2 | Variables and data types

2.1 Variables

As in any programming language, variables are one of the basics and Dart comes with support for type inference. A typical example of creation and initialization of a variable is the following:

```
var value = 18;
var myName = "Alberto"
```

In the example, `value` is an integer while `myName` is a string. Like Java and C#, Dart is able to **infer** the type of the variable by looking at the value you've assigned. In other words, the Dart compiler is smart enough to figure out by itself which is the correct type of the variable.

```
int value = 18;
String myName = "Alberto"
```

This code is identical to the preceding example with the only difference that here the types have been typed explicitly. There would also be a third valid way to initialize variables, but you should almost never use it.

```
dynamic value = 18;
dynamic myName = "Alberto"
```

`dynamic` can be used with any type, it's like a "jolly": any value can be assigned to it and the compiler won't complain. The type of a `dynamic` variable is evaluated at runtime and thus, for a proper usage, you'd need to work with checks and type casts. According with the Dart guidelines and our personal experience you should:

1. Prefer initializing variables with `var` as much as you can;
2. When the type is not so easy to guess, initialize it explicitly to increase the readability of the code;

3. Use `Object` or `dynamic` only if it's really needed but it's almost never the case.

Actually, we could say that `dynamic` is not really a type: it's more of a way to turn off static analysis and tell the compiler you know what you're doing. The only case in which you'll deal with it will come in the Flutter part in regard to JSON encoding and decoding.

2.1.1 Initialization

The official Dart guidelines [1] state that you should prefer, in most of the cases, the initialization with `var` rather than writing the type explicitly. Other than making the code shorter (programmers are lazy!) it can increase the readability in various scenarios, such as:

```
// BAD: hard to read due to nested generic types
List<List<Toppings>> pizza = List<List<Toppings>>();
for(List<Toppings> topping in pizza) {
    doSomething(topping);
}

// GOOD: the reader doesn't have to "parse" the code
// It's clearer what's going on
var pizza = List<List<Toppings>>();
for(var topping in pizza) {
    doSomething(topping);
}
```

Those code snippets use generics, classes and other Dart features we will discuss in depth in the next chapters. It's worth pointing out two examples in which you want to explicitly write the type instead of inferring it:

- When you don't want to initialize a variable immediately, use the `late` keyword. It will be explained in detail later in this chapter.

```
// Case 1
late List<String> names;

if (iWantFriends())
    names = friends.getNames();
else
    names = haters.getNames();
```

[1] https://dart.dev/guides/language/effective-dart/design#types

If you used `var` instead of `List<String>` the inferred type would have been `null` and that's **not** what we want. You'd also lose the type safety and readability.

- The type of the variable is not so obvious at first glance:

```
// Is this a list? I guess so, "People" is plural...
// but actually the function returns a String!
var people = getPeople(true, 100);

// Ok, this is better
String people = getPeople(true, 100);
```

However, there isn't a golden rule to follow because it's up to your discretion. In general `var` is fine, but if you feel that the type can make the code more readable you can definitely write it.

2.1.2 final

A variable declared as `final` can be set only once and if you try to change its content later, you'll get an error. For example, you won't be able to successfully compile this code:

```
final name = "Alberto";
name = "Albert"; // 'name' is final and cannot be changed
```

You can also notice that `final` can automatically infer the type exactly like `var` does. This keyword can be seen as a "restrictive var" as it deduces the type automatically but does not allow changes.

```
// Very popular - Automatic type deduction
final name = "Alberto";
// Generally unnecessary - With type annotation
final String nickName = "Robert";
```

If you want you can also specify the type but it's not required. So far we've only shown examples with strings, but of course both `final` and `var` can be used with complex data types (classes, enums) or methods.

```
final rand = getRandomInteger();

// rand = 0;
// ^ doesn't work because the variable is final
```

The type of `rand` is deduced by the return statement of the method and it cannot be re-assigned in a second moment. The same advice we've given in "2.1.1 Initialization" for `var` can be applied here as well.

