Chapter 3: Types & Operations

Data types in Dart

In Dart, a **type** is a way to tell the compiler how you plan to use some data. By this point in this book, you've already seen the following types:

- `int`
- `double`
- `num`
- `dynamic`
- `String`

The last one in that list, `String`, is the type used for text like `"Hello, Dart!"`.

As you learned in Chapter 2, **the root of all types is the `Object` type. This type defines a few core operations, such as testing for equality and describing the object in text. Every other type in Dart is a subtype of `Object`, and as a subtype, shares `Object`'s basic functionality.**

Type inference

In the previous chapter, you also got a sneak peak at type inference in Dart, but you'll take some time to look at it in a little more depth now.

Annotating variables explicitly

It's fine to always explicitly add the **type annotation** when you declare a variable. This means writing the data type before the variable name.

```
int myInteger = 10;
double myDouble = 3.14;
```

You annotated the first variable with `int` and the second with `double`.

Creating constant variables

Declaring variables the way you did above makes them mutable. If you want to make them immutable, but still keep the type annotation, you can add `const` or `final` in front.

These forms of declaration are fine with `const`:

```
const int myInteger = 10;
const double myDouble = 3.14;
```

They're also fine with `final`:

```
final int myInteger = 10;
final double myDouble = 3.14;
```

Note: Mutable data is convenient to work with because you can change it any time you like. **However, many experienced software engineers have come to appreciate the benefits of immutable data. When a value is immutable, that means you can trust that no one will change that value after you create it. Limiting your data in this way prevents many hard-to-find bugs from creeping in, and also makes the program easier to reason about and to test.**

Letting the compiler infer the type

While it's permissible to include the type annotation as in the example above, it's redundant. You're smart enough to know that 10 is an `int` and 3.14 is a `double`, and it turns out the Dart compiler can deduce this as well. The compiler doesn't need you to explicitly tell it the type every time — it can figure the type out on its own through a process called **type inference Not all programming languages have type inference, but Dart does — and it's a key component behind Dart's power as a language**.

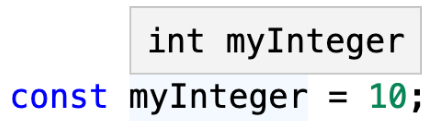
You can simply drop the type in most instances. For example, here are the constants from above without the type annotations:

```
const myInteger = 10;
const myDouble = 3.14;
```

In the absence of type annotation, Dart figures out the correct type for each constant above.

Checking the inferred type in VS Code

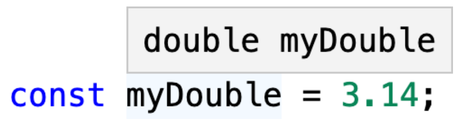
Sometimes, it can be useful to check the inferred type of a variable or constant. You can do this in VS Code by hovering your mouse pointer over the variable name. VS Code will display a popover like this:

- 

```
int myInteger
const myInteger = 10;
```

VS Code shows you the inferred type. In this example, the type is `int`.

It works for other types, too. Hovering your mouse pointer over `myDouble` shows that it's a `double`:



```
double myDouble
const myDouble = 3.14;
```

Note: There are times when you'll want (or need) to explicitly include the type, either because Dart doesn't have enough information to figure it out, or because you want your intent to be clear to the reader. However, you'll see type inference used for most of the code examples in this book.

Checking the type at runtime

Your code can't hover a mouse pointer over a variable to check the type, but Dart does have a programmatic way of doing the same thing: the `is` keyword.

```
num myNumber = 3.14;
print(myNumber is double);
print(myNumber is int);
```

Run this to see the following result:

```
true
false
```

Recall that both `double` and `int` are subtypes of num. That means `myNumber` could store either type. In this case, 3.14 is a `double`, and not an `int`, which is what the `is` keyword checks for and confirms by returning true and `false` respectively. You'll learn more about the type for true and `false` values in Chapter 4.

Another option to see the type at runtime is to use the `runtimeType` property that is available to all types.

```
print(myNumber.runtimeType);
```

This prints `double` as expected.

Type conversion

Sometimes, you'll have data in one type, but need to convert it to another. The naïve way to attempt this would be like so:

```
var integer = 100;
var decimal = 12.5;
integer = decimal;
```

Dart will complain if you try to do that:

```
A value of type 'double' can't be assigned to a
variable of type 'int'.
```

implicit conversion is a frequent source of software bugs and often hurts code performance**. Dart disallows** you from assigning a value of one type to another and avoids these issues.

Remember, computers rely on programmers to tell them what to do. In Dart, that includes being **explicit about type conversions**. If you want the conversion to happen, you have to say so!

You can convert this `double` to an `int` like so:

```
integer = decimal.toInt();
```

The assignment now tells Dart, unequivocally, that you want to convert from the original type, `double`, to the new type, `int`.

