4 | Classes

Up to now you've seen us saying many times the claim that Dart is an OOP language and now we're finally going to prove it. There are a lot of similarities with the most popular programming language so you probably are already familiar with the concepts.

```dart
class Person {
    // Instance variables
    String name;
    String surname;

    // Constructor
    Person(String name, String surname) {
        this.name = name;
        this.surname = surname;
    }
}
```

This syntax is almost identical to Java, C# or C++ and that's very good: if you're going to learn the language, there's nothing you've never seen before. Some keywords might be different but the essence is always the same.

> Every object is an instance of a class. Dart classes, even if it's not explicitly written in the declaration, descend from `Object` and in the next chapter you will see the benefits. In Delphi and C# as well, any class implicitly derives from `Object`.

In any class you have methods (it's the OOP way to call *functions*) which can be public or private. The keyword `this` refers to the current instance of the class. Dart has **NO** method overload so you cannot have more than a function with the same name. For this reason, you'll see how named constructors come to the help.

```
class Example {
    // Doesn't compile; you have to use different names
    void test(int a) {}
    void test(double x, double y) {}
}
```

You might be able to write this in other programming languages because methods have the same name but different signature. In Dart it's not possible, every function name (in the same class) must be unique. Before going into the details of classes, look at the *cascade notation*.

```
class Test {
  String val1 = "One";
  String val2 = "Two";

  int randomNumber() {
    print("Random!");
    return Random().nextInt(10);
  }
}
```

Given this class, you have two ways to give a value to the variables:

```
// first way, the "classic" one
test.val1 = "one";
test.val2 = "two";

// second way, using the cascade operator
test..val1 = "one"
    ..val2 = "two";
```

It's just a shorthand version you can use when there are multiple values of the same objects that has to be initialized. You can do the same even with methods but the returned value, if any, will be **ignored**. For this reason, the cascade notation is useful when calling a series of `void` methods on the same object.

```
Test()..randomNumber()
      ..randomNumber()
      ..randomNumber();
```

Here the integer returned by `randomNumber()` is discarded but the body is executed. If you run the snipped, you'll get `Random!` printed three times in the console. In case of nullable values...

```
MyClass? test = MyClass();

test?..one()
    ..two()
    ..three();
```

... the cascade notation has to start with `?..` in order to be null-checked before dereferencing. In Dart there **cannot** be nested classes.

4.1 Libraries and visibility

In the Dart world, when you talk about a "library" you're referring to the code inside a file with the `.dart` extension. If you want to use that particular library, you have to reference its content with the `import` keyword.

If you aren't sure you've understood the above statement, here's an example. Let's say there's the need to handle fractions in the form *numerator / denominator*: you are going to create a file named `fraction.dart`.

```
// === Contents of fraction.dart ===
class MyFraction {
    final int numerator;
    final int denominator;

    //... other code
}
```

Congratulations, you have just created the `fraction` library! It can be used by any other `.dart` file that references it via `import`.

```
import 'package:fraction.dart';

void main() {
    // You could have written 'new Fraction(1, 2)' but
    // starting from Dart 2.0 'new' is optional
    final frac = Fraction(1, 2);
    }
```

You can also use the `library` keyword to "name" your library as you prefer. This keyword really just names the library as you like, nothing more, and it's not required.

```
    library super_duper_fraction;

    class MyFraction {
        final int numerator;
        final int denominator;
        const MyFraction(this.numerator, this.denominator);
    }
```

The `import` directive accepts a string which must contain a particular scheme. For built-in libraries you have to use the `dart:` prefix followed by the library name:

```
import 'dart:math';
import 'dart:io';
import 'dart:html';
```

Everything else that doesn't belong to the Dart SDK, such as a custom library created by you or another developer in the community, must be prefixed by `package`.

```
import 'package:fraction.dart';
import 'package:path/to/file/library.dart';
```

It could happen that two different libraries have implemented a class with the same name; the only possible technique to avoid ambiguity is called "library aliases". It's a way to reference the contents of a library under a different name.

```
// Contains a class called 'MyClass'
import 'package:libraryOne.dart';
// Also contains a class called 'MyClass'
import 'package:libraryTwo.dart' as second;

void main() {
    // Uses MyClass from libraryOne
    var one = MyClass();

    //Uses MyClass from libraryTwo.
```

```
        var two = second.MyClass();
    }
```

You can selectively import or exclude types using the `show` and `hide` keywords:

- `import 'package:libraryOne.dart' show MyClass;` Imports only `MyClass` and discards all the rest.
- `import 'package:libraryTwo.dart' hide MyClass;` Imports everything except `MyClass`.

