

Programming Fundamentals

with JavaScript

The greatest teacher failure is.

- Yoda

Contents

1	The Fundamentals of Programming	6
1.1	Node	6
1.1.1	REPL Mode	7
1.1.2	Script Mode	7
1.1.3	Errors	8
1.2	Hello, World!	9
1.3	Additional Resources	10
2	Basic Types	11
2.1	Numbers	11
2.2	Strings	13
2.3	Additional Resources	15
3	Variables	16
3.1	Naming Variables	18
3.2	Template Strings	21
3.3	Additional Resources	21
4	Boolean Logic	22
4.1	Equality	23
4.2	Logic Rules	24
4.2.1	and (<code>&&</code>)	24
4.2.2	or (<code> </code>)	25
4.2.3	not (<code>!</code>)	25
4.3	Additional Resources	26
5	Conditionals	27
5.1	<code>if</code> statements	27
5.1.1	<code>else</code>	27
5.1.2	<code>else if</code>	28
5.2	The Ternary Operator	30
5.3	<code>switch</code> Statements	32

5.4 Additional Resources	33
6 Loops	34
6.1 <code>for</code> Loops	34
6.2 <code>while</code> Loops	35
6.3 Infinite Loops	35
6.4 Additional Resources	36
7 Functions	37
7.1 Declaring Functions	37
7.2 Calling Functions	38
7.3 Fat Arrow (<code>=></code>)	40
7.4 Examples	41
7.5 Writing Functions	43
7.6 Additional Resources	44
8 Advanced Functions	45
8.1 Scope	45
8.2 Function Composition	46
8.3 Pure Functions	47
8.4 Recursive Functions	49
8.5 Additional Resources	50
9 Arrays	51
9.1 Adding Values	53
9.2 Reading Values	53
9.3 Length	54
9.4 Iterating	54
9.5 All For One and One For All	56
9.6 Useful Operations	57
9.6.1 Sorting	57
9.6.2 Merging Arrays	57
9.6.3 Turning Arrays into Strings	57
9.6.4 Finding a Specific Value	58
9.6.5 Getting Part of an Array	58
9.7 The Spread Operator	59
9.8 Additional Resources	60
10 Array Iterator Methods	61
10.1 <code>forEach</code>	62
10.2 <code>find</code>	63
10.3 <code>filter</code>	63
10.4 <code>map</code>	64

10.5 <code>reduce</code>	64
10.5.1 Default Value	65
10.6 Which Iterator Method?	65
10.7 Functional Programming	66
10.8 Additional Resources	68
11 Objects	69
11.1 Reading Properties	70
11.2 Writing Properties	70
11.3 Methods	71
11.4 <code>this</code>	72
11.5 Additional Resources	73
12 The Standard Library	74
12.1 Strings	74
12.2 Date	75
12.3 Math	76
12.4 Additional Resources	77
13 Advanced Object Techniques	78
13.1 Destructuring	78
13.2 The Spread Operator	79
13.3 Keys and Values	81
13.4 Map, Filter, & Reduce	83
13.5 Additional Resources	83
14 Classes	84
14.1 An Example	85
14.2 Additional Resources	86
Glossary	87
Additional Resources	89

How To Use This Document

Bits of text in magenta are links and should be clicked at every opportunity. Bits of text in monospaced yellow represent code. Most the other text is just text: you should probably read it.

Copying and pasting code from a PDF can mess up indentation. For this reason large blocks of code will usually have a [View on GitHub] link underneath them. If you want to copy and paste the code you should follow the link and copy the file from GitHub.

Taking Notes

In earlier cohorts I experimented with giving out notes in an editable format. But I found that people would often unintentionally change the notes, which meant that the notes were then wrong. I've switched to using PDFs as they allow for the nicest formatting and are also immune from accidental changes.

Make sure you open the PDF in a PDF viewing app. If you open it in an app that converts it into some other format (e.g. Google Docs) you may well miss out on important formatting, which will make the notes harder to follow.

I make an effort to include all the necessary information in the notes, so you shouldn't need to take any additional notes. However, I know that this doesn't work for everyone. There are various tools that you can use to annotate PDFs:

- Preview (Mac)
- Edge (Windows)
- Hypothes.is
- Google Drive (*not* Google Docs)
- Dropbox

Do not use a word processor to take programming notes! (e.g. Google Docs, Word, Pages). Word processors have the nasty habit of converting double-quotes into “smart-quotes”. These can be almost impossible to spot in a text-editor, but will completely break your code.

Chapter 1

The Fundamentals of Programming

The most disastrous thing that you can ever learn is your first programming language

- Alan Kay

There are four fundamental concepts underpinning *all* programming languages:

- **Types**: sorts of things
- **Variables**: remembering things
- **Boolean Logic & Conditionals**: deciding things
- **Loops**: repeating things

We're going to be learning these concepts using JavaScript, but once you understand them you should be able to pick up new programming languages easily.

1.1 Node

When we're first learning to use a programming language it's useful to keep things as simple as possible. Although JavaScript was originally written to run in a web browser, it is also possible to run it in a more "pure" form, where we don't have to worry about HTML or CSS. One way to do this is in the browser's **console**, but we would still need a basic HTML page to load the JavaScript file in the first place.

We're going to be using **Node** to run JavaScript in the command-line. This will keep things simple and also allow you to practice using the command-line.

For our purposes, Node can be run in one of two ways: as a REPL (Read-Eval-Print Loop) or for running a JavaScript file.

1.1.1 REPL Mode

The REPL mode is useful if you want to quickly try something out. To use REPL mode simply type in **node** to the command-line and press return. You should get a welcome message and then see a **>** symbol on the left side of the screen. Node is waiting for you to type something.

If you type in some valid JavaScript and press return, Node will *read* the code, *evaluate* it (i.e. run it) and then *print* the answer (i.e. display it). It will then wait for further input (creating a *loop*).

You know you're in REPL mode if you can see the **>** symbol on the left hand side. If you try and run command-line commands (e.g. **cd**, **ls**) in this mode you'll get an error as it's almost certainly not valid JavaScript.

To leave REPL mode you can press **Ctrl¹+c** twice or type **.exit** and press return.

1.1.2 Script Mode

Script mode lets you run a JavaScript file. It will run through every line of code in the file and then exit back to the command-line. If there are any calls to **console.log()**, those will be printed on the screen, otherwise you won't see anything.

If we had a file in the current directory called **fish.js** we would run it in Node using:

```
node fish.js
```

Make sure you're in command-line mode if you want to run a script. If you've got a **>** on the left side of your screen, make sure you exit REPL mode first.

¹It's **Ctrl** even on a Mac

1.1.3 Errors

When something goes wrong in Node you will see an error. **Errors are your friend:** they normally tell you what you've done wrong. Ignore them at your peril.

In REPL mode errors are usually fairly easy to understand.

In script mode they are often harder to grok. Here is a standard Node error:

```
1 /Volumes/Projects/coding-fellowship/week-03/fish.js:2
2 let fish = cow + monkey;
3           ^
4
5 ReferenceError: monkey is not defined
6     at Object.<anonymous>
7     ↵ (/Volumes/Projects/coding-fellowship/week-03/fish.js:2:18)
8     at Module._compile (internal/modules/cjs/loader.js:774:30)
9     at Object.Module._extensions..js
10    ↵ (internal/modules/cjs/loader.js:785:10)
11    at Module.load (internal/modules/cjs/loader.js:641:32)
12    at Function.Module._load (internal/modules/cjs/loader.js:556:12)
13    at Function.Module.runMain (internal/modules/cjs/loader.js:837:10)
14    at internal/main/run_main_module.js:17:11
```

The error is made up of various parts:

- Line 1: tells us the file in which the error occurred (the part before the colon) and the line on which it occurred (the number after the colon). This *should* be the file that you're trying to run, if it's not then see below.
- Line 2: shows you the actual code where the error was.
- Line 3: a little arrow pointing at the exact part of the code that caused the issue (**monkey**).
- **Line 5:** what's gone wrong (**monkey** is not defined). This is the most useful part of the error message as it tells you what you need to do to fix it.
- Line 6+: a **stack trace**. Shows every bit of code that has run to get to the error. This can be very useful when you're working with multiple files, but for now you can probably ignore it, as it will mostly consist of Node's core files.

If Line 1 is not a file that you created, then Node probably wasn't able to run your code. This is most likely because the file you are trying to run doesn't exist:

```
internal/modules/cjs/loader.js:626
    throw err;
^

Error: Cannot find module
→ '/Volumes/Projects/coding-fellowship/week-03/test.js'
```

If you get an error like this make sure the file exists in the current directory.

Command-Line Fu

A few useful commands:

- **ls**: show all the files in the current directory
- **pwd**: shows the “present working directory” - e.g. the directory that you are in

It's also worth noting that if you press **up** on your keyboard it will bring up the previous command - which you'll be needing a lot to start with.

Also be sure to use the **tab** key to auto-complete directories and filenames. This will also stop you trying to load files that don't exist.

1.2 Hello, World!

It is traditional to write a “Hello, world” program before going any further.

Put the following in a file named **hello-world.js**:

```
console.log("Hello, world!"); // Hello, world!
```

Next, in the command-line go to the same directory as the file you just created and run **node hello-world.js**.

