

Learn Google Flutter Fast

65 Example Apps
MARK CLOW



1. Table of Contents

1. Table of Contents

2. Welcome

Introduction

Acknowledgements

Purpose

Work

Disclaimer

Revisions

Source Code

Location

Example & Exercise Names

Example – ‘gesture_app’

One File

3. The Big Picture

Introduction

Terminology

Compiler

Native and Non-Native Compilers

Cross-Platform Mobile Application Development

Mobile Applications

Before Cross-Platform Mobile Application Development

Early Cross-Platform Development Tools

Development Tools That Used Native Libraries

Development Tools That Didn’t Use Native Libraries

Modern Cross-Platform Development Tools

React Native

Google Flutter

Conclusion

4. Introduction to Dart

Introduction

Platforms

1. Within a Web Browser
2. As Interpreted Application
3. As Native Application

Dart SDK

1. Command-Line Tools
2. Command-Line Compilers
3. Libraries

5. Basic Dart

Introduction

Example Code

Entry Point

Example Code

Output

Introduction to Typing

1. Statically-typed languages.
2. Dynamically-typed languages.

Dart Typing

Static Types

Dynamic Types (aka Untyped)

There is a difference, but it is subtle.

This code wont compile. Dartpad displays the following error:

Type Inference

[Example of Inference #1:](#)

[Output](#)

[Example of Inference #2:](#)

[Output](#)

Type Matching

[Example Code](#)

[Output](#)

Type Information

[Example Code](#)

[Output](#)

Strings

[Interpolation](#)

[Raw Strings](#)

[Runes](#)

Object-Orientated Language Features

[Modules](#)

[Private Classes, Variables & Methods](#)

[Constructors](#)

[Instance Variables](#)

[Constructor and Method Parameters](#)

[1. Parameters - Positional Required](#)

[2. Parameters - Positional Optional](#)

[3. Parameters - Named](#)

[Interfaces](#)

Other

[Method Cascades](#)

6. More Advanced Dart

[Introduction](#)

[Arrow Functions \(Lambdas\)](#)

[Operator Overloading](#)

[Example](#)

[Warning - hashCode](#)

[Reflection](#)

[Mixins](#)

[Collections](#)

[Introduction](#)

[Lists](#)

[Maps](#)

[More-Specific Collection Classes](#)

[Assertions](#)

[Example Code](#)

[Output](#)

[Assertions & Modes \(Flutter\)](#)

[Further Reading](#)

[Errors & Exceptions](#)

[Why Have Error & Exception Handling?](#)

[Errors](#)

[Exceptions](#)

[Handling Errors](#)

[Handling Exceptions](#)

[Finally](#)

[Catch Exception](#)

[Catch Exception and Stack Trace](#)

[Catch Specific Exceptions](#)

[Throw Exception](#)

[Rethrow Exception](#)

[Create Custom Exceptions](#)

[Console Output](#)

[Example Code](#)

[Output](#)

[Asynchronicity](#)

[Introduction](#)

[Future](#)

[Invoking and Handling Asynchronous Operations](#)

[Future API](#)

[Async & Await Keywords](#)

[Reactive Programming](#)

[Nulls](#)

[?](#)

[Example](#)

[Static Analysis](#)

[Example ‘analysis_options.yaml’ File](#)

[Further Reading](#)

7. Introduction to Flutter

[Introduction](#)

[What is Flutter?](#)

[High Productivity](#)

[High Quality](#)

[High Performance](#)

[It is Free and Open.](#)

[Fuschsia](#)

[Flutter Source Code](#)

[Flutter SDK](#)

[Dart Platform](#)

[Flutter Engine](#)

[Foundation Library](#)

[Flutter Release Process](#)

[Flutter Channels](#)

8. Installing Flutter & Editor

Introduction

Note: Developing on a PC for iOS

Introduction

It's Not as Bad as it Seems

Install Flutter

Introduction

Step 1: Software Pre-Requisites

Step 2: Download the Flutter SDK

Step 3: Setup Your Path

Step 4: Run Flutter Doctor

Flutter Commands

Install Editor

Editors & UI Builders

Flutter Doctor Tells You to Install Android Studio

Android Studio

IntelliJ (Android Studio, IntelliJ)

Visual Studio Code

9. Create Default Flutter Project

Introduction

Default Flutter App

Create Project

Android Studio

Visual Studio Code

Command-Line

Project Created

Project Folders

Project Files

Project Application Code File

Note

Widgets & Composition

Introduction

Widgets are the Building Blocks of your UI

Project UI

UI As Code

Widget Tree

Custom Widgets

Flutter Widgets

Project Code

Entry Point

MyApp Widget

MyHomePage Widget

10. Setup Android Emulator & Run Project

Introduction

Emulators

Android Emulator

Setting Up the Android Emulator

Options

Mix

Setup Android Emulator in Android Studio

AVD Manager

AVD Manager Dialog

Setup Android Emulator in Visual Studio Code

Before You Start

View Flutter Commands

Launch Emulator

Create New Emulator

Setup Android Emulator in Command Line

[Introduction](#)

[Android SDK Path](#)

[To Create a New AVD to Test On:](#)

[To List Your Available AVDs:](#)

[To Launch an Available AVD:](#)

[Run Project](#)

[Android Studio](#)

[Visual Studio Code](#)

[Command-Line](#)

[11. Setup iOS Emulator & Run Project](#)

[Introduction](#)

[PCs](#)

[XCode](#)

[iOS Emulator](#)

[Open iOS Simulator/Emulator](#)

[Open from Xcode](#)

[Opening from Command-Line](#)

[Run Project](#)

[Android Studio](#)

[Visual Studio Code](#)

[Command-Line](#)

[12. Setup Device & Run Project](#)

[Introduction](#)

[Android Device](#)

[Introduction](#)

[iOS Device](#)

[Check Your Hardware First](#)

[Setup Your XCode Project](#)

[Open XCode Project](#)
[Create Signing Team](#)
[Set Bundle Identifier](#)
[Connect the Device to The Mac & Run](#)
[Further Reading / Instructions](#)

13. Hot Restarting & Reloading

[Introduction](#)
[Hot Reloads](#)
[The Official Documentation Says:](#)
[Two Options](#)
[Hot Restarting](#)
[Hot Reloading](#)
[Android Studio](#)
[Visual Studio Code](#)
[Command-Line](#)

14. Dependencies & Packages

[Introduction](#)
[Website](#)
[Core Packages](#)
[Non-Core Packages](#)
[Most Useful Non-Core Packages](#)
[How to Use an External Package](#)
[Declare Dependency in Project](#)
[Import Packages](#)
[Import & Use Package Code](#)
[Restart Your App](#)
[Package Version Numbers](#)

[Project Files](#)

[.packages](#)

[pubspec.lock](#)

[How to Clean & Reimport Your Packages](#)

[How to Publish Your Own Packages](#)

[Introduction](#)

[Setting Up a Dart Package](#)

[Adding Documentation](#)

[Final Review](#)

[Do a Publish Dry-Run](#)

[Publish](#)

[Further Reading](#)

[15. Introduction to Widgets](#)

[Introduction](#)

[What Are Widgets?](#)

[User Interface: Material & Cupertino](#)

[Material Design, According to Google](#)

[Most Flutter Widgets Work with Material Design](#)

[User Interface: Cupertino](#)

[Flutter Includes iOS-Styled Widgets](#)

[Building Widgets](#)

[Build Method](#)

[Build Context](#)

[Not All Widgets Are Equal](#)

[State](#)

[Mutable State](#)

[Immutable State](#)

[Flutter Widgets & State](#)

[Stateful Widgets](#)

[Stateless Widgets](#)

[Event Handling](#)

[Event Handlers](#)

[Styling](#)

[Margins, Borders & Padding](#)

[Further Reading](#)

[16. Stateless Widgets](#)

[Introduction](#)

[Not All Widgets Need to be Smart](#)

[Minimum Code](#)

[Creation](#)

[Example](#)

[Rendering](#)

[The ‘Build’ Method](#)

[When Does The ‘Build’ Method Execute?](#)

[Lifecycle](#)

[Exercise – ‘first_stateless’](#)

[Step 1 – Create Default Flutter App](#)

[Step 2 – Replace Application Code](#)

[Step 3 – Open Emulator & Run](#)

[Summary So Far](#)

[Step 4 – Add Some Padding](#)

[Step 5 – Add Scrolling](#)

[Step 6 – Add Border](#)

[Step 7 – Final Touch](#)

[Example – ‘stateless_widget_rebuild’](#)

[Optional](#)

[Purpose](#)

[Source Code](#)

17. Stateful Widgets

Introduction

Some Widgets Need to be Smart

Minimum Code

Two Classes

Class #1 – the class that extends StatefulWidget

Class #2 – the class that extends State

Creation

Rendering

The ‘Build’ Method

LifeCycle Methods

Class #1 – the class that extends StatefulWidget

Class #2 – the class that extends State

More Reading

Example – ‘stateful_widget_flowers’

Optional

App Purpose

App Widgets

Start App

Change the Flower

Add Blur

Source Code

18. Basic Material Widgets

Introduction

Text

Notes

Example – ‘text’

Image

Introduction

Exercise – ‘loading_image’

Icon

[Introduction](#)

[Example - ‘icon’](#)

[Further Reading](#)

Buttons

[Introduction](#)

[Enabling](#)

[Example – ‘buttons’](#)

19. Multi-Child Layout Widgets

Introduction

Multi-Child Layout Widgets

Column

[Spacing Out Children Using MainAxisAlignment](#)

[Expanding Children Using Expanded Widget](#)

Row

[Spacing Out Children Using MainAxisAlignment](#)

[Expanding Children Using Expanded Widget](#)

Flex

[Example – ‘flex’](#)

ListView & ListTile

[Example - ‘horizontal_list’](#)

[ListTile](#)

Stack

[Example – ‘stack_please_wait’](#)

20. Single-Child Layout Widgets

Introduction

[Most Important Single Child Layout Widgets](#)

Padding

Example – ‘padding’

Container

Example – ‘container’

Further Reading

Card

Example – ‘cards’

ConstrainedBox

Example – ‘constrained_box’

Expanded

Example – ‘expanded’

Flexible

Example – ‘flexible’

Center

GestureDetector

Example – ‘gesture_app’

Positioned

Example – ‘positioned’

SafeArea

Example: Non-Safe Area

Example: Safe Area

Example: Safe Area with Minimum Padding Set

SingleChildScrollView

Constructor Arguments Include:

Exercise – ‘single_child_scroll_view’