> Later on in the book we will analyze in detail the `const` keyword, which is the "brother" of `final`, and it has very important performance impacts on Flutter.

While coding you can keep this rule in mind: use `final` when you know that, once assigned, the value will **never** change in the future. If you know that the value might change during the time use `var` and think whether it's the case to annotate the type or not. Here's an example in which a `final` variable fits perfectly:

```
void main() {
    // Assume that the content of the file can't be edited
    final jsonFile = File('myfile.json').readAsString();

    checkSyntax(jsonFile);
    saveToDisk(jsonFile, 'file.json');
}
```

In this example the variable `jsonFile` has a content that doesn't have to be modified, it will always remain the same and so a `final` declaration is good:

- it won't be accidentally edited later;
- the compiler will give an error if you try to modify the value.

If you used `var` the code would have compiled anyway but it wouldn't have been the best choice. If the code was longer and way more complicated, you could accidentally change the content of `jsonFile` because there wouldn't be the "protection" of `final`.

2.2 Data types

Types in Dart can be initialized with "literals"; for example `true` is a boolean literal and `"test"` is a string literal. In chapter 6 we will analyze *generic* data types that are very commonly used for collections such as lists, sets and maps.

2.2.1 Numbers

Dart has two type of numbers:

- `int`. 64-bit at maximum, depending on the platform, integer values. This type ranges from -2^{63} to 2^{63}-1.
- `double`. 64-bit double-precision floating point numbers that follow the classic IEEE 754 standard definition.

Both `double` and `int` are subclasses of `num` which provides many useful methods such as:

- `parse(string)`,
- `abs()`,
- `ceil()`,
- `toString()`...

You should always use `double` or `int`. We will see, with generic types, a special case in which `num` is needed but in general you can avoid it. Some examples are always a good thing:

```
var a = 1; // int
var b = 1.0; // double

int x = 8;
double y = b + 6;
num z = 10 - y + x;

// 7 is a compile-time constant
const valueA = 7;
// Operations among constant values are constant
const valueB = 2 * valueA;
```

From Dart 2.1 onwards the assignment `double a = 5` is legal. In 2.0 and earlier versions you were forced to write 5.0, which is a *double* literal, because 5 is instead an *integer* literal and the compiler didn't automatically convert the values. Some special notations you might find useful are:

1. The exponential representation of a number, such as `var a = 1.35e2` which is the equivalent of 1.35 * 10^2;

2. The hexadecimal representation of a number, such as `var a = 0xF1A` where 0xF1A equals to F1A in base 16 (3866 in base 10).

2.2.1.1 Good practices

Very likely, during your coding journey, you'll have at some point the need to parse numbers from strings or similar kinds of manipulations. The language comes to the rescue with some really useful methods:

```
String value = "17";

var a = int.parse(value); // String-to-int conversion
var b = double.parse("0.98"); // String-to-double conversion
var c = int.parse("13", radix: 6); // Converts from 13 base 6
```

You should rely on these methods instead of writing functions on your own. In the opposite direction, which is the conversion into a string, there is `toString()` with all its variants:

```
String v1 = 100.toString(); // v1 = "100";
String v2 = 100.123.toString(); // v2 = "100.123";
String v3 = 100.123.toStringAsFixed(2); // v3 = "100.12";
```

Since we haven't covered functions yet you can come back to this point later or, if you're brave enough, you can continue the reading. When converting numbers from a string, the method `parse()` can fail if the input is malformed such as `"12_@4.49"`. You'd better use one of the following solutions (we will cover nullable types later):

```
// 1. If the string is not a number, val is null
double? val = double.tryParse("12@.3x_"); // null
double? val = double.tryParse("120.343"); // 120.343

// 2. The onError callback is called when parsing fails
var a = int.parse("1_6", onError: (value) => 0); // 0
var a = int.parse("16", onError: (value) => 0); // 16
```

Keep in mind that `parse()` is deprecated: you should prefer `tryParse()`. What's important to keep in mind is that a plain `parse("value")` call is risky because it assumes the string is already well-formed. Handling the potential errors as shown is safer.

