3 | Control flow and functions

3.1 If statement

This is probably the most famous statement of any programming language and in Dart it works exactly as you would expect. The `else` is optional and it can be omitted when not needed. You can avoid using brackets in case of one-liner statements.

```
void main() {
    final random = 13;

    if (random % 2 == 0)
        print("Got an even number");
    else
        print("Got an odd number");
}
```

Conditions must be boolean values. In C++ for example you can write `if (0) {...}` where zero is evaluated to `false` but in Dart it doesn't compile; you have to write `if (false) {...}`.

3.1.1 Conditional expressions

In Dart there are two shorthands for conditional expressions that can replace the if-else statement:

- `valueA ?? valueB`. If `valueA` is non-null, `valueA` is returned; otherwise `valueB` is evaluated and then returned. If the definition is too verbose, you can understand this syntax by looking at the following example.

```
String? status;  // This is null
```

```
// isAlive is a String declared somewhere before
if (status != null)
    isAlive = status;
else
    isAlive = "RIP";
```

Basically we want to know whether `status` is null or not and then decide the proper value to assign. The same logic can be expressed in another, more concise way:

```
String? status; // This is null
String isAlive = status ?? "RIP";
```

In the example `isAlive` doesn't need to be nullable as it's guaranteed to be initialized with a string. The `??` operator automatically checks if `status` is null and decides what to do:

- `status` is **not** null: return `status`;
- `status` is null: return the provided "default value" at the right of `??`

It's a very helpful syntax because it guarantees that a variable is properly initialized avoiding unwanted operations with `null`.

- `condition ? A : B;`. If `condition` is true A is returned, otherwise you get B. It's a pretty common pattern among modern languages so you might already be familiar with it.

```
String status;

if (correctAns >= 18)
    status = "Test passed!";
else
    status = "You didn't study enough..."
```

If it looks a bit too verbose, you can rewrite the logic in a more concise way:

```
String status = (correctAns >= 18) ?
    "Test passed!" : "You didn't study enough...";
```

We could call this the "*shorter if*" syntax in which you replace the `if` with the question mark (?) and the `else` with the colon (:). You can omit parenthesis.

3.1.2 Good practices

Simple boolean expressions are easy to read but complicated ones might require documentation and might also not fit well inside a single `if` statement like the following:

```
if ( (A && B || C && !A) || (!(A && C) || B) ) { ... }
```

You might get a headache while trying to figure out what's going on and there are also no comments at all. In such cases, you probably want to make the code more readable by splitting the conditions:

```
final usefulTestName1 = A && B || C && !A;
final usefulTestName2 = !(A && C)

if (usefulTestName1 || usefulTestName2 || B) { ... }
```

For sure it's more understandable and another programmer, or yourself in the future, will be very grateful. We also recommend to not underestimate the usefulness of variable names.

> The point is that you have to keep expressions short and easy to read. Break down long conditions into smaller pieces and give the variables good names to better understand what you want to check.

We also recommend the usage of the *short if* syntax only when there's one condition or at maximum two short ones. The longer the line is the harder it is to understand.

3.2 switch statement

When you have a series of cases to take into account, instead of using a long chain of if-else*s* you should go for the `switch` statement. It can compare many types:

1. compile-time constants
2. enums
3. integers
4. strings
5. classes

Classes must not override == if they want to be compared with this statement. At the bottom there's a `default` label used as fallback if none of the previous cases matches the item being compared.

```
enum Status { Ready, Paused, Terminated }

void main() {
    final status = Status.Paused;

    switch (status) {
        case Status.Ready:
            run();
            break;
        case Status.Paused
            pause();
            break;
        case Status.Terminated
            stop();
            break;
        default
            unknown();
    }
}
```

If the body of the `case` is **NOT** empty you must put a `break` otherwise your code won't compile. When you just want a fall-through to avoid code-replication, leave the body empty. Here's a few examples:

- This code is not going to compile because the first `case` has a body, containing `start()`, but there isn't a `break`.

```
switch (status) {
    case Status.Ready:
        start();
        //missing "break;" here
    case Status.Paused
        pause();
        break;
    }
}
```

- This code instead is fine because the `case` doesn't have a body; the method `pause()` is going to be called when `status` is ready or paused.

```
switch (status) {
    case Status.Ready:
    case Status.Paused
        pause();
        break;
    }
}
```

The above code is equivalent to...

```
switch (status) {
    case Status.Ready:
        pause();
        break;
    case Status.Paused
        pause();
        break;
    }
}
```

... but you should avoid code duplication which is always bad in terms of code maintenance. When you have two or more cases that must execute the same action, use the fall-through approach.

