APPENDIX A

Dart Language Overview

We use the Dart language when writing Flutter, but Dart isn't very popular (yet). Most developers jump right into Flutter with no prior knowledge of the language. In case that's you, we wanted to get you a little assistance.

In this appendix, we're making no attempt to teach you everything about Dart. Our goal here is to get you just enough Dart to be effective as you write Flutter. So this appendix is brief and to the point. We are only dealing with the things that would otherwise have slowed you down while writing Flutter. An example of this is the *rune* data type. Super cool and innovative Dart feature, but rarely used with Flutter so we omitted it. Please try to be tolerant of us if we left out your favorite feature. We didn't forget it. We just decided it wasn't as important as you thought it should be. Please forgive us.

What is Dart?

Dart is a compiled, statically typed, object-oriented, procedural programming language. It has a very mainstream structure much like other OO languages, making it awfully easy to pick up for folks who have experience with Java, C#, C++, or other OO, C-like languages. And it adds some features that developers in those other languages would not expect but are very cool nonetheless and make the language more than elegant.

In light of all that, we've organized this appendix in two sections:

- Expected features – A quick reference (aka a "cheatsheet") of mainstream features, the bare minimum of what you'll need to know for Flutter. You should tear through this section at lightning speed.
- Unexpected features – These are things that might be a surprise to developers who work in traditional OO languages. Since Dart departs from tradition in these areas, we thought it best to explain them briefly – very briefly.

Expected features – Dart Cheatsheet

This quick reference assumes that you're an experienced OO developer and ignores the stuff that would be painfully obvious to you. For a more in-depth and detailed look at Dart, please visit https://dart.dev/guides/language/language-tour.

Data types

```
int x = 10;          // Integers
double y = 2.0;      // IEEE754 floating point numbers
bool z = true;       // Booleans
String s = "hello"; // Strings
dynamic d;           // Dynamic variables can change types
d = x;               // at any time. Use sparingly!
d = y;
d = z;
```

Arrays/lists

```
// Square brackets means a list/array
// In Dart, arrays and lists are the same thing.
List<dynamic> list = [1, "two", 3];
// Optional angle brackets show the type - Dart supports Generics

// How to iterate a list
for (var d in list) {
  print(d);
}
// Another way to iterate a list
list.forEach((d) => print(d));
// Both of these would print "1", then "two", then "3"
```

Conditional expressions

```
// Traditional if/else statement
int x = 10;
if (x < 100) {
  print('Yes');
} else {
  print('No');
}
// Would print "Yes"

// Dart also supports ternaries
String response = (x < 100) ? 'Yes' : 'No';

// If name is set, use it. Otherwise use 'No name given'
String name;
String res = name ?? 'No name given';
```

```
//the "Elvis" operator. If the object is non-null, evaluate
//the property. Prevents null exceptions from throwing.
print(name?.length);
```

Looping

```
// A for loop
for (int i=1 ; i<10 ; i++) {
  print(i);
}
// Would print 1 thru 9

// A while loop
int i=1;
while(i<10) {
  print(i++);
}
// Would print 1 thru 9
```

Classes

```
class Name {
  String first;
  String last;
  String suffix;
}
class Person {
  // Classes have properties
  int id;
  Name name;    // Another class can be used as a type
  String email;
  String phone;
  // Classes have methods
  void save() {
    // Write to a database somehow.
  }
}
```

Class constructors

```
class Person {
  Name name;
  // Typical constructor
  Person() {
    name = Name();
    name.first = "";
    name.last = "";
  }
}
```

Unexpected things about Dart

The preceding Dart features were unsurprising to any experienced OO developers, but Dart has some pretty cool features that are unique. We'll cover these next, but since they're less familiar, let's take just a sentence or two for each and explain it briefly before giving you a code sample.