At the moment, you have the basics of libraries (simple creation and usage). In chapter 23 you'll learn how to create a *package* (a collection of libraries and tools) and how to publish it at https://pub.dev.

4.1.1 Encapsulation

You may be used to hide implementation details in your classes using *public*, *protected* and *private* keywords but there's no equivalent in Dart. Every member is public by default unless you append an underscore (_) which makes it private to its library. If you had this file...

```
// === File: test.dart ===
class Test {
    String nickname = "";
    String _realName = "";
}
```

... you could later import the library anywhere:

```
// === File: main.dart ===
import 'package:test.dart';

void main() {
    final obj = Test();

    // OK
    var name = obj.nickname;
    // ERROR, doesn't compile
    var real = obj._realName;
}
```

The variable `nickname` is public and everyone can see it but `_realName` can be seen **ONLY** inside `test.dart`. In other words, if you put the underscore in front of the name of a variable, it is visible only within that file.

```dart
// === File: main.dart ===
class Test {
    String nickname = "";
    String _realName = "";
}

void main() {
    final obj = Test();

    // Ok
    var name = obj.name;
    // Ok, it works because 'Test' is in the same file
    var real = obj._realName;
}
```

We've moved everything in the same file: now both `Test` and `main()` belong to the **same** library and so `_realName` is not private anymore.

ℹ The same rules on package private members also apply to classes and functions. For example, `void something()` is visible from the outside while `void _something()` is private to its library.

```dart
// Inside a file called users.dart
class Users { }

class _UsersHelper { }
```

In this case, `Users` is visible while `_UsersHelper` is package-private (exactly as it happens with variables and methods).

In Dart everything is "public" by default; if you append an underscore at the front, it becomes "private". There is no way to define "protected" members or variables (where `protected` is a typical OOP keyword that makes a member or variable accessible only by subclasses of a certain type.).

4.1.2 Good practices

We strongly recommend to **NOT** put everything in a single file (or a few ones) for the following reasons:

- Dealing with thousands of lines of code containing literally everything is not good at all. Maintenance is going to be hard and you're on the good way to become a professional "*spaghetti code*" writer!
- If you placed everything in the same file, you'd expose private details of the class. We've seen that private members exist only if classes are put in separated libraries (files).

Try to have one file per class or at maximum a few classes that are closely related (they should have the same purpose). When you write a Dart program, in general you have this folder structure:

```
root
 | -- lib
 |      | -- main.dart
 |      | -- routes
 |      | -- other_folders
 | -- tests
 | -- tools
```

You'll learn throughout the chapters how to create a robust folder hierarchy. The minimal Dart/Flutter project is made up of a `lib/` folder and a `main.dart` entry point file.

4.2 Constructors

The constructor is a special function with the same name of the class and doesn't carry a return type. To invoke it, the syntax is the most common one you can imagine:

```dart
final myObject = new MyClass();
```

Starting from Dart 2, the keyword `new` can be omitted. It's something you're always going to do, especially while writing Flutter apps.

```dart
final myObject = MyClass();
```

As example, in this chapter we are creating a library to work with fractions and rational numbers. To get started, there's the need to create a file called `fraction.dart` with the following contents:

```
class Fraction {
    int? _numerator;
    int? _denominator;

    Fraction(int numerator, int denominator) {
        _numerator = numerator;
        _denominator = denominator;
    }
}
```

In this case variables must be nullables because they are initialized, by default, to `null`. The body of the constructor is called **after** the variables initialization. If you want to get rid of nullables there are two options:

1. Declare variables as `late final` to tell the compiler to not emit an error. They are going to be initialized later but anyway before being accessed for the first time ever.

   ```
   class Fraction {
       late final int _numerator;
       late final int _denominator;

       Fraction(int numerator, int denominator) {
           _numerator = numerator;
           _denominator = denominator;
       }
   }
   ```

   Because of the `final` modifier, variables cannot be changed anymore after their initialization. You could have only used `late` but variables would be mutable then:

   ```
   class Fraction {
       late int _numerator;
       late int _denominator;

       Fraction(int numerator, int denominator) {
           _numerator = numerator;
           _denominator = denominator;
       }
   }
   ```

   In both cases, members are **NOT** initialized immediately because the body of the constructor is executed after the variable initialization phase.

