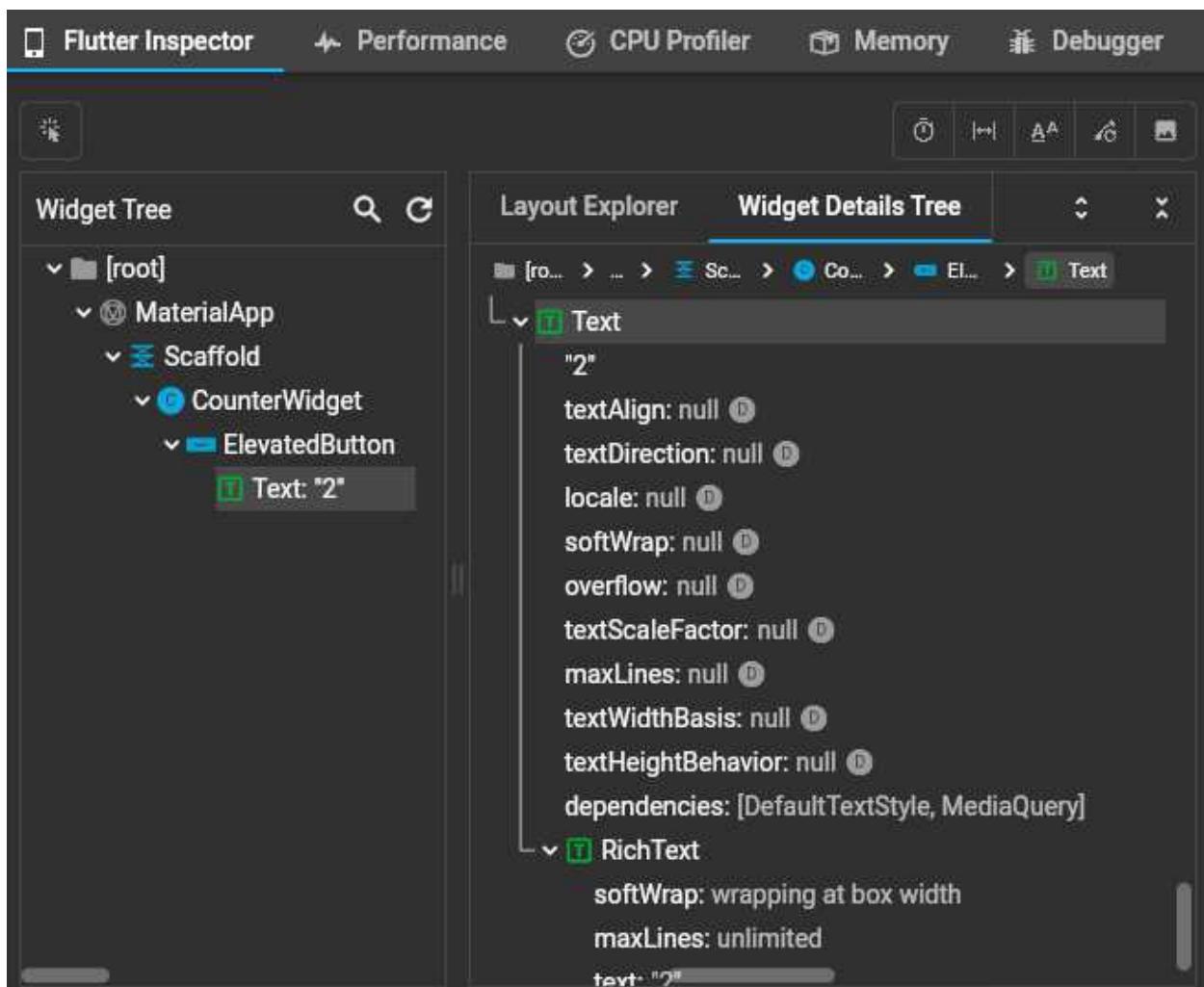


Figure A.2: Widget tree details of a counter application.

On the left, you can see a detailed representation of the widget tree. When you click on a widget, the box on the right refreshes to show you all the properties. In *Figure A.2* for example you can see all the properties of the `Text` widget in the tree. The “widget tree” column has two buttons in the top-left corner:

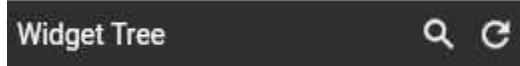


Figure A.3: The “Search” and “Refresh tree” buttons.

The *search* icon is used to look for a specific widget in the tree. The *refresh* icon instead refreshes the widget tree representation after a rebuild. For example, if you increased the counter, you would need to click the refresh button to update the widget tree to the latest status. In this example, since the `counter` variable is stored in the `CounterWidget` state class, you can hover the widget with the mouse and inspect its state:



Figure A.4: Hovering on the `CounterWidget` in DevTools shows the details of its state class.

Next to the widget details tab, you can find the *Layout Explorer*. It gives you size information about the selected widget, padding, and constraints (if any). You will also notice the presence of some visual debugging buttons (referred as *V. D. buttons*) in the top-left corner of the page:



Figure A.5: Visual debugging buttons of the Flutter inspector tool.