2.2.2 Strings

In Dart a string is an ordered sequence of UTF-16 values surrounded by either single or double quotes. A very nice feature of the language is the possibility of combining expressions into strings by using `${expr}` (a shorthand to call the `toString()` method).

```
// No differences between s and t
var s = "Double quoted";
var t = 'Single quoted';

// Interpolate an integer into a string
var age = 25;
var myAge = "I am $age years old";

// Expressions need '{' and '}' preceeded by $
var test = "${25.abs()}"

// This is redundant, don't do it because ${} already calls toString()
var redundant = "${25.toString()}";
```

A string can be either single or multiline. Single line strings are shown above using single or double quotes, and multiline strings are written using triple quotes. They might be useful when you want to nicely format the code to make it more readable.

```
// Very useful for SQL queries, for example
var query = """
    SELECT name, surname, age
    FROM people
    WHERE age >= 18
    ORDER BY name DESC
""";
```

In Dart there isn't a `char` type representing a single character because there are only strings. If you want to access a particular character of a string you have to use the `[]` operator:

```
final name = "Alberto";

print(name[0]); // prints "A"
print(name[2]); // prints "b";
```

The returned value of `name[0]` is a `String` whose length is 1. We encourage you to visit [2] the online Dart documentation about strings which is super useful and full of examples.

```
var s = 'I am ' + name + ' and I am ' + (23).toString() + ' y.o.';
```

You can concatenate strings very easily with the + operator, in the classic way that most programming languages support. The official Dart guidelines [3] suggest to prefer using interpolation to compose strings, which is shorter and cleaner:

```
var s = 'I am $name. I am ${25} years old';
```

In case of a string longer than a single line, avoid the + operator and prefer a simple line break. It's just something recommended by the Dart for styling reasons, there are no performance implications at all. Try to be as consistent as possible with the language guidelines!

```
// Ok
var s = 'I am going to the'
        'second line';

// Still ok but '+' can be omitted
var s = 'I am going to the' +
        'second line';
```

Since strings are immutable, making too many concatenations with the + operator might be inefficient. In such cases it'd be better if you used a `StringBuffer` which efficiently concatenates strings. For example:

```
var value = "";

for(var i = 0; i < 900000; ++i) {
    value += "$i ";
}
```

Each time the + operator is called, `value` is assigned with a **new** instance which merges the old value and the new one. In other words, this code creates for 900000 times a new `String` object, one for each iteration, and it's not optimal at all. Here's the way to go:

```
var buffer = StringBuffer();
```

[2] https://dart.dev/guides/libraries/library-tour#strings-and-regular-expressions
[3] https://dart.dev/guides/language/effective-dart/usage#prefer-using-interpolation-to-compose-strings-and-values

```
for(var i = 0; i < 900000; ++i)
    buffer.write("$i ");

var value = buffer.toString();
```

This is much better because `StringBuffer` doesn't internally create a new string on each iteration; the string is created only **once** at the moment in which `toString()` is called. When you have to do long loops that manipulate strings, avoid using the + operator and prefer a buffer. The same class can also be found in Java and C# for example.