2. The Dart team recommends going for the "initializing formal" [1] approach as it's more readable and it initializes the variables immediately.

   ```
   class Fraction {
       int _numerator;
       int _denominator;

       Fraction(this._numerator, this._denominator);
   }
   ```

   It's just syntactic sugar to immediately assign values to members. In this case, variables initialization is executed first so no need to use nullable types or `late`. This kind of initialization happens **before** the execution of the constructor's body.

Keep in mind that constructor bodies are executed **after** the variable initialization phase. The second approach is very common and you should get used to it, even if your class only declares `final` fields.

```
class Fraction {
    final int _numerator;
    final int _denominator;

    Fraction(this._numerator, this._denominator);
}
```

If your class doesn't define a constructor, the compiler automatically adds a *default* constructor with no parameters and an empty body. With the "initializing formal" you can still declare a body to perform additional setup for the class.

```
class Fraction {
    final int _numerator;
    final int _denominator;
    late final double _rational;

    Fraction(this._numerator, this._denominator) {
      _rational = _numerator / _denominator;
      doSomethingElse();
```

[1] https://dart.dev/guides/language/effective-dart/usage#do-use-initializing-formals-when-possible

```
        }
    };
```

You could only write `Fraction(this._numerator)` to initialize exclusively `_numerator` but then `_denominator` would be set to `null` by the compiler. Keep in mind that you cannot have **named** optional parameters starting with an underscore.

```
    // Doesn't compile
    Fraction({this._numerator, this._denominator});
```

However, you can have **positional** parameters starting with an underscore:

```
    // Ok but pay attention to non-nullability
    Fraction([this._numerator, this._denominator]);
```

4.2.1 Initializer list

When using the initializing formal approach, the names of the variables must match the ones declared in the constructor. This could lead to an undesired exposure of some internals of the class:

```
    class Test {
        int _secret;
        double _superSecret;

        Test(this._secret, this._superSecret);
    }
```

What if you wanted to keep `int _secret` (private) but with a different name in the constructor? Use an initializer list! It's executed before the body and thus variables are immediately initialized. No need for nullable types or `late`.

```
    class Test {
        int _secret;
        double _superSecret;

        Test(int age, double wallet)
            : _secret = age,
              _superSecret = wallet;
    }
```

In this way the constructor is asking you for `age` and `wallet` but the user has no idea that internally they're treated as `_secret` and `_superSecret`. It's basically a way to "rename" internal private properties you don't want to expose.

4.2.2 Named constructors

Named constructors are generally used to implement a default behavior the user expects from your class. They are the only alternative to have multiple constructors since Dart has no method overload.

```
class Fraction {
    int _numerator;
    int _denominator;

    Fraction(this._numerator, this._denominator);

    // denominator cannot be 0 because 0/0 is not defined!
    Fraction.zero() :
      _numerator = 0,
      _denominator = 1;
}
```

At this point you can use the named constructor in your code like if it were a static method call.

```
void main() {
    // "Traditional" initialization
    final fraction1 = Fraction(0, 1);

    // Same thing but with a named constructor
    final fraction2 = Fraction.zero();
}
```

In general constructors aren't inherited by a subclass so, if they are needed across the hierarchy, every subclass must implement its own named constructor. If we had written a named constructor with a body...

```
class Fraction {
    int? _numerator;
    int? _denominator;
```

```
        Fraction.zero() {
            _numerator = 0;
            _denominator = 1
        }
    }
```

... we would have had to use nullable instance variables as constructors' bodies are always executed **after** variables' initialization.

4.2.3 Redirecting constructors

Sometimes you might have a constructor that does almost the same thing already implemented by another one. It may be the case to use redirecting constructors in order to avoid code duplication:

```
Fraction(this._numerator, this._denominator);
// Represents '1/2'
Fraction.oneHalf() : this(1, 2);
// Represents integers, like '3' which is '3/1'
Fraction.whole(int val) : this(val, 1);
```

Where `Fraction.oneHalf()` is just another way to call `Fraction(1, 2)` but you've avoided code repetition. This feature is very powerful when mixed with named constructors.