Each of these buttons enables a special feature in the running application:

1. **Slow animations.** When enabled, this button runs animations 5x slower than usual. This is useful when you need to carefully observe an animation that does not seem to execute as you would expect.

2. Show guidelines. When enabled, Flutter renders guidelines around your widget to display boxes, alignments, scroll views, spacers and paddings. This is usually used with the layout explorer to find, for example, undesired padding around a widget.
3. Show baselines. A “baseline” is a horizontal line used to position text. When enabled, this feature shows all `Text` widgets baselines to see how the various strings are aligned.
4. Highlight repaints. When enabled, Flutter draws a colored border around the render box of a widget. The color changes whenever that box is repainted. When you see a large rainbow of colors, your application repaints the widget many times. For example:



Figure A.6: Repaint highlights of a box and a `CircularProgressIndicator` widget.

The borders of the outer box do not change color, so the widget is only painted once. The `CircularProgressIndicator` widget at the center is animated; consequently, the box's border color constantly changes. In some cases, if you see too many boxes whose border color changes, there might be a performance problem. We will discuss this issue in more detail in *Section 2.3.1 “Use `RepaintBoundary` judiciously”*.

5. Oversized images. When enabled, Flutter highlights images that are too large by inverting colors and flipping them vertically. These images use more memory than needed and can cause poor performance. In general, to fix this issue, you should resize or optimize the asset itself and (if you haven't already) make sure to define its dimensions explicitly:

```
const Image(
  image: AssetImage('assets/image.png'),
  width: 100, // very important
  height: 75, // very important
);
```

Thanks to `width` and `height`, the engine will decode the image using less memory.

The Flutter inspector should be used when the application is run in debug mode. It does not track runtime performance so you there is no need to run the application in profile mode.

### A.1.2 Performance view

The performance view is used to measure the performance of your Flutter application, which should always execute in profile mode with this tool. The main reason is that, in debug mode, the frame rendering rate is lower than normal and thus not representative of release performance. The most interesting part of this tool is the frames chart at the top:

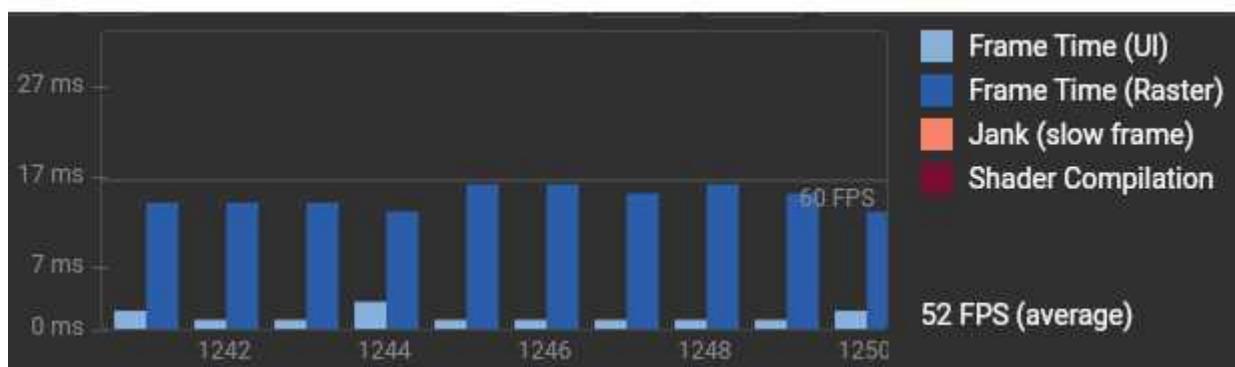


Figure A.7: The frames chart in the Performance view of a DevTools instance.

While you use the application, the chart updates in real-time with colored bars that highlight the different portions of work made while rendering a frame. The grey horizontal line indicates that, in order to run the application at 60 frames per second, frames should render in (approximately) less than 16 milliseconds. Each colored bar has a different meaning:

- **Frame Time (UI).** The UI thread executes all the Dart code within the Dart VM. When your application is run, the UI thread creates a lightweight object with painting commands and sends them to the raster thread. Make sure to never block this thread.
- **Frame Time (Raster).** The raster thread runs graphics code from the Flutter engine and talks with your device's GPU. This is where the graphic engine runs. If this thread is slow, then it's due to some Dart calls you're making.
- **Jank (slow frame).** When a frame takes more than 16 milliseconds to complete, it means that the event loop didn't process all the events on time. In this case, the frame is said to be

*janky*. In some cases, certain devices can run at 120 frames per second so the time budget lowers even more. Minimizing widgets rebuilds, offloading time-consuming tasks to other isolates and using asynchronous functions are great options to get rid of jank.

- **Shader Compilation.** This happens when the engine needs to compile some shaders the first time they're run. In other words, an animation might be janky the first time it runs but then it will always be smooth.