Type inference

If I said “x=10.0”, what data type would you guess that x is? Double? And how did you know? Because you looked to the right of the equal sign and *inferred* its type based upon the value being assigned to it. Dart can do that too. If you use the keyword var instead of a data type, Dart will infer what type it is and assign that type:

```
var i = 10;          // i is now defined as an int.
i = 12;              // Works, because 12 is an int.
i = "twelve";        // No! "twelve" is a String and not an int.
var str = "ten";     // str is now defined as a String.
str = "a million";  // Yep, works great.
str = 1000000.0;     // Nope! 1000000.0 is a double, not a string.
```

This is often confused with dynamic. Dynamic can hold any data type and can change at runtime. Var is strongly and statically typed.

final and const

final and const are Dart variable modifiers:

```
final int x = 10;
const double y = 2.0;
```

They both mean that once assigned, the value can’t change. But const goes a little farther – the value is set at compile time and is therefore embedded in the installation bundle.

final means that the variable can’t be reassigned. It does not mean that it can’t change. For example, this is allowed:

```
final Employee e = Employee();
e.employer = "The Bluth Company";
```

e changed, but it wasn't reassigned so that's okay. This, however, is not allowed:

```
const Employee e = Employee();
```

const is not allowed at all because this particular class has properties that could potentially change at runtime. final marks a variable as unchangeable, but const marks a value as unchangeable.

So in summary

- dynamic – Can store any data type. The data type can change at any time.
- var – The data type is inferred from the value on the right side of the "=". The data type does not change.
- final – The variable, once set, cannot be reassigned.
- const – The value is set at compile time, not runtime.

Variables are initialized to null

The default data type for most variables is null. The default return value of a function is null:

```
int x;
double y;
bool z;
String s;
dynamic d;
```

All of the preceding data are null since they haven't been assigned a value yet.

String interpolation with $

Interpolation saves devs from writing string concatenations. This …

```
String fullName = '$first $last, $suffix';
```

… is effectively the same thing as this …

```
String fullName = first + " " + last + ", " + suffix;
```

When the variable is part of a map or an object, the compiler can get confused, so you should wrap the interpolation in curly braces.

```
String fullName = '${name['first']} ${name['last']}';
```

Multiline strings

You can create multiline strings with three single or double quotes:

```
String introduction = """
Now the story of a wealthy family
who lost everything
And the one son who had no choice
but to keep them all together.
""";
```

Spread operator

The "…" operator will spread out the elements of an array, flattening them. This will be very familiar to JavaScript developers:

```
List fiveTo10 = [ 5, 6, 7, 8, 9, 10, ];
// Spreading the inner array with "...":
List numbers = [ 1, 2, 3, 4, ...fiveTo10, 11, 12];
// numbers now has [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

Map<foo, bar>

Maps are like a hash or dictionary. They're merely an object with a set of key-value pairs. The keys and values can be of any type:

```
// You set the value of a Map with curly braces:
Map<String, dynamic> person = {
  "first": "George",
  "last": "Bluth",
  "dob": DateTime.parse("1972-07-16"),
  "email": "amazingGob@gmail.com",
};
// Angle brackets on a Map set the data types of the keys and
// values. They're not required but are a good practice
```

```
// You reference a map member with square brackets:
String introduction = person['first'] + " was born "+
person['dob'].toString();
```

Functions are objects

Just like in JavaScript, functions are first-class objects. They can be passed around like data, returned from a function, passed into a function as a parameter, or set equal to a variable. You can do just about anything with a function that you can do with an object in Java or C#:

```
Function sayHi = (String name) => print('Hello, ' + name);
// You can pass sayHi around like data; it's an object!
Function meToo = sayHi;
meToo("Tobias");
```

Big arrow/Fat arrow

In the preceding example, we also saw the fat arrow syntax. When you have a function that returns a value in one line of code, you can put that returned value on the right side of a "=>" and the argument list on the left side. These are all the same:

```
int triple(int val) {
  return val * 3;
}
Function triple = (int val) {
  return val * 3;
};
Function triple = (int val) => val * 3;
```

The fat arrow is just syntactic sugar, allowing devs to be more expressive with less code.

Named function parameters

Positional parameters are great, but it can be less error-prone (albeit more typing) to have named parameters. Instead of calling a function like this:

```
sendEmail('ceo@bluthcompany.com','Popcorn in the breakroom');
```

You can call it like this:

```
sendEmail(subject:'Popcorn in the breakroom',
  toAddress:'ceo@bluthcompany.com');
```

Now the order of parameters is unimportant. Here is how you'd write the function to use named parameters. Note the curly braces:

```
void sendEmail({String toAddress, String subject}) {
  // send the email here
}
```

Named parameters also work great with class constructors where they are very commonly used in Flutter:

```
class Person {
  Name name;
  // Named parameters
  Person({String firstName, String lastName}) {
    name = Name()..first=firstName..last=lastName;
  }
}
```

Omitting “new” and “this.”

In Dart, it is possible – and encouraged – to avoid the use of the *new* keyword when instantiating a class:

```
// No. Avoid.
Person p = new Person();
// Yes
Person p = Person();
```

In the same way, inside of a class, the use of “this.” to refer to members of the class is not only unneeded because it is assumed, but it is also discouraged. The code is shorter and cleaner:

```
class Name {
  String first;
  String last;
  String suffix;
  String getFullName() {
    // No. Avoid "this.":
    String full=this.first+" "+this.last+", "+this.suffix;
    // Better.
    String full=first+" "+last+", "+suffix;
    return full;
  }
}
```

Class constructor parameter shorthand

Merely a shorter way of writing your Dart classes which receive parameters. When you write the constructor to receive "this.something" and have a class-scoped property with the same name, the compiler writes the assignments so you don't have to:

```
class Person {
  String email;
  String phone;
  // The parameters are assigned to properties automatically
  // because the parameters say "this."
  Person(this.email, this.phone) {}
}
```

The preceding code is equivalent to

```
class Person {
  String email;
  String phone;
  Person(String email, String phone) {
    this.email = email;
    this.phone = phone;
  }
}
```

Private class members

Dart does not use class visibility modifiers such as public, private, protected, package, or friend like other OO languages. All members are public by default. To make a class member private, put an underscore in front of the name:

```
class Person {
  int id;
  String email;
  String phone;
  String _password;

  set password(String value) {
    _password = value;
  }
  String get hashedPassword {
    return sha512.convert(utf8.encode(_password)).toString();
  }
}
```

In that example, id, email, and phone are public. _password is private because the first character in the name is "_", the underscore character.

Mixins

Mixins are baskets of properties and methods that can be added to any class. They look like classes but cannot be instantiated:

```
mixin Employment {
  String employer;
  String businessPhone;
  void callBoss() {
    print('Calling my boss');
  }
}
```

A mixin is added to a class when it uses the “with” keyword:

```
class Employee extends Person with Employment {
  String position;
}
```

This Employee class now has employer and businessPhone properties and a callBoss() method:

```
Employee e = Employee();
e.employer = "The Bluth Company";
e.callBoss();        // An employee can call its boss.
```

Dart, like Java and C#, only supports single inheritance. A class can only extend one thing. But mixin members are added to a class so any class can implement multiple mixins and a mixin can be used in multiple other classes.

The cascade operator (..)

When you see two dots, it means “return this class, but before you do, do something with a property or method.” We might do this

```
Person p = Person()..id=100..email='gob@bluth.com'..save();
```

which would be a more concise way of writing

```
Person p = Person();
p.id=100;
p.email='gob@bluth.com';
p.save();
```

No overloading

Dart does not support overloading methods. This includes constructors.

Named constructors

Since we can't have overloaded constructors, Dart supports a different way of doing essentially the same thing. They're called named constructors and they happen when you write a typical constructor, but you tack on a dot and another name:

```
class Person {
  // Typical constructor
  Person() {
    name = Name()..first=""..last="";
  }
  // A named constructor
  Person.withName({String firstName, String lastName}) {
    name = Name()
      ..first = firstName
      ..last = lastName;
  }
  // Another named constructor
  Person.byId(int id) {
    // Maybe go fetch from a service by the provided id
  }
}
```

And to use these named constructors, do this:

```
Person p = Person();
// p would be a person with a blank first and last name

Person p1 = Person.withName(firstName:"Lindsay",lastName:"Fünke");
// p1 has a first name of "Lindsay" and a last name of "Funke"

Person p3 = Person.byId(100);
// p3 would be fetched based on the id of 100
```

Index