JavaScript

# Introduction to JavaScript

## Lesson 1

### What is Code

Imagine you’re at home and sat at your kitchen table and you want to make some lunch because you’re hungry. What do you do?

```

Make lunch

```

Lunch isn’t much use unless you eat it though so we should add that to our instructions.

```

Make lunch

Eat lunch

```

Now you’re going to make and eat lunch, but what are you having for lunch?

```

Make beans on toast

Eat beans on toast

```

Are you going to be eating the beans cold?

```

Heat up beans

Eat beans on toast

```

But you didn’t put the beans on the toast?

```

Heat up beans

Put beans on toast

Eat beans on toast

```

Do you just have slices of toast lying around in your kitchen?

```

Toast bread

Heat up beans

Put beans on toast

Eat beans on toast

```

What we may have considered to be a pretty simple task, making some lunch, actually has a large number of steps that need to be followed for it to be completed.

Computers are stupid: they will do exactly what you tell them to do, even if what you tell them to do isn’t what you want them to do. They have to be guided and given specific instructions. They will not assume things and they will rarely interpret vague language correctly. Code is these instructions.

### How JavaScript Functions

A JavaScript program is a list of instructions or “statements”. These statements will be executed by the computer in the order the computer encounters them.

```

Let x = 1; // This will happen first.

Let y = x + 1; // Then this will happen.

Console.log(y); // Finally this will happen.

```

It’s important to note “the order the computer encounters them” is not necessarily the same as the order they’re written. The computer works from the top down but due to things like “loops”, “if statements”, “functions” and “asynchronous code” the computer will seem to jump about from line to line but it always does so according to very strict rules. We’ll be covering each of those items in future lessons so don’t worry if they don’t make any sense to you now, just remember the computer works from the top down.

Each statement must be ended with a semicolon (`;`) to let the computer know where a statement ends. Each line in a computer program can have multiple statements but this is rarely done as it can make the program difficult to interpret for a human.

```

Let x = 1;

Let y = x + 1;

Console.log(y);

```

Is the same as:

```

Let x = 1; Let y = x + 1; Console.log(y);

```

You may think the “best code” is the code that is the fastest or the most compact but in 99% of cases the best code is the code that’s easiest for a programmer to understand. Keep this in mind when writing your own code.

# Variables

## Lesson 1

### Declaration

In JavaScript we use variables to store information allowing us to use it later in the program. They make complex information and calculations easier to manipulate by giving them human-readable names.

Before a variable can be used it must be “declared”. You do this by using one of the three declaration keywords, specifically `const`, `let`, or `var`. For example:

```

Const daysInAYear;

Let myFavouriteFood;

Var dogsAreGood;

```

The differences between `const`, `let` and `var` will be covered in VARIABLES LESSON 2. Until you reach VARIABLES LESSON 2 you should use the `let` keyword to declare your variables.

The name we give to a variable is known as its “identifier”. An identifier must start with a letter and may contain letters, numbers, `$`, and `\_`. Identifiers are case sensitive, so `dogsAreGood` and `dogsaregood` will be considered two different variables.

Best practice for naming variables is to give them a name to represent their value and use “camelCase”. This is where you give each word in the identifier a capital letter except the first.

In general variables can only be declared once. If you declare a variable more than once you may receive an error message. This will be covered in more details in later lessons.

### Assignment

For a variable to be used it must be “assigned” a value. This is performed by using the equals sign (`=`) and is done as follows:

```

daysInAYear = 365;

myFavouriteFood = ‘sausages’;

dogsAreGood = true;

```

Variables can also be “reassigned” with a new value:

```

myFavouriteFood = ‘sausages’;

myFavouriteFood = ‘pizza’;

```

### Initialization

We can “assign” a variable at the same time we “declare” it. This is called “initialization”. It allows us to declare and assign a variable in a single line. For example:

```

Let peopleAtWork = 10;

```

Is equivalent to:

```

Let peopleAtWork;

peopleAtWork = 10;

```

## Lesson 2

“Use strict”

Special initializations (+= etc.)

Incrementations (++, --)

Difference between const, let and var. Principle of least privilege.

Multiple declarations and initializations

Destructuring

Scope

## Lesson 3

Identifier rules

# Comments and Console.log

## Lesson 1

### Comment

“Comments” are parts of our code that are not read by the computer. They are there for the people reading and interacting with our code.