21. App Scaffolding Widgets

Introduction

MaterialApp

[Navigator](#)
[Themes](#)
[Locales](#)
[Debugging Constructor Arguments](#)

[Scaffold](#)

[AppBar](#)

[Body](#)

[BottomNavigationBar](#)

[Drawer](#)

[BottomSheet](#)

[PersistentFooterButtons](#)

[Exercise – ‘scaffold’](#)

[Step 1 – Create Default Flutter App](#)

[Step 2 – Replace Application Code](#)

[Step 3 – Open Emulator & Run](#)

22. Other Widgets

[Introduction](#)

[Dialog](#)

[AlertDialog](#)

[SimpleDialog](#)

[Custom Dialog Widget](#)

[Dismissible](#)

[Example – ‘dismissible’](#)

[Source Code](#)

[ExpansionPanelList & ExpansionPanel](#)

[ExpansionPanelList](#)

[ExpansionPanel](#)

[Example – ‘expansion_panel’](#)

[GridView](#)

[Builder](#)

[GridTile](#)

[GridTileBar](#)

[Example – ‘gridview_app’](#)

[Further Reading](#)

[PopupMenuButton](#)

[Example – ‘popup_menu_button’](#)

[Radio](#)

[SnackBar](#)

[Example – ‘snack_bar’](#)

[Spacer](#)

[Flex Property](#)

[Exercise – ‘spacer’](#)

[Switch](#)

[TabBar, Tabs and TabBarView Widgets](#)

[TabBar & TabBarView vs iOS Tabs](#)

[Instructions:](#)

[Exercise – ‘tabs_simple’](#)

[Step 1 – Create Default Flutter App](#)

[Step 2 – Replace Application Code](#)

[Step 3 – Open Emulator & Run](#)

[Step 4 – Move Tabs to Bottom](#)

[Step 5 – Change Tab Styles To Look More Like iOS](#)

[Table](#)

[Introduction](#)

[Column Width Specifiers](#)

23. Builders

[Introduction](#)

[What is a Builder?](#)

[How Do You Use a Builder?](#)

[Nested Builders](#)

[Common Builders](#)

[AnimatedBuilder](#)

[GridView Builder](#)

[FutureBuilder](#)

[ListView Builder:](#)

[OrientationBuilder](#)

[PageRoutebuilder](#)

[StreamBuilder](#)

[StreamBuilder](#)

[Example – ‘nested_builders’](#)

[Source Code](#)

[24. Routing & Navigation](#)

[Introduction](#)

[Navigator Class](#)

[Stack of Routes](#)

[Invoking Navigation without Named Routes](#)

[Navigating Forward](#)

[Navigating Backwards](#)

[Data](#)

[Example – ‘routes_simple’](#)

[Invoking Navigation with Named Routes #1](#)

[Define Routes](#)

[Navigating Forward](#)

[See the problem yet?](#)

[Example – ‘routes_named’](#)

[Invoking Navigation with Named Routes #2](#)

[Attach Route Handler to MaterialApp](#)

[Code Route Handler](#)

[Navigating Forward](#)

[Example – ‘routes_named_with_parms’](#)

[PageView](#)

[Introduction](#)

[Child Widgets](#)

[Controller](#)

[Example – ‘page_view_navigation’](#)

[25. Forms](#)

[Introduction](#)

[Form](#)

[Form State](#)

[Form Validation](#)

[Form / Field Integration](#)

[Form Fields](#)

[Checkbox](#)

[DropdownButton](#)

[Radio](#)

[TextFormField, TextField](#)

[InputDecorator](#)

[Example – ‘form_details’](#)

[Dependencies](#)

[Source Code](#)

[Other Information](#)

[Input Decoration Themes](#)

[Enabling / Disabling Form Buttons](#)

[26. HTTP, APIs, REST & JSON](#)

[Introduction](#)

Asynchronous Communication

HTTP

[Introduction](#)

[Tools](#)

[Request](#)

[Response](#)

[Methods](#)

[URI](#)

[Status](#)

[Header](#)

[Body](#)

APIs

REST

[REST APIs should be stateless.](#)

[How REST Uses URLs](#)

[How REST Uses HTTP Method](#)

[Accessing Data with a REST API](#)

[Inserting Data with a REST API](#)

[Updating Data with a REST API](#)

[Deleting Data with a REST API](#)

JSON

[JSON For Passing an Object Containing Data.](#)

[JSON For Passing an Array](#)

[JSON For Passing an Array of Objects](#)

27. Flutter with HTTP, APIs, REST & JSON

[Introduction](#)

[Flutter & JSON](#)

[Introduction](#)

[Serializing & Deserializing JSON.](#)

[Generating Code for Serializing & Deserializing](#)

[Manually Writing Code for Serialization & Deserialization](#)

[Flutter & HTTP](#)

[Flutter HTTP Package](#)

[Dummy API](#)

[Error Handling](#)

[Example ‘http_employees’](#)

[Source Code](#)

[Other Information](#)

[Alice](#)

[HAL / HATEOS](#)

28. State

[Introduction](#)

[State & Events](#)

[Storing State](#)

[Kinds of State](#)

[How to Determine Where to Store State](#)

[Responding to Events](#)

[Introduction](#)

[Events Invoke Functions](#)

[Events Can Affect State](#)

[State & Events – Problems](#)

[State & Events – Different Approaches](#)

[Mixing Approaches](#)

[How I Decide Where to Put State](#)

[State & Events – Commonly-Used Approaches](#)

[Stateful Widget Approach](#)

[InheritedWidget Approach](#)

[Scoped Model Approach](#)

[BLoC w/Streams Approach](#)

29. State & Stateful Widget Approach

Introduction

Approach

Exercise – ‘state_and_stateful_widget’

Introduction

Step 1 – Create Default Flutter App

Step 2 – Replace Application Code

Step 3 – Open Emulator & Run

Summary

Step 4 – Add Car Selection

Further Reading

30. State & InheritedWidget Approach

Introduction

Approach

Exercise – ‘state_and_inherited_widget_add’

Step 1 – Create Default Flutter App

Step 2 – Replace Application Code

Step 3 – Open Emulator & Run

Summary

Exercise – ‘state_and_inherited_widget’

Step 1 – Create Default Flutter App

Step 2 – Replace Application Code

Step 3 – Open Emulator & Run

Summary

Conclusion

Further Reading

31. State & ScopedModel Approach

Introduction

[Approach](#)

[Package](#)

[Package Readme](#)

[Multiple Models](#)

[Exercise – ‘state_and_scoped_model’](#)

[Step 1 – Create Default Flutter App](#)

[Step 2 – Replace Application Code](#)

[Step 3 – Open Emulator & Run](#)

[Summary](#)

[Conclusion](#)

[**32. State & BLoCs w/Streams Approach**](#)

[Introduction](#)

[BLoC Pattern](#)

[Reactive Programming](#)

[RxDart](#)

[StreamBuilder](#)

[Exercise – ‘state_and_block_with_streams’](#)

[Step 1 – Create Default Flutter App](#)

[Step 2 – Add the RxDart Dependency](#)

[Step 3 – Replace Application Code](#)

[Step 4 – Open Emulator & Run](#)

[Summary](#)

[Conclusion](#)

[Further Reading](#)

[**33. Local Persistence**](#)

[Introduction](#)

[Your Options](#)

SQLite Database

[Introduction](#)

[Step 1 – Add Dependencies to Project](#)

[Step 2 – Define the Data Model](#)

[Step 3 – Open the Database](#)

[Retrieve Rows from Database](#)

[Executing SQL](#)

[Insert into Database](#)

[Update Row in Database](#)

[Delete Row in Database](#)

[Example – ‘sqlite_vocabulary’](#)

[Further Reading](#)

Local Files

[Introduction](#)

[Platform](#)

[Path Provider Package](#)

[Application Documents Directory](#)

[Directories](#)

[Files](#)

[Directory & File Methods](#)

[Reading & Writing Data to a File](#)

[Example ‘persistence_files’](#)

Shared Preferences

[Introduction](#)

[Methods](#)

[Further Reading](#)

[Example ‘persistence_shared_preferences’](#)

34. Mixins

[Introduction](#)

[Mixins](#)

Mixins & Code Generators

Example – ‘mixins’

Source Code

35. Animation

Introduction

Animations & State Changes

Animations Are Used to Make UI Changes Look Better

Animations Alter the Way the UI Change Occurs

Types of Animations

Implicit Animations

Example – ‘animated_container’

Example – ‘animated_text’

Example – ‘animated_list’

Explicit Animations

Animation

AnimationController

TickerProvider

Ticker

Example – ‘animated_progress_circle’

Curved Animations

Tweens

Custom Behavior

Transforms

Further Reading

36. Debugging & Performance Profiling

Introduction

Debugging

Performance Profiling

Programmatical Options

[Add Debugger Statements](#)

[Add Print & DebugPrint Statements](#)

[Add Assertions](#)

[Service Extensions](#)

[Introduction](#)

[Performance Overlay](#)

[Show Paint Baselines \(debugPaintSizeEnabled\)](#)

[Show Material Grid](#)

[Turn Service Extensions On/Off from Android Studio](#)

[Turn Service Extensions On/Off from Visual Studio Code](#)

[Turn Service Extensions On/Off from Command Line](#)

[Turn Service Extensions On/Off Programmatically](#)

[Dart Observatory](#)

[Introduction](#)

[Part of the Dart SDK](#)

[Starting the Observatory](#)

[Timeline](#)

[Profile Mode](#)

[Further Reading](#)

[Android Studio](#)

[Visual Studio Code](#)

[Command-Line](#)

[Further Reading](#)

37. Change Detection, Keys & Rendering

[Introduction](#)

[Change Detection](#)

[Widgets](#)

[Elements](#)

[Element Trees](#)

Widgets, Elements, Render Objects

Change Detection & Updates

Detecting Structural Changes

Matching Elements to Widgets

If there is a Match

If there is no Match

Optimizations

Render Tree

Render Objects

Keys

Introduction

Elements May or May Not Store a Reference to State

Elements for Stateless Widgets Have No Reference to any State

Elements for Stateful Widgets Have A Reference to the State

The ‘Losing State’ Problem

Global Keys

Further Reading

38. Other Performance Considerations

Introduction

Http Communication

Single Threaded

Isolates

Use Constants When Possible

Use Finals Whenever Possible

39. Publishing Your App

Introduction

Code Analysis

Platforms

Compilation

[Platform Class](#)

[Further Reading](#)

[Example – Detecting Platform at Runtime](#)

[Release Mode](#)

[Further Reading](#)

[Android Studio](#)

[Visual Studio Code](#)

[Command-Line](#)

[Application Icons](#)

[The Article I Used to Create Application Icons](#)

[Notes on Article](#)

[Android Deployment](#)

[Android-Specific Files](#)

[Dependency Management](#)

[Deployment Units](#)

[Deploying Your Android App to Google Play](#)

[iOS Deployment](#)

[iOS-Specific Files](#)

[Dependency Management](#)

[Deployment Units](#)

[Deploying Your iOS App to the App Store](#)

40. Flutter Resources

[Introduction](#)

[Official Resources](#)

[Other Resources](#)

2. Welcome

Introduction

The purpose of this chapter is to introduce the purpose of this book and acknowledge those people who have made this book possible. This chapter also contains a disclaimer and revision history, along with information on how to get to the source code for the examples.