For the best performance, your frame chart should never have red bars. A simplified version of the frame chart can also be displayed directly within the application while it's running. It appears at the top as a semi-transparent overlay, and it hovers over your entire application (without interfering):

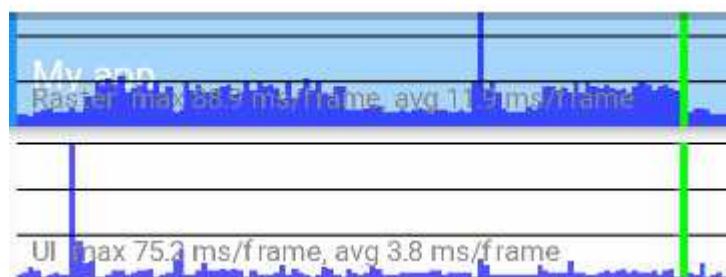


Figure A.8: The performance overlay on an Android application.

In general, when the raster thread takes too long to execute, it means that your Dart code is taking too much time to run an operation. In this case, try to see if you can optimize it. When you click on a frame, you can see a detailed timeline report in the Timeline chart (below the frame chart).

The timeline chart can be panned and zoomed, which is useful when you see jank on a frame and you want to try identifying where the problem is. The main goal of timeline events and the frames chart is to help you finding those areas that can be improved performance-wise.

### Note

Other than running the application in profile mode, you should also use this tool with different devices. Don't only analyze performance on a last generation and powerful device. When possible, try to see how your application behaves on (relatively) old and low-end devices and how much jank it produces.

### A.1.3 CPU profiler

The CPU profiler view is available for CLI Dart projects and Flutter applications that don't run on the web. If you want to analyze the CPU performance of a web application, use Chrome DevTools<sup>158</sup>. To start recording a CPU profile click Record and, when done, press Stop. The tool will automatically generate a report in three different tabs:

- Flame chart. It's a top-down stack trace where the width of each stack frame represents the time spent by the CPU on a task. Those stack frames that take too much time might be worth investigation for possible performance improvements.

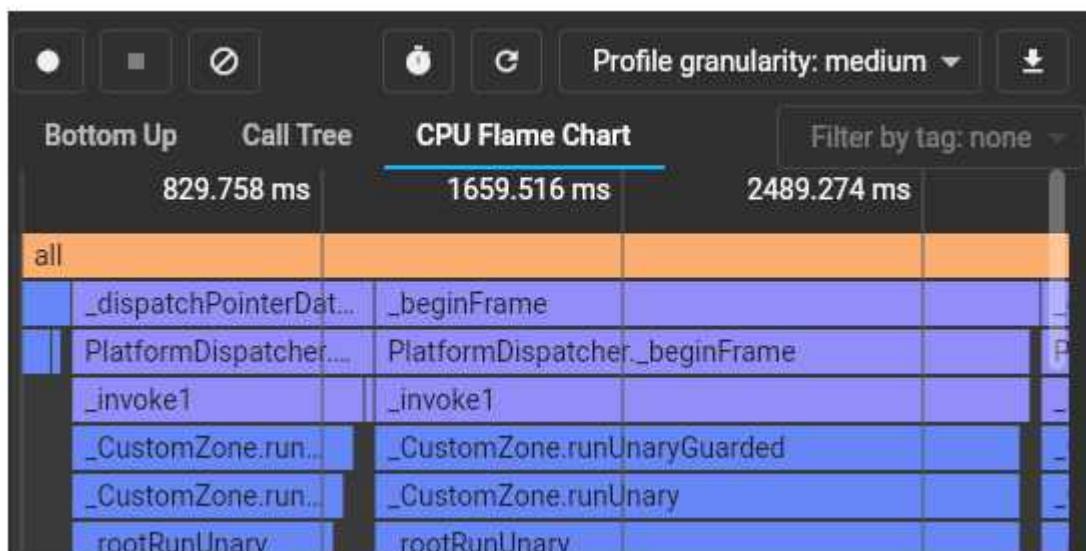


Figure A.9: The CPU profiler flame chart.

The orange stripe labeled with “all” represents the total profile duration. It basically is the elapsed time between the “Record” and “Stop” button presses.

- Call tree. A detailed method trace for the profiled sample. It's a top-down representation to see the method calls flow and how long they took to execute.
- Bottom up. Each top-level method in this table is the last method called in the call stack of the recorded CPU sample. Each entry can be pressed to see the caller's hierarchy.