2.2.3 Enumerated types

Also known as "enums", enumerated types are containers for constant values that can be declared with the `enum` keyword. A very straightforward example is the following:

```
enum Fruits { Apple, Pear, Grapes, Banana, Orange }

void main() {
    Fruits liked = Fruits.Apple;
    var disliked = Fruits.Banana;

    print(liked.toString()); // prints 'Fruits.Apple'
    print(disliked.toString()); // prints 'Fruits.Banana'
}
```

Each item of the enum has an associated number, called **index**, which corresponds to the zero-based position of the value in the declaration. You can access this number by using the `index` property.

```
enum Fruits { Apple, Pear, Grapes, Banana, Orange }

void main() {
    var a = Fruits.Apple.index;  // 0
    var b = Fruits.Pear.index;  // 1
    var c = Fruits.Grapes.index;  // 2
}
```

Note that when you need to use an `enum` you always have to fully qualify it. Using the name only doesn't work.

2.2.3.1 Good Practices

When you need a predefined list of values which represents some kind of textual or numeric data, you should prefer an `enum` over a primitive data type. In this way you can increase the readability of the code, the consistency and the compile-time checking. Look at these 2 ways of creating a function (*///* is used to document the code):

```
enum Chess { King, Queen, Rook, Bishop, Knight, Pawn }

/// METHOD 1. Checks if the piece can move in diagonal
bool diagonalMoveC(Chess item) { ... }

/// METHOD 2. Checks if a piece can move in diagonal: [item] can only be:
///  1. King
///  2. Queen
///  3. Rook
///  4. Bishop
///  5. Knight
///  6. Pawn
/// Any other number is not allowed.
bool diagonalMoveS(int item) { ... }
```

This example should convince you that going for the first method is for sure the right choice.

- `diagonalMoveC(Chess item)`. There's a big advantage here: we're guaranteed by the compiler that `item` can only be one of the values in `Chess`. There's no need for any particular check and we can understand immediately what the method wants us to pass.
- `diagonalMoveS(int item)`. There's a big disadvantage here: we can pass any number, not only the ones from 1 to 6. We're going to do extra work in the body because we don't have the help of the compiler, so we need to manually check if `item` contains a valid value.

In the second case, we'd have to make a series of `if` conditions to check whether the value ranges from 1 to 6. Using an `enum`, the compiler does the checks for us (by comparing the types) and we're guaranteed to work with valid values.

2.2.4 Booleans

You can assign to the `bool` type only the literals `true` or `false`, which are both compile-time constants. Here there are a few usage examples:

```
bool test = 5 == 0; // false
bool test2 = !test; // has the opposite value of test

var oops = 0.0 / 0.0; // evaluates to 'Not a Number' (NaN)
bool didIFail = oops.isNaN;
```

2.2.5 Arrays

Probably you're used to create arrays like this: `int[] array = new int[5];` which is the way that Java and C# offer. In Dart it doesn't really work like that because you can only deal with collections: an "array" in Dart is represented by a `List<T>`.

> `List<T>` is a generic container where `T` can be any type (such as `String` or a class). We will cover generics and collections in detail in chapter 6. Basically, Dart doesn't have "arrays" but only generic containers.

If this is not clear, you can look at this comparison. In both languages there is a generic container for the given type but only Java has "primitive" arrays.

- **Java**

```
// 1. Array
double[] test = new test[10];
// 2. Generic list
List<double> test = new ArrayList<>();
```

- **Dart**

```
// 1. Array
// (no equivalent)
// 2. Generic list
List<double> test = new List<double>();
```

In Dart you can work with arrays but they are intended to be instances of `List<T>`. Lists are 0-indexed collections and items can be randomly accessed using the `[]` operator, which will throw an exception if you exceed the bounds.

```
//use var or final
final myList = [-3.1, 5, 3.0, 4.4];
final value = myList[1];
```

A consequence of the usage of a `List<T>` as container is that the instance exposes many useful methods, typical of collections:

- `length`,
- `add(T value)`,
- `isEmpty`,
- `contains(T value)`

... and much more.