Acknowledgements

First and foremost, thanks to my wife Jill and her patience. I hope she is enjoying herself doing her favorite things like Paddle boarding, Kayaking and being at one with nature. I hope she never reads this book because it would bore her.

I would also like to acknowledge the amazing work done by the Google engineers. When I ran the profiler and saw just how fast the UI was redrawing, I was blown-away. Google has some amazing talent, what a team of engineers.

Purpose

I wrote this book to broaden my own knowledge of this technology and I still have a way to go. However, I learnt a lot in writing this book and I hope it helps

others.

Work

I have to be honest; I also wrote this book for some self-promotion. After this book is published, I intend to continue working on apps for my wife's business and find some part-time Flutter work. If you are interested, shoot me an email at markclow@hotmail.com or communicate with me via my LinkedIn page here: <https://www.linkedin.com/in/mark-clow-9a61362/>.

Disclaimer

Let's get this over with as quickly as possible. Some of this information in this book may be incorrect (I am a human being that makes mistakes) and that this publication is somewhat opinionated. I am trying my best to be as technically accurate as possible, but I am still learning a lot and have much to learn about Flutter and Dart. I have opinions but please don't take them too seriously. I do not intend to harm anything or anyone, I am not smart enough for that.

Revisions

This book has taken a long time to write and I will continue to improve it whenever I have time, adding

more content when possible. So, if you get an earlier version of the book it may be slightly less complete than later on. If this is the case, email me a proof of purchase at markclow@hotmail.com and I will send you a PDF version, which will be watermarked with your name (sorry but it's to prevent copying). I welcome (constructive) criticism and input so if you have any, please email me at markclow@hotmail.com.

| Date | Description |
|------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 3/31/2019 | Initial version. |
| 4/7/2019 | Added chapter name to footer. Applied corrections from printed proof notes – many! Simplified chapters re setup. Added Animations Chapter. Added Dismissible Widget. |
| 4/25/2019 | Adding more content to Stack. |
| 4/29/2019 | Fixes for missing images on Kindle. |
| 5/12/2019 | Isolates. |
| 5/15/2019 | InputDecoration – added info on customizing errors. |
| 5/20/2019 | Channels. |
| 5/23/2019 | ConstrainedBox. |
| 6/15/2019 | How to Clean & Reimport Your Packages |
| | |

| | |
|------------------|----------------------------------------------------------------------------------|
| 6/19/2019 | Application Icons. |
| 6/20/2019 | Publishing Your Android App. |
| 6/21/2019 | Publishing Your iOS App. |
| 6/28/2019 | Apple Guidelines. |
| 7/2/2019 | Flutter commands. |
| 7/28/2019 | Flutter folders and files. Event handlers. |
| 8/11/2019 | Lambda functions. Native compilation on platforms. UI as code. Named parameters. |
| 8/17/2019 | State. |
| 8/18/2019 | Final variables. Borders, padding & margin. |

Source Code

Location

This book has many examples & exercises. The source code is available here:

https://github.com/markclow/flutter_book_examples

Example & Exercise Names

Each example or exercise should have a name in quotes (see the example below in the box). The name in quotes is the directory in which the source code is

located.

This is the format:

Example – ‘gesture_app’

The source code for this example is located in
‘gesture_app’.

One File

Another thing to bear in mind is that the exercises have all been written to use a single file. This was so that there could be one single continuous listing in the book. In reality, you would obviously split your project up into many smaller files.

3. The Big Picture

Introduction

The purpose of this chapter is to give the reader a quick introduction to the world of Mobile Application development, and to introduce him or her the problem of cross-platform development, and how it was approached by different companies.

Before we go into this, lets introduce some of the terminology first.

Terminology

Compiler

A computer program consists of a set of instructions for the computer to perform a specific task. Most computer programs are written using high-level programming languages. Thus, the computer does not understand these programs. Therefore, they are converted to machine understandable, machine language. A compiler is a software that performs this conversion. It converts the source program into machine language.

Native and Non-Native Compilers

A native compiler converts the source program into machine code for the platform on which it runs. The code will not require any translation to be interpreted and executed by the CPU.

A non-native compiler converts the source program into a more general format where it can be run on multiple platforms. The code will require translation to be interpreted and executed by the CPU.

Cross-Platform Mobile Application Development

Mobile Applications

A mobile application (shortened to ‘mobile app’) is a computer program or software application designed to run on a mobile device such as a phone/tablet or watch. Writing mobile apps sounds easy but is complicated by the number of platforms that are available. Your app could run on an iPhone, it could run on an iPad, it could run on an Android Phone etc. Also remember that these platforms could change quickly as new devices appear on the market.

Before Cross-Platform Mobile Application Development

In the past, in order to produce performant applications, developers had to write the application code specifically for each platform, writing what is called native apps. There would often be one codebase (and developers) for iOS (iPhone) and another codebase (and developers) for Android. For native iOS, Objective-C and Swift are the preferred programming languages. For native Android, Java and Kotlin are the preferred languages.

This complicated matters:

- You had to keep two sets of code in sync.
 - If you change the iPhone code, you should change the Android code to match.
- You had to have developers with multiple skillsets.
 - Expensive.
- Sometimes the app for one platform would look very different from the other platforms.

Early Cross-Platform Development Tools

Anyway, Silicon Valley soon realized what a problem this was and set to work on developing tools for cross-

platform mobile application development. They quickly split into two groups of development tools: those that used native libraries and those that didn't.

Development Tools That Used Native Libraries

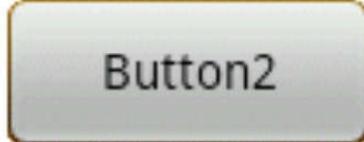
These tools created a ‘Unified’ API on top of the native SDK supplied by Apple and Google. Many of these development tools, for example Xamarin, Appcelerator, Nativescript are still around.

Unified API Does Not Cover 100%

The problem with these types of applications is that the ‘Unified API’ does not cover 100% and leaves the developers with many burdens, such as having to still write a large chunk of platform-specific code.

Use Widget SDKs

Also, these apps use the Widgets from the SDKs. Therefore, the apps can look different because they use different Widgets from different SDKs for different platforms. Here is an example of how different button Widgets are used for different platforms:

| ios | Android |
|------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------|
| <u>uibutton</u>  | <u>widget.button</u>  |

Development Tools That Didn't Use Native Libraries

These tools took a different approach. Most of these attempted to bypass the SDK approach and write code that runs on the platform's browser. This had the advantage of being able to use many of the HTML5 and JavaScript capabilities already built-in. The app would run in a ‘web view’. A “webview” is a browser bundled inside of a mobile application producing what is called a hybrid app. Using a webview allows mobile apps to be built using Web technologies (HTML, JavaScript, CSS, etc.) but still package it as a native app and put it in the app store.

The problem with these types of applications is speed. They are not running natively in compiled machine code, they are running on a hidden web browser. Many of these development tools, for example Cordova, PhoneGap are still around.

Modern Cross-Platform Development Tools

More recently, two main rivals have emerged and look to be leading the field of mobile app development tools: Facebook React Native and Google Flutter.

React Native



React.JS is an excellent JavaScript framework that has been popular for years and works with both mobile and non-mobile websites equally well. Developers write user interfaces with Component objects, like lego blocks. These Components can contain code so that they can react to the user's input and produce an interactive user interface. React Native is like React, but it uses native components instead of web components as building blocks.

How Does It Work?

React Native runs in two parts.

1. The UI.
 - It displays the ui and receives user input.

2. The JavaScript engine.
 - It interprets and executes the JavaScript application code.

The two parts communicate with a bridge.

Conclusion

React Native is an excellent framework. It has the great advantage of being the more established player because it has been out since 2015. There are also a lot of React developers out there who can quickly cross-train to use React Native rather than React JS. React Native is also a very productive tool because it has many ready-to-use components.

React native apps are not completely native compiled. A lot of the deployed code is native but your part of the app runs as embedded JavaScript, communicating through a bridge with the native components. This is not the optimum solution for performance.