> **Note**: In this case, assigning the decimal value to the integer results in a loss of precision: The integer variable ends up with the value 12 instead of 12.5. This is why it's important to be explicit. Dart wants to make sure you know what you're doing and that you may end up losing data by performing the type conversion.

Operators with mixed types

So far, you've only seen operators acting independently on integers or doubles. But what if you have an integer that you want to multiply with a double?

Take this example:

```
const hourlyRate = 19.5;
const hoursWorked = 10;
const totalCost = hourlyRate * hoursWorked;
```

hourlyRate is a double and hoursWorked is an int. What will the type of totalCost be? It turns out that Dart will make totalCost a double. This is the safest choice, since making it an int could cause a loss of precision.

If you actually do want an int as the result, then you need to perform the conversion explicitly:

```
const totalCost = (hourlyRate * hoursWorked).toI
nt();
```

The parentheses tell Dart to do the multiplication first, and after that, to take the result and convert it to an integer value. However, the compiler complains about this:

```
Const variables must be initialized with a const
ant value.
```

The problem is that toInt is a runtime method. This means that totalCost can't be determined at compile time, so making it const isn't valid. No problem; there's an easy fix. Just change const to final:

```
final totalCost = (hourlyRate * hoursWorked).toIn
t();
```

Now totalCost is an int.

Ensuring a certain type

Sometimes you want to define a constant or variable and ensure it remains a certain type, even though what you're assigning to it is of a different type. You saw earlier how you can convert from one type to another. For example, consider the following:

```
const wantADouble = 3;
```

Here, Dart infers the type of `wantADouble` as `int`. But what if you wanted the constant to store a `double` instead?

One thing you could do is the following:

```
final actuallyDouble = 3.toDouble();
```

This uses type conversion to convert 3 into a `double` before assignment, as you saw earlier in this chapter.

Another option would be to not use type inference at all, and to add the `double` annotation:

```
const double actuallyDouble = 3;
```

The number 3 is an integer, but literal number values that contain a decimal point cannot be integers, which means you could have avoided this entire discussion had you started with:

```
const wantADouble = 3.0;
```

Sorry! :]

Casting down

At other times, you may have a variable of some general supertype, but you need functionality that is only available in a subtype. If you're sure that the value of the variable actually is the subtype you need, then you can use the as keyword to change the type. This is known as **type casting**.

Here's an example:

```
num someNumber = 3;
```

You have a number, and you want to check if it's even. You know that integers have an `isEven` property, so you attempt the following:

```
print(someNumber.isEven);
```

However, the compiler gives you an error:

```
The getter 'isEven' isn't defined for the type 'n
um'.
```

`num` is too general of a type to know anything about even or odd numbers. Only integers can be even or odd; so the issue is that `num` could potentially be a `double` at runtime, since `num` includes both `double` and `int`. In this case, though, you're sure that 3 is an integer, so you can cast `someNumber` to `int`.

```
final someInt = someNumber as int;
print(someInt.isEven);
```

The as keyword causes the compiler to recognize someInt as an int, so your code is now able to use the isEven property that belongs to the int type. Since 3 isn't even, Dart prints false.

You need to be careful with type casting, though. If you cast to the wrong type, you'll get a runtime error:

```
num someNumber = 3;
final someDouble = someNumber as double;
```

This will crash with the following message:

```
_CastError (type 'int' is not a subtype of type
'double' in type cast)
```

The runtime type of someNumber is int, not double. In Dart, you're not allowed to cast to a sibling type, such as int to double. You can only cast down to a subtype.

If you do need to convert an int to a double at runtime, use the toDouble method that you saw earlier:

```
final someDouble = someNumber.toDouble();
```

Mini-exercises

1. Create a constant called age1 and set it equal to 42. Create another constant called age2 and set it equal to 21. Check that the type for both constants has been inferred correctly as int by hovering your mouse pointer over the variable names in VS Code.

2. Create a constant called `averageAge` and set it equal to the average of `age1` and `age2` using the operation `(age1 + age2) / 2`. Hover your mouse pointer over `averageAge` to check the type. Then check the result of `averageAge`. Why is it a `double` if the components are all `int`?

Strings

Working with strings in Dart

Dart, like any good programming language, can work directly with strings. It does so through the `String` data type. In this section, you'll learn about this data type and how to work with it.