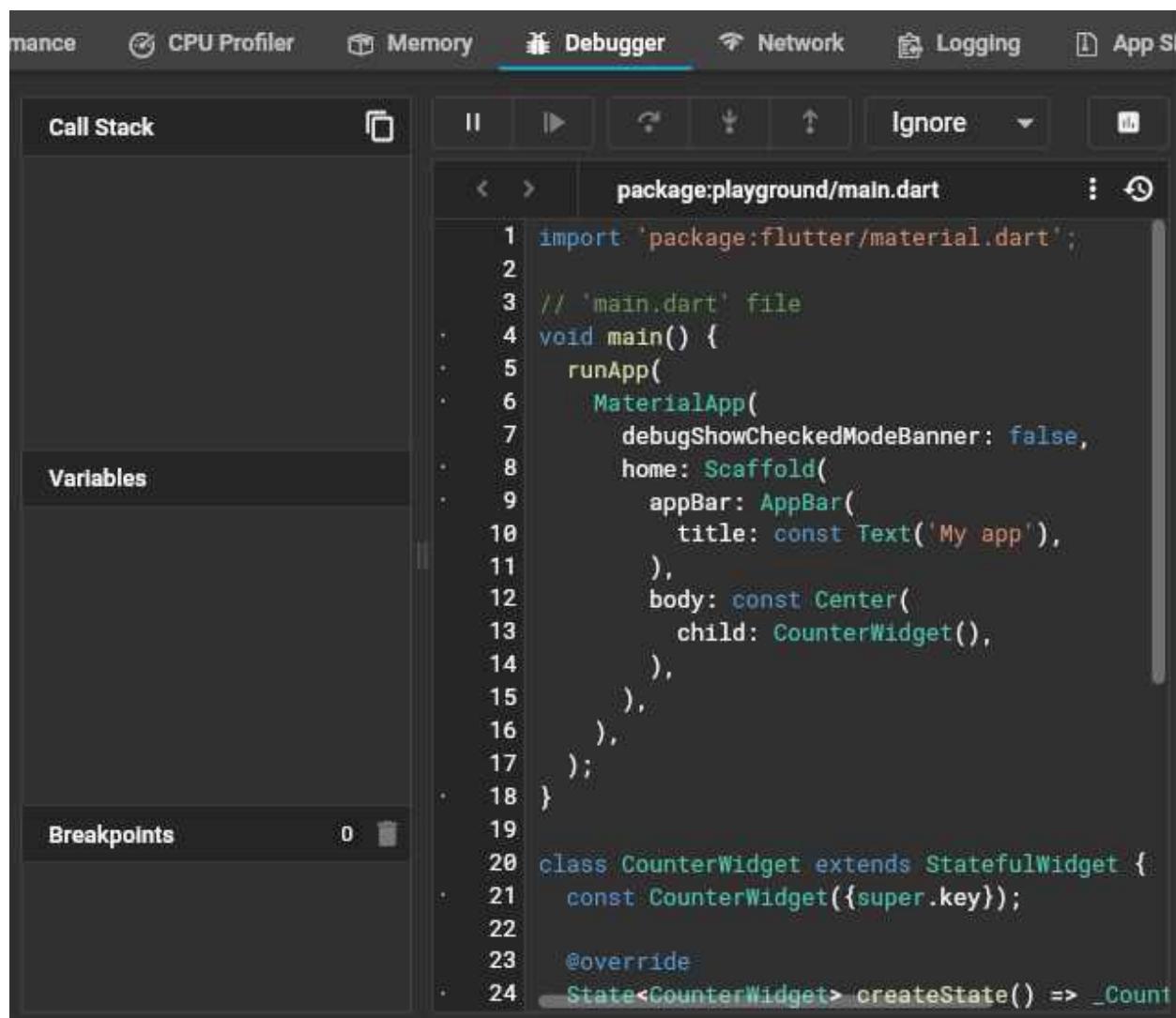
---

<sup>158</sup> <https://developer.chrome.com/docs/devtools/>

This tool should always be used on an application built in profile mode to get closer performance to a release build.

#### A.1.4 Debugger

The debugger works with all Dart and Flutter applications. This is not visible if DevTools is launched on VS Code because the IDE already has its built-in debugger. When you open the debugger tab, you should see the source of your application's main entry-point



The screenshot shows the Flutter DevTools interface with the 'Debugger' tab selected. The main area displays the source code of the main.dart file. The code defines the main entry point of the application, which runs an MaterialApp with a Scaffold containing a CounterWidget. A CounterWidget is a StatefulWidget that creates a State<CounterWidget> object named \_Count. The code is color-coded to highlight keywords and syntax. On the left, there are tabs for 'Call Stack', 'Variables', and 'Breakpoints'. The 'Breakpoints' tab shows 0 breakpoints.

```
import 'package:flutter/material.dart';
// 'main.dart' file
void main() {
  runApp(
    MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Scaffold(
        appBar: AppBar(
          title: const Text('My app'),
        ),
        body: const Center(
          child: CounterWidget(),
        ),
      ),
    ),
  );
}
class CounterWidget extends StatefulWidget {
  const CounterWidget({super.key});
  @override
  State<CounterWidget> createState() => _Count
```

Figure A.10: The debugger view of Flutter DevTools.

The top-right icon allows you to open the File Explorer to visualize other Dart files. You can of course set breakpoints just by clicking the left margin, next to the line number you’re interested in. When the application encounters a breakpoint, it pauses there and populates the views on the left.

You can use the five buttons above the code view to resume the execution, step into a method invocation, step over a method invocation, or step out of the current method. When the DevTools window is large enough, since the tool is responsive, you also see the text:



Figure A.11: Extended view of the action buttons in the debugger.

