

JavaScript Crash Course

Learn The Basics Of JavaScript

Author: Raheem Soomro

A large, bold, black 'Js' logo is centered on the page. The 'J' is a simple, thick vertical stroke with a curved bottom. The 's' is a thick, rounded, cursive-style letter. The entire logo is set against a solid yellow background.

Contents

Chapter 1 – Introduction to JavaScript	4
What is JavaScript?	4
Getting Started.....	5
Installing Extensions.....	6
How To Read JavaScript Code in HTML Files	7
Adding Comments To Your File.....	8
Chapter 2 - Basics of JavaScript	9
Console Window	9
Data Types.....	9
Variables	10
Combining Strings and Variables	11
Chapter 2 Quiz	12
Chapter 3 - JavaScript Operators	14
Arithmetic Operators	14
Comparison Operators.....	14
Logical Operators	15
Assignment Operators	16
Typeof Operator.....	17
Chapter 3 Quiz	18
Chapter 4 - Conditional Statements.....	20
Conditional Statements	20
If... else Statements.....	20
Ternary Operator	22
Adding Logical and Comparison Operators To Our Conditions	23
Switch Statements	24
Chapter 4 Quiz	26
Chapter 5 - Functions	29
What Are Functions?.....	29
Function Declaration.....	30
Return	33
Function Expressions	34
Scope.....	35

Hoisting	36
Chapter 5 Quiz	39
Chapter 6 - Arrays	42
What Are Arrays?	42
Accessing Elements in an Array	43
Replacing Elements in an Array	43
Using the Length Property	44
Useful Array Methods	44
Push Method	44
Pop Method	45
Shift Method	45
Unshift Method	45
IndexOf Method	46
Splice Method	46
Chapter 6 Quiz	48
Chapter 6 - Loops	50
What Are Loops?	50
Looping Through Arrays	52
While Loop	52
Chapter 6 Quiz	54
Chapter 7 - Objects	56
What Are Objects?	56
Accessing Properties of Objects	56
Assigning Properties	57
Adding Methods to Objects	58
Nested Objects	59
This Keyword	62
Chapter 7 - Quiz	64
Chapter 8 - ES6 Features	66
Let and Const	66
Template Literals	69
Default Parameters	70
Arrow Functions	71

Chapter 8 Quiz	73
Chapter 9 - Debugging	76
Error Messages	76
How To Debug – Example	78
Chapter 8 Quiz	89
Quiz Answers.....	92
Chapter 2.....	92
Chapter 3.....	92
Chapter 4.....	92
Chapter 5.....	93
Chapter 6.....	93
Chapter 7.....	94
Chapter 8.....	94
Chapter 9.....	94

Chapter 1 – Introduction to JavaScript

What is JavaScript?

JavaScript is a programming language that is commonly used on the internet. It allows you to implement several complicated features on websites. Whenever you see a website that has interactive features, contains animated graphics, slideshows and interactive forms, JavaScript was most likely used to build that website. JavaScript also has the ability to calculate and manipulate data on a website. JavaScript, along with HTML and CSS, form the basis of the internet and is an important component in web development. To break it down, these are the roles of HTML, CSS and JavaScript:

- HTML: Basic structure of any website such as text and images
- CSS: Allows you to control a websites design such as color schemes, fonts, background colors etc.
- JavaScript: Allows your website to be dynamic and adds interactive features

JavaScript was first introduced in 1995 and has since become one of the most important languages today. The rise of the internet has contributed to this and almost every mainstream browser has a built-in JavaScript engine that interprets and executes the JavaScript code on the user's device. It is actually amazing when you think about what you can achieve by writing a few simple lines of JavaScript code! JavaScript can also be used in several other settings, such as the development of mobile applications and creating games on websites.

JavaScript is primarily known as a client-side scripting language as opposed to a server-side language. What this means is that JavaScript code is processed on the user's device, as opposed to code that is processed on a web server and then downloaded onto the user's device

upon request. However, JavaScript can also be utilized as a server-side language using the Node.js environment. This book will only cover the client-side aspect of JavaScript, but once you are confident with coding, feel free to try out Node.js as well!