3.3 for and while loops

The iteration with a `for` loop is the most traditional one and doesn't need many explanations. You can omit brackets in case of one-liner statements. The index cannot be nullable (using `int? i = 0` doesn't work).

```
for(var i = 0; i <= 10; ++i)
    print("Number $i");
```

As you'd expect, the output prints a series of "*Number (i)*" in the console. An equivalent version can be written with a classic `while` loop:

```
var i = 0;

while (i <= 10) {
    print("Number $i");
```

```
    ++i;
}
```

The language also has the `do while` loop that always executes at least **one** iteration because the condition is evaluated only at the end of the cycle.

```
var i = 0;

do {
    print("Number $i");
    ++i;
} while (i <= 10)
```

The difference is that the `while` evaluates the condition at the beginning so the loop could never start. The `do while` instead runs at least once because the condition check is placed at the end. If you wanted to alter the flow of the loop you could use:

- `break`. It immediately stops the loop in which it is called. In case of nested loops, only the one whose scope contains `break` is stopped. For example:

  ```
  for (var i = 0; i <= 3; ++i) {    // 1.
      for(var j = 0; j <= 5; ++j) { // 2.
          if (j == 5)
              break;
      }
  }
  ```

  In this case only loop *2* is terminated when `j` is 5 but loop *1* executes normally until `i` reaches 3. In practical terms, we can say `break` stops only 1 loop.

- `continue`. It skips to the next iteration and, like we've seen before, in case of nested loops it does the jump only for the loop containing it, not the others.

3.3.1 for-in loop

There are some cases in which you want to completely traverse a string or a container and you don't care about the index. Look at this very easy example:

```
final List<String> friendsList = ["A", "B", "C", "D", "E"];

for(var i = 0; i < friendsList.length; ++i)
    print(friendsList[i]);
```

That's perfectly fine but you're using `i` just to retrieve the element at the i-th position and nothing more. There are no calculations based on the index as it's just used to traverse the list. In such cases you should do the following:

```
List<String> friendsList = ["A", "B", "C", "D", "E"];

for(final friend in friendsList)
    print(friend);
```

This version is less verbose and clearer. You're still traversing the entire list but now, instead of the index `i`, you have declared `final friend` that represents an item at each iteration.

3.4 Assertions

While writing the code you can use assertions to throw an exception [1] if the given condition evaluates to false. For example:

```
// the method returns a json-encoded string
final json = getJSON();

// if length > 0 is false --> runtime exception
assert(json.length > 0, "String cannot be empty");

// other actions
doParse(json);
```

The first parameter of `assert` must be an expression returning a boolean value. The second parameter is an optional string you can use to tell what's gone wrong; it will appear in the IDE error message window if the condition evaluates to `false`.

> In release mode, every `assert` is **ignored** by the compiler and you're guaranteed that they won't interfere with the execution flow. Assertions work only in debug mode.

When you hit **Run** on Android Studio or VS Code your Flutter app is compiled in debug mode so assertions are enabled.

[1] Exceptions will be discussed in detail in chapter 5

3.5 Good practices

Sometimes you have to implement a complicated algorithm and you don't want to make it even more complex by writing code hard to understand. Here's what we recommend.

- Try to always use brackets, even if they can be omitted, so that you can avoid unexpected behaviors. Imagine you had written this code...

```
// Version 1
if ("A" == "A")
    if ("B" == "B")
        print("Oh well!");
else
    print("Oops...");
```

  ... but in reality you wanted to write this, with a better indentation:

```
// Version 2
if ("A" == "A")
    if ("B" == "B")
        print("Oh well!");
    else
        print("Oops...");
```

  There's a high possibility that, at first glance, in version 1 you associated the `else` to the first `if` but it'd be wrong! While it may seem obvious, in a complex architecture with thousands of lines you might misread and get tricked.

- When you have to traverse an entire list and you **don't** care about the position in which you are during the iteration, use a `for-in` loop. As we've already said, it's less verbose and so more understandable.

Use assertions, in particular when you create Flutter apps, to control the behavior of your software. Don't remove them when you're ready to deploy the code to the production world because they will be automatically discarded.