To adjust the stop-on-exceptions behavior you can click the *Ignore* dropdown button and override the configuration.

### A.1.5 Network and logging views

The network and logging views are used to respectively inspect network calls and collect events from the Dart runtime and Flutter. When you make HTTP, HTTPS, or web socket calls from Dart or Flutter, the Network tab keeps track of a log. For example:

Network				Search
Method	Url	Status	Type	Overview
GET	https://google.com	200	html	Request url: https://google.com
GET	InternetAddress	101	ws	Method: GET
GET	https://www.google.com	200	html	Status: 200
GET	InternetAddress	101	ws	Port: 56127
GET	https://google.com	200	html	Content type: [text/html; charset=ISO-8859-1]
GET	InternetAddress	101	ws	

Figure A.12: The network tab of a DevTools instance.

Network traffic is recorded by default, but you can also pause and resume it with the buttons on the top-left corner. “Search” and “filter” controls are helpful when your application makes many calls and you’re looking for a specific one. For example, if you click the “Filter” button next to the search bar, you will get this dialog:

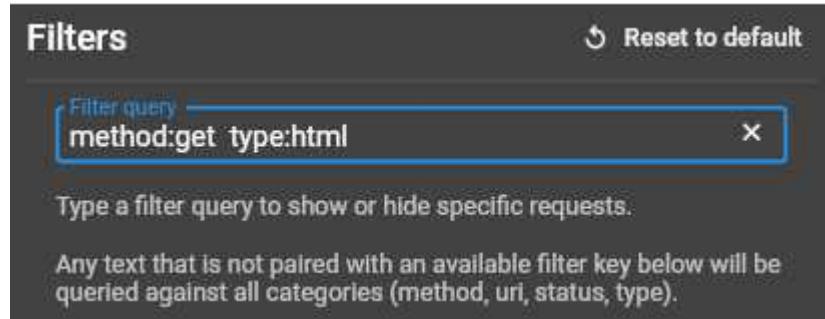


Figure A.13: The filter dialog of the Network tab.

In *Figure A.13* we filter all network calls to only display GET request whose `type` is HTML. In other words, we only want to see entries of fetched HTML pages. Filters are separated by a space and can only contain the following keywords:

- `method`, for the HTTP method type;
- `status`, for the HTTP status (such as 200, 404, or 500);
- `type`, for the request type (such as json, html, txt or ws).

To see any other kind of event (not network-related), you can use the Logging view. It reports events from the garbage collector, the Flutter framework, stdout or stderr calls, and custom entries from external applications.

## A.1.6 Memory view

Dart objects are created with constructors and live in a memory area called `heap`, which is managed by the Dart VM. In particular, the VM allocates the memory for you and deallocates it automatically when objects are not needed anymore.

### Note

Memory management and garbage collection are complicated subjects that go beyond the scope of this book. As such, we will not go into the details on the “Memory view” tab. We will just provide an overview and some links in case you wanted to learn more.

Dart uses a garbage collector to find objects on the heap that the program does not use anymore and can be removed. Note that a garbage collector is not able to prevent all kinds of memory leaks, so you still have to pay attention to some coding patterns. In short, a GC periodically looks for all

unreachable objects in the memory and disposes them. In some cases, objects may be unused but their reference isn't, so they are not recognized as "garbage". For example, this code might cause a memory leak:

```
void myFunction() {  
  final test = Test();  
  final callback = () => test.doSomething();  
  
  // Imagine that this function is defined somewhere in your code  
  addHandler(callback);  
}
```

The scope of the `test` object is limited to the function body, but its reference is stored in the closure context and passed to the `addHandler` function. In other words, `test` will not be garbage collected as long as `callback` is reachable in the code. This example is (relatively) easy, but the leak is tough to spot.

### Note

The `leak_tracker`<sup>159</sup> package, created and maintained by the Dart team, was created to help you find memory leaks. It is very helpful because memory leaks are hard to spot by eye. This package can also automate the process of finding memory leaks via regular unit or widget testing.

The memory view of the DevTools suite helps you investigating garbage collection events, memory allocations, and much more. The page is divided into two main areas:

1. **Expandable chart:** It shows a high-level and real-time chart of memory allocation and events. For example, it shows when the VM schedules a garbage collection event or when a new memory is requested/retained.
2. **Tab:** The "profile" tab shows the current memory allocation status. The "diff" tab is used to investigate the memory management of a feature (for example, you could see the memory consumption of a worker isolate). The "trace" tab is used to check the memory management of a series of classes.