Everything to the right of two forward slashes, `//`, in a line of code will be a comment.

```

// This line is a comment.

Let x = 6 // We can also have comments on the same line as other code.

```

Another way to insert comments into a file is with a “block comment”. Block comments are started with a forward slash and an asterisk, `/\*` and ended with an asterisk and a forward slash, `\*/`. Block comments can span multiple lines and are useful when wanting to write comments over multiple lines.

```

/\* A block comment can be on one line. \*/

/\* Or it can be

On multiple

Lines. \*/

Let x = / \*They can even be inside code, but that can be very confusing. \*/ 20

```

### Console Log

`console.log` is an important function for you to know while following these lessons. It allows us to print information to the “console” so we can monitor our programs.

```

Console.log(‘Hello’);

// The word ‘Hello’ will appear in the console.

```

You can also put a variable into a console log and it will log the value of that variable.

```

Let favouriteNumber = 36;

Console.log(favouriteNumber);

// The number ‘36’ will appear in the console.

```

Multiple values can be put within the parenthesis and separated by commas to log them all.

```

Let favouriteNumber = 36;

Let leastFavouriteNumber = 17;

Console.log(favouriteNumber, leastFavouriteNumber);

// The numbers ‘36’ and ‘17’ will appear in the console.

```

We can enter almost any kind of data into a console log and it will be coerced into a “string” for printing to the console. We will be covering data types in a future lesson so don’t worry too much about what that exactly means for now. Knowing the basic function of the console log command is important for following these lessons and for checking your own programs.

# Data Types

## Lesson 1

### Introduction

In JavaScript data is information that we want to use in our code. This data can be numbers, words, whether something is true or false or a more complex data type. Each data type has different properties and different uses.

Data types broadly fit into one of two categories: “primitive” or “object”.

### Primitive Data

There are currently seven primitive data types in JavaScript. These are:

* Number
  + These can be negative or positive and can be whole numbers or fractions. Examples are `13`, `-7`, `2.25` and `-1.01`.
* String
  + Strings contain text. They can contain letters, numbers, symbols, white space and special characters such as tabs or line breaks. Examples are `’hello’`, `’Oh no, René!?’`, `’23 bats’` and `’16’`
* Boolean
  + Booleans can be either `true` or `false`.
* Null
  + The null data type only has the value `null`.
* Undefined
  + The undefined data type only has the value `undefined`.
* BigInt
  + BigInts are whole numbers. They do not have any fractional data. They can be positive or negative. They can maintain accuracy when calculating very large integers. Examples are `10000000000000000n`, `-84512526689454125n`, `0n` and `128n`.
* Symbol
  + Symbols are values that are completely unique.

Primitive data types are “immutable”, this means they can not be changed. If we assigned a primitive data type to a variable there is no way to change the value of that variable without reassigning it. Don’t worry if this doesn’t make sense at the moment, we will be going over most of these data types in the coming lessons.

### Objects

Objects are a complex data type. They can contain values and other objects within them. They can even contain themselves as a “circular reference”.

There are many built in object types in JavaScript. These include:

* Array
* Date
* Error
* Function
* Math
* Object
* RegExp

Objects are “mutatable” meaning they can be changed. Take a look at the following example.

```

// First we declare a variable called “today” and assign an object that is the current date.

let today = new Date(Date.now())

// We log the value of the variable “today” after it’s been converted to a string.

console.log(today.toString()); // "Thu Jul 04 2019 12:26:31 GMT+0100 (British Summer Time)"

// We call a “method” on the Date object called “setDate” that allows us to change the day field of the date to the 8th.

today.setDate(8);

// We log the value of the variable “today” after it’s been converted to a string.

console.log(today.toString()); // "Mon Jul 08 2019 12:26:31 GMT+0100 (British Summer Time)"

```

Notice that the value of `today` changed even though we never reassigned it. This is an important property of objects that we will be covering in more detail in a future lesson.

# Numbers

## Lesson 1

### Introduction

You create a number in JavaScript by typing the number. Use a dash, `-`, to mark the number as negative and a full stop, `.`, for the decimal point if you require one.

```

Let x = 12

Let y = -500

Let z = 6.625

```

You should not use any other characters in your number. Some common mistakes are using separators for thousands such as `1,000,000` or `1 000 000` and placing the number in quotation marks such as `’17’` or `”71”`.