4.2.4 Factory constructors

The `factory` keyword returns an instance of the given class that's not necessarily a new one. It can be useful when:

- You want to return an instance of a subclass instead of the class itself,
- You want to implement a singleton (the Singleton pattern),
- You want to return an instance from a cache.

Factory constructors are like static methods and so they don't have access to `this`. There cannot be together a factory and a "normal" constructor with the same name.

```
class Test {
    static final _objects = List<BigObject>();

    factory Test(BigObject obj) {
            if (!objects.contains(obj))
                objects.add(obj);

            return Test._default();
        }

        // This is a private named constructor and thus it can't be called
        // from the outside
        Test._default() {
            //do something...
        }
    }
```

In the example, since `BigObject` requires a lot of memory and the list is very very long, we've declared *objects* as `static`. This technique is often used to save memory and reuse the same object across multiple objects of the same type.

> If the list weren't `static` (just a normal instance variable), it would be created **every** time that a Test object is instantiated. It'd be a waste of memory and a performance problem; in this way we're guaranteed that there's an unique list created only once.

In this case the factory constructor is essential because it takes care of updating the `_objects` cache. Factories are called "normally" like if they were a regular constructor:

```
// Calls the factory constructor
final a = Test();
```

4.2.5 Instance variables initialization

As we've already seen, "normal" non-nullable variables have to be initialized either via initializing formal or initializer list. In any other case, they're set to `null` because constructor bodies run after the instance initialization phase.

```
// Initializing formal
class Example {
    int _a;  // Ok - 'a' initialized by the constructor
    Example(this._a);
}

class Example {
    int _a;  // Error - 'a' not initialized
    Example(int a) {
        _a = a;
    }
}
```

If variables aren't initialized immediately and you want them to be non-nullanle, you can use the `late` modifier. It works like a "lazy initialization" because with this keyword you allow a non-nullable to be initialized later (but anyway before it gets accessed for the first time ever).

```
// For Dart 2.10 and earlier versions, 'late' does not exist so just
// remove it. Not initialized variables will be set to 'null' by default.
class Example {
    late int a;

    void printExample() {
        a = 5;
        print("$a");
        a = 2;
        print("$a");
    }
}
```

If you tried to use the variable before it gets assigned for the first time, you would get a compilation error. Always be sure to have the initialization done before using it.

```
class Example {
    late int a;

    void printExample() {
        // Compilation error
        print("$a");
        a = 5;
    }
}
```

Using `a` like above causes an error because it's accessed before being initialized. There's also the possibility to declare a `late final` variable which behaves in the same way but with the only exception that it can be assigned only once.

```
class Example {
    late final int a;

    void printExample() {
        a = 5;
        print("$a");
        //a = 6; <-- This would be an error
    }
}
```

Once `a` is set with `a = 5;` you can't re-assign it anymore because of the `final` modifier. Instead, if it were a simple `late int a;`, you could have re-assigned it multiple times.

```
late final double a = takesLongTime();
```

Thanks to the usage of the `late final` combination you can lazily initialize a variable that is going to hold a value computed from a function. You can assign it immediately or, as we've just seen, in a second moment. The function `takesLongTime()` will only be called once `a` is accessed.

4.2.6 Good practices

Following what the official documentation [2] suggests, here's some tips you should consider while writing constructors for your classes:

1. Prefer using the "initializing formal" approach rather than initializing variables directly in the body of the constructor. Doing so, you'll avoid the usage of nullable types or `late`.
2. When you use the initializing formal, the types of the variables are deduced automatically to reduce the verbosity. Omit the types because they're useless.

   ```
   // Ok but useless
   Constructor(String this.a, double this.b) {}
   // OK
   Constructor(this.a, this.b);
   ```

3. When you have an empty constructor with no body, use the semicolon instead of the empty brackets.

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

```
// Bad
Constructor(this.a, this.b) {}
// Good
Constructor(this.a, this.b);
```

4. Do not use the `new` keyword when creating new instances of objects. In modern Dart and Flutter code, `new` never appears.
5. We recommend using redirecting constructors to avoid code duplication. It makes maintenance easier.
6. Try to not use the `late` keyword because it could lead to hard maintenance (you'd have to **manually** keep an eye on the initialization of variables). Whenever possible, initialize variables as soon as they are declared.
7. In the second part of the book we'll show a case where `late` or `late final` are required. They'll be used to initialize values inside the state of a `Widget`.