---

<sup>159</sup> [https://pub.dev/packages/leak\\_tracker](https://pub.dev/packages/leak_tracker)

The expandable chart at the top of the memory tab is updated in real-time and shows the state of the Dart or Flutter application memory. The horizontal axis is for timeline events, while the vertical axis is for the time (the chart refreshes every 500ms by default). This is an example of how it looks:

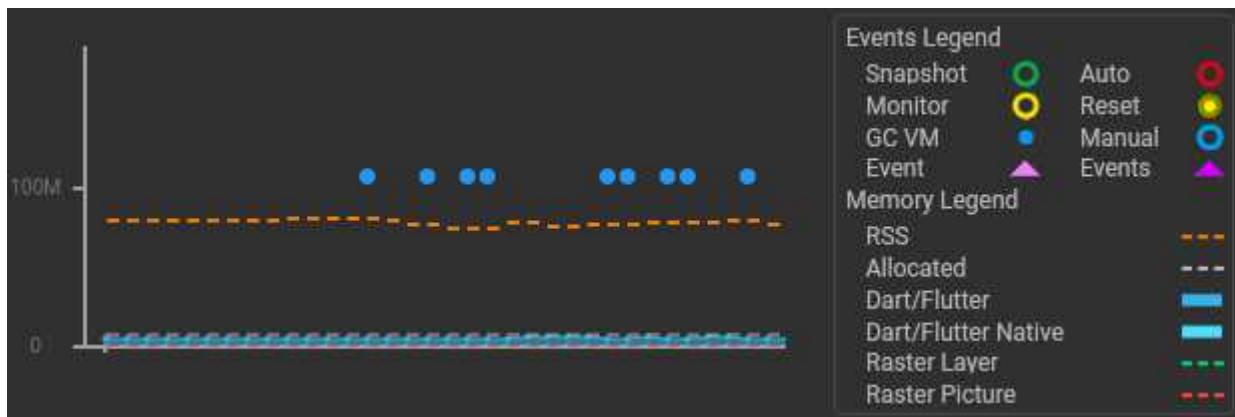


Figure A.14: The memory chart at the top of the the “Memory” tab.

One notable difference: “Dart/Flutter” is about Dart objects on the heap while “Dart/Flutter native” is about native objects that are exposed from the platform to the VM. For more information about memory management and low-level bits, check the official documentation.

## A.2 Performance best practices

Even if Flutter applications are very performant by default, avoiding common pitfalls guarantees an even more enhanced runtime experience. In this section, we have decided to gather a series of best practices from both the Flutter team guidelines and our professional expertise.

### A.2.1 Minimize widgets rebuilds

Reducing rebuilds probably is one of the most important performance optimizations you can make. The `build` method should only return widgets and make no time-consuming computations because it could be invoked very often. To reduce the number of rebuilds on the widget tree, ensure to:

1. use `const` constructors as much as possible;
2. do NOT use functions to return widgets (prefer a `StatelessWidget` instead).

In *chapter 10 – Section 3.2 “Performance considerations”* we have discussed in detail why constant constructors and stateless widgets are important to reduce the number of rebuilds on the various

trees. We also want to remind you that when a widget never changes but cannot be `const` because of external dependencies, you can manually cache it in a stateful widget like this:

```
class _MyWidgetState extends State<MyWidget> {
    late var cached = LargeAndImmutable(title: widget.title);

    @override
    void didUpdateWidget(covariant CounterWidget oldWidget) {
        super.didUpdateWidget(oldWidget);

        if (oldWidget.title != widget.title) {
            cached = LargeAndImmutable(title: widget.title);
        }
    }

    @override
    Widget build(BuildContext context) {
        return Column(
            children: [
                const Text('Hello!'),
                cached,
            ],
        );
    }
}
```

Since the widget is created within the state class, the reference never changes and persists across rebuilds. Consequently, `cached` won't be rebuilt more than once. Remember that the traversal to rebuild all children stops when the same instance of the child widget as the previous frame is found. This is the same idea as using the `child` parameter of a builder. For example:

```
AnimatedBuilder(
    animation: controller,
    builder: (context, child) {
        return Transform.rotate(
            angle: controller.value * 2 * math.pi,
            child: child,
        );
    },
    child: MyWidget(
        width: something.width,
        color: Colors.green,
        child: const Text('Hi!'),
    ),
);
```

The `child` is always passed to the builder function and does not get rebuilt on each animation tick. When possible, use the `child` parameter when you have a widget that does NOT depend on the animation status. In addition:

- prefer making `setState` calls on leaf widgets;
- avoid `setState` calls too high in the tree because it would schedule a rebuild for the current widget and its whole subtree;
- prefer `InheritedWidget` and listenable classes (like `ChangeNotifier`) instead of `setState` when multiple widgets in various locations of the tree need to react to state changes. This allows a more granular rebuild control.

See *chapter 12 – “State management”* to learn about `setState` and inherited widgets.