### Operations

Addition is performed with the plus, `+`, symbol. This adds two numbers together.

```

Let x = 5 + 3;

Console.log(x); // Gives us 8

Let y = x + 10 + 2; // We can perform multiple operations on the same line.

Console.log(y); // Gives us 20

```

Subtraction is performed with the minus, `-`, symbol, also known as a dash. The number on the right will be subtracted from the number on the left.

```

Let x = 8 - 3;

Console.log(x); // Gives us 5

Let y = x - 4 - 2; // We can perform multiple operations on the same line.

Console.log(y); // Gives us -1

```

The minus symbol can also be used to “negate” the following number. This is known as “unary negation”. This will make a positive number negative or a negative number positive.

```

Let x = 5;

Console.log(- x); // Gives us -5

Let a = 10 - x;

Console.log(a); // Gives us 5

Let b = 10 + - x; // Adding a negative number is equivalent to subtracting it.

Console.log(b); // Gives us 5

Let c = 10 - - x; // Subtracting a negative number is equivalent to adding it.

Console.log(c); // Gives us 15

```

Multiplication is performed with the asterisk (`\*`) symbol. It multiplies the numbers to the left and right of itself.

```

Let x = 4 \* 5;

Console.log(x); // Gives us 20

Let y = x \* 0.5;

Console.log(y); // Gives us 10

Let z = y \* -3 \* 2;

Console.log(z); // Gives us -60

```

Division is performed using the forward slash (`/`) symbol. The number on the left is divided by the number on the right.

```

Let x = 6 / 2;

Console.log(x); // Gives us 3

Let y = 1 / x;

Console.log(y); // Gives us 0.333333…

Let z = 2 / y / 4;

Console.log(z); // Gives us 1.5

```

The modulo operation, also known as the remainder operation, is performed using the percent (`%`) symbol. It gives us the remainder after the number to the left is divided by the number on the right. For example 11 modulo 3 (`11 % 3`) gives us the value 2 as 11 will divide into 3 three times and you would have two left over. 11 = 3 \* 3 + 2. The modulo will always give a number that is greater than or equal to zero and less than the number to the right of the modulo.

```

Let x = 8 % 3;

Console.log(x); // Gives us 2.

Let y = 12 % 2;

Console.log(y); // Gives us 0. This is often used to check if a number is even.

Let z = 4.7 % 0.5;

Console.log(z); // Gives us 0.20000000000000018. Module can be used on fractions as well as integers.

/\* But the remainder for 4.7 divided by 0.5 should be 0.2. Why does JavaScript say it’s 0.2 and a tiny bit? This is one of JavaScripts little “quirks”. We will be covering these quirks in later lessons. \*/

```

Raising a number to a power, sometimes called exponentiation, is performed with two asterisk (`\*\*`) symbols. The number to the left, the “base”, will be raised to the power of the number on the right, the “exponent”. When a base is raised to an exponent it is multiplied by itself that many times. For example `2 \*\* 3` is equal to `2 \* 2 \* 2` and `10 \*\* 5` is equal to `10 \* 10 \* 10 \* 10 \* 10`. This is often depicted by writing the base and then having the exponent written in a smaller size to the top right, such as 2³ or 10². A number raised to a power of two is said to be “squared” and a number raised to the power three is said to be “cubed”. Names for exponents higher than this are not used.

```

Let x = 2 \*\* 3;

Console.log(x); Gives us 8. 2 x 2 x 2.

Let y = x \*\* 2;

Console.log(y); Gives us 64. 8 x 8.

Let z = 4 \*\* 3 \*\* 2;

Console.log(z); // Gives us 262144. 4 \*\* (3 )

Let a = 100 \*\* 0.5

Console.log(a); Gives us 10. This is the square root of 100.

/\* Fractions and negative number can be used as exponents but these are more complicated in their nature and will be tackled in their own lessons in my series on mathematics. \*/

```

When raising a base to a power you cannot use a unary negation on the base as the meaning is ambiguous. You must be clear on whether you want to negate the base or the entire exponentiation.

```

Let A = 2 \*\* 2; // Gives 4.

Let B = -2 \*\* 2; // Gives an error.

Let C = 0 – 2 \*\* 2; // Gives -4.

Let D = (-2) \*\* 2; // Gives 4.

Let E = -(2 \*\* 2); // Gives -4.

/\* The parenthesis (“()”) are used to force JavaScript to perform certain operations before other one. We will be covering this later in this lesson. \*/

```

### Order of Operations

Much like in classic mathematics there are rules that dictate the order we should perform operations in JavaScript has its own set of internal rules for performing operations. Without these rules operations would be ambiguous. For example `10 / 5 \* 2` could be interpreted as `10 / 10` to equal 1 or `2 \* 2` to equal four.