   ```
   class Example extends State {
       late int value;

       @override
       void initState() {
           super.initState();
           value = 0;
       }
   }
   ```

   In short, subclasses of `State` cannot define a constructor and you have to perform the initialization of the variables in the `initState()` method. In order do to this, `late` is essential.

The syntax for private constructors in Dart might seem a bit weird at first. Generally, a private constructor is used in conjunction with a `factory` that returns a subtype or an instance of the actual object with certain criteria.

```
class Example {
    final a;
    // Private constructor
    Example._(this.a);

    factory Example(int value) {
```

```
            final c = value * 3;
            return Example._(c);
        }
    }
```

A private constructor is declared using the `._()` notation. In the example, the class can still be instantiated but only because we've defined a `factory`. In this case...

```
final ex = Example(10);
```

... we're not calling the "normal" constructor (because it's package private) but instead the `factory` one.

4.3 const keyword

The `const` keyword can be used when you have to deal with compile-time constant values such as strings or numbers. It can automatically deduce the type.

```
// type of 'number' is int
const number = 5
// explicitly write the type
const String name = 'Alberto';

const sum = 5.6 + 7.34;
```

It's true that `final` and `const` are very similar at first glance and they also share the same syntax style. A very intuitive way to determine which one you can choose is ask yourself: is this value already well defined? is it known at compile time? Let's find out why.

- `final`. Use it when the value is not known at compile time because it will be evaluated/obtained at runtime. Common usages are I/O from the disk or HTTP requests. For example, this is how you read a text file from the disk (more on this in A.1):

  ```
  final contents = File('myFile.txt').readAsString();

  // const contents = File('myFile.txt').readAsString();
  // ^ does not compile!
  ```

  The compiler doesn't know in advance which is the content of `myFile.txt` because it will be read only when the program will be running (so *after* the compilation). For this reason, you can only use final.

- `const`. Use it when the value is computed at compile time, for example with integers, doubles, Strings or classes with a *constant constructor* (more on it in the next section).

  ```
  const a = 1;
  // final a = 1 -> it works as well
  ```

  If it works with `const`, it works also with `final` because anything that is `const` is also `final`.

Instance variables can only be declared as `final` while `const` can be applied in combination with the `static` keyword.

```
class Example {
    // OK
    final double a = 0.0;
    // NO, instance variables can only be 'final'
    const double b = 0.0;

    // OK
    static const double PI = 3.14;
    // OK but without type annotation
    static const PI = 3.14;
}
```

The real power of `const` comes when combined with constructors and, in Flutter, it can lead to an important performance boost.

ℹ Variables and methods marked with the `static` modifier are available on the class itself and not on instances. In practice it means that you can use them without having to create an object.

```
class Example {
    static const name = "Flutter";
    static String test() => "Hello, I am $name!";
}

void main() {
    final name = Example.name;
    final text = Example.test();
}
```

Both variables are strings and they've been retrieved without creating an instance of `Example`; `static` members belong to the "class scope" and you cannot use `this`.

```dart
class Example {
    int a = 0;
    static void test() {
        // Doesn't compile
        final version = this.a;
        print("$version");
    };
}
```

Since you don't have to create objects to call static methods, `this` cannot work because it refers to the current instance that gets never created.

4.3.1 const constructors

In Dart you can append the `const` keyword in front of a constructor only if you're going to initialize a series of `final` (immutable) variables.

```dart
// Compiles
class Compiles {
    final int a;
    final int b;
    const Compiles(this.a, this.b);
}

// Does not compile because a is mutable (not final)
class DoesNot {
    int a;
    final int b;
    const DoesNot(this.a, this.b);
}
```

If your class only has `final` variables it's said to be an "immutable class" and you should really instantiate it with a `const` constructor. The compiler can perform some optimizations.

```dart
final example1 = const Compiles(); // (1) constant object
final example2 = Compiles();       // (2) not a constant object!
```

In example *(1)* we're calling the constant constructor but in *(2)* we're **not**. Even if your class only has constant constructors, objects can be instantiated as constants only with the `const` keyword. When put in front of a collection, such as a list, everything inside that container will automatically be `const` (if it is allowed to be constant).

```dart
class Test {
    // constant constructor
    const Test();
}

const List<Test> listConst = [Test(), Test()];  // (1)
final List<Test> listConst2 = [Test(), Test()];  // (2)
```

