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](https://dart.dev/guides/language/language-tour).

#### Data types

```dart
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);
    i++; // Increment 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:

```dart
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:

```dart
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:

```dart
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 “=” sign. 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:

```dart
// 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#:

```dart
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:

```dart
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:

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

You can call it like this:

```dart
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:

```dart
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;
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;
  }
}

// Better.
String full = first + " +last", +suffix;
return full;
}
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:

```dart
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:

```dart
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:

```dart
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

A
AlertDialog, 154
Android
    emulator, 14, 15
Android Studio, 11
Android Virtual Device (AVD)
    manager, 14, 15
Anti-RaisedButton, 81
API call, 228
API requests, 228, 229
AppBar widget, 103, 104
async, 215
await, 214

B
BLoC, 200, 201
Boolean value
    property, 60
BoxConstraints, 106
BoxFit options, 52
BoxFit.scaleDown, 51
Box model, 124
BoxShape, 173
build.gradle file, 276
Button widgets, 78

C
Cascade operator (..), 300
Class visibility modifiers, 299
Cloud firestore, 257, 258
Cloud functions, 258, 259
ColorCircle, 197
ColorMixer, 195
ColorValueChanger, 197
Column widget, 111
Compile-to-native cross-platform frameworks, 7
Componentization, 34
Container
    alignment property, 126, 127
    <div>, 125
    properties, 125
    size, 128–130
crossAxisAlignment, 115, 117
Cross-platform development categories, 6
CRUD app, API service, 241
    DELETE request, 247
    Flutter app, creation, 243
    GET request, 247
    PeopleList widget, 244–246
    PeopleUpsert.dart, 248–252
INDEX

CRUD app, API service (cont.)
  Pipedream.com, 242
  POST call, 252
  PUT call, 252
  strongly typed business
class, 243, 244
testing, 242
curl command, 242
CustomPainter, 174, 175

D
Dart
  cascade operator (..), 300
  class constructors, 291
  classes, 290, 291
  expected features
    arrays/lists, 289
    conditional
    expressions, 289–290
    data types, 288
    looping, 290
  Flutter, 211
  mixins, 299
  named constructors, 301, 302
  overloading methods, 301
  private class members, 299
  unexpected features
    big arrow/fat arrow, 296
    class constructor, 298
    final and const, 292–293
    function parameters, 296–297
    functions, 295
    map, 295
  multiline strings, 294
  spread operator, 294
  string interpolation, 294
  type inference, 292
  variables, 293
  Debugging, 25
  Development process
    Flutter project, creation, 19, 20
    running, app
      debugging, 25
      hot reloading, 24
      play/debug button, 22
      tethered device, 24
      web app, 23
  DevTools, 12
  Dialog widget, 153
  Dismissible, 83
  Drawer navigation, 144–150
  DropdownButton, 63–65
  DRY principle, 248

E
  Embedded images, 50
  Event loop, 211
  Expanded widget
    example, 117, 118
    flex factor, 119, 120

F
  Firebase Authentication, 259
  Firestore.instance.
collection('Foo'), 279
Flutter app, 207
  async, 215, 216
  await, 214
  code to create the future, 212, 213
  compile-to-native solutions, 7
cross-platform development, 5, 6
Dart language, 211, 287
data from a Future, 213, 214
data with an external server, 207
data within the device, 207
definition, 4
event loop, 211
finding a library, 208, 209
future, 211
Google CRM team, 5
importing the library, 210
including a file, 216
iOS apps, 5
memory to JSON, 221
pubspec.yaml, 210
reading a file, 219, 220
reading JSON, 222, 223
shared preferences, 223
to read preferences, 224
to write preferences, 224
using JSON, 220, 221
using the library, 211
writing a file, 218
Flutter checkbox widget, 60
Flutter DevTools, 12
Flutter doctor, 16, 17
Flutter radio widgets, 61, 62
Flutter SDK, 10
Flutter styles, 160, 162
Flutter Switch widget, 61
Flutter toolchain
  emulators, 13
  flutter doctor, 16
  flutter upgrade, 17, 18
  IDE DevTools, 12
  IDEs (see IDEs)
  SDK, installation, 10
Flutter upgrade, 17, 18
Flutter widgets
  layout widgets, 36
  navigation widgets, 37
  value widgets, 36
fontFamily, 165
Form example, 71, 73, 74, 76
FormField widget, 67, 68
  Form.autovalidate, 69, 71
  save() method, 69
  validator, 69
Form widget, 65–67
FutureBuilder, 234, 235
Genymotion, 14
Gesture
  add a new item, 89
  associate with behavior, 87
INDEX