Each operations has a “precedence”, it’s priority or importance, and, if applicable, an “associativity”. Operations are performed in descending order of precedence and then by their associativity. Associativity will be either “left-to-right” or “right-to-left”. This is why `10 / 5 / 2` equals `2 / 2` and not `10 / 2.5` but `4 \*\* 3 \*\* 2` equals `4 \*\* 9` and not `64 \*\* 2`; division has left-to-right associativity and exponentiation has right-to-left associativity. All operations with the same precedence have the same associativity.

Let’s take a large, complicated expression and work through it to help us understand the order of operations.

```

Let a = 5 \* 16 / 4 + 2 \* (6 – 3) \*\* 2 \*\* 2 – 4 \*\* (4 – 5 + 3) / 8 % 5 + 9;

console.log(a); // Gives us 189.

```

First, we need to know each operation’s precedence and associativity. In this example we have:

* Assignment - `=`
* Multiplication - `\*`
* Division - `/`
* Addition - `+`
* Grouping - `()`
* Subtraction - `-`
* Modulo - `%`

Looking at the <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence> we get the following precedences and associativities:

* 20 – Grouping - `()`
* 15 - Exponentiation – Right-to-Left - `\*\*`
* 14 – Multiplication, Division, Modulo – Left-to-Right - `\*`, `/`, `%`
* 13 – Addition, Subtraction – Left-to-Right - `+`, `-`
* 3 – Assignment – Right-to-Left - `=`

“Grouping” with parenthesis (`()`) has a higher precedent than any other operation. Grouping is used to force the code to perform operation in a certain order. For example, if you want to add two numbers then divide it by two you could use parenthesis. `(a + b) / 2` will give you the correct answer when `a + b / 2` would not.

Let’s take out expression and start breaking it down by order of precedence.

```

a = 5 \* 16 / 4 + 2 \* (6 – 3) \*\* 2 \*\* 2 – 4 \*\* (4 – 5 + 3) / 8 % 5 + 9;

```

We will deal with the groups first as they always have the highest precedence. The first group contains only a single operation, `6 - 3`, so we can perform this without worrying about any precedence.

```

a = 5 \* 16 / 4 + 2 \* (3) \*\* 2 \*\* 2 – 4 \*\* (4 – 5 + 3) / 8 % 5 + 9;

```

The second group contains two operations, addition and subtraction. These have the same precedence, 13, so we perform the operations by their associativity which is left-to-right. `4 – 5 + 3` becomes `-1 + 3` and this finally becomes `2`.

```

a = 5 \* 16 / 4 + 2 \* (3) \*\* 2 \*\* 2 – (4) \*\* 2 / 8 % 5 + 9;

```

These groups now contain no operations and just contain a value. We can now remove the parenthesis as they have no further effect on the order of operations.

```

a = 5 \* 16 / 4 + 2 \* 3 \*\* 2 \*\* 2 – 4 \*\* 2 / 8 % 5 + 9;

```

The highest precedence operation present now is the exponentiation operations. These have right-to-left associativity so we will start from the right and calculate the exponentiations in order, starting with `4 \*\* 2` which gives `16`.

```

a = 5 \* 16 / 4 + 2 \* 3 \*\* 2 \*\* 2 – 16 / 8 % 5 + 9;

```

Next is the `2 \*\* 2` expression which equates to `4`.

```

a = 5 \* 16 / 4 + 2 \* 3 \*\* 4 – 16 / 8 % 5 + 9;

```

Finally we calculate `3 \*\* 4` which equates to `81`.

```

a = 5 \* 16 / 4 + 2 \* 81 – 16 / 8 % 5 + 9;