2.3 Nullable and Non-nullable types

Starting from Dart 2.10, variables will be **n**on-**n**ullable **b**y **d**efault (nnbd) which means they're not allowed to hold the `null` value. This feature has been officially introduced in June 2020 as *tech preview* in the dev channel of the Dart SDK.

```
// Trying to access a variable before it's been assigned will cause a
// compilation error.
int value;
print("$value"); // Illegal, doesn't compile
```

If you don't initialize a variable, it's automatically set to `null` but that's an error because Dart has non-nullability enabled by default. In order to successfully compile you have to initialize the variable as soon as it's declared:

```
// 1.
int value = 0;
print("$value");

// 2.
int value;
value = 0;
print("$value");
```

In the first case the variable is assigned immediately and that's what we recommend to do as much as possible. The second case is still valid because `value` is assigned **before** it's ever accessed. It wouldn't have worked if you had written this:

```
// OK - assignment made before the usage
```

```
int value;
value = 0;
print("$value");

// ERROR - usage made before assignment
int value;
print("$value");
value = 0;
```

Non-nullability is very powerful because it adds another level of type safety to the language and, by consequence, lower possibilities for the developer to encounter runtime exceptions related to `null`. For example, you won't have the need to do this:

```
String name = "Alberto";

void main() {
    if (name != null) {
        print(name)
    }
}
```

The compiler guarantees that it can't be `null` and thus no null-checks are required. To sum up, what's important to keep in mind while writing Dart 2.10 code (and above) is:

- By default, variables cannot be `null` and they must **always** be initialized before being used. It would be better if you immediately initialized them, but you could also do it in a second moment before they ever get utilized.
- Don't do null-checks on "standard" non-nullable variables because it's useless.

In Dart you can also declare *nullable* types which doesn't require to be initialized before being accessed and thus they're allowed to be `null`. Nullables are the counterpart of non-nullable types because the usage of `null` is allowed (but the additional type safety degree is lost).

```
int? value;
print("$value"); // Legal, it prints 'null'
```

If you append a question mark at the end of the type, you get a nullable type. For safety, they would require a manual null checks in order to avoid undesired exceptions but, in most of the cases, sticking with the default non-nullability is fine.

```
// Non-nullable version - default behavior
```

```
int value = 0;
print("$value"); // prints '0'

// Nullable version - requires the ? at the end of the type
int? value;
print("$value"); // prints 'null'
```

Nullable types that support the index operator `[]` need to be called with the `?[]` syntax. `null` is returned if the variable is also `null`.

```
String? name = "Alberto";
String? first = name?[0]; // first = 'A';

String? name;
String? first = name?[0]; // first = 'null';
```

We recommend to stick with the defaults, which is the usage of non-nullable types, as they're safer to use. Nullables should be avoided or used only when working with legacy code that depends on `null`. Last but not least, here are the only possible conversions between nullables and non nullables:

- When you're sure that a nullable expression isn't null, you can add a `!` at the end to convert it to the non-nullable version.

  ```
  int? nullable = 0;
  int notNullable = nullable!;
  ```

  The `!` (called "bang operator") converts a nullable value (`int?`) into a non-nullable value (`int`) of the same type. An exception is thrown if the nullable value is actually `null`.

  ```
  int? nullable;
  // An exception is thrown
  int notNullable = nullable!;
  ```

- If you need to convert a nullable variable into a non-nullable subtype, use the typecast operator `as` (more on it later):

  ```
  num? value = 5;
  int otherValue = value as int;
  ```

  You wouldn't be able to do `int otherValue = value!` because the bang operator works only when the type is the same. In this example, we have a `num` and an `int` so there's the need for a cast.

- Even if it isn't a real conversion, the operator `??` can be used to produce a non-nullable value from a nullable one.

```
int? nullable = 10;
int nonNullable = nullable ?? 0;
```

  If the member on the left (`nullable`) is non-null, return its value; otherwise, evaluate and return the member of the right (`0`).