Congratulations, you're now a programmer!

Hello, world?

We use the “Hello, world!” app because it lets us compare the same functionality between different languages.

For example, here it is in PHP:

```
<?php  
echo "Hello, world!";
```

In Java:

```
public class HelloWorld  
{  
    public static void main(String[] args)  
    {  
        System.out.println("Hello, world!");  
    }  
}
```

And Haskell:

```
main :: IO ()  
main = putStrLn "Hello, world!"
```

Just by looking at these few lines of code you can work out quite a lot about a language: PHP seems to need to be told that it’s PHP, Java is a tad verbose, and Haskell is... different.

You can see “Hello, world” in almost every programming language on [Rosetta Code](#)

1.3 Additional Resources

- [The Weird History of JavaScript](#)

Chapter 2

Basic Types

Types are the different sorts of things that a programming language can recognise and work with. Almost all languages have basic types representing numbers, strings (sequences of characters), and booleans (true and false). Most languages also have types representing more complex “data structures” such as arrays and objects, which we’ll be looking at later in the week.

2.1 Numbers

In JavaScript there is a single “Number” type. All of the following are valid number values:

```
10          // an integer
-10         // a negative number
1.2345      // a number with a decimal point
1.5e3        // scientific notation for 1500
0           // zero
-0          // negative zero!
Infinity    // the concept of Infinity
```

Numerical Limitations

Many languages have separate types for integers and decimal point numbers (often referred to as “floats”), but JavaScript isn’t fussy. This is a mixed blessing: you don’t have to worry about converting between different types of number, but you also get some fairly weird results at times:

```
0.1 + 0.2 // 0.30000000000000004
```

We get this weird result because JavaScript stores numbers in a binary system that’s not optimised for precision (the [IEEE 754](#) format). It’s like how we can express one third as a fraction exactly ($\frac{1}{3}$) but we can’t represent it exactly in the decimal system (0.333333...)

The intricacies of how numbers are stored in JavaScript and why you get these weird results are all covered in [an excellent talk by Bartek Szopka](#).

Basic types normally have a number of associated “operators”. Numbers have the following:

Operator	Name	Description
+	addition	adds two numbers together
-	subtraction	subtracts the second number from the first number
*	multiplication	multiplies two numbers
/	division	divides the first number by the second
%	modulus	remainder after dividing the first number by the second

You’re probably familiar with these concepts, except for **modulus** - but don’t forget that one: it comes in handy in all sorts of places.

Arity

These are all **binary operators**, meaning that they require two values (have an *arity* of 2), one on each side: `40 + 2`. You also get **unary** and **ternary** operators, which we’ll come across shortly.

Try the following in **node**:

```
10 + 10 // 20
10 - 20 // -10
50 / 3 // 16.6666666667
10 / 5 // 2
100 * 2 // 200
11 % 3 // 2
```

2.2 Strings

Strings represent a sequence of characters. It's probably easiest to think of them as storing words, but that's not quite right as they can store parts of a word, whole sentences, entire books, or just a single smiling poop emoji¹.

In order to get JavaScript to recognise a string we surround it with quotes:

```
"cow"
'a string'
"an even longer string"
```

You can use double or single quote marks around strings², but try and stick to one or the other.

There is a special string known as the **empty string**, which is simply two sets of quote marks with nothing in between (not even a space): `" "`. You'll probably use this a lot.

Strings only have a single operator, `+`, known as the **concatenation** operator. It joins two strings together:

```
"hello" + " " + "world" // "hello world"
"fish" + "sticks"      // "fishsticks"
```

¹It's also worth noting that "word" has a [technical meaning in computing](#)

²This is not true in all programming languages: some only allow double quotes and some (e.g. PHP) treat single and double quotes slightly differently

The above examples are a little contrived as you could (and, in real code, *should*) write them both as a single string:

```
"hello world" // "hello world"  
"fishsticks" // "fishsticks"
```

However, until we learn about variables there's no other way to demonstrate concatenation.

Strings & Numbers

You need to be careful when using numbers and strings together: they won't always do what you want.

The `+` operator has two meanings (it is **overloaded**): addition and concatenation. So JavaScript has some rules to work out what it should be:

```
12 + 12      // 24 - a number  
"12" + 12    // "1212" - a string  
120 + "1"    // "1201" - a string  
5 + 6 + "1"  // "111" - a string
```

Basically, if it comes across a string everything from that point on will be treated as a string too.

You can guard against this by putting an additional `+` symbol before a value that might not be a number. This **casts** the string value into a number value:

```
+ "12" + 12 // 24
```

Again, this is a somewhat contrived example, as in the case above you could simply not write the quote marks (`12 + 12`), but once we start storing values in variables it will make more sense.

It is often necessary to cast a string to a number when getting values from the browser (e.g. an input's value will come back as a string).

2.3 Additional Resources

- Eloquent JavaScript: Values, Types, and Operators
- JavaScript for Impatient Programmers: Variables & Values
- The Why Behind the Wat: An Explanation of JavaScript's Weird Type System
- WTFJS: The idiosyncrasies of JS

Chapter 3

Variables

Variables are a way of storing a value using a name so that we can refer to it later.

This serves two purposes:

1. We can store the results of complex calculations so that we only have to calculate them once
2. If we're sensible about what we call our variables it makes our code much easier to follow

Before we use a variable we must **declare** it using the **let** keyword:

```
let email;  
let age;
```

Declaring a variable lets JavaScript know that from that point on, if we use that series of characters in our code, it represents a value.

Once we've declared the variable, we can **assign** it a value:

```
email = "orb@is.horse";  
age = 32;
```

We only need to declare a variable once. From that point on we can reassign the value if we want to:

```
// elsewhere  
email = "farm@wisdom.com";  
  
// elsewhere  
email = "shrimp.heaven@now.plumbing";
```

Once we've stored a value in a variable we can use it to represent that value elsewhere in our code:

```
let pointless;  
pointless = email + age; // "shrimp.heaven@now.plumbing32"
```

Generally we declare and assign variables at the same time:¹

```
let name = "Archie";  
  
let age = 4, // can also declare multiple variables in one go  
houseNumber = 21;  
  
// using variables  
let notUseful = age + houseNumber; // 25
```

¹If you see old JavaScript code that uses `var`, you will often find all the variables declared at the top of the file and values assigned to them later. This is because of something called “**hoisting**”. Luckily it’s not necessary if you use `let`.

Variable Types

let

The most commonly used in modern JS. Use this unless you can think of a good reason not to.

```
let value = 10;
```

const

Useful if you want to make sure a value can't be changed, for example if you had a variable that stored some configuration.

```
const maxVolume = 10;  
maxVolume = 11; // Error - can't assign a new value to maxVolume
```

var

Very common in older JS. Works almost identically to `let` except when it comes to **scoping**. Stick to `let` unless you're dealing with legacy code.

```
var meh = 10;
```

3.1 Naming Variables

We can call a variable pretty much anything we want, but it's best to pick something that represents its purpose.

```
let name = "Ben"; // good  
let a = 394; // possibly ok for short bits of code  
let aRidiculouslyLongVariableName = 83; // maybe a bit long  
let appleSauce = 394; // huh?  
let name = "Not Ben"; // already used that...
```

A variable name can contain:

- alphanumeric characters

- underscores
- the dollar sign

It cannot:

- contain spaces
- contain hyphens
- start with a number
- be a **reserved word** (e.g. `class`, `let`, `var`, `function`, `if`, and many more)

We tend to use camel-case (lowercase first letter, uppercase beginnings of words - `likeThis`) - as opposed to snake-case (all lowercase, underscores between words - `like_this`). You don't have to, but if you don't your code will look weird to everyone else and your friendship group will slowly dwindle.

If we pick our variable names sensibly then it's easy to see what our code does:

```
let username = "potus";
let password = "00000000";

armNuclearWeapons(username, password);
```

If we pick our variables names poorly it can be impossible to work out what's going on:

```
let a = "potus";
let b = "00000000";

doWhatEvs(a, b); // we just started a thermonuclear war - oops!
```

Comments

It's a good idea to explain unusual parts of your code. You can do this with comments.

If you put `//` on a line in JavaScript then everything after that will be ignored when your code runs:

```
// The number of milliseconds in a year
// 1000 * 60 * 60 * 24 * 365.2425
let millisecondsPerYear = 31556952000;
let another = 12345; // you can comment at the end of a line too
```

You can also do multi-line comments using `/*` and `*/`. Everything between the opening `/*` and the closing `*/` will be ignored.

This can be useful if you need to temporarily disable a bit of code. But *make sure you don't leave unused code lying around once everything is working*.

```
/*
 * The number of milliseconds in a year
 * Calculated using: 1000 * 60 * 60 24 * 365.2425
 * Required for date calculations
 */
let millisecondsPerYear = 31556952000;
```

The extra `*` at the beginning of each line isn't necessary - but it looks nicer.

If you change a bit of code, *make sure you update the corresponding comments*: old/incorrect comments are worse than no comments at all.

3.2 Template Strings

We often want to include something stored in a variable as part of a string.