```

Next is multiplication, division and modulo which all have a precedence of 14 and an associativity of left-to-right. For each step I will show the current expression to be calculated by using parenthesis.

```

a = (5 \* 16) / 4 + 2 \* 81 – 16 / 8 % 5 + 9;

a = 80 / 4 + 2 \* 81 – 16 / 8 % 5 + 9;

a = (80 / 4) + 2 \* 81 – 16 / 8 % 5 + 9;

a = 20 + 2 \* 81 – 16 / 8 % 5 + 9;

a = 20 + (2 \* 81) – 16 / 8 % 5 + 9;

a = 20 + 162 – 16 / 8 % 5 + 9;

a = 20 + 162 – (16 / 8) % 5 + 9;

a = 20 + 162 – 2 % 5 + 9;

a = 20 + 162 – (2 % 5) + 9;

a = 20 + 162 – 2 + 9;

```

Next is addition and subtraction which have a precedence of 13 and an associativity of left-to-right.

```

a = (20 + 162) – 2 + 9;

a = 182 – 2 + 9;

a = (182 – 2) + 9;

a = 180 + 9;

a = 189;

```

Finally the value of `189` is assigned to the variable `a`. Assignments have one of the lowest precedences which means we almost never have to worry about them when considering the order of operations.

While it’s important to know about the order of operations it isn’t vital unless you are doing a lot of calculations. Even then the use of multiple variables and grouping calculations with parenthesis will make your calculations not only work how you want them to but also much clearer.

### Special Numbers

There are three special number values, these are `Infinity`, `-Infinity` and `NaN`. `Infinity` and `-Infinity` occur when a number is too large or too small to be represented by JavaScript. This occurs at around 10^300 from zero, that’s a 1 followed by three hundred zeros. `NaN`, which stand for “Not a Number” but, confusingly, is still considered a type of number by JavaScript, occurs when JavaScript tries to perform a numerical operation with something that is not a number. `NaN` usually indicates you’ve made a mistake somewhere in your calculations.

```

Const x = 10 \*\* 10 \*\* 10 \*\* 10; / This is a “googolplex”, a number much too large for JavaScript.

Console.log(x); // Gives Infinity.

Const y = -1 / 0;

Console.log(y); // Gives -Infinity. JavaScript will estimate any non-zero number divided by zero to be infinitely large, either positive or negative.

Const z = 5 / ‘Hello’;

Console.log(z); // Gives NaN.

```

# Practical Lesson: Converting Degrees

You have now learned enough to start making your own JavaScript functions. In this lesson I will write a function to convert degrees Celsius to degrees Fahrenheit. This function will:

1. Take a temperature in degrees Celsius.
2. Console log the given temperature in degrees Fahrenheit.

We will not be using the symbols °C or °F in this function, we will only be dealing with the numbers.

First we will declare a variable for storing the temperature in degrees Celsius.

```

Let temperatureFahrenheit;

```

Now we will assign that variable a temperature. You can choose any temperature you like.

```

Let temperatureFahrenheit;

temperatureFahrenheit = 14;

```

We will want a variable to store the new temperature so let’s declare that as well.

```

Let temperatureFahrenheit;

temperatureFahrenheit = 14;

let temperatureCelsius;

```

To calculate the temperature in Fahrenheit we take the temperature in Celsius, subtract 32, multiply by 5 and then divide by 9. We will assign the temperatureCelsius variable the result of this calculation.

```

Let temperatureFahrenheit;

temperatureFahrenheit = 14;

let temperatureCelsius;

temperatureCelsisus = temperatureFahrenheit – 32 \* 5 / 9;

```

Finally we will add a console log to print the temperature to the console.

```

Let temperatureFahrenheit;

temperatureFahrenheit = 14;

let temperatureCelsius;

temperatureCelsisus = temperatureFahrenheit – 32 \* 5 / 9;

console.log(temperatureCelsisus);

```

If we did the calculation correctly we should get the answer as -10 but we instead get -3.777…. Something has gone wrong.

If we look at the calculation and remember back to NUMBERS – LESSON 1 we realize that the reason this calculation doesn’t work is due to the order the operations calculate in. Multiplications and divisions happen before additions and subtractions. The calculation we are actually performing is.

```

temperatureCelsisus = temperatureFahrenheit – 32 \* 5 / 9;

temperatureCelsisus = 14 – 32 \* 5 / 9;

temperatureCelsisus = 14 – 160 / 9;

temperatureCelsisus = 14 – 17.777…;

temperatureCelsisus = -3.777…;

```

We need to make sure the subtraction of 32 happens before the multiplication and division. The easiest and cleanest way to do this is to add some parenthesis to our calculation.

```

temperatureCelsisus = (temperatureFahrenheit – 32) \* 5 / 9;

temperatureCelsisus = (14 – 32) \* 5 / 9;

temperatureCelsisus = -18 \* 5 / 9;

temperatureCelsisus = -90 / 9;

temperatureCelsisus = -10;

```

We are now getting the answer we expected to get. We will add this in to our program.

```

Let temperatureFahrenheit;

temperatureFahrenheit = 14;

let temperatureCelsius;

temperatureCelsisus = (temperatureFahrenheit – 32) \* 5 / 9;

console.log(temperatureCelsisus);

```

This now gives us the answer we expect.