### A.2.2 Good practices for scrollable widgets

When it comes to scrollable widgets, lazy loading is easy to implement and also very powerful. We have already seen in *chapter 14 – Section 3 “Scrollable widgets”* that scrollable widgets can lazily build their children using the `builder` named constructor. It should always be used when you know the total list length in advance. For example:

```
ListView.builder(  
    itemExtent: 35,  
    itemCount: itemList.length,  
    itemBuilder: (context, index) {  
        return Text('${itemList[index]}');  
    },  
);
```

This constructor creates children on demand, meaning that `itemBuilder` is only called for currently visible children. The `itemCount` property is used to improve the ability to estimate the maximum scroll extent. Two other important considerations are:

1. When possible, define the `itemExtent` property. It sets how much space each child takes in the scrolling direction. If a list is horizontally scrolled for example, this property defines each child's height. When a value for `itemExtent` is given, the scrollable widget is more efficient because it doesn't have to compute the extent of each child before laying it down.
2. The `itemBuilder` callback should always create new widget instances. Don't use a builder that returns a previously-constructed widget. When children are created in advance, using the `ListView` default constructor is more efficient.

Another very important suggestion is to avoid the usage of `shrinkWrap` when nesting scrollable widgets. Prefer slivers instead. The problem is that `shrinkWrap` forces Flutter to evaluate the entire list of widgets, even if they are lazily loaded. For example:

```
class MyWidget extends StatelessWidget {
  const MyWidget({super.key});

  @override
  Widget build(BuildContext context) {
    return ListView(
      children: [
        ListView.builder(
          shrinkWrap: true, // Bad!
          itemExtent: 50,
          itemCount: itemList.length,
          itemBuilder: (_, index) => Text('${itemList[index]}'),
        ),
        ListView.builder(
          shrinkWrap: true, // Bad!
          itemCount: otherList.length,
          itemBuilder: (_, index) => Text('${otherList[index]}'),
        ),
      ],
    );
  }
}
```

In general, `shrinkWrap` avoids errors when a scrollable widget is nested inside another one. The problem in the above example is that the inner list (even if it's created with a `builder`) is entirely evaluated because of `shrinkWrap`. In other words, lazy loading does not happen anymore: all 500 children are created when the widget is inserted in the tree. This becomes even more problematic if the list also has animations.

### Note

Even if you'll never scroll anything, all 500 children of the inner list would still be created. The `builder` constructor loses its ability to create children lazily and it's forced to render everything (even those widgets that will never appear on the screen).

Using `shrinkWrap` on large lists (especially if the children contain animations) has a good chance of creating jank on frames. Prefer using `CustomScrollView` with slivers instead, which gives you the same result with better performance. For example:

```

CustomScrollView(
  slivers: [
    SliverFixedExtentList(
      itemExtent: 50,
      delegate: SliverChildBuilderDelegate(
        (_, index) => Text('${itemList[index]}'),
        childCount: itemList.length,
      ),
    ),
    SliverList(
      delegate: SliverChildBuilderDelegate(
        (_, index) => Text('${itemList[index]}'),
        childCount: itemList.length,
      ),
    ),
  ],
),

```

This is an equivalent (but more performant) version of the previous example that uses lazy loading. The same is also valid for grids: avoid `shrinkWrap` on `GridView` and prefer its sliver counterpart. Slivers and `CustomScrollView` are covered in *chapter 14 – Section 3.3 “Slivers”*.

### A.2.3 Widgets usage recommendations

Lowering the frame render time below 16 milliseconds on mobile devices improves the battery life and reduces the chances of creating janky frames (even on slower devices). In addition to the good practices we have already discussed, there are more recommendations from the documentation:

- Use the `Opacity` widget only when needed. If you need to add transparency to an image or to a `Color` value, prefer using their opacity properties other than wrapping them with the `Opacity` widget.
- Sometimes you might need to load an image from the web and, when it's ready, you may want to apply a fade-in transition. In this case, don't animate an `Opacity` widget. Prefer the `FadeInImage` widget, which efficiently applies opacity using the GPU's fragment shader. Alternatively, use `AnimatedOpacity` but avoid animating the `Opacity` widget.
- Use clippers like `ClipRect` or `ClipRRect` only when needed. In general, “clipping” may be a costly operation in terms of performance. Use the `borderRadius` property (if possible) to create a rectangle with rounded borders. For example, avoid using a clipper on a `Container`: instead, use its `decoration` property to set the border-radius.

There also is another important thing to mention. In some cases, Flutter could call `saveLayer`. It is an expensive function that implements various visual effects. Even if you don't directly call it in your code, some other widgets or packages you're using might still invoke `saveLayer` behind the scenes (and you don't know about it). If your application calls this function too often, you might start to see janky frames. From this, you understand that `saveLayer` calls should be minimized<sup>160</sup> to keep a high frame rate.

### Note

Calling `saveLayer` allocates an offscreen buffer for drawing contents, but this operation might trigger a “render target” switch. On mobile GPUs, this is quite bad especially if multiple buffers are created. This function is used, for example, when displaying various shapes with transparency that might overlap.

To see how often your application calls `saveLayer`, you need the *Performance* tab of a DevTools instance and look for “`saveLayer`” events in the timeline. This is the best way to check where the method is called. Regardless, there are a few cases where you know that `saveLayer` is called:

- when using the `ShaderMask` or `ColorFilter` widgets;
- when using `Text` with an `overflowShader`;
- when using `Chip` with a particular disabled color alpha;
- when using clippers whose `behavior` property is set to `Clip.antiAliasWithSaveLayer`.