In *(1)* everything inside the list is automatically "converted" into a `const` value while in *(2)* it doesn't happen, meaning that the contents of `listConst2` aren't constant. The `final` keyword in front of a list just makes it impossible to change the reference assigned to `listConst2` but does **NOT** call the constant constructor (while example *(1)* does).

```dart
// Bad
const List<Test> list1 = [const Test(), const Test()]; // (1)
// Good
const List<Test> list2 = [Test(), Test()];  // (2)
```

You should avoid version *(1)* because calling `const Test()` is not necessary. Any constant collection initializes its children calling their `const` constructor (if any, otherwise a compilation error occurs).

4.3.2 Good practices and annotations

If you know that variables in your class will never change, you really should make them `final` and use a `const` constructor. As we've already said, constant constructors play a very important role in Flutter because they allow "caching" on instances.

ℹ Don't get "obsessed" by immutable classes and `const` constructors because you simply can't use them in every situation. If your instance variables cannot be `final` that's perfectly fine; the environment and the compiler are very powerful and your final product won't suffer of speed degradations due to the lack of immutability.

Any class with a constant constructor can be used as **annotation**: they're are generally put before the name of a class or method. An annotation is preceded by the "at" sign (@). In the next section, we will see that overriding methods is usually done in the following way:

```
class MySubclass extends SuperClass {
    @override
    void defineMethod() {}
}
```

The `@override` annotation does nothing in practice: it just tells the developer that `defineMethod` has been overridden. If you looked at how the override annotation is declared in the Dart SDK, you'll find simply the following:

```
// This class has a constant constructor and so it can be used as annotation
class _Override {
    const _Override();
}

// the actual "@override" annotation
const Object override = _Override();
```

The class has been made private (`_Override`) because its instantiation is useless as it does nothing. However, thanks to the `const Object override` variable being public, there's an "alias" of the `_Override` class which can be used as annotation. They're generally used for:

- reminding the developer about something, such as in the case of `@override`;
- before Dart 2.10, the `@required` annotation was used by the IDE to bring the developer's attention to the fact that a named optional parameter is required;
- some packages, such as *json_serializable* we're going to cover in chapter 15, rely on annotations to add additional information about a class, a method or a member. Annotations can be used to pass data to code generation tools.

Annotations can also have parameters but in this case you aren't doing the above "trick" of declaring a global variable exposing a private class. Just create a normal class, with a `const` constructor and use it as follows:

```
// Use this as annotation but it takes a param
class Something {
    final int value;
    const Something(this.value);
}
```

```
@Something(10)
class Test {}
```

4.4 Getters and setters

When a public variable is declared, anyone can freely manipulate it but it may not a good idea because the class partially loses control of its members. If we had written this...

```
class Fraction {
    int numerator;
    int denominator;
    Fraction(this.numerator, this.denominator);
}
```

... at a certain point someone could have changed the numerator and the denominator without any control introducing unexpected behaviors due to a wrong internal state.

```
void main() {
    final frac = Fraction(1, 7);

    frac.numerator = 0;
    frac.denominator = 0;
}
```

Having set both numerator and denominator to 0 there will be problems at runtime due to an invalid division operation. We can fix this problem using a *getter*, which makes the variables read-only.

```
class Fraction {
    int _numerator;
    int _denominator;
    Fraction(this._numerator, this._denominator);

    // Getters are read-only
    int get numerator => _numerator;
    int get denominator {
        return _denominator;
    }
}
```

The getter `numerator` returns an `int`, `_numerator`; the getter `denominator` does the same thing with an equivalent syntax. Like it happens with methods, when there is an one-liner expression or value to return, the => (arrow) syntax can be used.

```
void main() {
    final frac = Fraction(1, 7);

    // Compilation error, numerator is read-only
    frac.numerator = 0;
    // No problems here, we can read its value
    final num = frac.numerator;
}
```