## Refactoring

“Refactoring” is the process of improving your code without changing its functionality. This is to make the code less complex and more concise. It is important to refactor your code regularly. We will refactor the code we’ve just written.

```

Let temperatureFahrenheit;

temperatureFahrenheit = 14;

let temperatureCelsius;

temperatureCelsisus = (temperatureFahrenheit – 32) \* 5 / 9;

console.log(temperatureCelsisus);

```

Notice that we’re declaring the variable `temperatureFahrenheit` and immediately assigning it with a value. We can make this more concise by initializing the variable with the value straight away. We can do the same thing for the `temperatureCelsius` variable.

```

Let temperatureFahrenheit = 14;

let temperatureCelsius = (temperatureFahrenheit – 32) \* 5 / 9;

console.log(temperatureCelsisus);

```

The code is now less complex as it involves fewer steps but it is still clear what is happening in it. You may be tempted to the number of lines even further, to something such as this:

```

console.log((14 – 32) \* 5 / 9);

```

This will give exactly the same result as the previous block of code but it is very unclear what is happening or what the purpose of the code is. This is bad code.

Refactoring is a bit of an art form and each developer has their own preference for how to do it. We will be going over refactoring techniques and practices throughout these lessons and as we do you will develop your own preference and style.

# Semicolons and Whitespace

We said in an earlier lesson that every statement required a semicolon. This is technically true, but JavaScript will often put any missed semicolons in for you when it runs. Despite this I recommend manually placing semicolons after each statement as there can be situations where a manually inserted semicolon is necessary for the code to work as expected. It’s easier to insert manual semicolons than it is to learn where they can be left out and where they are required.

“Whitespace” is the name given to any text characters that appear blank such as spaces, tabs and “newline”, a special character that is inserted when you press the Enter key that represents the start of a new line of characters. This whitespace is used to separate things like keywords and variable names

# Strings

## Lesson 1

“Strings” in JavaScript are, at their most basic, words. A string is a list of characters that could be a single letter, a name, a sentence or even an entire novel.

To write a string in JavaScript it must start and end with either apostrophes (`’`) or quotation marks (`”`).

```

Let firstName = ‘Dan’;

Let surname = “Soup”;

Console.log(firstName, surname); // This will print “Dan”, “Soup”

```

Different people have different preferences for whether to use apostrophes or quotation marks for their strings in JavaScript.

# Functions and Methods Overview

## Why Learn Functions and Methods Now?

Many online tutorials will teach you how to use functions before teaching you what they actually are. When learning JavaScript myself I found I learned how these functions looked and how to use them rather then understanding them and learning how they worked. For these reasons I have decided to give you an overview of functions early on to help you understand what is happening when we use them in future lessons. If you don’t fully get your head around functions after this lesson don’t panic, we will be covering them in more detail in a later lesson.

## Functions

Functions are reusable blocks of code that can take “arguments” and “return” values. They let us write code that can be run multiple times with either the same or different conditions. Let’s look at an example.

```

Let countToThree = function () {

Console.log(1);

Console.log(2);

Console.log(3);

};

countToThree();

countToThree();

countToThree();

```

When this code is run we will see `1 2 3 1 2 3 1 2 3` printed to the console. Let’s break down what is happening in this code.

`let` is a declaration keyword that allows us to declare a variable. `countToThree` is the identifier of the variable we are declaring. `=` is assigning a value to `countToThree`. So far this all works exactly the same as what we have been doing previously; we’re initializing a variable with a value.

`function` is a keyword that tells JavaScript that what is following is a function. The parentheses, `()`, are there to contain any “parameters” that the function will use. In this case there are no parameters, so the parentheses are opened and immediately closed.

Next we have an opening “curly brace”, `{`, that indicates the start of a “code block”. A “code block” is a group of zero or more statements that JavaScript will interpret as a single statement. This is how we tell the function which statements to execute.