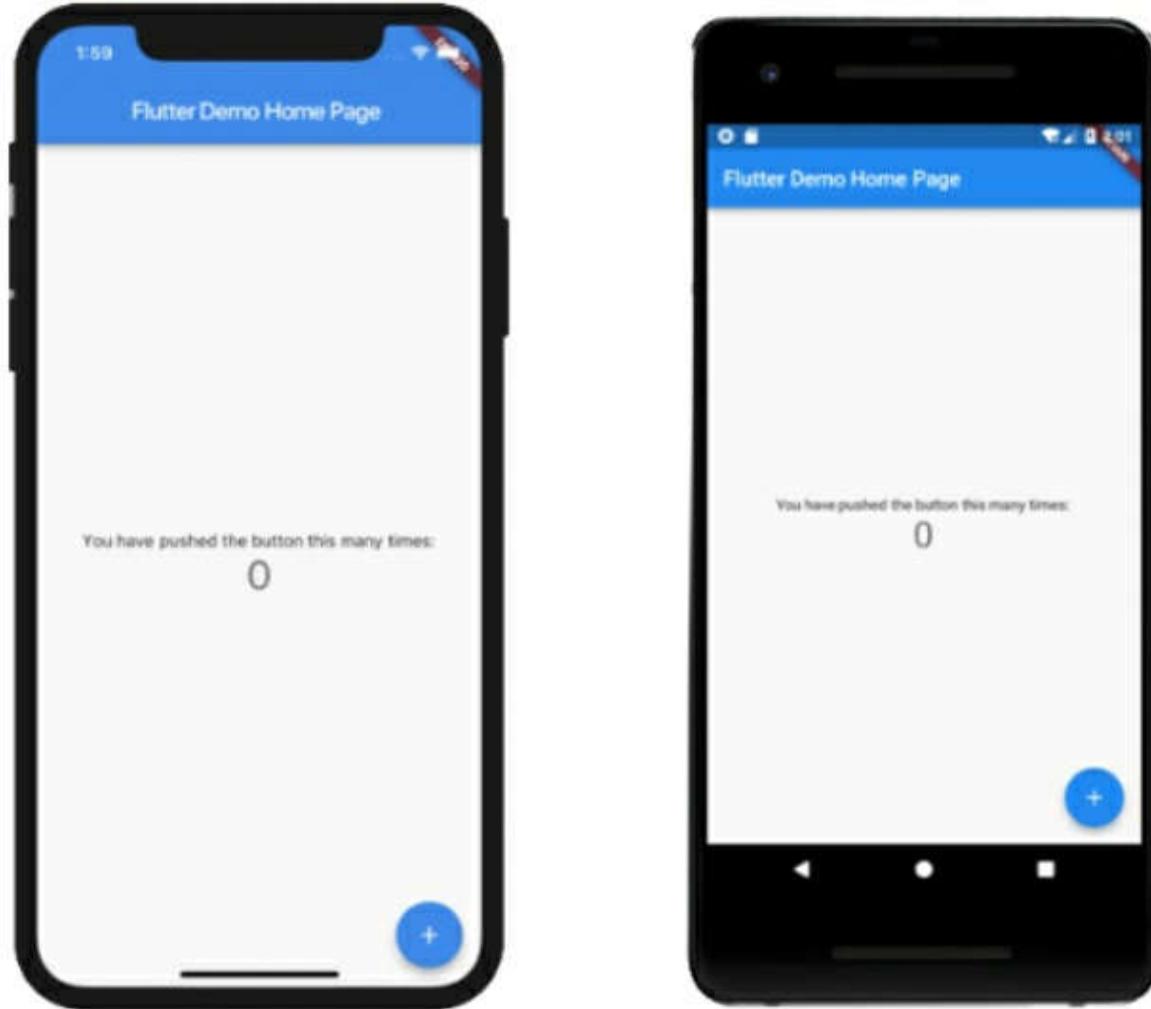
Google Flutter



Google Flutter has only been available since 2017 but it is making waves because it takes a different approach to cross-platform mobile app development. Google is currently working on the successor to its Android operating system called Fuchsia and it is writing it using Flutter. So, Flutter is very important to Google.

You write user interfaces using Google Flutter user interface widgets, not the native iOS or Android UI widgets shipped with their respective SDKs. A Flutter app made using Flutter Widgets will look exactly the same on iOS as it does on Android because it will use the same Widgets from the same library. Flutter comes with many widgets, including those that mimic Google's Material look & feel and those that mimic Apple's iOS look & feel.

The Flutter Default App Running on iPhone on Left, Android on Right



Google Flutter uses its own high-performance rendering engine to draw these widgets and they have been designed to work on all mobile platforms. Also, these widgets are extendable.

You write the application code in Google's Dart language and it is compiled ahead-of-time into machine-code for native-like performance, thus offering a performance advantage over React Native.

There is no bridge between the user interface and the application code.

The only downside that is currently obvious is that developers will have to learn Dart, rather than reuse their existing JavaScript expertise.

Conclusion

If you want to write cross-platform mobile web apps that are performant then Google Flutter appears to be the best choice at the moment. However, things move quickly and that may not be for long!

4. Introduction to Dart



Introduction

The purpose of this chapter is to give the reader a quick introduction to Dart before installing it and starting to use it.

Dart is a general-purpose programming language which was created by Google in 2011. Like Java and C#, it has a similar syntax to ‘C’.

Platforms

Unlike conventional languages, Dart has been optimized to be deployed to run on a variety of platforms:

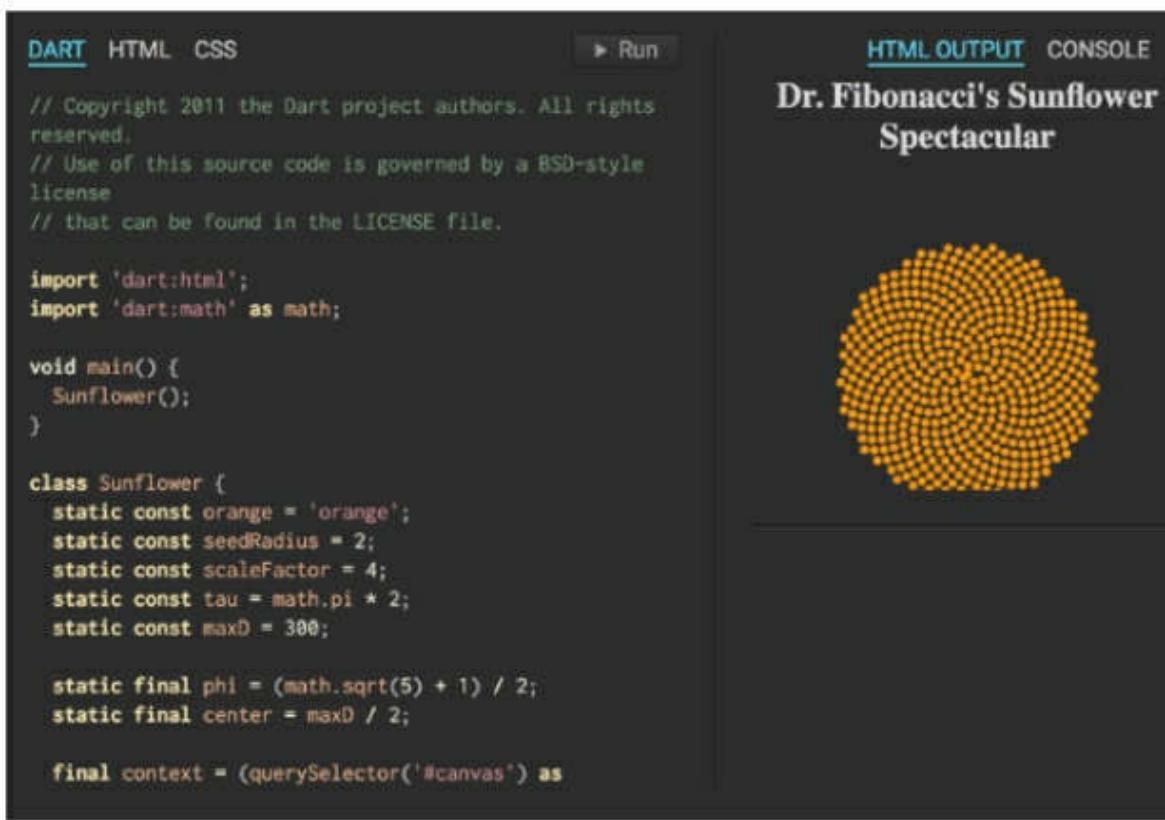
1. Within a web browser as JavaScript
2. As an interpreted application
3. As a native application

1. Within a Web Browser

Dart provides an SDK, which provides command-line

tools to transpile Dart source code into JavaScript. This has been developed so efficiently that the resulting transpiled JavaScript is more efficient than its hand-coded equivalent!

You can try out Dart in your web browser by Navigating to <https://dartpad.dartlang.org/>. You can write your own code or run the sample code. See the ‘Sunflower’ sample below.



The screenshot shows the DartPad interface. On the left, the Dart code for generating a sunflower pattern is displayed:

```
// Copyright 2011 the Dart project authors. All rights reserved.  
// Use of this source code is governed by a BSD-style license  
// that can be found in the LICENSE file.  
  
import 'dart:html';  
import 'dart:math' as math;  
  
void main() {  
  Sunflower();  
}  
  
class Sunflower {  
  static const orange = 'orange';  
  static const seedRadius = 2;  
  static const scaleFactor = 4;  
  static const tau = math.pi * 2;  
  static const maxD = 300;  
  
  static final phi = (math.sqrt(5) + 1) / 2;  
  static final center = maxD / 2;  
  
  final context = (querySelector('#canvas')) as
```

On the right, the output section shows the title "Dr. Fibonacci's Sunflower Spectacular" and a large, intricate sunflower pattern composed of small yellow dots.

Just remember that not everything will always be the same.

For example, you cannot read from stdin when running

from a browser. I tried to develop a Dart program on dartpad.dartlang.org that would accept user input and it would never work.

2. As Interpreted Application

The Dart SDK includes a Virtual Machine. A virtual machine is a sandbox in which code may run without directly communicating with the underlying operating system. This enables Dart code to be invoked from the command-line, using the ‘dart’ command-line tool in the SDK. This code is compiled on demand just-in-time as it runs.

Using Dart in this way is a great way to write server-side applications and it performs at a similar level to Java / .Net.

Hot Reloading / Hot Replacing

If the developer is running the Dart application in the Dart virtual machine from the command-line (interpreted), the JIT compiler can reload the code when the underlying source code changes, often while preserving the application state (variables) whenever possible. So, the developer can write and run the code at almost the same time. This makes application development very fast indeed. Yet at the end of the development process, the code can be compiled using

the ahead-of-time compiler and deployed as a native application.

Flutter Development (Debug Mode)

When you are developing a Flutter Application, most of the time you run it in Debug Mode and the code is JIT compiled & interpreted. This mode is known as ‘check’ or ‘slow’ mode. Under this mode, the assertion functions, including all debugging information, service extensions, and debugging aids such as “observatory,” are enabled. This mode is optimized for rapid development and operation, but not for execution speed, package size, or deployment.

Once your app is written you can build it to run in Release Mode as a native application and it will perform much better.

3. As Native Application

Dart code can be compiled ahead-of-time so that the code may be deployed as machine-code.

Flutter was mostly written using Dart and runs natively. This makes Flutter fast, as well as customizable (as the Flutter widgets were written in Dart).

Dart SDK

The Dart SDK is available to download here:

<https://www.dartlang.org/tools/sdk>

The Dart SDK comprises of three main elements:

1. Command-line tools.
2. Command-line compilers.
3. Libraries.

1. Command-Line Tools

The Dart SDK contains the following command line tools:

| Name | Description |
|---------------------|-----------------------------------------------------------------------------------------------------------------|
| dart | Enables you to execute a .dart file within the Dart Virtual Machine. |
| dart2js | Compiles dart source code to JavaScript. |
| dartanalyser | Analyses dart source code. This is used by many of the code editors to provide error and warning highlighting. |
| dartdevc | Compiles dart source code to JavaScript. Similar to dart2js except that it supports incremental compilation, |

| | |
|----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| | which lends itself to developers. |
| dartdoc | Generates Dart documentation from source code. As the seminal book ‘Domain-Driven Design’ by Eric Evans states: ‘the code is the model and the model is the code’. |
| dartfmt | Formats Dart source code. This is used by many of the code editors to provide Dart formatting. |
| pub | This is Google’s Package Manager. This is important and we will cover this in a later chapter. |

2. Command-Line Compilers

Dartium, WebDev and Build_Runner

You can run Dart in a browser called Dartium without compiling it to JavaScript. Dartium is basically Chrome with a Dart VM. However, the mainstream Dart web development route is now writing the code with Dart but compiling and running as JavaScript using the dart2js and dartdevc JavaScript compilers in combination with the webdev and build_runner utilities.

