

# Chapter 2: Expressions, Variables & Constants

## Commenting Code

Dart, like most other programming languages, allows you to document your code through the use of comments. These allow you to write any text directly along side your code and are ignored by the compiler.

The first way to write a comment in Dart is like so:

```
// This is a comment. It is not executed.
```

This is a single line comment.

You can stack up single line comments to allow you to write multi-line comments as shown below:

```
// This is also a comment,  
// over multiple lines.
```

You may also create comment blocks by putting your comment text between `/*` and `*/`:

```
/* This is also a comment. Over many...  
many...  
many lines. */
```

The start is denoted by `/*` and the end is denoted by `*/`. Simple!

Dart also allows you to nest comments, like so:

```
/* This is a comment.  
/* And inside it is  
another comment. */  
Back to the first. */
```

In addition to these two ways of writing comments, Dart also has a third type called **documentation comments**. Single-line documentation comments begin with `///`, while block documentation comments are enclosed between `/**` and `*/`. Here's an example of each:

```
/// I am a documentation comment
/// at your service.

/**
 * Me, too!
 */
```

The Flutter and Dart documentation is well-known for its detailed comments. And since the code for Flutter and Dart is open source, simply browsing through it is an excellent way to learn how great documentation comments are written.

```
void main() {
  print('Hello, Dart!');
}
```

In VS Code, **Command+Click** on a Mac, or **Control+Click** on a PC, the `print` keyword. VS Code will take you to the source code for that keyword and you'll see the documentation comments for `print`:

```
/// Prints a string representation of the object to the console.
void print(Object? object) {
```

Speaking of `print`, that's another useful tool when writing Dart code.

## Printing output

`print` will output whatever you want to the debug console.

For example, consider the following code:

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

Run this code and it'll output a nice message to the debug console, like so:



The screenshot shows the Dart IDE's debug console with four tabs: PROBLEMS, OUTPUT, DEBUG CONSOLE, and TERMINAL. The DEBUG CONSOLE tab is selected and shows two lines of output: "Hello, Dart Apprentice reader!" in blue text and "Exited" in orange text.

Adding `print` statements into your code is an easy way to monitor what's happening at a particular point in your code. Later, when you're ready to take your debugging to the next level, you can check out some of the more detailed logging packages on [pub.dev](https://pub.dev).

You can print any expression in Dart. To learn what an expression is, though, keep reading.

## Statements and expressions

Two important words that you'll often hear thrown about in programming language documentation are **statement** and **expression**. It's helpful to understand the difference between the two.

### Statements

A **statement** is a command, something you tell the computer to do. In Dart, all simple statements end with a semicolon. You've already seen that with the `print` statement:

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

The semicolon on the right finishes the statement.

One example of a complex statement is the if statement:

```
if (someCondition) {  
  // code block  
}
```

No semicolons are needed on the lines with the opening or closing curly braces. You'll learn more about if statements and other control flow statements in Chapter 4.

## Expressions

Unlike a statement, an **expression** doesn't *do* something; it *is* something. That is, an expression is a value, or is something that can be calculated as a value.

Here are a few examples of expressions in Dart:

```
42  
3 + 2  
'Hello, Dart Apprentice reader!'  
x
```

The values can be numbers, text, or some other type. They can even be variables such as `x` whose value isn't known until runtime.

Coming up next, you'll see many more examples of expressions.

## Arithmetic operations

## Simple operations

- Add: +
- Subtract: -
- Multiply: \*
- Divide: /

These operators are used like so:

```
2 + 6
10 - 2
2 * 4
24 / 3
```

Check the answers yourself in VS Code using a print statement:

```
print(2 + 6);
```

Dart ignores whitespace, so you can remove the spaces surrounding the operator:

```
2+6
```

However, it's often easier to read expressions when you have white space on either side. In fact, the **dart format** tool in the Dart SDK will format your code according to the standard whitespace formatting rules. In VS Code, you can apply **dart format** with the keyboard shortcut **Shift+Option+F** on a Mac or **Shift+Alt+F** on a PC .