3.6 The basics of functions

Functions in Dart have the same structure you're used to see in the most popular programming languages and so you'll find this example self-explanatory. You can mark a parameter with `final` but in practice it does nothing.

```
bool checkEven(int value) {
    return value % 2 == 0
}
```

When the body of the function contains only one line, you can omit the braces and the `return` statement in favor of the "arrow syntax". It works with **expressions** and not with statements.

```
// Arrow syntax
bool checkEven(int value) => value % 2 == 0;

// Arrow syntax with method calls
bool checkEven(int value) => someOtherFunction(value);

// Does NOT work
bool checkEven(int value) => if (value % 2 == 0) ... ;
```

The second example is a conditional statement so it doesn't work with the arrow syntax. The first example instead is still a condition but it's written as **expression** and so it works fine.

```
// 1. This function does not return a value
void test() {}

// 2. No return type so this function returns dynamic. Don't do this.
test() {}
```

When you don't need a function to return a value, simply make it `void` like you'd do in Java for example. If you omitted the return type like in (*2.*), the compiler would automatically append `return dynamic` at the end of the body.

```
void test() => print("Alberto");
```

Interestingly, if you have a void function with an one-liner body, you can use the arrow syntax. The function doesn't return anything because of the `void` but you're allowed to do it anyway.

> ℹ Try to **always** specify the return type or use `void`. Avoid ambiguity; you could avoid the return type for laziness (you just don't want to write `void`) but someone else could think you're returning `dynamic` on purpose.

3.6.1 The Function type

Dart is truly an OOP language because even functions are objects and the type is called... `Function`! A return type is required while the parameters list is optional:

```
// Declare a function
bool checkEven(int value) => value % 2 == 0;

void main() {
  // Assign a function to a variable
  bool Function(int) checker = checkEven;

  // Use the variable that represents the function
  print(checker(8)); // true
}
```

It's nothing new: you're just writing a type (`bool Function(int)`), its name (`checker`) and then you're assigning it a value (`checkEven`). You may find this declaration a bit weird because it's made up of many keywords but it's a simple assignment. This is a comparison to clarify the idea:

- `28`: It's an integer and its type is `int`.
- `"Pizza"`: It's a string and its type is `String`.
- `bool checkEven(int value) => ...`: It's a function and its type is `bool Function(int)`.

This particular syntax is very expressive; you have to declare the return type and the **exact** order of the type(s) it takes. In other words, signatures must match. If you think it's too verbose, you can use the typical automatic type deduction you're getting used to see:

```
bool checkEven(int value)  => value % 2 == 0;

void main() {
  final checker1 = checkEven;
  var checker2 = checkEven;

  print(checker1(8)); // true
  print(checker2(8)); // true
}
```

Both `var` and `final` will be evaluated to `bool Function(int)`. There's still something to say

about this type but you'll have to wait until the next chapter where we'll talk about classes and the special `call()` method.

> When you declare a variable you can only write `Function(int) name` without the return type. However, automatic type deduction is generally the best choice because it reduces a lot the verbosity.

It might not seem very useful and we'd agree with you because there is no usage context, at the moment. When you'll arrive at part 2 of the book you'll see that the `Function` type is super handy in Flutter because it's used to create "function callbacks".

3.7 Anonymous functions

So far you've only seen *named functions* such as `bool checkEven(int value)` where *checkEven* is the name. Dart gives you the possibility to create nameless functions called anonymous functions.

```
void main() {
    bool Function(int) isEven = (int value) => value % 2 == 0;

    print(isEven(19)); //false
}
```

This syntax allows you to create functions "*on the fly*" that are immediately assigned to a variable. If you want an anonymous function with no parameters, just leave the parenthesis blank `()`. Of course you can use `final` and `var` to automatically deduce the type.

- **Single line**. You can use the arrow syntax when you have one-liner statements. This example declares a function with no parameters that returns a double.

```
final anon = () => 5.8 + 12;
```

- **Multiple lines**. Use brackets and `return` when you have to implement a logic that's longer than one line.

```
final anon = (String nickname) {
    var myName = "Alberto";
    myName += nickname;
```

```
        return myName;
    };
```

We recommend to always write down the type of the parameter even if it's not required by the compiler. You can decide whether the type has to appear or not. Using `final` and `var` is allowed but it doesn't make much sense.

```
String Function(String) printName = (String n) => n.toUpperCase();
String Function(String) printName = (final n) => n.toUpperCase();
String Function(String) printName = (var n) => n.toUpperCase();
String Function(String) printName = (n) => n.toUpperCase();
```

Any variant compiles with success but none of them is the best option, the decision is up to your discretion. Before moving on, we're going to show a simple scenario you'll encounter many times in Flutter.

```
// 1.
void test(void Function(int) action) {
    // 2.
    final list = [1, 2, 3, 4, 5];

    // 3.
    for(final item in list)
        action(item);
}

void main() {
    // 4.
    test(
        // 5.
        (int value) { print("Number $value"); }
    );
}
```

The `action` parameter commonly known as *callback* because it executes an action given from the outside.