More Reading:

<https://webdev.dartlang.org/tools/webdev>.

Dart2js and DartDevC

These two JavaScript compilers have different use cases. Normally these are used with the tool webddev and you don't usually have to worry about which compiler you're using, because it chooses the right compiler for your use case. When you're developing your app, webdev chooses [dartdevc](#), which supports incremental compilation so you can quickly see the results of your edits. When you're building your app for deployment, webdev chooses [dart2js](#), which uses techniques such as tree shaking to produce optimized code.

3. Libraries

| Name | Description |
|---------------------|----------------------------------------------------------------------------------------------------------------------------|
| dart:core | Built-in types, collections, and other core functionality. This library is automatically imported into every Dart program. |
| dart:async | Support for asynchronous programming, with classes such as Future and Stream. |
| dart:math | Mathematical constants and functions, plus a random number generator. |
| dart:convert | Encoders and decoders for converting |

between different data representations, including JSON and UTF-8.

5. Basic Dart

Introduction

The purpose of this chapter is to introduce some of the more basic Dart concepts and syntaxes.

Example Code

All the example code for this chapter should be executed on the following website:

dartpad.dartlang.org

Entry Point

Dart is a bit like Java, every Dart app must start with a main function.

Example Code

```
void main(){
    print("App started");
    new App();
    print("App finished");
}

class App{
    App(){
        print("Constructing a class.");
    }
}
```

```
| }  
| }
```

Output

```
| App started  
| Constructing a class.  
| App finished
```

Introduction to Typing

Typically, computer languages have fallen into two camps:

1. Statically-typed languages.
2. Dynamically-typed languages.

1. Statically-typed languages.

These languages have specific variable types and the developer compiles the code using an ‘ahead-of-time’ compiler. The compiler type checking is performed before the code is run. This is an excellent way to develop software as the compiler performs static-analysis of the code as part of the compilation, alerting the developer when issues arise. Software typically takes longer to develop in this method, but the software developed in this manner typically works better in complex scenarios.

2. Dynamically-typed languages.

These languages don't have specific variable types and no ahead-of-time compilation is performed.

Dynamically-typed languages make the development process very quick as the developer does not typically need to recompile the code. However, code developed in this manner tends to lend itself to simpler scenarios as it can be more error-prone.

Dart Typing

Dart is different because Dart code can be run with both static types and dynamic type variables. The type system in Dart 1 had some issues and they introduced a 'strong mode' for stronger type checking. This mode has become the typing system in Dart 2.0 and it offers strong guarantees that an expression of one type cannot produce a value of another type.

Dart performs type checking at two different times:

- When the code is compiled (code is reloaded / or compiled ahead-of-time).
- When the code is run (runtime).

Static Types

These are the most-commonly used and built-in Dart

types:

| Type | Description |
|-----------------------|------------------------------------|
| int | Integers (no decimals). |
| double | Decimal number (double precision). |
| bool | Boolean true or false. |
| String | Immutable string. |
| StringBuffer | Mutable string. |
| RegExp | Regular expressions. |
| List, Map, Set | Dart provides Collection classes. |
| DateTime | A point in time. |
| Duration | A span of time. |
| Uri | Uniform Resource Identifier |
| Error | Error information |

Dynamic Types (aka Untyped)

You can define untyped variables by declaring them using the ‘var’ or ‘dynamic’ keywords.

- The ‘var’ keyword declares a variable without specifying its type, leaving the variable as a dynamic.
- The ‘dynamic’ keyword declares a variable of the type ‘dynamic’ with optional typing.

There is a difference, but it is subtle.

```
void main() {  
    print (multiplyMethod1(2,4));  
    print (multiplyMethod2(2,4));  
}  
  
dynamic multiplyMethod1(int a, int b){  
    return a * b;  
}  
  
var multiplyMethod2(int a, int b){  
    return a * b;  
}
```

This code wont compile. Dartpad displays the following error:

```
Error compiling to JavaScript: main.dart:10:1: Error: The  
return type can't be 'var'. var multiplyMethod2(int a, int b){  
  ^^^ Error: Compilation failed.
```

This is because methods need to return a type and a ‘var’ does not specify a type.

Type Inference

Often, the variable types are ‘inferred’ when the program runs. In other words, when the program runs, the runtime figures out what the variable types are

based on the values they are set to. This usually works well – see ('Example of Inference #1') but can cause problems if a variable type is inferred at one point in the code then another type is inferred later on – see 'Example of Inference #2' below.

Example of Inference #1:

```
void main() {  
    dynamic x = 1;  
    if (x is int){  
        print('integer');  
    }  
}
```

Output

```
| integer |
```

Example of Inference #2:

```
void main() {  
    dynamic x = 'test';  
    if (x is String){  
        print('String');  
    }  
    x += 1;  
}
```

Output

```
String Uncaught exception: TypeError: 1: type 'JSInt' is not a subtype of type 'String'
```

Type Matching

Dart allows users to check for types using the ‘is’ keyword.

Example Code

```
main(){
    printType(23);
    printType('mark');
}

printType(dynamic d){
    if (d is int){
        print ('Its an Integer');
    }
    if (d is String){
        print ('Its a String');
    }
}
```

Output

```
Its an Integer
Its a String
```

Type Information

Dart gives the developer a way to get information about an Object's type at runtime. You can use Object's runtimeType property, which returns a Type object.

Example Code

```
void main() {  
  var v1 = 10;  
  print(v1.runtimeType);  
  
  var v2 = 'hello';  
  print(v2.runtimeType);  
}
```

Output

```
int  
String
```

Strings

Interpolation

One very useful feature of Dart is its string interpolation. You can put the value of an expression inside a string by using \${expression}.

Example Code

```
class Person{  
    String firstName;  
    String lastName;  
    int age;  
    Person(this.firstName, this.lastName, this.age);  
}  
  
main(){  
    Person p = new Person('mark','smith', 22);  
    print('The persons name is ${p.firstName} ${p.lastName}  
and he is ${p.age}');  
}
```

Output

```
| The persons name is mark smith and he is 22 |
```

Raw Strings

In Dart, normally you can add escape characters to format your string. For example: ‘\n’ means ‘new line’. However, you can prefix the string with an ‘r’ to indicate to tell Dart to treat the string differently, to ignore escape characters.

Example Code – ‘New Lines’:

```
main(){  
    print('this\nstring\nhas\nescape\ncharacters');
```

```
    print("");
    print(r'this\nstring\nhas\nescape\ncharacters');
}
```

Output

```
this
string
has
escape
characters

this\nstring\nhas\nescape\ncharacters
```

Example Code – ‘Dollar Sign’:

```
void main() {
    double price = 100.75;
    print('Price is: \$\$${price}');
}
```

Output

```
| Price is: $100.75 |
```

Runes

Runes are also special characters encoded into a string.
Here is a link with a lot of the run codes:

<https://www.compart.com/en/unicode/block/U+1F300>

Example Code

```
main() {  
  var clapping = '\u{1f44f}';  
  print(clapping);  
}
```

Output



Object-Orientated Language Features

Modules

Unlike Java and C#, Dart allows you to declare multiple objects within a single Dart file.

This has made our example code a single cut-n-paste!

Private Classes, Variables & Methods

Unlike Java, Dart doesn't have the keywords public, protected, and private to specify the visibilities of fields or properties. If a class name, instance variable or method starts with an underscore, it's private and cannot be accessed outside the Dart file in which it is declared.

You should replace:

```
class ContactInfo {  
  private String name;  
  private String phone;
```

```
| }
```

with

```
| class ContactInfo {  
|     String _name;  
|     String _phone;  
| }
```

Constructors

Default Constructor

If you do not specify a constructor, a default constructor will be created for you without arguments. If you do specify a constructor, the default constructor won't be created for you.

Constructor Syntax Shortcut

If you want to set the value of an instance variable in a constructor, you can use the ‘this.[instance variable name]’ to set it in the constructor signature.

Example Code

```
| class Name{  
|     String firstName;  
|     String lastName;  
  
|     Name(this.firstName, this.lastName);  
| }
```

```
main(){  
    Name name = new Name('mark','smith');  
    print(name.firstName);  
    print(name.lastName);  
}
```

Output

```
mark  
smith
```

New Keyword

Dart doesn't need you to use the 'new' keyword when invoking constructors. However, you can keep it if you want.

Example Code

```
void main() {  
    Car car = Car("BMW","M3");  
    print(car.getBadge());  
  
    Car car2 = new Car("BMW","M3");  
    print(car2.getBadge());  
}  
  
class Car{  
    String _make;  
    String _model;  
  
    Car(this._make, this._model){}  
}
```

```
String getBadge(){  
    return _make + " - " + _model;  
}  
}
```

Output

```
BMW - M3  
BMW - M3
```

Named Constructors

Dart allows named constructors and I have found them very useful indeed if you want to instantiate the same class in different ways. Named constructors (if named correctly) can also improve code readability & intent.

Example

A good example of a Flutter class that uses multiple named constructors is EdgeInsets:

- EdgeInsets.fromLTRB
- EdgeInsets.all
- EdgeInsets.only
- EdgeInsets.symmetric
- EdgeInsets.fromWindowPadding

Example Code

```
class ProcessingResult{  
    bool _error;  
    String _errorMessage;
```

```
ProcessingResult.success(){
    _error = false;
    _errorMessage = "";
}

ProcessingResult.failure(this._errorMessage){ //shortcut
    this._error = true;
}

String toString(){
    return 'Error: ' + _error.toString() + ' Message: ' +
_errorMessage;
}

void main() {
    print(ProcessingResult.success().toString());
    print(ProcessingResult.failure('it broke').toString());
}
```

Output

```
Error: false Message:  
Error: true Message: it broke
```

Constructor Parameters

Constructors can accept different kinds of parameters, similar to methods.

Factory Constructors

You can use the factory keyword when implementing a constructor that doesn't always create a new instance of its class. The factory keyword allows you to return a variable at the end of the constructor. This is useful when you want the constructor to return an instance from a variable or a cache.