In the `paint` method of a custom painter, you can explicitly call `canvas.saveLayer` but there are very few cases where this method call is needed. Most of the time, especially if you're unsure about its purpose, it's better to not use `canvas.saveLayer`.

#### A.2.3.1 Use `RepaintBoundary` judiciously

In some cases (often when animations are involved), Flutter might end up repainting more layers than needed. Note that we are not talking about widget tree rebuilds. We are talking about repaints made by the Flutter engine to “refresh” the pixels on the screen. Let's make an example to give more context:

---

<sup>160</sup> <https://docs.flutter.dev/perf/best-practices#use-savelayer-thoughtfully>

```
// Assume this is placed inside a Scaffold with an AppBar
Row(
  mainAxisSize: MainAxisSize.min,
  children: [
    RotationTransition(
      turns: controller,
      child: const Text('Wohooo!'),
    ),
    const SizedBox(width: 20),
    const Text('Hello!'),
  ],
),
)
```

On the widgets side, all is good. We use `const` constructors where possible and the animation is performed efficiently by a `RotationTransition` widget. On the engine side, we need to evaluate how often layers are being repainted. Let's open a DevTools instance and click the *Highlight repaints* button in the Flutter inspector tab. When the application is run, this is the result:

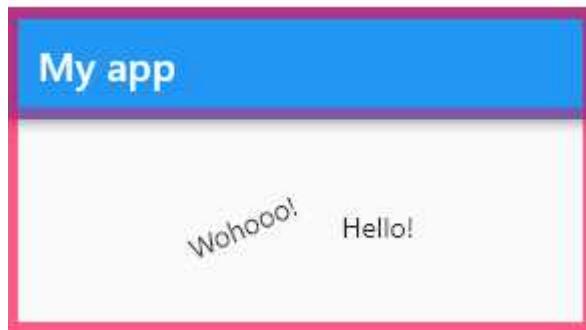


Figure A.15: Layer repaints highlighted via DevTools.

Notice a rainbow of colors that quickly changes all around the application. Even if most widgets are not being rebuilt, Flutter still requires render objects to repaint because the animation shares the same layer as the other UI parts. Let's fix this issue:

```
RepaintBoundary( // Wrap the animated widget with 'RepaintBoundary'
  child: RotationTransition(
    turns: controller,
    child: const Text('Wohooo!'),
  ),
),
```

The `RepaintBoundary` widget creates a separate layer for the given child to isolate repaints. If you try to run DevTools again with this fix, you will see that the rainbow of colors only surrounds the

animated widget now. In other words, only the animation layer is repainted, but the rest of the application is not:

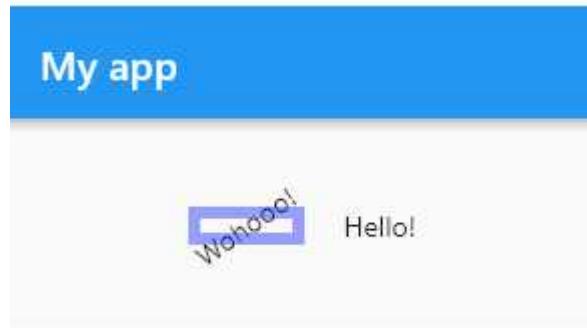


Figure A.16: Thanks to `RepaintBoundary`, layer repaints are limited to the animation.

This performance improvement reduces the number of layer repaints made by the engine. Under the hood, the `RepaintBoundary` widget creates a new `RenderObject` that always has a dedicated layer (separated from the others). There are a few considerations to make:

- Don't confuse layer repaints with widget rebuilds. In this section, we talked about a strategy that minimizes how many times `render objects` are repainted. This is a different thing from the widget rebuild system. Constant widgets will never be rebuilt, but the engine might still decide to update the associated render object.
- The `RepaintBoundary` widget has a small CPU and memory cost that can add up if used too often (and cause more problems than it solves<sup>161</sup>). For this reason, do not wrap all of your animations with `RepaintBoundary`. Use DevTools to see which parts repaint too often and also check for jank in the performance view. Note that a widget might already internally use `RepaintBoundary` so do not use it more than once.
- Layers repaints are not only caused by animations. You should run DevTools and see which parts repaint too often. If you see lots of repaints on a certain area, but your application still runs at 60 fps or more, you can avoid using `RepaintBoundary` to save some memory and CPU cycles.

---

<sup>161</sup> <https://www.youtube.com/watch?v=cVAGLDuc2xE> (RepaintBoundary – Widget of the Week)

- The `RepaintBoundary` child should only contain the portion of the widget tree that you want to be isolated in its own layer. In fact, in our example, it only wraps the animation and not the entire `Row`.

In summary, `RepaintBoundary` is very helpful when creating dedicated layers for certain portions of the widget tree. Specifically, it can save layer repaints and improve the rendering time. However, it adds small CPU and memory overhead so you should use it only when needed.

THE END!

*We hope this book has been a valuable resource for you on your journey to becoming a skilled Dart and Flutter developer :)*