1. This function doesn't return a value because of the `void`. The parameter, called `action`, accepts a `void` function with a single integer value.
2. It's a simple list of integer values

3. We iterate through the entire list and, for each item, we call the function.
4. `test(...);` is how you normally call a function
5. This is an anonymous function returning nothing (`void`) and asking for a single integer parameter.

The flexibility of callbacks lies on the fact that you can reuse the same function `test()` with different implementations. The caller doesn't care about the body of the anonymous function, it just invokes it as long as the signature matches.

```
// The same method (test) outputs different values
// because anonymous functions have different bodies
test( (int value) => print("$value") );
test( (int value) => print("${value + 2}") );
```

You will often encounter the `forEach()` method on collections, which accepts a callback to be executed while elements are traversed. Again, the same function (`forEach()`) is reused multiple times regardless the implementation (thanks to callbacks).

```
void main() {
    // Declare the list
    final list = [1, 2, 3, 4, 5];
    // Iterate
    list.forEach((int x) => print("Number $x"));
}
```

This is an even shorter way that doesn't use a `for-in` loop. You pass an anonymous function to the method and it executes the given action for every item. Pay attention because the documentation suggests to avoid using anonymous functions in `forEach()` calls.

> ℹ A very handy feature you'll see very often is the possibility to put an underscore when one or more parameters of a function aren't needed. For example, in Flutter the `BuildContext` object is often given as a callback param but it's not always essential.
>
> ```
> builder: (BuildContext context) {
>     return Text("Hello");
> }
> ```
>
> Since the variable `context` isn't used, but it must be there anyway to match the method signature, you can use an underscore to "hide" it:

```
builder: (_) {
    return Text("Hello");
}
```

It's less code for you to write and the reader focuses more on what's really important. In case of multiple values that you don't use, just chain a series of underscores.

```
builder: (_, value, __) {
    return Text("$value");
}
```

3.8 Optional parameters

In Dart function parameters can be **optional** in the sense that if you don't provide them, the compiler will assign `null` or a default value you've specified.

3.8.1 Named parameters

In the simplest case, a function can have optional parameters whose names must be explicitly written in order to be assigned. Pay attention to null-safety in case you don't plan to give the variables a default value.

Declaration

```
void test({int? a, int? b}) {
    print("$a");
    print("$b");
}
```

Calling

```
void main() {
    // Prints '2' and '-6'
    test(a: 2, b: -6);
}
```

When calling a function with optional named parameters, the order **doesn't** matter but the names of the variables names must be explicit. For example, you could have called `test(b: -6, a: 2);` and it would have worked anyway. When a parameter is missing, the default value is given:

Declaration

```
void test({int? a, int? b}) {
    print("$a");
    print("$b");
}
```

Calling

```
void main() {
    // Prins '2' and 'null'
    test(a: 2);
}
```

Calling `test(a: 2);` initializes only `a` because `b`, which is omitted, is set to `null` by the compiler. `null` is the default value of nullable types. You can manually give a default value to an optional named parameter just with a simple assignment:

Declaration

```
void test({int? a, int b = 0}) {
    print("$a");
    print("$b");
}
```

Calling

```
void main() {
    // Prints '2' and '0'
    test(a: 2);
}
```

Note that `b` doesn't need to be nullable anymore thanks to the default value. In Dart 2.9 (an lower) nnbd was not enabled so you were able to successfully compile this code, which initializes both `a` and `b` to `null`:

Declaration

```
// Dart 2.9 and lower
void test({int a, int b}) {
    print("$a");
    print("$b");
}
```

Calling

```
// Dart 2.9 and lower
void main() {
    // Prints 'null' and 'null'
    test();
}
```

The `required` modifier, introduced in Dart 2.10 with nnbd, forces an optional parameter to be set. You won't be able to compile if a required parameter is not used when calling the function.

```
void test({int a = 0, required int b}) {
    print("$a");
    print("$b");
```

```
}

void main() {
    test(a: 5, b: 3); // Ok
    test(a: 5);       // Compilation error, 'b' is required
}
```

Even if you had written `required int? b` you'd have to assign `b` anyway because it's required. Version 2.9 of Dart and lower didn't have this keyword: you had to use instead an *annotation* which just produced a warning (and not a compilation error) by default.

```
// Dart 2.9 and lower
void test({int a = 0, @required int b}) {...}
```

In Flutter, for a better readability, some methods only use named optional params together with `required` to force the explicit name in the code.Y ou can mix optional named parameters with "classic" ones:

Declaration

```
void test(int a, {int b = 0}) {
    print("$a");
    print("$b");
}
```

Calling

```
void main() {
    // Prints '2' and '3'
    test(2, b: 3);
}
```

Optional parameters must stay at the end of the list.

```
// it compiles
void test(int a, {int? b}) { }

// it doesn't compile
void test({int? a}, int b) { }
```

3.8.2 Positional parameters

You can also use optional parameters without being forced to write down the name. Optional positional parameters follow the same rules we've just seen for named params but instead of using curly braces (`{ }`) they're declared with square brackets (`[ ]`).