Example Code

```
class Printer{  
    static final Printer _singleton = Printer._construct();  
  
    factory Printer(){  
        return _singleton;  
    }  
  
    Printer._construct(){  
        print('private constructor');  
    }  
  
    printSomething(String text){  
        print(text);  
    }  
}  
  
void main() {  
    Printer().printSomething("this");  
}
```

```
    Printer().printSomething("and");
    Printer().printSomething("that");
}
```

Output

Note how the constructor was only invoked once.

```
private constructor
this
and
that
```

Instance Variables

Unspecified Visibility

You don't have to specify the visibility of instance variables and if you don't then they are made public.

```
class Name {
  String firstName;
  String lastName;
}
```

Default Values

The default values of instance variables are null.

Constructor and Method Parameters

Flutter is very flexible in regard to constructor & method parameters. There are several different kinds:

1. Positional Required
2. Positional Optional
3. Named

1. Parameters - Positional Required

These are declared first.

These are required.

Constructor with required parameters:

```
class Car{  
    String _make;  
    String _model;  
    Car(this._make,this._model){}  
}
```

2. Parameters - Positional Optional

These are declared second.

You can make parameters optional, by using the square brackets.

If an optional parameter is not supplied, it has a null value.

Example Code

```
void main() {  
    Car car1 = Car("Nissan","350Z");  
    Car car2 = Car("Nissan");
```

```
}
```

```
class Car{
```

```
    String _make;
```

```
    String _model;
```

```
    Car(this._make,[this._model]) {
```

```
        print('${_make} ${_model}');
```

```
    }
```

```
}
```

Output

```
Nissan 350Z
```

```
Nissan null
```

3. Parameters - Named

All named parameters are optional.

These are declared last.

You can make parameters named, by using the curly brackets.

If a named parameter is not supplied, it has a null value.

Example Code

```
void main() {
```

```
    Car car1 = Car("Nissan", model:"350Z", color: "yellow");
```

```
    Car car2 = Car("Nissan", color:"red");
```

```
    Car car3 = Car("Nissan");
```

```
}
```

```
class Car{
```

```
    String make;
```

```
    String model;
```

```
    String color;
```

```
    Car(this.make,{this.model,this.color}){
```

```
        print('${make}${getOptional(model)}${getOptional(color)})'
```

```
    }
```



```
    String getOptional(String str) {
```

```
        return str == null ? "" : " " + str;
```

```
    }
```

```
}
```

Output

```
Nissan 350Z yellow
```

```
Nissan red
```

```
Nissan
```

Required Decorator

You can add the ‘@required’ decorator to named parameters to make them required.

This is not a part of Dart, but it is part of Flutter.
Therefore, it won’t work with Dartpad.

Example Code

We define a constructor for SelectButton that requires

both ‘text’ and ‘onTap’ named parameters.

```
| SelectButton({@required this.text, @required this.onTap}); |
```

If you declare a named parameter as ‘@required’ and the developer writes code that does not supply that parameter:

```
| SelectButton(text: "YES"), |
```

then the following compilation error occurs:

```
| warning: The parameter 'onTap' is required.  
| (missing_required_param at [yes_no] lib/main.dart:58) |
```

Interfaces

Dart uses implicit interfaces.

Example Code

```
abstract class IsSilly {  
  void makePeopleLaugh();  
}  
  
class Clown implements IsSilly {  
  void makePeopleLaugh() {  
    // Here is where the magic happens  
  }  
}  
  
class Comedian implements IsSilly {
```

```
void makePeopleLaugh() {  
    // Here is where the magic happens  
}  
}
```

Further Reading

<https://www.dartlang.org/guides/language/language-tour - implicit-interfaces>

Other

Method Cascades

Method cascades can help with the brevity of your code.

Example Code

```
class Logger {  
    void log(dynamic v){  
        print(DateTime.now().toString() + ' ' + v);  
    }  
}  
main(){  
  
    // Without method cascades  
    new Logger().log('program started');  
    new Logger().log('doing something');  
    new Logger().log('program finished');
```

```
// With method cascades  
new Logger()  
    ..log('program started')  
    ..log('going something')  
    ..log('program finished');  
}
```

Output

```
2018-12-30 09:28:39.686 program started  
2018-12-30 09:28:39.686 doing something  
2018-12-30 09:28:39.686 program finished  
2018-12-30 09:28:39.686 program started  
2018-12-30 09:28:39.686 going something  
2018-12-30 09:28:39.686 program finished
```

6. More Advanced Dart

Introduction

The purpose of this chapter is to introduce some of the more advanced Dart concepts and syntaxes.

Arrow Functions (Lambdas)

Dart offers arrow functions, which enable the developer to shorten single-line functions that calculate & return something.

You can use:

| => xxx |

instead of:

| { return xxx; } |

Arrow functions are often used by event handlers and when you set state (more on that later).

Example Code

```
num divideNonLambda(num arg1, num arg2) {  
    return arg1 / arg2;  
}  
  
num divideLambda(num arg1, num arg2) => arg1 / arg2;  
  
void main() {  
    print('non-lambda ${divideNonLambda(6, 2)}');
```

```
print('non-lambda ${divideNonLambda(9, 2)}');
print('non-lambda ${divideNonLambda(9, 2.5)}');

print('lambda ${divideLambda(6, 2)}');
print('lambda ${divideLambda(9, 2)}');
print('lambda ${divideLambda(9, 2.5)}');

}
```

Output

```
non-lambda 3
non-lambda 4.5
non-lambda 3.6
lambda 3
lambda 4.5
lambda 3.6
```

Operator Overloading

In Dart, you compare equality using the ‘==’ operator rather than an ‘equals’ method. Sometimes you need to override it this operator in your class to ensure that instances of the classes are compared correctly.

Example

If you want to be able to compare two Car objects for equality in this way:

```
| car1 == car2 |
```

and your equality test is:

‘car make and model should match’

then you would have similar code to that below:

```
class Car {  
  String _make;  
  String _model;  
  String _imageSrc;  
  
  Car(this._make, this._model, this._imageSrc);  
  
  operator ==(other) =>  
    (other is Car) && (_make == other._make) && (_model  
    == other._model);  
  
  int get hashCode => _make.hashCode ^ _model.hashCode ^  
    _imageSrc.hashCode;  
}
```

Warning - hashCode

Note that when you override the ‘==’, you need to override the ‘hashCode’ method as well. If you don’t do that then Flutter will give you a warning.

You should override the two together because the collections framework uses the ‘hashCode’ method to determine equality, array indexes etc. You don’t want

equality working in one place and not the other.

Reflection

Reflection allows the inspection of classes, interfaces, fields and methods at runtime without knowing the names of the interfaces, fields, methods at compile time. It enables software to inspect itself. For example, one class can inspect another class (or itself) to see what methods it has available. It also allows instantiation of new objects and invocation of methods. Dart has a library called ‘mirrors’ that enables developers to use reflection in Dart code.

Mixins

A Mixin is a class that contains methods for use by other classes without it having to be the parent class of those other classes.

So, a Mixin is a class you can use code from without having to inherit from.

You can refer to the [Mixins](#) chapter.

Collections

Introduction

When developing, you often need to keep track of

information (objects) in memory. This enables you to search them, sort them, insert them, manipulate them or delete them. That is what the Collection classes are for. Collection classes are used all the time.

Dart offers support for Collections in both its core library and its collection library. The most-commonly used Collection classes are maintained in the core library and the more specific ones are maintained in the collection library.

Lists

A List is an ordered Collection (sometimes called a sequence). Lists may contain duplicate elements.

Unlike other languages, an Array and a List have been combined together and are the same thing. Note how the List in the example below is declared using square brackets, which are normally used for declaring Arrays.

Example Code

This dart code creates a list then sorts it:

```
| class Person{  
|   String _firstName;  
|   String _lastName;
```

```
String _phone;

Person(this._firstName, this._lastName, this._phone);

toString(){
    return "${_firstName} ${_lastName} ${_phone}";
}

}

void main() {
    List<Person> list = [
        Person("Mark", "Clow", "4043124462"),
        Person("Brant", "Sandermine", "4243124462"),
        Person("Phillip", "Perry", "4243124444")
    ];
    print("Not sorted: ${list}");

    list.sort((a, b) => a._firstName.compareTo(b._firstName));
    print("Sorted by first name: ${list}");

    list.sort((a, b) => a._lastName.compareTo(b._lastName));
    print("Sorted by last name: ${list}");
}
```

Output

```
Not sorted: [Mark Clow 4043124462, Brant Sandermine  
4243124462, Phillip Perry 4243124444]
```

```
Sorted by first name: [Brant Sandermine 4243124462, Mark  
Clow 4043124462, Phillip Perry 4243124444]
```

Sorted by last name: [Brant Sandermine 4243124462, Mark Clow 4043124462, Phillip Perry 4243124444]

Flutter Uses Lists Everywhere!

When you write UI code in Flutter, you will end up using Lists all the time.

In the example code below, we use a list to specify the child widgets of a parent widget.

Note the use of Generics ('<Widget>') to specify that the list is of objects of the datatype 'Widget'. Generics are optional.

```
children: <Widget>[  
    new Text(  
        'You have pushed the button this many times:',  
    ),  
    new Text(  
        '$_counter',  
        style: Theme.of(context).textTheme.display1,  
    ),  
],
```

Maps

An object that maps keys to values. Both keys and values in a map may be of any type. A Map is a dynamic collection. In other words, Maps can grow and shrink at runtime.

Example Code

```
void main() {  
    Map<String, String> stateNamesByStateCode =  
        {"AL": "Alamaba",  
         "AK": "Alaska",  
         "AR": "Arkansas",  
         "AZ": "Arizona"};  
  
    stateNamesByStateCode["GA"] = "Georgia";  
  
    for (String key in stateNamesByStateCode.keys){  
        print(stateNamesByStateCode[key]);  
    }  
  
    print("\nGet just one: ${stateNamesByStateCode["AK"]});  
}
```

Output

```
Alamaba  
Alaska  
Arkansas  
Arizona  
Georgia  
  
Just one: Alaska
```

More-Specific Collection Classes

These classes are contained in the ‘dart:collection’ library.

To use this library in your code:

```
| import 'dart:collection'; |
```

Assertions

When you are developing code, you will frequently come across bugs, where things aren’t going as expected. For example, you have a variable with a value that you never expected.

This is where assertions come in. An assertion is a statement that something is expected to be always true at that point in the code. If not, the assertion will throw an exception.