The code is now safe because we can expose both numerator and denominator but it's guaranteed that they cannot be freely modified. Internally, `_numerator` and `_denominator` are "safe" because they aren't visible from the outside. Just as example, we're going to see how to write a **setter** for the denominator. So far it's read-only but with a setter it becomes editable:

```
class Fraction {
    int _numerator;
    int _denominator;
    Fraction(this._numerator, this._denominator);

    // getters
    int get numerator => _numerator;
    int get denominator => _denominator;

    // setter
    set denominator(int value) {
        if (value == 0) {
            // Or better, throw an exception...
            _denominator = 1;
        } else {
            _denominator = value;
        }
    }
}
```

There can be the same name for a setter and a getter so that a property can be read/written using the same identifier. Setters should be used to make "safe edits" on variables; they often contain a validation logic which makes sure that the internal state of the class doesn't get corrupted.

```
void main() {
    final frac = Fraction(1, 7);

    var den1 = frac.denominator; // den1 = 7
    frac.denominator = 0; // the setter changes it to 1
    den1 = frac.denominator      // den1 = 1
}
```

To sum it up, getters and setters are used to control the reading/writing on variables. They are methods under the hood but with a "special" syntax that uses the `get` and `set` keywords.

4.4.1 Good practices

Our recommendation is to keep the body of getters and setters as short as possible in benefit of code readability. You shouldn't put any loop that might slow down the assignment/retrieval of a value. The official documentation [3] also has something to say:

- When a variable has to be both public and read-only, just mark it as `final` without associating a getter to it.

  ```
  // Bad
  class Example {
      final _address = "https://fluttercompletereference.com";
      String get address => _address;
  }

  // Good
  class Example {
      final address = "https://fluttercompletereference.com";
  }
  ```

  If it's `final`, it's already a read-only variable because nothing can change its content. In this case a getter is simply useless.

- Avoid wrapping public variables with getters and setters if there's no validation logic.

  ```
  class Example {
      var _address = "https://fluttercompletereference.com";
  ```

[3]https://dart.dev/guides/language/language-tour#getters-and-setters

```
        String get address => _address;
        set address(String value) => _address = value;
    }
```

It compiles but there's no point in doing that: both getter and setter don't perform any particular logic as they just serve the variable as it is. Prefer doing this:

```
    class Example {
        var address = "https://fluttercompletereference.com";
    }
```

In general, use getters when you want to expose a variable but in "read-only mode" and setters when you want to filter/check the value that is going to be assigned.

4.5 Operators overload

When you deal with primitive types you use operators very often: `5 + 8` is a sum between two `int` types happening with the + operator. We are going to do the same with `Fraction` class.

```
class Fraction {
    Fraction operator+(Fraction other) =>
        Fraction(
          _numerator * other._denominator +
          _denominator * other._numerator,
          _denominator * other._denominator
        );

    Fraction operator-(Fraction other) => ...

    Fraction operator*(Fraction other) => ...

    Fraction operator/(Fraction other) => ...
}
```

Operator overloading gives the possibility to customize the usage of operators in your classes. We have overloaded the + operator so that we can easily sum two fractions instead of having to create an `add(Fraction value)` method, like it happens with Java.

```
void main() {
        // 2/5
        final frac1 = Fraction(2, 5);
        // 1/3
        final frac2 = Fraction(1, 3);

        // 2/5 + 1/3 = 11/15
        final sum = frac1 + frac2
    }
```

They work like normal methods with the only exception that the name must be in the form *operator{sign}* where *sign* is a supported Dart operator:

- **Arithmetic** operators like +, -, *, or /.
- **Relational** operators such as >=, <=, > or <.
- **Equality** operators like != and ==

And many more. There are no restrictions on the types you can handle with the operators, meaning that we could also sum fractions with integers: `operator+(int other)`. You cannot overload the same operator more than once in the same class.

4.5.1 callable classes

There is a special `call()` method which is very closely related to an operator overload because it allows classes to be called like if they were functions with the `()` operator.

ℹ You can give `call()` as many parameters as you want as there are no restrictions on their types. The function can return something or it can simply be `void`.

Let's give a look at this example.

```
class Example {
    double call(double a, double b) => a + b;
}

void main() {
    final ex = Example();         // 1.
    final value = ex(1.3, -2.2); // 2.
```

```
        print("$value");
    }
```

1. Classic creation of an instance of the class
2. The object `ex` can act like if it were a function. The `call()` method allows an object to be treated like a function.