One option would be to concatenate the variables:

```
let name = "Chetna";
let greeting = "Hello " + name + ", how are you?";
```

However, rather than using quotation marks, we can put backticks (`) around our strings. This allows us to **interpolate** values:

```
let name = "Chetna";
let greeting = `Hello ${name}, how are you?`;
```

As you can see, we use `${variable}` inside the backticks to insert the value contained in a variable.

3.3 Additional Resources

- [Eloquent JavaScript: Values, Types, and Operators](#)
- [JavaScript for Impatient Programmers: Variables & Values](#)
- [Template Strings](#)

Chapter 4

Boolean Logic

The best thing about a boolean is even if you are wrong, you are only off by a bit

- Anonymous

Modern digital computers¹ use **boolean** logic: true and false. Because these are such fundamental ideas in computing, JavaScript has the special **true** and **false** values (lowercase, no quotation marks).

```
// setting variables to boolean values
let news = true;
let lies = false;

// don't use strings!
let fakeNews = "false"; // as far as JS is concerned, this is true
```

¹Analogue computers are another matter

4.1 Equality

The ideas of `true` and `false` are most useful when it comes to comparing things.

We can compare things with various operators:

Operator	Name	Description
<code>==</code>	strict equality	<code>true</code> if the values are the same
<code>!=</code>	non-equality	<code>false</code> if the values are the same
<code><</code>	less than	<code>true</code> if the first value is less than the second value
<code>></code>	greater than	<code>true</code> if the first value is greater than the second value
<code><=</code>	less than or equal to	<code>true</code> if the first value is less than or equal to the second value
<code>>=</code>	greater than or equal to	<code>true</code> if the first value is greater than or equal to the second value

For example:

```
10 === 10;    // true
10 === 12;    // false
"12" === 12; // false - a string is not a number
10 <= 12;    // true
10 < 10;     // false
10 >= 12;    // false
10 > 9;      // true
10 != 14;    // true
```

Sort of Equal

In many languages if you tried to compare a string and a number they'd think you were mad. But JavaScript isn't fussy about what types of things your variables store. That means you can compare different sorts of things and JavaScript will give it a go.

Because of this JavaScript also has the `==` and `!=` operators. These **type cast**: they convert one or the other side of the operator to be the same sort of thing as the other side before checking if the values are the same.

This lets you do things like `12 == "12"` and get `true` back.

This might seem really useful, but using it suggests you don't know what types of values you're dealing with, which means you don't really understand what your code is doing. So you should stick to `==`, which first checks if both values are the same type and immediately returns `false` if they aren't.

4.2 Logic Rules

There are a number of operators that we can use when working with boolean values, these represent the key rules of boolean logic: **and**, **or**, and **not**.

4.2.1 and (`&&`)

If either value is `false`, the result is `false`:

```
true && true; // true
false && true; // false
true && false; // false
false && false; // false

(10 > 12) && (1 < 2); // false
(10 < 12) && (1 < 2); // true
(10 > 12) && (1 > 2); // false
```

If you think about the phrase “My name is Mark and I live in Bristol”, we'd say that the whole phrase is false if either (or both) of the sides are false: “My name is Brian and I live in Bristol” is false, even though the right side is true. The phrase as a whole can only be true if both sides are true.

4.2.2 or (||)

If either value is **true**, the result is **true**:

```
true || true; // true
false || true; // true
true || false; // true
false || false; // false

(10 === 10) || (2 !== 1); // true
(10 === 12) || (1 !== 2); // true
(10 >= 12) || (2 <= 1); // false
```

This one doesn't work quite so well with the common sense notion of "or". If you think about the phrase "My name is Mark or I live in Bristol", some people might be inclined to think that it's true when exactly one side is true². However, the standard interpretation in boolean logic is that as long as at least one side of the phrase is true, then the whole phrase is true.

4.2.3 not (!)

Reverses the truth value. Turns **true** to **false** and **false** to **true**:

```
!true; // false
!false; // true

!!true; // true
!(10 > 12); // true
!!(10 > 12); // false
```

Notice that **not** is a **unary** operator: it only takes a single value.

²This is a useful concept and is known as **exclusive or** or **XOR**

Casting to Boolean

When we use `!` twice, it first flips the boolean value (either from `true` to `false` or vice versa) and then flips it again, so you end up with the original boolean value.

This is completely pointless if the values are already boolean, but it can be useful for casting a non-boolean value to a boolean:

```
!!0;      // false
!!10;     // true
!!"";     // false
!!"false"; // true
```

4.3 Additional Resources

- [JavaScript “Loose” Comparison Step by Step](#)
- [JavaScript Equality Table](#)

Chapter 5

Conditionals

5.1 if statements

Computers need to be able to make decisions: if such-and-such is the case then do something. Without this ability we couldn't write programs that do anything other than basic calculations.

We can use our logic rules in combination with `if` statements to decide what our program will do:

```
let value = 8;
let max = 10;

if (value <= max) {
    console.log("Valid"); // this will run
}
```

The part inside the `if` brackets is the **truth condition**¹ - this will always be evaluated as either `true` or `false`. If it evaluates to `true`, then the block of code will execute, if it evaluates to `false` then it won't.

5.1.1 else

You can also add an `else` block. If the conditional is `true` then the first block of code runs, otherwise the second block of code will run.

¹Hence “conditional”

```
let username = "mark";

// the truth condition
if (username === "admin") {
    // this runs if it's true
    console.log("Hello Admin"); // won't run
} else {
    // this runs if it's false
    console.log("Unauthorized!"); // will run
}
```

5.1.2 else if

You can also use `else if` blocks. These let you check multiple conditions. You can have as many of these as you like.

As in the previous case, one and only one of these blocks of code will run. If the first condition is `false` then the first `else if` condition is checked; if that is also `false` then it will move onto the next; and so on until it reaches the final `else` block, which will only run if *all* of the previous conditions have returned `false`.

```
let average = (10 + 13 + 15) / 3;

if (average <= 10) {
    console.log("Less than 10"); // won't run
} else if (average < 20) {
    console.log("Less than 20"); // will run
} else {
    console.log("20 or more"); // won't run
}
```

If we hadn't included the final `else` block, then it's possible that none of the code blocks would run:

```
let average = (30 + 50 + 100) / 3;

if (average <= 10) {
    console.log("Less than 10"); // won't run
} else if (average < 20) {
    console.log("Less than 20"); // won't run
}
```

Truthy & Falsey Values

Because JavaScript isn't that fussy about types, if you try and use a non-boolean value as a condition it will do its best to make things work:

```
let value = "Hello";

if (value) {
    // even though the condition isn't a boolean value
    // this will run as a non-empty string is "truthy"
}
```

Falsey values (ones that type cast to `false`) in JavaScript are:

- `false`
- `0`
- `""` (the empty string)
- `null`
- `undefined`
- `Nan`

Everything else is truthy (i.e. it type casts to `true`).

5.2 The Ternary Operator

If the blocks of your conditional are both short, it can sometimes save space to use the [ternary operator](#).

Unlike an [if statement](#) the ternary operator is an [expression](#), meaning that it equates to a value.

```
let current = 3;

// set the value of next, dependent on the value of current
let next = current > 5 ? 0 : current + 1; // next is set to 4
```

Does the same as:

```
let current = 3;
let next;

if (current > 5) {
    next = 0;
} else {
    next = current + 1;
}
```

The ternary operator consists of three parts:

1. The condition (before the `?`)
2. The *if true* result (before the `:`)
3. The *if false* result (after the `:`)

In other words, if the bit before the `?` is true, then evaluate to the bit before the `:`, otherwise evaluate to the bit after the `:`.

Statements & Expressions

An **expression** is any bit of code that equates to some value. If you could assign the whole bit of code to a variable, then it's an expression. An expression can be made up of sub-expressions.

For example, all of the following equate to a value, so they are expressions:

```
// equates to 46
12 + 34;

// equates to true
((true && false) || false) === false;

// equates to some number, assuming celsius is a number
(celsius * 9/5) + 32;
```

A **statement** is just an instruction. Things like **if** statements or variable assignments: they do something, but aren't equal to a value. Statements can be made up of other statements and expressions. All expressions are also statements.

5.3 switch Statements

If your `if` statement is just a series of checks against the same value, then a `switch` statement can sometimes save space:

```
let username = "admin";

switch (username) {
    case "admin": console.log("Authorized"); break;
    case "Admin": console.log("Authorized"); break;
    case "Jeff": console.log("Back door! Authorized"); break;
    default: console.log("Unauthorized");
}
```

We use the `switch` keyword and then in brackets we put the value we want to check against. Inside the block, we list a series of `case` statements: if the value in the `switch` matches a specific `case` then that case statement will run. It will only run the first matching `case`. We can also have a `default` case which works much like an `else` block: it will run if none of the case statements are true.

`switch` statements are neater than multiple `else if` statements, but make sure you remember to use `break` at the end of each `case`, otherwise you'll get unexpected behaviour².