This is a form of Defensive Programming.

Example Code

```
void main() {  
    // .. some good code that calculates age  
    int age1 = 50;  
    checkAge(age1);  
    // .. some good code that calculates age  
  
    // .. some bad code that calculates age incorrectly  
    int age2 = 150;  
    checkAge(age2);
```

```
// .. some bad code that calculates age incorrectly
}

void checkAge(int age) {
  assert(age < 112, "bad age ${age}");
}
```

Output

```
Uncaught exception:  
Assertion failed: "bad age 150"
```

Assertions & Modes (Flutter)

When you are developing your Dart code, you can add assertions to check that it is working as expected. Later on (once the code is mostly bug-free), you can run the same code without the assertions being executed (without the assertions slowing things down).

You develop your Flutter code in Checked (or Debug) Mode, which checks things like assertions. It also turns on the Dart Observatory. More on that here: [Dart Observatory](#). Later on, you can deploy the compiled code that runs in Release mode, speeding things up.

Further Reading

<https://github.com/flutter/flutter/wiki/Flutter's-modes>

Errors & Exceptions

Why Have Error & Exception Handling?

Most software systems are complicated and written by a team of people.

Complexity arises from multiple sources:

- The business domain.
- The act of writing software.
- From multiple people working together, each one having different viewpoints.
- etc

The complexity can result in misunderstandings, errors & exceptions.

This is not the end of the world if the code has good error handling.

- If you don't handle your errors & exceptions, your software may act unpredictably, and users may suffer a catastrophic error without knowing it or being able to detect when it happened.
- If you do handle your errors & exceptions, the user may be able to continue using the program even

with the error / exception and the developers can find the problems over time and improve the software.

Good error & exception handling should not blind the end user with technical jargon, but it should also provide enough information for the developers to trace down the problem.

Dart can throw Errors & Exceptions when problems occur running a Dart program. When an Error or an Exception occurs, normal flow of the program is disrupted, and the program terminates abnormally.



Errors

Errors are serious issues that cannot be caught and ‘dealt with’. Non-recoverable.

Examples

- RangeError – programmatic bug where user is

- attempting to use an invalid index to retrieve a List element.
- OutOfMemoryError

Exceptions

Exceptions are less-serious issues that can be caught and ‘dealt with’.

Recoverable.

Examples

- FormatException – could not parse a String.

Handling Errors

Trying to handle non-recoverable errors is impossible. How can you catch and just handle an out of memory error?

The best thing to do is to log what happened and where so that the developers can deal with them. The approach to this is to add a handler to the top level of your application, for example Sentry or Catcher.

Further Reading

<https://medium.com/flutter-community/handling-flutter-errors-with-catcher-efce74397862>

Handling Exceptions

Try to handle these to prevent the application from terminating abruptly. If you want your code to handle exceptions then you need to place it in a ‘try..catch..finally’ block. The finally part is optional.

Finally

Dart also provides a finally block that will always be executed no matter if any exception is thrown or not.

```
void main() {  
    try {  
        // do something here  
    } catch (e) {  
        // print exception  
        print(e);  
    } finally {  
        // always executed  
        print('I will always be executed!');  
    }  
}
```

Catch Exception

The first argument to the catch is the Exception.

Example Code

This code catches the Exception and prints it out.

```
void main() {  
    print('start');  
    try {  
        int.parse("mark");  
    } catch (ex) {  
        print(ex);  
    }  
    print('finish');  
}
```

Output

```
start  
FormatException: mark  
finish
```

Catch Exception and Stack Trace

The second argument to the catch is the StackTrace.

Example Code

This code catches the Exception and StackTrace. It prints out the StackTrace.

```
void main() {  
    print('start');  
    try {  
        int.parse("mark");  
    } catch (ex) {  
        print(ex);  
        print(ex.stackTrace);  
    }  
}
```

```
    } catch (ex, stacktrace) {  
      print(stacktrace);  
    }  
    print('finish');  
  }
```

Output

```
start  
FormatException: mark  
FormatException: mark  
  at Object.wrapException (<anonymous>:370:17)  
  at Object.int_parse (<anonymous>:1555:15)  
  at main (<anonymous>:1702:11)  
  at dartMainRunner (<anonymous>:9:5)  
  at <anonymous>:2206:7  
  at <anonymous>:2192:7  
  at dartProgram (<anonymous>:2203:5)  
  at <anonymous>:2210:3  
  at replaceJavaScript  
(https://dartpad.dartlang.org/scripts/frame.html:39:17)  
  at https://dartpad.dartlang.org/scripts/frame.html:69:7  
finish
```

Catch Specific Exceptions

If you know you want to catch a specific Exception then you can use an ‘on’ instead of a ‘catch’. Consider

leaving a ‘catch’ at the bottom to catch other Exceptions.

You can optionally add the ‘catch(e)’ or catch(e, s)’ after if you want the Exception and StackTrace data as arguments.

Example Code

```
void main() {  
    print('start');  
    try {  
        int.parse("mark");  
    } on FormatException{  
        print('invalid string');  
    } catch (ex,stacktrace) {  
        print(stacktrace);  
    }  
    print('finish');  
}
```

Output

```
start  
invalid string  
finish
```

Throw Exception

To throw an Exception simply use the ‘throws’

keyword and instantiate the Exception.

Example Code

```
| throw new TooOldForServiceException(); |
```

Rethrow Exception

Once you have caught an Exception, you have the option of rethrowing it so that it bubbles up to the next level. So, you could catch an Exception, log it then rethrow it so it is dealt with at a higher level.

Example Code

```
void misbehave() {  
    try {  
        dynamic foo = true;  
        print(foo++); // Runtime error  
    } catch (e) {  
        print('misbehave() partially handled ${e.runtimeType}.');  
        rethrow; // Allow callers to see the exception.  
    }  
}  
  
void main() {  
    try {  
        misbehave();  
    } catch (e) {  
        print('main() finished handling ${e.runtimeType}.');  
    }  
}
```

```
|    }  
|}  
|
```

Output

```
| misbehave() partially handled JsNoSuchMethodError.  
| main() finished handling JsNoSuchMethodError.
```

Create Custom Exceptions

It is very simple to create your own custom Exception.
Simply implement the Exception interface.

Example Code

```
class TooOldForServiceException implements Exception {  
    Cadet _cadet;  
  
    TooOldForServiceException(this._cadet);  
  
    toString(){  
        return "${_cadet.name} is too old to be in military  
service.";  
    }  
}  
  
class Cadet {  
    String _name;  
    int _age;  
  
    Cadet(this._name, this._age);  
}
```

```
get age{
    return _age;
}

get name{
    return _name;
}

void main() {
    print('start');

    List<Cadet> cadetList = [
        Cadet("Tom", 21),
        Cadet("Dick", 37),
        Cadet("Harry", 51),
        Cadet("Mark", 52),
    ];

    List<Cadet> validCadetList = [];
    for (Cadet cadet in cadetList){
        try {
            validateCadet(cadet);
            validCadetList.add(cadet);
        } on TooOldForServiceException catch(ex) {
            print(ex);
        } // .. other validation exceptions ...
    }
}
```

```
    }

    print('finish: ${validCadetList.length} of ${cadetList.length}
cadets are valid.');

}

void validateCadet(Cadet cadet){
    if (cadet.age > 50){
        throw new TooOldForServiceException(cadet);
    }
    // .. other validations ...
}
```

Output

```
start
Harry is too old to be in military service.
Mark is too old to be in military service.
finish: 2 of 4 cadets are valid.
```

Console Output

Dart allows you to print to the console using the ‘print’ command.

Remember the following:

- Printing a variable attempts to call its ‘`toString()`’ method go get what to print.
- You can use string interpolation and special characters to format the output.

Example Code

```
void main() {  
    int oneVariable = 12;  
    String anotherVariable = 'some text';  
    print('noneVariable: ${oneVariable} \n\nanotherVariable:  
\\'$anotherVariable}\\');  
}
```

Output

```
noneVariable: 12  
anotherVariable: 'some text'
```

Asynchronicity

Introduction

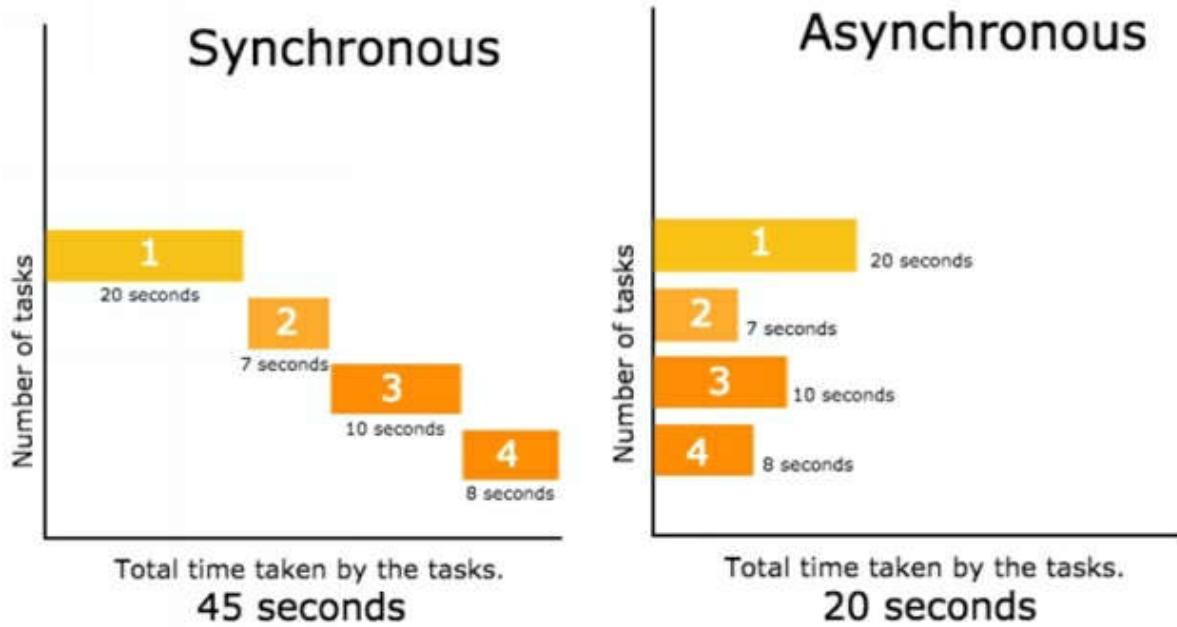
Asynchronicity is the ability to do multiple things at the same time.