Any class that implements `call()` is said to be a **callable class**. In Dart, everything is an object and you've seen in 3.6.1 that even functions are objects. You can now understand why with the following example:

```
void test(String something) {
    print(something);
}
```

This is a typical `void` function asking for a single parameter. Actually, the above code can be converted into a callable class that overrides `call()` returning nothing and asking for a string.

```
// Create this inside 'my_test.dart' for example
class _Test {
    const _Test();

    void call(String something) {
        print(something);
    }
}

const test = _Test();

// Somewhere else, for example in main.dart
import 'package:myapp/my_test.dart';

void main() {
    test("Hello");
}
```

The function is nothing more than a package private class that overrides `call()` with a certain signature. Thanks to `const test = _Test();` in the last line we're "hiding" the class and exposing a callable object to be used as function.

4.6 Cloning objects

Even if it's not mentioned in the official Dart documentation, there is a standard "*pattern*" to follow when it comes to cloning objects. Unlike Java, there is no `clone()` method to override but still you might need to create deep copies of objects:

```
class Person {
    final String name;
    final int age;
    const Person({
        required this.name,
        required this.age,
    });
}

void main() {
    const me = const Person(
        name: "Alberto",
        age: 25
    );
    const anotherMe = me;
}
```

As you already know, the variable `anotherMe` just holds a reference to `me` and thus they point to the same object. Changes applied to `me` will also reflect on `anotherMe`. If you want to make deep copies in Dart (cloning objects and making them independent), this is the way:

```
class Person {
    final String name;
    final int age;
    const Person({
        required this.name,
        required this.age,
    });

    Person copyWith({
        String? name,
        int? age,
    }) => Person(
            name: name ?? this.name,
            age: age ?? this.age
        );

        @override
        String toString() => "$name, $age";
    }
```

This method is called `copyWith()` by convention and it takes the same number (and name) of parameters required by the constructor. It creates a **new**, independent copy of the object (a clone) with the possibility to change some parameters:

```
    const me = const Person(
        name: "Alberto",
        age: 25
    );

    // Create a deep copy of 'me'.
    final anotherMe = me.copyWith();

    // Create a deep copy of 'me' with a different age.
    final futureMe = const.copyWith(age: 35);

    print("$me");        // Alberto, 25
    print("$anotherMe"); // Alberto, 25
    print("$futureMe");  // Alberto, 35
```

Both `anotherMe` and `futureMe` have **no** side effects on `me` because the reference is not the same. In fact, `copyWith()` returns a fresh new instance by copying internal data. Let's take a look at this line:

```
    name: name ?? this.name,
```

Thanks to the ?? operator, if `name` is `null` then initialize the clone with value of `this.name` taken from the instance. In other words, if you don't pass a custom `name` to `copyWith()`, by default a copy of `this.name` is made. Pay attention to generic containers and objects in general:

```
    class Skills {...}

    class Person {
        final List<Skills> skills;
        const Person({
            required this.skills
        });

        Person copyWith({
            List<Skills>? skills,
        }) => Person(
            skills: skills ?? this.skills
        );
    }
```

This code doesn't do what you'd expect because `List<T>`, like any other generic container, is an object and not a primitive type. With the above code you're just copying **references** and not making copies. The correct solution is the following:

```
    Person(
        skills: skills ?? this.skills.map((p) => p.copyWith()).toList();
    );
```

In this way you're making a copy of the entire list rather than passing a reference. The above code is just an one-liner way to iterate on each element of the source, making deep copies using `copyWith` and returning a new list. However, when the list is made up of primitive types, you could use a shortcut:

```
    class Person {
        final List<int> values;
        const Person({
            required this.values
        });

        Person copyWith({
            List<int>? values,
        }) => Person(
            values: values ?? []..addAll(this.values)
        );
    }
```

Primitive types are automatically copied so instead of using `map()` (which would be perfectly fine as well) we can use `addAll()` for a shorter syntax. There is no difference however because it still iterates on every element of the source list. The same example also applies to `Map<K, V>` and

`Set<K>`. To sum it up, what you have to keep in mind is:

- Deep copies in Dart are made using the `copyWith()` method. You can give it any other name but you'd better follow the conventions.
- When making copies, be sure that classes (like generic containers) are deep copied using the convenient `map((x) => x.copyWith())` strategy.
- If you have a list of primitive types (like `double`s or `int`) you can use the `[]..addAll()` shortcut. Do this **only** with primitive types.