²If you don't use `break` then every `case` statement below the one that was true will also run. This is called **fall-through** and is occasionally useful, but if you see it in code it's hard to know whether it's deliberate or not, so it's best not to use it

Indenting

You'll notice that all of the code inside the `if` blocks is indented:

```
// anything between the opening and closing
// curly braces should be indented
if (x === 10) {
    console.log("x is 10"); // indented
} // <-- first block block stops here, so don't indent anymore

// new block, so indent everything inside the braces
if (x < 20) {
    console.log("Less than 20"); // indented

    // a new block, so indent another level
    if (average < 20) {
        console.log("Less than 20"); // indented two times
    } // <-- end of a block, indent one less
} // <-- end of outermost block, so stop indenting

console.log("Less than 20"); // <-- outside of blocks, not indented
```

This makes it clear which bits of the code are part of the if blocks and which bits are not.

Basically, when you get to a `{` you should indent another level and when you get to a `}` you should de-indent.³ Your text editor will probably do this for you and, at this stage, it probably knows best.

Fix indenting problems as soon as you spot them - it will save you time later!

5.4 Additional Resources

- [MDN: Conditionals](#)
- [Eloquent JavaScript: Program Structure](#)
- [JavaScript for Impatient Programmers: Control Flow](#)

³Unindent? Outdent? Dent?

Chapter 6

Loops

We use a loop when we want to do something similar more than once.

6.1 for Loops

for loops are useful when you know how many times the loop should run.

They consist of three parts:

1. `let i = 0`: setup a variable that we use as a counter
2. `i < 10`: keep running the loop as long as this is true
3. `i += 1`: increment `i` by 1 every time the loop runs

```
// will keep running until i is 9
for (let i = 0; i < 10; i += 1) {
    console.log(i);
}

// 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

It is traditional to use `i` as the counter variable - one of the few places it's good practice to use a single letter variable name. If you need a loop inside a loop then just keep going down the alphabet (`j`, `k`, ...).

6.2 while Loops

We use a `while` loop if we aren't sure how many times the loop needs to run.

For example, say we wanted to add 1 to 2 to 3 to 4 and so on until we get to a number bigger than 100. We know when we want it to stop (when the total is bigger than 100), but we don't know how many times it needs to run.

```
let i = 0;      // keeps track of which number we're adding
let total = 0; // keeps track of the total so far

// will keep running until total is more than 100
while (total <= 100) {
    total += i;
    i += 1;
}

console.log(total); // 105
console.log(i); // 15 - so the loop ran 15 times
```

6.3 Infinite Loops

We need to be careful to avoid infinite loops: loops that never stop running. These sometimes occur because of typos, but more often because you use a variable that isn't equal to what you're expecting.

```
for (let i = 0; i < 10; i -= 1) {
    // will never stop
    // why not?
}
```

An infinite loop will keep running until you kill the process that's running it. If you're running code in `node` and you think you've got an infinite loop then press `Ctrl+C`, which will kill the `node` process.

6.4 Additional Resources

- MDN: Loops
- Eloquent JavaScript: Program Structure
- JavaScript for Impatient Programmers: Control Flow
- Radiolab: Loops: Nothing to do with programming loops, but really interesting!

Chapter 7

Functions

In programs we often need to do the same thing multiple times in different places. Because of this almost all programming languages have the concept of **functions**.

A function is a block of code that performs a specific action. We usually give the function a name, which allows us to use it elsewhere in our code. A function can accept and return values. For example, a function called `authorize` might accept a username and password and return `true` if the combination is valid or `false` if not.

7.1 Declaring Functions

A function looks like this:¹:

```
// a function to add two numbers together
// this function gets given two values
function (a, b) {
    // it adds the two values together
    // and then returns the sum
    return a + b;
}
```

The above function isn't useful because we haven't given it a name (an **anonymous function**), so we don't have any way to use it in our code.

¹In fact, functions in JavaScript can look a number of different ways

But a function is a value in JavaScript², so, as with any other value, we can assign functions to variables:

```
let add = function (a, b) {  
    return a + b;  
};
```

We can now refer to the function using the `add` variable.

7.2 Calling Functions

Functions do not run until you `call` them. You call a function by using the function name followed by a pair of brackets.

When you call it, you can send a function values (known as `arguments`) which it can use.

For example, the `add` function takes two arguments:

```
add(1, 2); // call add, passing the values 1 and 2  
add(3, 7); // call add, passing the values 3 and 7
```

The arguments are passed to the function in the order that they are given.³ We can then use the passed values in the function:

```
// if we call add(1, 2) then inside the function  
// a = 1 and b = 2 as the values are passed in order  
let add = function (a, b) {  
    return a + b;  
};
```

In the above example `a` and `b` are effectively variables that are temporarily assigned the values that you pass to the function when you call it. As with any other variable you can call them whatever you like.

²This is *not* true in a lot of languages

³Also worth noting that in JavaScript it's possible to pass too many or too few arguments and the code will still run - although it generally won't do what you want it to.

Most the time functions will return a value:

```
let onePlusTwo = add(1, 2); // 3
let threePlusFive = add(onePlusTwo, 5); // 8
```

A function can contain as much code as you like, although well-written functions should only try to do one thing.

A function that doesn't explicitly return anything returns **undefined**:

```
let oops = text => {
  text + "!";
};

let value = oops("Hello");
console.log(value); // undefined
```

If **undefined** sneaks into your calculations you're likely to start getting strange results.

Arguments & Parameters

The values that we pass to the function when we call it are the **arguments**. When we accept those values in the function declaration they are called **parameters**.

```
// a and b are the function parameters
// the variable names we use inside the function body
let add = function (a, b) {
  return a + b;
};

// 1 and 2 are the arguments
// the values we call the function with
add(1, 2);
```

This is a fairly technical distinction that you can easily get by without knowing. A lot of programmers will use the term "arguments" in both cases.

7.3 Fat Arrow (`=>`)

In modern browsers we can use fat arrow (`=>`) to neaten up our function declarations:

```
let add = (a, b) => a + b;  
  
// works exactly as before  
let onePlusTwo = add(1, 2); // 3  
let threePlusFive = add(onePlusTwo, 5); // 8
```

If the function does something simple then this can save quite a lot of boilerplate: you no longer need to write out `function` and as long as the function fits on one line you also don't need curly braces or a `return` - it automatically returns whatever the right-hand side evaluates to.

If the function only takes one argument, you can even skip the brackets:

```
let square = n => n * n;  
  
// call add with an argument  
let twoSquared = square(2); // 4  
let fiveSquared = square(5); // 25
```

You can still use `=>` for multi-line functions:

```
let sum41 = (a, b) => {  
  let total = a + b;  
  return total === 41;  
};
```

We need to use curly braces in this instance - which also means we need to manually `return` a value.

It's worth noting that fat arrow syntax is not identical to a traditional function declaration: it inherits its parent's `context`. However, this is a technicality and is unlikely to cause you any issues.

Unless you have a good reason not to, you should use fat arrow syntax.

7.4 Examples

A function to greet someone:

```
// greet takes one argument
// multi-line, so we need curly-brackets
let greet = name => {
    // get the current hour of the day
    let hour = new Date().getHours();

    if (hour < 12) {
        return "Good morning " + name;
    } else if (hour < 18) {
        return "Good afternoon " + name;
    }

    // when a function returns a value it stops running
    // so this will only ever run if the two return statements
    // above don't run
    return "Good evening " + name;
};

let greeting = greet("Ezra");
console.log(greeting);
```

A function to multiply three numbers:

```
// product takes three arguments
let product3 = (a, b, c) => a * b * c;

// call product, separating arguments with commas
let result = product3(2, 3, 4); // 24
```

Returning Booleans in Conditionals

If you ever have a conditional where you return `true` in one case and `false` in the other:

```
if (expression) {  
    return true;  
} else {  
    return false;  
}
```

`expression` should always be a boolean value, as it's a condition in the `if` statement. So you can return the condition (`expression`) instead of using a conditional:

```
return expression;
```

Likewise, you can always replace:

```
if (expression) {  
    return false;  
} else {  
    return true;  
}
```

with:

```
return !expression;
```

7.5 Writing Functions

Here's the process to go through when you're writing a function:

1. Think of a sensible name for the function: a short way of describing its purpose
2. Think about how many arguments the function needs to accept: this will depend on what you're trying to do
3. Think about what type of thing the function should return
4. Write out the boilerplate
5. The thinky bit: Work about how to turn the arguments into the return value
6. Test it with a few values you know the answer to

For example, a function to convert Fahrenheit to Celsius:

1. Let's call it `celsius`
2. It should take a single argument: a number (the temperature in Fahrenheit), let's call the parameter `fahrenheit`
3. It should return a number
4. First put in the boilerplate:

```
let celsius = fahrenheit => {
    // needs to return a number: fahrenheit in celsius
};
```

5. To convert Fahrenheit into Celsius you need to take away 32 and divide by 1.8 (see [DuckDuckGo](#))

```
let celsius = fahrenheit => {
    return (fahrenheit - 32) / 1.8;
};
```

6. Check it with a few values:

```
celsius(45); // 7.222222
celsius(32); // 0
```

7.6 Additional Resources

- Eloquent JavaScript: Functions
- MDN: Functions
- A Gentle Introduction to Functional JavaScript
- What the heck is the event loop anyway?

Chapter 8

Advanced Functions

8.1 Scope

The **scope** of a variable is the parts of code that you can use the variable name in.

let and **const** are **block scoped**.¹ This means that the variable's value is only accessible from within the block in which it is declared (including any blocks within that block).

Variables created without declaring them will go into **global** scope.