Example

When a modern web application needs to get data from a server, it sends out a request and waits for the result to come back. However, the application should still be able to do things in the meantime, like respond to user input.

Doing Multiple Things at the Same Time Can

Save Time



Future

Normally an asynchronous operation results in something, you have a method with asynchronous code that returns something once its finished.

A Future starts off as uncompleted then later ends up being completed (or completed with an error).

Example

The user communicates with a web server to get information and returns the information. Dart uses the Future object to represent the result of an asynchronous operation, starting off as incomplete then later on completed with a value.

Result Type

Futures can complete with result objects. These objects are generics, i.e. they have a specified type.

Example 1: if you are asynchronously getting a Customer object, you would use a Future<Customer>.

Example 2: if your asynchronous operation is not returning any object, you would use a Future<void>.

Exceptions

Futures can fail to complete and can result in exceptions, which you can catch.

Invoking and Handling Asynchronous Operations

Dart offers two ways of handling asynchronous code: using the Future API and using Async-Await. The Future API is the older, more established way of doing things and the Async-Await is the more convenient modern way.

Future API

Before async and await were added in Dart 1.9, you had to use the Future API. You might still see the Future API used in older code and in code that needs

more functionality than `async-await` offers.

As an asynchronous operation can have two possible outcomes (success and failure, otherwise knowns completion and error), the Future API enables a developer to call asynchronous code with callback handlers, one for success and one for failure (optional). The success handler is the ‘then’ and the failure handler is the ‘`catchError`’.

Exercise

This exercise shows how we can asynchronously run some code that creates a string of numbers using the Future API (callbacks).

Step 1

Open your browser and navigate to

<https://dartpad.dartlang.org/>

Step 2

Paste the following code into the left-side.

```
import 'dart:async';

String countUp(int count){
  print('start count up');
  StringBuffer sb = new StringBuffer();
  for (int i = 0; i < count; i++) {
    sb.write(" ${i}");
  }
  return sb.toString();
}
```

```
    }

    print('finish count up');
    return sb.toString();
}

Future<String> createFutureCounter(int count) {
    return new Future(() { return countUp(count); });
}

void main() {
    print('start main');
    Future<String> future = createFutureCounter(100);
    print('adding Future API callbacks');
    future.then((value) => handleCompletion(value));
    print('finish main');
}

void handleError(err){
    print('Async operation errored: ${err}');
}

void handleCompletion(value){
    print('Async operation succeeded: ${value}');
}
```

Step 3

Hit the run button and you should see the following output:

```
start main
adding Future API callbacks
finish main
start count up
finish count up
Async operation succeeded: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54
55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74
75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94
95 96 97 98 99
```

Step 4 – Summary So Far

- The ‘main’ method is short-lived. It calls ‘createFutureCounter’, is returned a future, adds a callback to the future and finishes. It finishes almost immediately, that means that it was not blocked by invocation of heavy synchronous code.
- The ‘createFutureCounter’ method is called by the main and returns a new Future object containing a lambda which is executed asynchronously, calling the ‘countUp’ method.
- The ‘countUp’ method then does the relatively slow work of counting up the numbers.
- Once the ‘count up’ completes then the callback (the one that was added in the ‘main’ method) is fired and we see ‘Async operation succeeded’.

Step 5 – Add Error Handling

Replace the code in the left side with the following:

```
import 'dart:async';

String countUp(int count){
    print('start count up');
    StringBuffer sb = new StringBuffer();
    for (int i = 0; i < count; i++) {
        if (i > 500){
            throw new Exception("Over 500 not allowed.");
        }
        sb.write(" ${i}");
    }
    print('finish count up');
    return sb.toString();
}

Future<String> createFutureCounter(int count) {
    return new Future(() { return countUp(count); });
}

void main() {
    print('start main');
    Future<String> future = createFutureCounter(1000);
    print('adding Future API callbacks');
    future.then((value) =>
handleCompletion(value)).catchError((err) =>
handleError(err));
```

```
    print('finish main');
}

void handleCompletion(value){
    print('Async operation succeeded: ${value}');
}

void handleError(err){
    print('Async operation errored: ${err}');
}
```

Step 6

Hit the run button and you should see the following output:

```
start main
adding Future API callbacks
finish main
start count up
Async operation errored: Exception: Over 500 not allowed.
```

Step 7 – Final Summary

- The ‘main’ method is short-lived. It calls ‘createFutureCounter’, is returned a future, adds two callbacks to the future (one for completion, one for error) and finishes. It finishes almost immediately, that means that it was not blocked by invocation of heavy synchronous code.

- As before, the ‘createFutureCounter’ method is called by the main and returns a new Future object containing a lambda which is executed asynchronously, calling the ‘countUp’ method.
- The ‘countUp’ method then does the relatively slow work of counting up the numbers but artificially throws an Exception once it gets to 500.
- The ‘count up’ never completes but invokes the ‘error’ callback (the second one that was added in the ‘main’ method) is fired and we see ‘Async operation errored’.

Async & Await Keywords

Async

When an async method is called, a Future is immediately returned, and the body of the method is executed later. Later on, as the body of the async function is executed, the Future returned by the function call will be completed along with its result. At the end of the async method, the value (from the completed Future) can be returned.

Await

Await expressions are used in async methods. They

enable you to invoke asynchronous code (that returns a Future). Once the asynchronous code is invoked, the currently running function is suspended until the Future has completed or there is an Error or Exception.

Exercise

This exercise shows how we can asynchronously run some code that creates a string of numbers using the `Async & Await` keywords.

Step 1

Open your browser and navigate to
<https://dartpad.dartlang.org/>

Step 2

Paste the following code into the left-side.

```
import 'dart:async';

String countUp(int count) {
  print('start count up');
  StringBuffer sb = new StringBuffer();
  for (int i = 0; i < count; i++) {
    sb.write(" ${i}");
  }
  print('finish count up');
  return sb.toString();
}
```

```
Future<String> createFutureCounter(int count) {  
    return new Future(() {  
        return countUp(count);  
    });  
}  
  
void countUpAsynchronously(int count) async {  
    print('Async operation start');  
    String value = await createFutureCounter(count);  
    print('Async operation succeeded: ${value}');  
}  
  
void main() {  
    print('start main');  
    countUpAsynchronously(100);  
    print('finish main');  
}
```

Step 3

Hit the run button and you should see the following output:

```
start main  
Async operation start  
finish main  
start count up  
finish count up  
Async operation succeeded: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14  
15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
```

```
35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54  
55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74  
75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94  
95 96 97 98 99
```

Step 4 – Summary So Far

- The ‘main’ method is short-lived. It calls ‘countUpAsynchronously’ and exits.
- The ‘countUpAsynchronously’ method is an `async` method. That means a `Future` is immediately returned and the body of the method is executed later. The body of the method is executed after the main completes and it invokes the ‘createFutureCounter’ and waits for it to finish. Once its finished it prints out the counts.
- The ‘createFutureCounter’ method is called by the main and returns a new `Future` object containing a lambda which is executed asynchronously, calling the ‘countUp’ method.

Step 5 – Add Error Handling

Paste the following code into the left-side.

```
import 'dart:async';

String countUp(int count) {
  print('start count up');
  StringBuffer sb = new StringBuffer();
  for (int i = 0; i < count; i++) {
```

```
if (i > 500) {
    throw new Exception("Over 500 not allowed.");
}
sb.write(" ${i}");
}
print('finish count up');
return sb.toString();
}

Future<String> createFutureCounter(int count) {
    return new Future<String>() {
        return countUp(count);
    );
}
}

void countUpAsynchronously(int count) async {
    print('Async operation start');
    String value;
    try {
        value = await createFutureCounter(count);
        print('Async operation succeeded: ${value}');
    } catch (ex) {
        print('Async operation errored: ${ex}');
    }
}

void main() {
```

```
    print('start main');
    countUpAsynchronously(1000);
    print('finish main');
}
```

Step 6

Hit the run button and you should see the following output:

```
start main
Async operation start
finish main
start count up
Async operation errored: Exception: Over 500 not allowed.
```

Step 7 – Final Summary

- The ‘main’ method is short-lived. It calls ‘countUpAsynchronously’ and exits.
- The ‘countUpAsynchronously’ method is an async method. That means a Future is immediately returned and the body of the method is executed later. Later, the body of the method is executed, and it invokes the ‘createFutureCounter’ method.
- The ‘createFutureCounter’ method returns a new Future object containing a lambda which is executed asynchronously, calling the ‘countUp’ method, which throws the Exception. That

exception is then caught by method ‘countUpAsynchronously’ and the exception is printed out.

Reactive Programming

Reactive programming is a declarative programming paradigm concerned with data streams and the propagation of change. With this paradigm, it is possible to express static (e.g., arrays) or dynamic (e.g., event emitters) data streams and write simple code to process these streams as required.

The Dart language has built-in Stream APIs that are well suited for reactive-like programming.

Nulls

Dart has some unexpected ways of dealing with nulls:

?.

The ?. operator short-circuits to null if the left-hand side is null.

??=

The ?? operator returns the left-hand side if it is not null, and the right-hand side otherwise.

Example

Source Code

```
class Person{  
    String _ssn;  
    String _name;  
  
    Person(this._ssn, this._name);  
  
    String get ssn {  
        return _ssn;  
    }  
  
    String get name {  
        return _name;  
    }  
  
}  
  
void main() {  
  
    Person person1 = null;  
    Person person2 = Person("223232323", "Peter Jones");  
  
    String name = person1?.name;  
    print("Person 1 Name: ${name}");  
  
    Person person1IfPossibleOtherwisePerson2OtherwiseNull =  
(person1??=person2);
```

```
name =  
person1IfPossibleOtherwisePerson2OtherwiseNull?.name;  
    print("A Name from Person1 If Possible, Otherwise Person2:  
    ${name}");  
}
```

Outputs

```
Person 1 Name: null  
A Name from Person1 If Possible, Otherwise Person2: Peter  
Jones
```

Static Analysis

When you edit your Dart source code in your project, the Flutter SDK displays an analysis of the code in real time. Android Studio displays this static analysis in the ‘Dart Analysis’ tab at the bottom.

You can modify the static analysis options by adding the file ‘analysis_options.yaml’ to the root of the project.

Example ‘analysis_options.yaml’ File

```
include: package:pedantic/analysis_options.yaml  
  
linter:  
  rules:  
    - camel_case_types
```

```
| analyzer:  
| #  exclude:  
| #    - path/to/excluded/files/**
```

Further Reading

<https://medium.com/dartlang/making-dart-a-better-language-for-ui-f1ccaf9f546c>