**Note:** This book won't always explicitly tell you to `print(X)`, where X is some Dart expression that you're learning about, but you should proactively do this yourself to check the value.

## Decimal numbers

All of the operations above use whole numbers, more formally known as **integers**. However, as you know, not every number is whole.

For example, consider the following:

```
22 / 7
```

If you're used to another language that uses integer division by default, you might expect the result to be 3. However, Dart gives you the standard decimal answer:

```
3.142857142857143
```

If you actually did want to perform integer division, then you could use the `~/` operator:

```
22 ~/ 7
```

This produces a result of 3.

The `~/` operator is officially called the **truncating division operator** when applied to numbers.

## The Euclidean modulo operation

The first of these is the **Euclidean modulo operation**.

In Dart, the Euclidean modulo operator is the `%` symbol. You use it like so:

```
28 % 10
```

In this case, the result equals 8, because 10 goes into 28 twice with a remainder of 8.

## Order of operations

```
((8000 / (5 * 10)) - 32) ~/ (29 % 5)
```



Note the use of parentheses, which in Dart serve two purposes: to make it clear to anyone reading the code — including yourself — what you meant, and to disambiguate the intended order of operations. For example, consider the following:

```
350 / 5 + 2
```

Does this equal 72?

Dart uses the same reasoning and achieves this through what's known as **operator precedence**. The division operator (/) has a higher precedence than the addition operator (+), so in this example, the code executes the division operation first.

If you wanted Dart to perform the addition first — that is, so the expression will return 50 instead of 72 — then you could use parentheses, like so:

```
350 / (5 + 2)
```

The precedence rules are the same as you learned in school. Multiplication and division have equal precedence. Addition and subtraction are equal in precedence to each other, but are lower in precedence than multiplication and division.

The ~/ and % operators have the same precedence as multiplication and division. If you're ever uncertain about what precedence an operator has, you can always use parentheses to be sure the expression evaluates as you want it to.

## Math functions

Dart also has a large range of math functions.

To use these math functions, add the following import to the top of your file:

```
import 'dart:math';
```

`dart:math` is one of Dart's core libraries. Adding the `import` statement tells the compiler that you want to use something from that library.

Now you can write the following:

```
sin(45 * pi / 180)  
// 0.7071067811865475  
  
cos(135 * pi / 180)  
// -0.7071067811865475
```

These convert an angle from degrees to radians, and then compute the sine and cosine respectively. Notice how both make use of `pi`, which is a constant Dart provides you. Nice!

**Note:** Remember that if you want to see the values of these mathematical expressions, you need to put them inside a `print` statement like this:

```
print(sin(45 * pi / 180));
```

Then there's this:

```
sqrt(2)  
// 1.4142135623730951
```

This computes the square root of 2.

Not mentioning these would be a shame:

```
max(5, 10)
// 10

min(-5, -10)
// -10
```

These compute the maximum and minimum of two numbers respectively.

If you're particularly adventurous you can even combine these functions like so:

```
max(sqrt(2), pi / 2)
// 1.5707963267948966
```

## Mini-exercise

Now print the value of 1 over the square root of 2 in Dart. Confirm that it equals the sine of 45°.

## Naming data

- Identifiers can include both, characters and digits. However, the identifier cannot begin with a digit.
- Identifiers cannot include special symbols except for underscore (\_) or a dollar sign (\$).
- Identifiers cannot be keywords.
- They must be unique.
- Identifiers are case-sensitive.
- Identifiers cannot contain spaces.