Within this code block we have three lines of code; three `console.log` statements that print the numbers 1 to 3. Then we have a closing curly brace, `}`, to signify the end of the code block.

Next we have the statement `countToThree()`. `countToThree` is the name of the variable we assigned the function to. The parentheses following it, `()`, indicate that we are “calling” that function. When a function is “called” the code within the function’s code block is executed, in this case the numbers `1`, `2` and `3` are printed to the console. We do this two more times so the function is called a total of three times, printing `1 2 3 1 2 3 1 2 3` to the console.

An important thing to note is that the code inside the function’s code block only runs when the function is called and it doesn’t run when the function is declared.

```

Let countToThree = function () {

Console.log(1);

Console.log(2);

Console.log(3);

};

```

In this case we have removed the `countToThree()` function calls from the code. If we run this program nothing will be printed to the console as the code block in the function containing the console logs will only be executed when the function is called.

## Functions with Parameters

“Parameters” allow us to alter the behaviour of a function when we call it. We define the parameters when we create the function. We will make a function that console logs a number that has been multiplies by two.

```

Let multiplyByTwo = function (numberToMultiply) {

Console.log(numberToMultiply \* 2);

};

multiplyByTwo(4); // This will log 8.

```

We declare a function `multiplyByTwo` and define the parameter `numberToMultiply` in the parenthesis. Just like naming a variable we can name this whatever we want but it is best practice to give it a name that makes sense. We then console log this parameter multiplied by two.

When we call the function, `multiplyByTwo(4)`, we provide an “argument”. In this case the argument is `4`. Now the function will execute with the parameter `numberToMultiply` now being equal to the argument provided, `4`. When the function performs `console.log(numberToMultiply \* 2)` what is actually being executed is `console.log(4 \* 2)` which gives us the `8` that we see in the console.

What we call the parameter isn’t important as long as it follows the rules of naming variables. We can name it whatever we want but it should be given a name that makes sense.

```

Let multiplyByTwo = function (aBigFluffyCat) {

Console.log(aBigFluffyCat \* 2);

};

multiplyByTwo(7.5); // This will log 15.

```

With the parameter now named `aBigFluffyCat` the function still works exactly the same.

Functions can be defined with any number of parameters. These parameters must be separated by commas. When you call the function you must add arguments for each parameter. These values will be mapped to the parameters in the order they are written.

```

Let findMidPoint = function (firstNumber, secondNumber) {

Console.log((firstNumber + secondNumber) / 2);

};

findMidPoint(4, 8); // This will log 6.

```

In this example the value `4` will be assigned to `firstNumber` and the value `8` will be assigned to `secondNumber` because that is the order they were written in.

In the following example the order we enter out arguments in affects the number that gets logged.

```

Let myDivideFunction = function (numerator, denominator) {

Console.log(numerator / denominator);

};

myDivideFunction(5, 10); // This will log 0.5.

myDivideFunction(10, 5); // This will log 2.

```

`5 / 10` is 0.5, but `10 / 5` is 2. For the majority of functions you must enter your arguments in the correct order or you will get unexpected results.

### Returning

Functions can “return” a value when they finish executing. This value can then be used in other parts of the program.

```

Let getNextNumber = function (firstNumber, secondNumber) {

Let difference = secondNumber – firstNumber;

Let nextNumber = secondNumber + difference;

};

getNextNumber(5, 7);

```

This function takes the difference between two number and adds that difference to the second number to find the next number in the sequence. For example, if we entered `5` and `7` as arguments the next value would be `9`, as 7 – 5 is 2 and 7 + 2 is 9.

If we execute this program we won’t see anything as there are no console logs to display the information to the terminal. Let’s add a console log and log the function call.

```

Let getNextNumber = function (firstNumber, secondNumber) {

Let difference = secondNumber – firstNumber;

Let nextNumber = secondNumber + difference;

};

Console.log(getNextNumber(5, 7)); // This will log “undefined”.

```

When a function is called the function call becomes the value that is “returned” from the function. As we are currently not returning anything from the function the function call has the value “undefined”. We will now add a “return” statement to the function.

```

Let getNextNumber = function (firstNumber, secondNumber) {

Let difference = secondNumber – firstNumber;

Let nextNumber = secondNumber + difference;

Return nextNumber;

};

Console.log(getNextNumber(5, 7)); // This will log 9.

```