Gesture (cont.)
  custom widget, 85
  GestureDetector widget, 86
long press, 87
pinching-to-zoom, 92
swiping, 90
GestureArena, 92
GestureDetector widget, 86, 87
Gestures and behaviors, 84
Google Firebase
  adding FlutterFire
    plugins, 277, 278
  advantage, 255
create Android app
  google-services.json file, 275
Gradle files, 276, 277
  id or package name, 273
package name/nickname, 274
time, 273
creating database, 263, 267
creating iOS app, 267, 268, 272
creating project, 260–262
delete, 282, 283
Foo, 279, 280
Podfile.xcworkspace, 271
query, 281
server, 256
tools, 257
  Cloud firestore, 257, 258
  Cloud functions, 258, 259
  firebase authentication, 259
upsert, 281, 282
GoogleService-info.plist, 270
Gradle files, 276, 277
GridView, 131
  GridView.count(), 132–134
  GridView.extent(), 131, 132

H
Hooks, 201
HTTP GET or DELETE request, 230
HTTP responses
  brute force, 233
  FutureBuilder, 234, 235
  StreamBuilder, 236, 237

I
IconButton, 81
Icon widget, 48
IDEs
  Android emulator, 14, 15
  Android Studio and IntelliJ, 11
  DevTools, 12
  iOS simulator, 13
  VS code, 11
Image sizing, 51
Image widget, 36, 49
InputDecoration widget, 56
IntelliJ, 11
Internet, 36
iOS simulator, 13

J
JSON, 220
K
keyboardType property, 58
Kotlin, 5

L
Layout widgets, 36
Lifting state up, 193
ListView.builder, 121, 123
ListView.custom, 122
ListView.separated, 121

M
MainAxisAlignment, 113, 114
MaterialApp widget, 100, 101
Microsoft Visual Studio, 11
Mixins, 299

N
Navigation
dialog, 140
drawers, 139
stacks, 139
tabs, 139
widgets, 37
Network images, 51

O
Omitting “new” and “this.”, 297, 298
onChanged property, 55
onHorizontalDragEnd, 92

P, Q
PATCH request, 231
path_provider 1.3.0, 209
.plist file, 273
Positioned widget, 178–180
POST request, 231
Private class members
cascade operator (..), 300
Mixins, 299, 300
named constructors, 301
overloading, 301
pubspec.yaml, 210
PUT request, 231

R
RaisedButton, 80
React, 34
React Native, 15
Redux, 202
Row and column, widgets, 110, 112
Row widget, 111

S
SafeArea widget, 104, 105
Scaffold widget, 101, 102
ScopedModel, 201
showDialog(), 154
SimpleDialog, 154
INDEX

SizedBox(), 121
Slider, 62
SnackBar widget, 105, 106
.snapshots() method, 279
Spacer(), 121
Stacking widgets, 176, 177
Stack navigation, 140–144
StatefulWidget, 45, 130,
  187, 189, 194
  important rule about, 190, 191
  lifting the state up, 192, 193
  passing down, 191, 192
  state management, 193–197
  usage of stage, 198
StreamBuilder widget, 236, 237, 279
Strongly typed class
  business class, 238
  .fromJson() method, 239, 240
  typed deserialization, 238
Subwidgets, 39, 40

T, U
TabBarView, 151
TabBar widget, 150
TabController, 151
Table widget
  columnWidths, 137
  rows and columns, 134, 135
  TableRow, 135, 136
Tab navigation, 150–153
TextField widget, 55, 188
TextStyle, 163–165
Text widget, 36, 39, 47

V
Value widgets, 36

W
.where() function, 281
Widget
  BLoC, 200, 201
  gestures, 83
  inheritedWidget, 200
  key, 41
  passing value in, 42–44
  provider, 202
  Redux, 202
  ScopedModel, 201
  StatefulWidget, 189, 190
  styling, 159
  value, 36
Widgets, layout
  entire screen, 94
  extra space, 96
  fine-tuned spacing, 97
  ListView, 94, 95
  PersonCard, 94–96
  Row widget, 96
  visual, 97–99
  VS Code, 99
Widget’s size
  layout
    BoxConstraints, 109
    branches, 108
    LimiteBox(), 110
    RenderBox, 106
    unbounded height error, 107, 108
Widgets style
   border, 170, 171
   BorderRadius, 172, 173
   BoxShape, 173–175
   Card widget, 180
   colors, 161–163
   container
      decorations, 168–170
   custom fonts, 165–167
   fonts, 166, 167
   positioned widget, 178–180
   shadows, 168, 169
   stacking widgets, 176, 177
   TextStyle, 163–165
   themes, 181–183

X, Y, Z

Xamarin, 15
.xcworkspace file, 270