19

The following tables lists a few examples of valid and invalid identifiers – **Valid identifiers**

firstName	Var
first_name	first name
num1	first-name
\$result	1number

### Invalid identifiers

## Keywords in Dart

Keywords have a special meaning in the context of a language. The following table lists some keywords in Dart.

abstract 1	continue	false	new	this
as 1	default	final	null	throw
assert	deferred 1	finally	operator 1	true
async 2	do	for	part 1	try
async* 2	dynamic 1	get 1	rethrow	typedef 1
await 2	else	if	return	var
break	enum	implements 1	set 1	void
case	export 1	import 1	static 1	while
catch	external 1	in	super	with
class	extends	is	switch	yield 2
const	factory 1	library 1	sync* 2	yield* 2

### • Variables

Take a look at the following:

```
int number = 10;
```

This statement declares a variable called `number` of type `int`. It then sets the value of the variable to the number 10. The `int` part of the statement is known as a **type annotation** which tells Dart explicitly what the type is.

If you want to change the value of a variable, then you can just give it a different value of the same type:

```
int number = 10;  
number = 15;
```

The type `int` can store integers. The way you store decimal numbers is like so:

```
double apple = 3.14159;
```

This is similar to the `int` example. This time, though, the variable is a `double`, a type that can store decimals with high precision.

For readers who are familiar with object-oriented programming, you'll be interested to learn that 10, 3.14159 and every other value that you can assign to a variable are **objects in Dart**. In fact, Dart doesn't have the primitive variable types that exist in some languages. Everything is an object. **Although `int` and `double` look like primitives, they're subclasses of `num`, which is a subclass of `Object`.**

With numbers as objects, this lets you do some interesting things:

```
10.isEven  
// true  
  
3.14159.round()  
// 3
```

## Type safety

Dart is a **type-safe language**. That means once you tell Dart what a variable's type is, **you can't change it later**. Here's an example:

```
int myInteger = 10;  
myInteger = 3.14159; // No, no, no. That's not  
allowed.
```

3.14159 is a **double**, but you already defined `myInteger` as an **int**. No changes allowed!

Of course, sometimes it's useful to be able to assign related types to the same variable. That's still possible. For example, you could solve the problem above, where you want `myNumber` to store both an integer and floating-point value, like so:

```
num myNumber;  
myNumber = 10;           // OK  
myNumber = 3.14159;      // OK  
myNumber = 't';          // No, no, no.
```

The `num` type can be either an **int** or a **double**, so the first two assignments work. However, the string value `'t'` is of a different type, so the compiler complains.

use the **dynamic** type.

*This lets you assign any data type you like to your variable, and the compiler won't warn you about anything.*

```
dynamic myVariable;  
myVariable = 10;          // OK  
myVariable = 3.14159;     // OK  
myVariable = 'ten';       // OK
```

But, honestly, don't do that. Friends don't let friends use **dynamic**.

## Type inference



Dart is smart in a lot of ways. You don't even have to tell it the type of a variable, and Dart can still figure out what you mean. The `var` keyword says to Dart, "Use whatever type is appropriate."

```
var someNumber = 10;
```

There's no need to tell Dart that 10 is an integer. Dart **infers the type** and makes `someNumber` an `int`. Type safety still applies, though:

```
var someNumber = 10;  
someNumber = 15;           // OK  
someNumber = 3.14159; // No, no, no.
```

Trying to set `someNumber` to a `double` will result in an error. Your program won't even compile. Quick catch; time saved. Thanks, Dart!

## Constants

Dart has two different types of "variables" whose values never change. They are declared with the `const` and `final` keywords, and the following sections will show the difference between the two.

### `const` constants

Variables whose value you can change are known as **mutable data**. Mutables certainly have their place in programs, but can also present problems. It's easy to lose track of all the places in your code that can change the value of a particular

variable. For that reason, you should use **constants** rather than variables whenever possible. These unchangeable variables are known as **immutable data**

To create a constant in Dart, use the `const` keyword:

```
const myConstant = 10;
```

Just as with `var`, Dart uses type inference to determine that `myConstant` is an `int`.