Remember that when you're working with nullable values, the member access operator (`.`) is not available. Instead, you have to use the **null-aware** member access operator (`?.`):

```
double? pi = 3.14;

final round1 = pi.round();   // No
final round2 = pi?.round();  // Ok
```

2.4 Data type operators

In Dart expressions are built using operators, such as `+` and `-` on primitive data types. The language also supports operator overloading for classes as we will cover in chapter 4.

2.4.1 Arithmetic operators

Arithmetic operators are commonly used on `int` and `double` to build expressions. As you already know, the `+` operator can also be used to concatenate strings.

Symbol	Meaning	Example
+	Add two values	`2 + 3 //5`
-	Subtract two values	`2 - 3 //-1`
*	Multiply two values	`6 * 3 //18`
/	Divide two values	`9 / 2 //4.5`
~ /	Integer division of two values	`9 ~/ 2 //4`

%	Remainder (modulo) of an int division	`5 % 2 //1`

Prefix and postfix increment or decrement work as you're used to see in many languages.

```
int a = 10;
++a; // a = 11
a++; // a = 12

int b = 5;
--b; // b = 4;
b--; // b = 3;

int c = 6;
c += 6 // c = 12
```

As a reminder, both postfix and prefix increment/decrement have the same result but they work in a **different** way. In particular:

- in the prefix version (`++x`) the value is first incremented and then "returned";
- in the postfix version (`x++`) the valie is first "returned" and then incremented

2.4.2 Relational operators

Equality and relational operators are used in boolean expression, generally inside `if` statements or as a stop condition of a `while` loop.

Symbol	Meaning	Example
==	Equality test	`2 == 6`
!=	Inquality test	`2 != 6`
>	Greather than	`2 > 6`

<	Smaller than	`2 < 6`
>=	Greater or equal to	`2 >= 6`
<=	Smaller or equal to	`2 <= 6`

Testing the equality of two objects *a* and *b* always happens with the == operator because, unlike Java or C#, there is no `equals()` method. In chapter 6 we will analyze in detail how classes can be properly compared by overriding the equality operator. In general here's how the == works:

1. If *a* or *b* is null, return `true` if both are null or `false` if only one is null. Otherwise...
2. ... return the result of == according with the logic you've defined in the method override.

Of course, == works only with objects of the same type.

2.4.3 Type test operators

They are used to check the type of an object at runtime.

Symbol	Meaning	Example
`as`	Cast a type to another	`obj as String`
`is`	True if the object has a certain type	`obj is double`
`is!`	False if the object has a certain type	`obj is! int`

Let's say you've defined a new type like `class Fruit {}`. You can cast an object to `Fruit` using the `as` operator like this:

```
(grapes as Fruit).color = "Green";
```

The code compiles but it's unsafe: if `grapes` was `null` or if it wasn't a `Fruit`, you would get an exception. It's always a good practice checking whether the cast is doable before doing it:

```
if (grapes is Fruit) {
    (grapes as Fruit).color = "Green";
}
```

Now you're guaranteed the cast will happen only if it's possible and no runtime exceptions can happen. Actually, the compiler is smart enough to understand that you're doing a type check with `is` and it can do a *smart cast*.

```
if (grapes is Fruit) {
    grapes.color = "Green";
}
```

You can avoid writing the explicit cast `(grapes as Fruit)` because, inside the scope of the condition, the variable `grapes` is automatically casted to the `Fruit` type.

2.4.4 Logical operators

When you have to create complex conditional expressions you can use the logical operators:

Symbol	Meaning
`!expr`	Toggles true to false and vice versa
`expr1 && expr2`	Logical AND (true if both sides are true)
`expr1 || expr2`	Logical OR (true if at least one is true)

2.4.5 Bitwise and shift operators

You'll never use these operators unless you're doing some low level data manipulation but in Flutter this never happens.

Symbol	Meaning
`a & b`	Bitwise AND
`a | b`	Bitwise OR
`a ^ b`	Bitwise XOR
`~ a`	Bitwise complement
`a >> b`	Right shift
`a << b`	Left shift