```
let x = 10;

if (x === 10) { // new block
  let x = 20;
  y = 50; // goes into global scope, no variable declaration
  console.log(x); // 20 - different x
}

// logs 10: the x inside the if statement is scoped to that block
console.log(x);
```

Function parameters are always scoped to the function they belong to. Variables created with **var** are also scoped to the containing function, *not* block scoped like **let** and **const**.

¹This is probably the most common form of scoping in programming languages

```
var x = 1; // in the global scope

var fn = y => { // y is only available inside the function
  var z = 2; // z is only available inside the function

  if (z < 3) {
    var z = 4; // overwrites z above - var not block scoped
  }

  // can reference x, it was declared outside of the function
  return z + x + y;
};

var result = fn(12); // we can only access x, result, and fn here
```

8.2 Function Composition

If you write a function that works, then don't be tempted to add additional functionality later. Instead, write a new function that calls the one you've already written.

For example, say you had a function that works out if a number is divisible by 3. You then need to write a function that works out if a number is divisible by 3 *and* a square number. Rather than editing your existing `divisibleBy3` function, create a new function that calls the `divisibleBy3` function.

```
// does number divide by 3 with no remainder
let divisibleBy3 = n => n % 3 === 0;

// is square root of number an integer
let isSquare = n => Math.sqrt(n) % 1 === 0;

// combine the two bits of functionality in one new function
let divisibleBy3Square = n => divisibleBy3(n) && isSquare(n);
```

If you get used to writing short functions that do one thing well, you'll find it much easier to perfect the art of **function composition**, arguably one of the most important skills in writing complex programs.

8.3 Pure Functions

You may sometimes hear people talk about **pure** functions. A pure function is one that fits the *mathematical* concept of a function: given the same inputs you will always get back the same outputs.

For example, a function like **add** is a *pure* function because if you give call it with **add(4, 5)** you will *always* get back **9**. There are no two inputs you can pass it that will sometimes return one answer but other times return something else.

A pure function need not be doing maths to fit the mathematical concept. The function below is also pure, even though it deals with strings:

```
let shock = a => a + "!";
```

The following function is pure too. It will return the string “spooky” for *any* input:

```
let ghost = () => "spooky";
```

The function **greet** (from the [previous chapter](#)) is *not* pure: depending on the time of day it will return *different* values given the same string. We *could* make it pure by passing the current hour of the day in as a second argument, although this would make it harder to use.

Pure functions also shouldn’t cause **side-effects**: they should only work with the values they are given and not change anything else.

The function below causes side-effects:

```
let x = 10;

let thing = b => {
  x = b + 1;
  return x;
};
```

Even though this function will always return the same value given the same inputs, it changes the value of **x**, which lives outside the function.

The following is also not pure:

```
let thing = b => {
  console.log(b);
  return b + 1;
};
```

Although this presumably doesn't change any values outside of the function it does call `console.log()`, which outputs text somewhere, thus changing *the world itself* - which is a pretty major side-effect!

It's worth noting that pure functions can call *other* pure functions without losing their purity, so the `divisibleBy3Square` function in the previous section is pure, as both of the functions it calls are also pure.

We can't write a useful program consisting *entirely* of pure functions, as it wouldn't be able to output anything or respond to a user's input. But, because they are self-contained, pure functions are generally much easier to reason about. So it's a good idea to use pure functions wherever you can.

8.4 Recursive Functions

In order to understand recursion, one must first understand recursion

- Some smart-arse

A function can call itself. This can be surprisingly powerful:

```
// works out the factorial of n
let factorial = n => {
    // this condition stops the function calling itself forever
    if (n <= 1) {
        return 1;
    }

    // to find the factorial of n
    // times n by the factorial of n - 1
    // e.g. 5! = 5 * 4! = 5 * 4 * 3! etc.
    return n * factorial(n - 1);
};
```

Or if that's too long:

```
let factorial = n => (n <= 1) ? 1 : n * factorial(n - 1);
```

A recursive function creates a looping structure.

Anything we can do with recursion can also be done using traditional loops, but it's often much less elegant.

As with other types of loop, we need to be careful not to create an infinite loop. If we didn't have the `n <= 1` check in the `factorial` function, it would keep going forever.

8.5 Additional Resources

- Scope in JavaScript
- What is Function Composition?
- What is a Pure Function?
- Recursion in Functional JavaScript
- Algorithms: Recursion: *deep* dive into recursion

Chapter 9

Arrays

Sometimes it's useful to work with multiple bits of data that are related in some way. In order to do this we'll need **data structures**: more complex types that allows us to more than one piece of data.

Arrays are ordered collections of values. It's probably easiest to think of them as being a list of values¹.

We can create an array using square brackets and put in values separated by commas:

```
let values = []; // an empty array

let numbers = [1, 2, 3, 4, 5, 6, 7]; // an array of numbers

let animals = [
  "cow",
  "chicken",
  "fish",
  "wombat"
]; // an array of strings
```

Arrays can contain any of the value types: numbers, strings, booleans, functions, objects (which we'll look at later), and, of course, other arrays.

¹It's worth noting that in some programming languages there are different types of list, that have different performance characteristics

JavaScript isn't fussy about types, so a single array can contain different types of values:

```
// this is totally fine in JavaScript
let nope = [1, "fish", n => n * n, [1, 2, 3], true];
```

However, just because you *can*, doesn't mean you *should*. When we're working with arrays we almost always want to go over every value in the array and run the same bit of code for each one. If you start having different types of values in your array you're going to end up with a lot of conditionals. So, *an array should use the same type for all its values*.

Multi-Dimensional Arrays

An array containing other arrays is often called a **multi-dimensional array** as it can represent multi-dimensional values. For example, an array of arrays can represent a table:

```
let table = [
  [1, 2, 3], // first row
  [4, 5, 6], // second row
  [7, 8, 9], // third row
];
```

An array of arrays of arrays can represent three-dimensional structure; four levels deep can represent a four-dimensional structure (e.g. an animated 3D object); and so on.

9.1 Adding Values

We can add items to the end of the array:

```
values.push("a"); // ["a"]
numbers.push(8); // [1, 2, 3, 4, 5, 6, 7, 8]
animals.push("cat"); // ["cow", "chicken", "fish", "wombat", "cat"]
```

And to the beginning:

```
values.unshift("g"); // ["g", "a"]
numbers.unshift(0); // [0, 1, 2, 3, 4, 5, 6, 7, 8]
```

It's important to note that these *change the original array*.

9.2 Reading Values

You can reading values from an array using square brackets and passing in the **index** of the item you want. Arrays are **zero-indexed**², which means the first item has index **0**:

```
let animals = ["cow", "chicken", "fish", "wombat"];
animals[0]; // "cow" - arrays are "zero-indexed"
animals[2]; // "fish"
```

Reading values this way does *not* change the array.

Getting values out of the array, changing the array:

```
let animals = ["cow", "chicken", "fish", "wombat"];
let last = animals.pop(); // "wombat" - removed from end of array
let first = animals.shift(); // "cow" - removed from beginning of array

console.log(animals); // ["chicken", "fish"]
```

²This is true in most languages and goes back to when computers had tiny amount of memory and every bit was precious. The language **Lua** starts indexes at **1** by default.

9.3 Length

We can use the `.length` property to find out how many items there are in an array.

```
let animals = ["cow", "chicken", "fish", "wombat"];
console.log(animals.length); // 4
```

```
let values = [17, 12];
console.log(values.length); // 2
```

Note that `.length` is a **property**, not a function, so it doesn't require brackets to get the value. We'll learn more about properties tomorrow.

9.4 Iterating

So, how could we do something with every item in an array?

- We know how long an array is using `.length`
- We know we can read individual items using `arr[0]`, `arr[1]`, `arr[2]`, etc

We can use a `for` loop to do something with every item in an array. This is called **iterating** over an array.

If an array has a `.length` of `5`, then we know that the indexes will go from `0` to `4`. And we know we can access a specific index using the `values[0]` (where `0` is the index) notation. So, we just need to loop from `0` to `4` and get back that index:

```
let animals = ["cow", "chicken", "fish", "wombat", "kangaroo"];

// start at 0, because arrays are zero-indexed
// finish one less than the length of array
// so, this will loop i from 0 to 4
for (let i = 0; i < animals.length; i += 1) {
  let current = animals[i];
  console.log(current);
}

// "cow", "chicken", "fish", "wombat", "kangaroo"
```

We could, for example, use this to add up all the values in an array:

```
let values = [1, 2, 3, 4, 5, 6];

// keep track of the cumulative total
let total = 0;

// iterate over each item in the array
// adding it to total
for (let i = 0; i < values.length; i += 1) {
  let current = values[i];
  total += current; // the value of total will go up each iteration
}

console.log(total); // 21
```

Or we could use it to create a new array without odd numbers:

```
let values = [1, 2, 3, 4, 5, 6];

// an array to put the even numbers into
let even = [];

// iterate over each item in the array
for (let i = 0; i < values.length; i += 1) {
  let current = values[i];