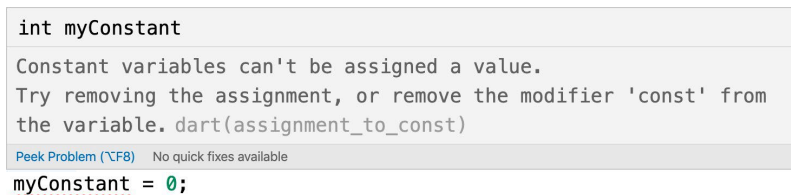
Once you've declared a constant, you can't change its data. For example, consider the following example:

```
const myConstant = 10;  
myConstant = 0; // Not allowed.
```

This code produces an error:

```
Constant variables can't be assigned a value.
```

In VS Code, you would see the error represented this way:



The screenshot shows a code editor with the following content:

```
int myConstant  
Constant variables can't be assigned a value.  
Try removing the assignment, or remove the modifier 'const' from  
the variable. dart(assignment_to_const)  
Peek Problem (⌘F8) No quick fixes available  
myConstant = 0;
```

The error message is displayed in a light gray box above the code. The code itself is in a dark theme. The variable `myConstant` is highlighted with a red squiggly line underneath it, indicating an error.

If you think “constant variable” sounds a little oxymoronic, just remember that it’s in good company: virtual reality, advanced BASIC, readable Perl and internet security.

## final constants

Often, you know you'll want a constant in your program, but you don't know what its value is at compile time. You'll only know the value after the program starts running. This kind of constant is known as a **.runtime constant**

In Dart `const` is only used for **compile-time constants** ; that is, for values that can be determined by the compiler before the program ever starts running.

If you can't create a `const` variable because you don't know its value at compile time, then you must use the `final` keyword to make it a runtime constant. There are many reasons you might not know a value until after your program is running. For example, you might need to fetch a value from the server, or query the device settings, or ask a user to input their age.

Here is another example of a runtime value:

```
final hoursSinceMidnight = DateTime.now().hour;
```

`DateTime.now()` is a Dart function that tells you the current date and time when the code is run. Adding `hour` to that tells you the number of hours that have passed since the beginning of the day. Since that code will produce a different results depending on the time of day, this is most definitely a runtime value. So to make `hoursSinceMidnight` a constant, you must use the `final` keyword instead of `const`.

If you try to change the `final` constant after it's already been set:

```
hoursSinceMidnight = 0;
```

This will produce the following error:

```
The final variable 'hoursSinceMidnight' can only be set once.
```

## Using meaningful names

Always try to choose meaningful names for your variables and constants. Good names act as documentation and make your code easy to read.

A good name specifically describes the role of a variable or constant. Here are some examples of good names:

- `personAge`
- `numberOfPeople`

- `gradePointAverage`

Often a bad name is simply not descriptive enough. Here are some examples of bad names:

- `a`
- `temp`
- `average`

Also, note how the names above are written. In Dart, it's standard to use **lowerCamelCase** for variable and constant names. Follow these rules to properly case your names:

- Start with a lowercase letter.
- If the name is made up of multiple words, join them together and start every word after the first one with an uppercase letter.
- Treat abbreviations like words (for example, `sourceUrl` and `urlDescription`).

## Mini-exercises

If you haven't been following along with these exercises in VS Code, now's the time to create a new project and try some exercises to test yourself!

1. Declare a constant of type `int` called `myAge` and set it to your age.
2. Declare a variable of type `double` called `averageAge`. Initially, set the variable to your own age. Then, set it to the average of your age and your best friend's age.
3. Create a constant called `testNumber` and initialize it with whatever integer you'd like. Next, create another constant called `evenOdd` and set it equal to `testNumber` modulo 2. Now change `testNumber` to various numbers. What do you notice about `evenOdd`?