Strings and characters

You've already seen a Dart string back in Chapter 1 where you printed one:

```
print('Hello, Dart!');
```

You can extract that same string as a named variable:

```
var greeting = 'Hello, Dart!';
print(greeting);
```

The right-hand side of this expression is known as a **string literal**. Due to type inference, Dart knows that `greeting` is of type `String`. Since you used the `var` keyword, you're allowed to reassign the value of `greeting` as long as the new value is still a string.

```
var greeting = 'Hello, Dart!';
greeting = 'Hello, Flutter!';
```

Even though you changed the value of `greeting` here, you didn't modify the string itself. That's because strings are immutable in Dart. It's not like you replaced `Dart` in the first string with `Flutter`. No, you completely discarded the string `'Hello, Dart!'` and replaced it with a whole new string whose value was `'Hello, Flutter!'`.

If you're familiar with other programming languages, you may be wondering about a `Character` or `char` type. Dart doesn't have that. Take a look at this example:

```
const letter = 'a';
```

So here, even though `letter` is only one character long, it's still of type `String`.

Single-quotes vs. double-quotes

Dart allows you to use either single-quotes or double-quotes for string literals. Both of these are fine:

```
'I like cats'
"I like cats"
```

Although Dart doesn't have a recommended practice, the Flutter style guide does recommend using single-quotes, so this book will also follow that practice.

You might want to use double-quotes, though, if your string includes any apostrophes.

```
"my cat's food"
```

Otherwise you would need to use the backslash \ as an escape character so that Dart knows that the string isn't ending early:

```
'my cat\'s food'
```

Concatenation

You can do much more than create simple strings. Sometimes you need to manipulate a string, and one common way to do so is to combine it with another string. This is called **concatenation**...with no relation to the aforementioned felines.

In Dart, you can concatenate strings simply by using the addition operator. Just as you can add numbers, you can add strings:

```
var message = 'Hello' + ' my name is ';
const name = 'Ray';
message += name;
// 'Hello my name is Ray'
```

You need to declare message as a variable, rather than a constant, because you want to modify it. You can add string literals together, as in the first line, and you can add string variables or constants together, as in the third line.

If you find yourself doing a lot of concatenation, you should use a string buffer, which is more efficient.

```
final message = StringBuffer();
message.write('Hello');
message.write(' my name is ');
message.write('Ray');
message.toString();
// "Hello my name is Ray"
```

This creates a mutable location in memory where you can add to the string without having to create a new string for every change. When you're all done, you use `toString` to convert the string buffer to the `String` type. This is similar to what you saw with type conversion earlier with `toInt`.

Interpolation

You can also build up a string by using **interpolation**, which is a special Dart syntax that lets you build a string in a manner that's easy for other people reading your code to understand:

```
const name = 'Ray';
const introduction = 'Hello my name is $name';
// 'Hello my name is Ray'
```

This is much more readable than the example in the previous section. It's an extension of the string literal syntax, in which you replace certain parts of the string with other values. You add a dollar sign ($) in front of the value that you want to insert.

The syntax works in the same way to build a string from other data types such as numbers:

```
const oneThird = 1 / 3;
const sentence = 'One third is $oneThird.';
```

Here, you use a `double` for the interpolation. Your `sentence` constant will contain the following value:

```
One third is 0.3333333333333333.
```

Of course, it would actually take an infinite number of characters to represent one-third as a decimal, because it's a repeating decimal. You can control the number of decimal places shown on a `double` by using `toStringAsFixed` along with the number of decimal places to show:

```
final sentence = 'One third is ${oneThird.toStrin
gAsFixed(3)}.';
```

There are a few items of interest here:

- You're requesting the string to show only three decimal places.
- Since you're performing an operation on `oneThird`, you need to surround the expression with curly braces after the dollar sign. This lets Dart know that the dot (`.`) after `oneThird` isn't just a regular period.
- The sentence variable needs to be `final` now instead of const because `toStringAsFixed(3)` is calculated at runtime.

Here's the result:

```
One third is 0.333.
```

Multi-line strings

Dart has a neat way to express strings that contain multiple lines, which can be rather useful when you need to use very long strings in your code.

You can support multi-line text like so:

```
const bigString = '''
You can have a string
that contains multiple
lines
by
doing this.''';
print(bigString);
```

The three single-quotes (''') signify that this is a multi-line string. Three double-quotes (""") would have done the same thing.

The example above will print the following:

```
You can have a string
that contains multiple
lines
by
doing this.
```

Notice that all of the newline locations are preserved. If you just want to use multiple lines in code but don't want the output string to contain newline characters, then you can surround each line with single-quotes:

```
const oneLine = 'This is only '
    'a single '
    'line '
    'at runtime.';
```

That's because Dart ignores whitespace outside of quoted text. This does the same thing as if you concatenated each of those lines with the + operator:

```
const oneLine = 'This is only ' +
    'a single '  +
    'line ' +
    'at runtime.';
```

Either way, this is what you get:

```
This is only a single line at runtime.
```

Like many languages, if you want to insert a newline character, you can use \n.

```
const twoLines = 'This is\ntwo lines.';
```

This gives:

```
This is
two lines.
```

However, sometimes you want to ignore any special characters that a string might contain. To do that, you can create a **raw string** by putting **r** in front of the string literal.

```
const rawString = r'My name \n is $name.';
```

And that's exactly what you get:

```
My name \n is $name.
```

Mini-exercises

1. Create a string constant called `firstName` and initialize it to your first name. Also create a string constant called `lastName` and initialize it to your last name.