  // if the value is even add it to the even array
  if (current % 2 === 0) {
    even.push(current);
  }
}

console.log(even); // [2, 4, 6]
```

9.5 All For One and One For All

Although an array *contains* multiple values, we treat it like a single value:

```
// we store it in a single variable
let x = [1, 2, 3, 4, 5, 6];
```

A function being passed arrays is being passed however many *arrays* it is called with, not the number of items in the array:

```
// when we pass it to a function it is a single value
let total = (arr) => {
    let sum = 0;

    for (let i = 0; i < arr.length; i += 1) {
        let current = arr[i];
        sum += current;
    }

    return sum;
}

// even though this array contains 6 values
// it gets passed to the function as a single value
// it's the number of arrays passed in that matters
total([1, 2, 3, 4, 5, 6]);
```

This is very useful as it lets us create functions that can deal with as many values as we like: we can pass in an array with one value, a hundred values, two values, or even no values and the function should work.

9.6 Useful Operations

9.6.1 Sorting

We can sort an array alphabetically:

```
let values = ["b", "c", "a", "d"];
values.sort();

console.log(values); // ["a", "b", "c", "d"]
```

However, this won't be much use if you want to sort numbers:

```
let values = [9, 11, 40, 112, 89, 380];
values.sort();

// sorts numbers alphabetically
console.log(values); // [11, 112, 380, 40, 89, 9]
```

9.6.2 Merging Arrays

We can merge two arrays:

```
let v1 = [1, 2, 3, 4];
let v2 = [3, 4, 5, 6];
let merged = v1.concat(v2);

console.log(merged); // [1, 2, 3, 4, 3, 4, 5, 6]
```

9.6.3 Turning Arrays into Strings

It's often useful to join an array of string into a single string:

```
let letters = ["a", "b", "c", "d"];
let alphabet = letters.join(" - ");

console.log(alphabet); // "a - b - c - d"
```

You could pass an empty string in to `.join("")` which would join the strings with nothing between.

9.6.4 Finding a Specific Value

Finding a value:

```
let letters = ["a", "b", "c", "d"];
console.log(letters.indexOf("b")); // 1
console.log(letters.indexOf("d")); // 3
console.log(letters.indexOf("e")); // -1
```

We get back the index of the first match. If no match is found you get back `-1`.³

9.6.5 Getting Part of an Array

Getting part of an array:

```
let numbers = [3, 4, 5, 6, 7, 8, 9];
// first argument is index to start on
// second argument is index to finish before
let middle = numbers.slice(2, 5); // [5, 6, 7]
```

³It doesn't return `false` because it's a good idea for functions to always return the same type of value

9.7 The Spread Operator

We can use the **spread operator** (...) to copy an array:

```
let numbers = [3, 4, 5, 6, 7, 8, 9];

// ...numbers represents all the values in the array
// but as separate values
let copied = [...numbers]; // [3, 4, 5, 6, 7, 8, 9]

// without the spread operator we'd get an array inside an array
let oops = [numbers]; // [[3, 4, 5, 6, 7, 8, 9]]
```

The ...numbers bit effectively stands in for each item in the array as a separate element. So we're creating a new array with the [and] and then putting all of the elements from numbers into it.

We can also use it to merge two arrays into a new array:

```
let odd = [3, 5, 7, 9];
let even = [4, 6, 8];

let mergedOddEven = [...odd, ...even]; // [3, 5, 7, 9, 4, 6, 8]
let mergedEvenOdd = [...even, ...odd]; // [4, 6, 8, 3, 5, 7, 9]
```

This is useful because a lot of operations on arrays (like **pop** and **push**) alter the original array, which can lead to confusing results:

```
// a badly written function to get the last item in an array
let last = arr => arr.pop();

let values = [1, 2, 3]
last(values); // 3 - but values is now [1, 2]
```

If we make a copy first, we don't need to worry about this:

```
let last = arr => [...arr].pop();

let values = [1, 2, 3]
last(values); // 3 - values still [1, 2, 3]
```

You can also use this to add items to the end of a copy of an array:

```
let odd = [3, 5, 7, 9];

// add a few items on the end
let odder = [...odd, 11, 13]; // [3, 5, 7, 9, 11, 13]
```

9.8 Additional Resources

- [Eloquent JavaScript: Objects and Arrays](#)
- [MDN: Arrays](#)
- [What You Should Know About JavaScript Arrays](#)

Chapter 10

Array Iterator Methods

Because we almost always want to loop over an array, JS has some built in functions for looping over arrays in commonly used ways. This saves us from writing the boilerplate for a `for` loop every time.

We're going to look at:

- `forEach`
- `find`
- `filter`
- `some`
- `every`
- `map`
- `reduce`

These can (and *should*) be used in place of a `for` loop.

All of the iterator methods *accept a function as their first argument*. Remember, functions are values just like any other type in JavaScript, so we can pass them as arguments to functions too.¹

So far you've always called the function yourself, but the iterator methods *call the function for you*. This can take a bit of getting used to, but - once you get the hang of it - you'll write much more elegant array iterating code.

¹Functions that accept or return other functions are called [Higher-Order Functions](#).

Performance vs Elegance

There are some posts on the internet that will tell you to always use a **for** loop as it is much more performant than using the iterator methods. While it is true that the iterator methods are not quite as quick as using a **for** loop, that's no reason not to use them.

Firstly, underlying performance can be optimised: how different bits of code perform is always being worked on by the people working on the JavaScript engines in browsers. Secondly, they're both incredibly quick: if your app is having performance issues then there are probably much bigger issues to sort out than using the array iterator methods. And lastly, your main aim when writing code should be to make it maintainable: that means easy to understand and make changes to. The iterator methods are much tidier than using a **for** loop.

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of non-critical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies: premature optimization is the root of all evil.

- Donald Knuth

10.1 **forEach**

forEach goes over each item in the array. You can always use it instead of a **for** loop for iterating arrays.

```
let numbers = [1, 2, 3, 4, 5, 6];

numbers.forEach(val => console.log(val));
numbers.forEach((val, index) => console.log(index));
```

This is tidier than using a **for** loop and is the preferred method. Although, more often than not, **filter**, **map**, and **reduce** can be used to create more succinct code.

10.2 find

`find` goes over every item in an array until it finds the first value that returns `true` when passed into the given function², it then returns the value it found.

```
let words = ["fish", "cow", "monkey"];  
  
// each item in the array is passed into the function  
// returns the first value in the array for which  
// the given function returns true  
let result = words.find(word => word.length === 3); // "cow"
```

10.3 filter

`filter` can be used to remove items from an array:

```
let numbers = [1, 2, 3, 4, 5, 6];  
  
// iterates over an array passing each value into the supplied function  
// removes any items that the function returns false for  
let evenNumbers = numbers.filter(val => val % 2 === 0);  
  
console.log(evenNumbers); // [2, 4, 6]
```

Iterates over an array and returns a new array containing the **same number or fewer** items.

If the given function takes two arguments the second one will be the current index.

Some & Every

There are also `some` and `every` methods which work like `filter`. `some` returns `true` if any of the test functions return `true`. `every` returns `true` if all of the test functions return `true`.

²A function that returns `true` or `false` for a given value is sometimes called a **predicate**.

10.4 map

map transforms each value in array to another value:

```
let numbers = [1, 2, 3, 4, 5, 6];

// iterates over an array passing each item into the supplied
// function transforms the value using the supplied function
let squares = numbers.map(val => val * val);

console.log(squares); // [1, 4, 9, 16, 25, 36]
```

Iterates over an array and returns a new array containing the **same** number of items.

If the given function takes two arguments the second one will be the current index.

10.5 reduce

map and filter always return an array. However, it's often useful to turn an array into some other value type.

reduce turns an array into a single value:

```
let numbers = [1, 2, 3, 4, 5, 6];

// iterates over the array, passing in previous return value and new
// value - the final value is the return value from the final iteration
// the value of total the first time it runs is 0, reduce's second
// argument
let sum = numbers.reduce((total, val) => total + val, 0);

console.log(sum); // 21
```

Iterates over an array and returns some value (number, string, boolean, array, object, function, etc.).

Unlike the other iterator methods, reduce takes a second argument. This represents the **initial value**: the value that the first argument passed to the function will

take on the first iteration.

If the given function takes *three* arguments the *third* one will be the current index.

Make sure you use both of the first two arguments inside the function you pass to `reduce`, otherwise you don't need a reduce.

10.5.1 Default Value

If you do not pass in an initial value as the second argument to `reduce`, JavaScript will automatically use the *first* value in the array. It will also skip the first item in the array when iterating, as if doing a `for` loop that starts at index `1`.

You should be careful relying on the default value: if the array happens to be empty you will get a JavaScript error, which will stop the rest of your code from running. There are a few occasions where it can be slightly more efficient to not provide the default value, but it's a minor performance improvement for a potentially broken app.

10.6 Which Iterator Method?

Firstly, you can only use iterator methods on arrays, so if you're not working with an array, you can't use the iterator methods (although, if you're sneaky, you might be able to turn the problem into one involving arrays).

- Do you need to filter out some values? Use `filter`
- Do you need to transform each value? Use `map`
- Do you need to turn the array into some other value? Use `reduce`
- Do you need to turn the array into another array, but which isn't just filtered or mapped? Use `reduce`
- Do you just need to run some code, but you're not interested in getting back a result? Use `forEach`

You shouldn't ever need to use a `for` loop for working with arrays.

10.7 Functional Programming

The beauty of the iterator methods is that we can use functions that we've already written:

```
let numbers = [1, 2, 3, 4, 5, 6];  
  
let sum = numbers.reduce((total, val) => total + val, 0);
```

The function we've passed into sum is just a function that takes two arguments and adds them together - which is what `add` did:

```
let numbers = [1, 2, 3, 4, 5, 6];  
let add = (a, b) => a + b;  
  
let sum = numbers.reduce(add, 0);
```

So rather than passing in anonymous functions, we can *reuse functions we've already written*. This is one of the key ideas behind **functional programming**.

Range

We can only use the iterator methods if we have an array to work with, which won't always be the case. For example, say you just want to do something 100 times: unless you have an array with 100 things in lying around you'll still need to use a **for** loop.

For the **for** loop averse, there is a way to turn *any* looping problem into an array iterator method:

```
let range = (start, end, increment) => {
  let arr = [];

  // allows us to ignore the increment argument
  increment = increment ? increment : 1;

  for (let i = start; i <= end; i += increment) {
    arr.push(i);
  }

  return arr;
};
```

Now we can use the **range()** function to create an array of arbitrary length:

```
// get all the numbers between 1 and 100 that are divisible by 5 or
// 9
let div5or9 = range(1, 100).filter(i => i % 5 === 0 || i % 9 ===
0);

// all the numbers between 3 and 100 that are divisible by 3
let div3 = range(3, 100, 3);
```

A function like **range** is included in many popular JS libraries - so in the real world you wouldn't need to write it yourself.

10.8 Additional Resources

- JavaScript's Map, Reduce, and Filter
- JavaScript Functional Programming: Map, Filter, and Reduce
- An Illustrated (and Musical) Guide to Map, Reduce, and Filter Array Methods
- How to create range in JavaScript
- MDN: Filter
- MDN: Map
- Reduce
 - MDN: Reduce
 - How to use array reduce for more than just numbers
 - Five ways to calculate an average with array reduce

Chapter 11

Objects

Objects are *unordered* collections of values. We use objects to collect together a set of related values. Objects have **keys** (names) and **properties** (values).

Properties can be any type of value (numbers, strings, booleans, functions, arrays, and other objects). We tend not to iterate over the items in an object¹, so it's fine for different properties to store different types of values.

```
let empty = {} // an empty object

let address = { // an object representing an address
    number: 54, // assign the value 54 to the property "number"
    road: "Park Street",
    postcode: "BS5 9LD",
};

// an object representing a person
let person = {
    name: "Kofi",
    address: address, // using the address object from above
    favouriteColours: ["purple", "green"],
};
```

When we write objects out like this, we call them **object literals**. Later on we'll look at classes, which give us a different way of creating objects.

¹Although we'll look at **maps** later, which are an exception to this

11.1 Reading Properties

We use **dot-notation** to read the values of properties:

```
console.log(address.number); // 54  
console.log(person.name); // "Kofi"
```

You can also use array-style notation (although dot-notation is preferred):

```
console.log(address["number"]); // 54 - using array style notation
```

11.2 Writing Properties

You can change the value of a property:

```
address.number = 82;  
address.road = "Bath Road";
```

You can also use square bracket notation. This can be useful if you have the property name stored in a variable:

```
let property = "road";  
  
// elsewhere  
address[property] = "Bristol Road";
```

You can also add new properties to an object after it's been created:

```
// creates a new property, "country", and gives it "UK" as a value  
address.country = "UK";
```

Nested Properties

If we wanted to access the `road` property of the `address` property of the `person` object we can do this by extending the dot-notation:

```
console.log(person.address.road); // "Park Street"  
person.address.road = "Bath Road";  
console.log(person.address.road); // "Bath Road"
```

You can do this to any level in the hierarchy: just make sure you include *every* level.

11.3 Methods

Objects can have functions as property values. We normally refer to these as **methods**. We can write them with `=>` if we want, but there is an object-specific way of writing functions:

```
let basicMaths = {  
    pi: 3.141592654,  
  
    // object-style method (function)  
    add(a, b) {  
        return a + b;  
    },  
  
    // fat-arrow style method  
    minus: (a, b) => a - b  
};  
  
basicMaths.add(1, 2); // 3  
basicMaths.minus(1, 2); // -1  
basicMaths.pi; // 3.141592654
```

When you used `arr.sort()`, `str.split(" ")`, etc. you were calling methods of strings and arrays.

Methods are still properties, so you can't have a method that has the same name as another property (e.g. you can't have a `.pi()` method and a `.pi` property)

11.4 this

Objects can refer to themselves using the `this` keyword:

```
let tori = {
  name: "Tori",
  birthdate: new Date("1984-11-22"),

  getAge() {
    let now = new Date();
    let millisecondsPerYear = 31556952000;

    // this.birthdate is the birthdate property above
    let years = (now - this.birthdate) / millisecondsPerYear;

    return Math.floor(years);
  }
};

tori.getAge(); // 34
```

In the example above, it looks like we could have used `tori` instead of `this`. But objects aren't always assigned to named variables and what's stored in a specific variable can change, whereas the value of `this` will always point to the object it belongs to.²

Because methods have access to an object's properties via `this` it is more common for them to not take any arguments. In the above example we don't need to pass anything to the `getAge()` method because all of the relevant data are stored as properties of the object.

²That's not strictly true, `this` is a bit broken in JavaScript. But we'll get to that in the React week

11.5 Additional Resources

- Eloquent JavaScript: Objects and Arrays
- MDN: Objects
- MDN: Object Basics
- The Chronicles of JavaScript Objects
- What Is `this`? The Inner Workings of JavaScript Objects

Chapter 12

The Standard Library

12.1 Strings

Under the hood strings are actually objects. That means that strings have various properties and methods:

```
let str = "A String";  
  
// properties  
str.length; // 8  
  
// methods  
str.charAt(2); // "S"  
str.substring(2, 5); // "Str" - start at index 2, end before index 5  
str.toLowerCase(); // "a string"  
str.toUpperCase(); // "A STRING"
```

A full list of properties can be found on [MDN](#).

12.2 Date

The built-in `Date` object allows you to manipulate dates.

```
let now = new Date(); // a date object representing now

// a date representing 5:08 am on 24th August 2018
let birthdate = new Date("2018-08-24T05:08:00");

birthdate.getFullYear(); // 2018
birthdate.getDate(); // 24
birthdate.getDay(); // 5 (0 - 6, where 0 is Sunday and 6 is Saturday)
birthdate.getMonth(); // 7 (0 - 11, where 0 = January - c'est
→ stupide!)
birthdate.getTime(); // 1535083680000
```

Generally it's easier to use a library like `moment.js`.

The Beginning of Time

You might be wondering what the `1535083680000` value from `.getTime()` represents.

It is, *of course*, the number of milliseconds between the given date and 00:00 GMT on the 1st of January 1970.

For an entertaining look at how computers handle dates, check out [UTC is enough for everyone, right?](#)

12.3 Math

The **Math** object lets you do more complex mathematical operations:

```
// Useful properties
Math.PI; // 3.141592653589793
Math.E; // 2.718281828459045

// Rounding
Math.floor(3.45); // 3
Math.ceil(3.45); // 4
Math.round(3.45); // 3

// Exponents
Math.sqrt(4); // 2
Math.pow(2, 3); // 8

// Other mathematical functions
Math.log(6); // 1.791759469228055
Math.cos(45); // 0.5253219888177297
```

Putting the Java in JavaScript

The **Date** and **Math** objects both feel decidedly un-JavaScripty. That's because they were taken directly from Java.

JavaScript was originally going to be called "Mocha" and then "LiveScript". But Netscape, the creators of JavaScript, settled on "JavaScript" after making a deal with Sun, the creators of Java. Sun agreed to give Netscape some money as long as they called their new language "JavaScript" to make it sound like a toy version of Java. Sun also insisted that JavaScript include the **Date** and **Math** objects from Java - despite the fact that Java and JavaScript have barely anything in common other than that they both ran in a browser.

Ironically JavaScript went on to eclipse Java as the language of choice for apps that would run on any system.

This also means that Oracle, who bought Sun in 2010, own the trademark on the name "JavaScript". That's why it's generally referred to as ECMAScript (after the European Computer Manufacturers Association, who control the standard) in any technical documentation.

You can see the origins of the insane **getMonth()** method on the [Java documentation](#)

12.4 Additional Resources

- [MDN: Math](#)
- [MDN: Date](#)
- [The JavaScript Date Object](#)
- [Everything You Need to Know About Date in JavaScript](#)
- [Handling Time Zone in JavaScript](#)

Chapter 13

Advanced Object Techniques

13.1 Destructuring

Destructuring allows us to pull property values out of an object and into variables of the same name as the original property:

```
let person = {
  firstName: "Mark",
  lastName: "Wales",
  age: 67,
};

// creates the firstName and lastName variables
// from those properties of person
let { firstName, lastName } = person;
```

When passing an object to a function we can use destructuring syntax in the function parameters, which can make the function a bit tidier:

```
// destructure in the parameters
let fullName = ({ firstName, lastName }) => firstName + " " + lastName;

fullName(person); // "Mark Wales"
```

Without:

```
let fullName = ob => ob.firstName + " " + ob.lastName;
```

We need to be careful if we try to use destructuring on two objects with the same property names at the same time, as they would need to use the same variable name:

```
// creates the firstName and lastName variables from
// those properties of person
let { firstName, lastName } = person1;

// won't work, because firstName and lastName already taken
let { firstName, lastName } = person2;
```

It is possible to avoid this issue by assigning new variable names during the de-structuring:

```
let { firstName: p2FirstName, lastName: p2LastName } = person2;

console.log(p2FirstName);
console.log(p2LastName);
```

13.2 The Spread Operator

The spread operator with objects is similar to the array spread operator. It lets us create a copy of an object:

```
let person = {
  name: "Sandy",
  age: 54,
};

// creates a copy of person
let copied = { ...person };
```

We can also change a property whilst copying:

```
let person = {  
    name: "Sandy",  
    age: 54,  
};  
  
// creates a copy of person and changes the age property  
let copied = { ...person, age: 55 };
```

And we can merge two objects together:

```
let personProps = {  
    name: "Sandy",  
    age: 54,  
};  
  
let otherPersonProps = {  
    name: "Noel",  
    favouriteColour: "orange",  
};  
  
// merges two objects  
// the second object overwrites any matching properties of the first  
let merged = { ...personProps, ...otherPersonProps };  
// gives us: { name: "Noel", age: 54, favouriteColour: "orange"}
```

As with arrays, this can be useful if we need to make sure we're not changing the original object.

This can get quite sophisticated if you have a complex hierarchy of objects (and arrays):

```
let data = {
  name: "Heena",
  age: 43,
  address: {
    street: "12 Flamingo Road",
    town: "Newcastle",
  }
};

// to update the town property...
let newData = {
  ...data, // first copy all of the properties of data
  // we need to update the address object too
  address: {
    ...data.address, // copy across all the address properties
    town: "Liverpool", // update the value we want to change
  }
}
```

13.3 Keys and Values

It is sometimes useful to get just the keys or values of an object. To do this we can use the `Object.keys()` and `Object.values()` functions:

```
let person = {
  firstName: "Mark",
  lastName: "Wales",
};

let keys = Object.keys(person);
let values = Object.values(person);

console.log(keys); // ["firstName", "lastName"]
console.log(values); // ["Mark", "Wales"]

keys.forEach(key => console.log(person[key]));
```

These are useful when you treat an object more like an array: as a collection of the same sorts of thing.

This can be useful as it allows us to immediately access an item based on its key without having to go over every item in the structure:

```
let people = {  
    345: {  
        id: 345,  
        name: "Ta-Nehisi",  
        age: 43,  
    },  
    789: {  
        id: 789,  
        name: "Reni",  
        age: 29,  
    }  
};
```

When objects are used this way we call them **maps**¹.

¹Modern JavaScript actually has a **Map** type built in, but it's not used much

13.4 Map, Filter, & Reduce

If you have an array of objects you can do all sorts of data manipulation using `map`, `filter`, and `reduce`.

```
let people = [{  
    id: 345,  
    name: "Ta-Nehisi",  
    age: 43,  
}, {  
    id: 789,  
    name: "Reni",  
    age: 29,  
}];  
  
// find all the people age 30 or over  
let over30s = people.filter(person => person.age >= 30);  
  
// return an array of just the names  
let names = people.map(person => person.name);  
  
// return the oldest person  
let oldest = people.reduce((oldest, person) => (  
    person.age > oldest.age ? person : oldest  
) , people[0]);
```

We could even use destructuring for the `filter` and `map`:

```
let over30s = people.filter(({age}) => age >= 30);  
let names = people.map(({name}) => name);
```

We can't use destructuring for the `reduce` as we need to return the full object.

13.5 Additional Resources

- MDN: Object Destructuring
- ES6: In Depth Destructuring

Chapter 14

Classes

We can also create abstract concepts (**classes**) of objects:

```
class Person {
    constructor(name, dob) {
        this.name = name;
        this.dob = dob;
    }

    getAge() {
        let now = new Date();
        let millisecondsPerYear = 31556952000;
        let years = (now - this.dob) / millisecondsPerYear;

        return Math.floor(years);
    }
}
```

This allows us to create multiple object **instances** with the same structure, without having to rewrite the same object literal each time.

Each instance has its own set of properties and internally **this** refers to the specific instance it belongs to:

```
// we use the "new" keyword to create an "instance" of Person
let kye = new Person("Kye", new Date("1965-03-19"));
let jane = new Person("Jane", new Date("1973-11-27"));
```

```
// we get back two separate ages, as they are different instances
console.log(kye.getAge(), jane.getAge());
```

Note that we give classes names with capital letters: `Person` rather than `person`. You don't have to, but it makes it much more obvious what your code is doing.

14.1 An Example

A book class:

```
class Book {
    constructor(title, author) {
        this.title = title;
        this.author = author;
        this.price = null;
    }

    setPrice(value) {
        this.price = value;
        return this;
    }

    getPrice() {
        if (this.price === null) {
            return "Unknown";
        }

        return "£" + this.price.toFixed(2);
    }
}

let book = new Book("Lord of the Rings", "JRRRRR Tolkien");
console.log(book.getPrice()); // "Unknown"

book.setPrice(9.9);
console.log(book.getPrice()); // "£9.90"
```

It's considered good practice to write **getter** and **setter** methods for reading and writing properties when using classes.

In the example above if we just did `book.price` we'd get back `null` some of the time. Whereas if we write a **getter** function we can guarantee that it always comes back with a useful value that we can show to a user.

Returning `this`

We often return `this` from **setter** methods: i.e. methods that accept values but don't have an obvious return value.

```
class Book {  
    // ...  
  
    setPrice(value) {  
        this.price = value;  
  
        // return this from setPrice  
        return this;  
    }  
  
    // ...  
}
```

This allows us to **chain** together methods on an object:

```
// because .setPrice returns the book object  
// we can then run .getPrice on it  
book.setPrice(99).getPrice();
```

Note: it wouldn't make sense to return `this` from **getter** methods, as the whole point of a getter method is to return a specific value

14.2 Additional Resources

- [MDN: Classes](#)
- [ES5 Getters and Setters](#) - an alternative way to do getters/setters
- [Public and private class fields](#)

Glossary

- **Array**: a collection of items stored in a specific order, accessed via their position (index)
- **Boolean**: a value that is either true (1) or false (0)
- **Class**: an abstract representation of a specific object structure which can be used to create object instances
- **Conditional**: a piece of code that will do different things depending on the value of a truth condition
- **Destructuring**: a way to store the value of specified object properties into variables
- **Expression**: a piece of code that is equal to some value - can be stored in a variable
- **Function**: a piece of code can be run zero or more times by calling it, often passing arguments and usually returning a value
- **Index**: the position of an item in an array, starting at 0 in most programming languages
- **Infinite Loop**: a loop with a looping condition that is always true so it never stops running
- **Iterator Methods**: array methods that run for each value in the array
- **Loop**: a piece of code that runs zero or more times depending on the value of a looping condition
- **Object**: a collection of items stored using named keys
- **Property**: a value stored on an object
- **Scope**: where a value is visible within code
- **String**: a sequence of characters

- **Turing Complete:** a programming language that can approximately simulate the computational aspects of any other programming language¹
- **Type:** the different sorts of things that a programming language understands (e.g. numbers, characters, lists)
- **Variable:** a way to store a value using a name

¹https://en.wikipedia.org/wiki/Turing_completeness

Additional Resources

Practice

- [A List of Practical Projects Anyone Can Solve in Any Programming Language](#)
Some more programming challenges to try
- [codewars](#)
Great place to practice your programming skills
- [exercism](#)
Code practice with online mentorship

Programming

- [Advice to new Programmers](#)
Some advice from a lots of experienced developers
- [Things I Don't Know as of 2018](#)
One of the core React developers lists all the things he *doesn't* know
- [A Brief, Incomplete, and Mostly Wrong History of Programming Languages](#)
This gets funnier every year as you learn more about programming
- [7 absolute truths I unlearned as junior developer](#)
- [It Runs on Java 8](#)

Computer Science

Computer science education cannot make anybody an expert programmer any more than studying brushes and pigment can make somebody an expert painter

- Eric S. Raymond

But, if you're interested in how things work:

- **Code: The Hidden Language of Computer Hardware and Software**
This best introduction to how computers work from the ground up
- **Nand2Tetris**
Build your own virtual computer from the most basic logic gates
- **Unravelling the JPEG**
Fascinating interactive look at how JPEG images work
- **Turing Completeness**
A key theorem in Computer Science that's fascinating and also worth knowing about
- **The case of the 500-mile email**

Colophon

Created using T_EX

Fonts

- **Feijoa** by Klim Type Foundry
- **Avenir Next** by Adrian Frutiger, Akira Kobayashi & Akaki Razmadze
- **Fira Mono** by Carrois Apostrophe

Colour Palette

- **Solarized** by Ethan Shoonover

Written by Mark Wales

∞

July 12, 2019