## Increment and decrement

A common operation that you'll need is to be able to **increment** or **decrement** a variable. In Dart, this is achieved like so:

```
var counter = 0;

counter += 1;
// counter = 1;

counter -= 1;
// counter = 0;
```

The counter variable begins as 0. The increment sets its value to 1, and then the decrement sets its value back to 0.

The += and -= operators are similar to the assignment operator (=), except they also perform an addition or subtraction. They take the current value of the variable, add or subtract the given value, and assign the result back to the variable.

In other words, the code above is shorthand for the following:

```
var counter = 0;  
counter = counter + 1;  
counter = counter - 1;
```

If you only need to increment or decrement by 1, then you can use the ++ or -- operators:

```
var counter = 0;  
counter++; // 1  
counter--; // 0
```

The \*= and /= operators perform similar operations for multiplication and division, respectively:

```
double myValue = 10;

myValue *= 3; // same as myValue = myValue * 3;
// myValue = 30.0;

myValue /= 2; // same as myValue = myValue / 2;
// myValue = 15.0;
```

Division in Dart produces a result with a type of `double`, so `myValue` could not be an `int`.

## Challenges

Before moving on, here are some challenges to test your knowledge of variables and constants. It's best if you try to solve them yourself, but solutions are available with the supplementary materials for this book if you get stuck.

### Challenge 1: Variables

Declare a constant `int` called `myAge` and set it equal to your age. Also declare an `int` variable called `dogs` and set that equal to the number of dogs you own. Then imagine you bought a new puppy and increment the `dogs` variable by one.

### Challenge 2: Make it compile

Given the following code:



```
age = 16;  
print(age);  
age = 30;  
print(age);
```

Modify the first line so that the code compiles. Did you use `var`, `int`, `final` or `const`?

### Challenge 3: Compute the answer

Consider the following code:

```
const x = 46;  
const y = 10;
```

Work out what each answer equals when you add the following lines of code to the code above:

```
const answer1 = (x * 100) + y;  
const answer2 = (x * 100) + (y * 100);  
const answer3 = (x * 100) + (y / 10);
```

### Challenge 4: Average rating

Declare three constants called `rating1`, `rating2` and `rating3` of type `double` and assign each a value. Calculate the average of the three and store the result in a constant named `averageRating`.

### Challenge 5: Quadratic equations

A quadratic equation is something of the form

$$a \cdot x^2 + b \cdot x + c = 0.$$

The values of  $x$  which satisfy this can be solved by using the equation

$$x = \frac{-b \pm \sqrt{b^2 - 4 \cdot a \cdot c}}{2 \cdot a}.$$

Declare three constants named `a`, `b` and `c` of type `double`. Then calculate the two values for  $x$  using the equation above (noting that the  $\pm$  means plus or minus, so one value of  $x$  for each). Store the results in constants called `root1` and `root2` of type `double`.

## Key points

- Code comments are denoted by a line starting with `//`, or by multiple lines bookended with `/*` and `*/`.
- Documentation comments are denoted by a line starting with `///` or multiple lines bookended with `/**` and `*/`.
- You can use `print` to write to the debug console.
- The arithmetic operators are:

```
Addition: +  
Subtraction: -  
Multiplication: *  
Division: /  
Truncating division: ~/  
Modulo (remainder): %
```

- Dart has many functions including `min`, `max`, `sqrt`, `sin` and `cos`. You'll learn many more throughout this book.
- Constants and variables give names to data.
- Once you've declared a constant, you can't change its data, but you can change a variable's data at any time.
- If a variable's type can be inferred, you can replace the type with the `var` keyword.
- The `const` keyword is used for compile-time constants while `final` is used for runtime constants.
- Always give variables and constants meaningful names to save yourself and your colleagues headaches later.
- Operators that perform arithmetic, and then assign back to the variable, are:

```
Add and assign: +=
Subtract and assign: -=
Multiply and assign: *=
Divide and assign: /=
Increment by 1: ++
Decrement by 1: --
```