2. Create a string constant called `fullName` by adding the `firstName` and `lastName` constants together, separated by a space.

3. Using interpolation, create a string constant called `myDetails` that uses the `fullName` constant to create a string introducing yourself. For example, Ray Wenderlich's string would read: `Hello, my name is Ray Wenderlich.`

Object and dynamic types

Dart grew out of the desire to solve some problems inherent in JavaScript. JavaScript is a **dynamically-typed** language. Dynamic means that something can change, and for JavaScript that means the types can change at runtime.

Here is an example in JavaScript:

```
var myVariable = 42;
myVariable = "hello";
```

In JavaScript, the first line is a `number` and the second line a `string`. Changing the types on the fly like this is completely valid in JavaScript. While this may be convenient at times, it makes it *really* easy to write buggy code. For example, you may be erroneously thinking that `myVariable` is still a number, so you write the following code:

```
var answer = myVariable * 3; // runtime error
```

Oops! That's an error because `myVariable` is actually a string, and the computer doesn't know what to do with "hello" times 3. Not only is it an error, you won't even discover the error until you run the code.

You can totally prevent mistakes like that in Dart because it's an **optionally-typed** language. That means you can choose to use Dart as a dynamically typed language or as **statically-typed** language. Static means that something *cannot* change; once you tell Dart what type a variable is, you're not allowed to change it anymore.

If you try to do the following in Dart:

```
var myVariable = 42;
myVariable = 'hello'; // compile-time error
```

The Dart compiler will immediately tell you that it's an error. That makes type errors trivial to detect.

As you saw in Chapter 2, the creators of Dart did include a `dynamic` type for those who wish write their programs in a dynamically-typed way.

```
dynamic myVariable = 42;
myVariable = 'hello'; // OK
```

In fact, this is the default if you use `var` and don't initialize your variable:

```
var myVariable; // defaults to dynamic
myVariable = 42;      // OK
myVariable = 'hello'; // OK
```

While `dynamic` is built into the system, it's more of a concession rather than an encouragement to use it. You should still embrace static typing in your code as it will prevent you from making silly mistakes.

If you need to explicitly say that any type is allowed, you should consider using the `Object?` type.

```
Object? myVariable = 42;
myVariable = 'hello'; // OK
```

At runtime, `Object?` and `dynamic` behave nearly the same. However, when you explicitly declare a variable as `Object?`, you're telling everyone that you generalized your variable on purpose, and that they'll need to check its type at runtime if they want to do anything specific with it. Using `dynamic`, on the other hand, is more like saying you don't know what the type is; you're telling people they can do what they like with this variable, but it's completely on them if their code crashes.

Note: You may be wondering what that question mark at the end of `Object?` is. That means that the type can include the `null` value. You'll learn more about nullability in Chapter 7.

Challenges

Challenge 1: Teacher's grading

You're a teacher, and in your class, attendance is worth 20% of the grade, the homework is worth 30% and the exam is worth 50%. Your student got 90 points for her attendance, 80 points for her homework and 94 points on her exam. Calculate her grade as an integer percentage rounded down.

Challenge 2: Find the error

What is wrong with the following code?

```
const name = 'Ray';
name += ' Wenderlich';
```

Challenge 3: What type?

What's the type of value?

```
const value = 10 / 2;
```

Challenge 6: In summary

What is the value of the constant named summary?

```
const number = 10;
const multiplier = 5;
final summary = '$number * $multiplier = ${n umber
* multiplier}';
```

Key points

- Type conversion allows you to convert values of one type into another.
- When doing operations with basic arithmetic operators (+, -, *, /) and mixed types, the result will be a double.
- Type inference allows you to omit the type when Dart can figure it out.
- Unicode is the standard representation for mapping characters to numbers.
- Dart uses UTF-16 values known as code units to encode Unicode strings.
- A single mapping in Unicode is called a code point, which is known as a rune in Dart.

- User-perceived characters may be composed of one or more code points and are called grapheme characters.

- You can combine strings by using the addition operator.

- You can make multi-line strings using three single-quotes or double quotes.

- You can use string interpolation to build a string in-place.

- Dart is an optionally-typed language. While it's preferable to choose statically-typed variables, you may write Dart code in a dynamically-typed way by explicitly adding the `dynamic` type annotation in front of variables.