4. Introduction to Dart

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

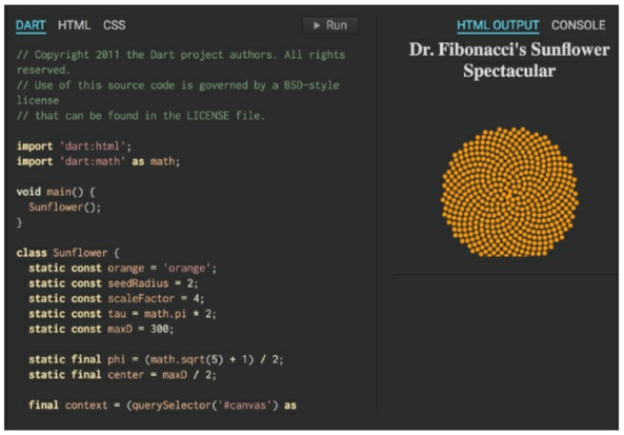
Unlike conventional languages, Dart has been optimized to be deployed to run on a variety of platforms:

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.



```
// 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
```

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 <u>not</u> 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](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 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, 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

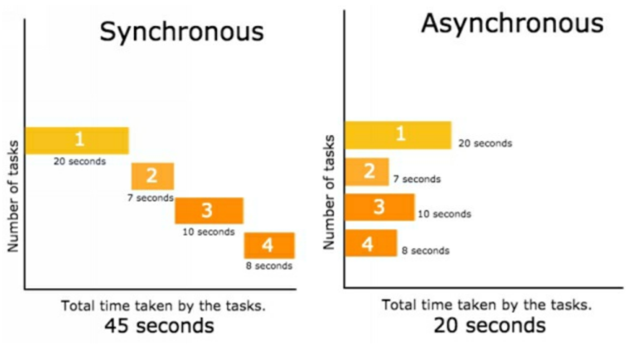
Asynchronicity is the ability to do multiple things at the same time.

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}");
  }
  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 succeded’.

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(() {
    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