Declaration

```
void test([int? a, int? b]) {
    print("$a");
    print("$b");
}
```

Calling

```
void main() {
    // Prints '2' and '-6'
    test(2, -6);
}
```

All the examples we've made for named parameters also apply here. They really have the same usage but the practical difference is that, in this case, the name of the parameter(s) doesn't have to be written in the function call.

3.9 Nested functions

The language allows you to declare functions inside other functions visible only within the scope in which they're declared. In other words, nested functions can be called only inside the function containing them; if you try from the outside, you'll get a compilation error.

> From a practical side, the **scope** is the "area" surrounded by two brackets `{ }`

This example shows how you can nest two functions, where `testInner()` is called "**outer** function" and `randomValue()` is called "**inner** function".

```
void testInner(int value) {
    // Nested function
    int randomValue() => Random().nextInt(10);

    // Using the nested funcion
    final number = value + randomValue();
    print("$number");
}
```

As we've just seen, functions are types in Dart so a "nested function" is nothing more than a `Function` type assignment. Given this declaration, we're able to successfully compile the following:

```
void main() {
```

```
    // testInner internally calls randomValue
    testInner(20);
}
```

An error is going to occur if we try to directly call `randomValue` from a place that's not inside the scope of its **outer** function.

```
void main() {
    // Compilation error
    var value = randomValue();
}
```

3.10 Good practices

Following the official Dart guidelines [2] we strongly encourage you to follow these suggestions in order to guarantee consistency with what the community recognizes as a good practice.

- Older versions of Dart allow the specification of a default value using a colon (`:`); don't do it, prefer using `=`. In both cases, the code compiles successfully.

  ```
  // Good
  void test([int a = 0]) {}

  // Bad
  void test([int a : 0]) {}
  ```

  The colon-initialization *might* be removed in the future.

- When no default values are given, the compiler already assigns `null` to the variable so you don't have to explicitly write it.

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

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

  In general, you should **never** initialize nullables with `null` because the compiler already does that by default.

[2] https://dart.dev/guides/language/effective-dart/usage#functions

Dart gives the possibility to write only the name of a function, with no parenthesis, and automatically pass proper parameters to it. It's a sort of "method reference". We are going to convince you with this example:

```
void showNumber(int value) {
    print("$value");
}

void main() {
    // List of values
    final numbers = [2, 4, 6, 8, 10];

    // Good
    numbers.forEach(showNumber);

    // Bad
    numbers.forEach((int val) { showNumber(val); });
}
```

The bad example compiles but you can avoid that syntax in favor of a shorter one.

- The `forEach()` method asks for a function with a single integer parameter and no return type (`void`).
- The `showNumber()` function accepts an integer as parameter and returns nothing (`void`).

The signatures match! If you pass the function name directly inside the method, the compiler automatically initializes the parameters. This **tear-off** is very useful and you might already have seen it somewhere else under the name of "method reference" (Java).

3.11 Using typedefs

The `typedef` keyword simply gives another name to a function type so that it can be easily reused. Imagine you had to write a callback function for many methods:

```
void printIntegers(void Function(String msg) logger) {
    logger("Done.");
}

void printDoubles(void Function(String msg) logger) {
```

```
    logger("Done.");
}
```

Alternatively, rather than repeating the declaration every time, which leads to code duplication, you can give it an alias using the `typedef` keyword.

```
typedef LoggerFunction = void Function(String msg);

void printIntegers(LoggerFunction logger) {
    logger("Done int.");
}

void printDoubles(LoggerFunction logger) {
    logger("Done double.");
}
```

You are going to encounter this technique very often, especially in Flutter, in callbacks for classes or methods. For instance, `VoidCallback` [3] is just a function alias for a void function taking no parameters.

```
typedef VoidCallback = void Function();
```

In a future version of Dart, probably later than 2.10, `typedef` will also be used to define new type names. At the moment it's not possible, but in the future there will be the possibility to compile the following code:

```
typedef listMap = List<Map<int,double>>;
```

The reason is that generic types can become very verbose and so an alias could improve the readability. Currently, `typedef` only works with functions.

[3] https://api.flutter.dev/flutter/dart-ui/VoidCallback.html