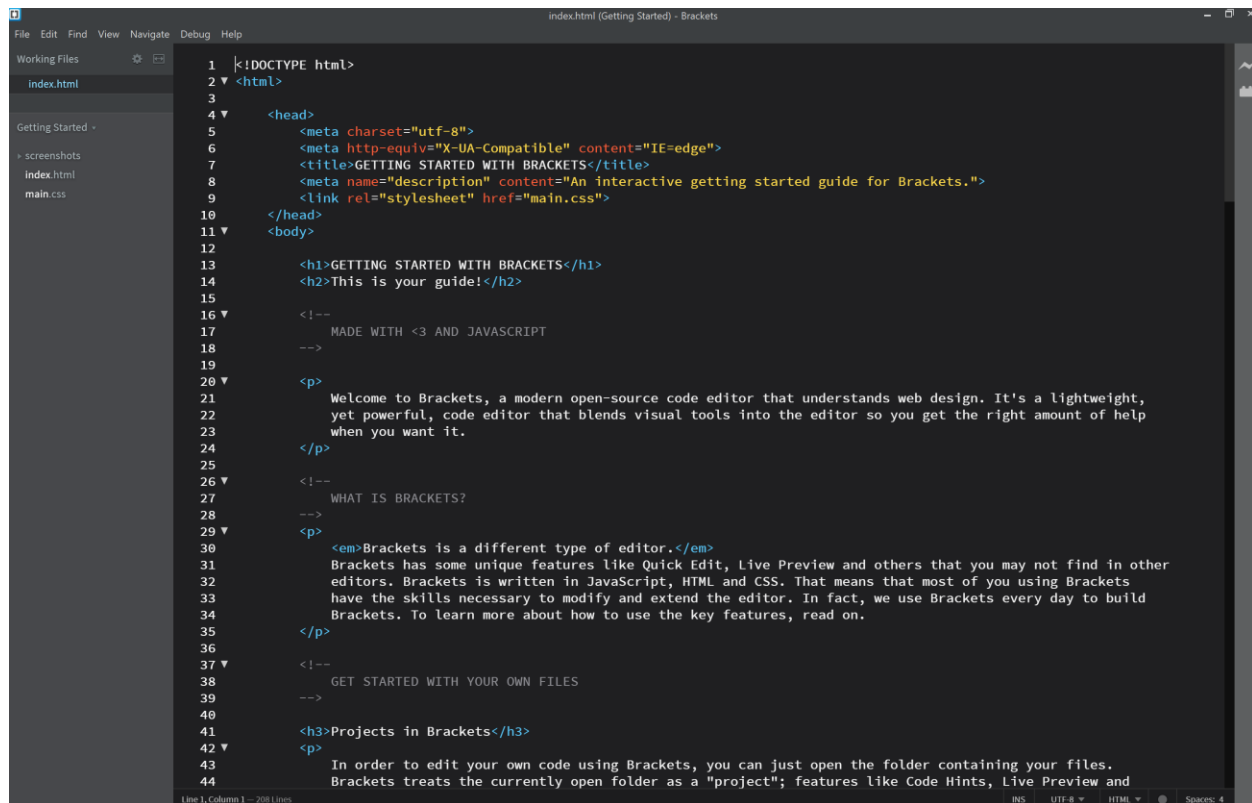
Getting Started

To get started on coding JavaScript, you will first need a program called a text editor. There are several text editors that you can use to code. Even the most basic text editor that comes with your operating system can be used to code! But as a beginner, it is recommended that you use something called an Integrated Development Environment or an IDE. IDE's contain a set of tools that include text editors, compilers and debugging features that makes coding much simpler for beginners. There are several IDE's that you can use to code JavaScript, including Visual Studio Code, Atom, Brackets and NotePad++. As a beginner, I highly recommend using Brackets because of its minimalist design and simplistic nature. Brackets can be downloaded from here:

<http://brackets.io/>

If you already have prior coding experience and are comfortable with another IDE, please feel free to use that! It is important in coding that you use whatever software you are comfortable with.

Once you have downloaded the Brackets application, simply follow the on-screen instructions to install your software. Once you have successfully installed it on your computer, the software should look something like this:



```
1 |<!DOCTYPE html>
2 |<html>
3 |
4 |<head>
5 |<meta charset="utf-8">
6 |<meta http-equiv="X-UA-Compatible" content="IE=edge">
7 |<title>GETTING STARTED WITH BRACKETS</title>
8 |<meta name="description" content="An interactive getting started guide for Brackets.">
9 |<link rel="stylesheet" href="main.css">
10|</head>
11|<body>
12|
13|<h1>GETTING STARTED WITH BRACKETS</h1>
14|<h2>This is your guide!</h2>
15|
16|<!--
17|    MADE WITH <3 AND JAVASCRIPT
18|-->
19|
20|<p>
21|    Welcome to Brackets, a modern open-source code editor that understands web design. It's a lightweight,
22|    yet powerful, code editor that blends visual tools into the editor so you get the right amount of help
23|    when you want it.
24|</p>
25|
26|<!--
27|    WHAT IS BRACKETS?
28|-->
29|
30|<p>
31|    <em>Brackets is a different type of editor.</em>
32|    Brackets has some unique features like Quick Edit, Live Preview and others that you may not find in other
33|    editors. Brackets is written in JavaScript, HTML and CSS. That means that most of you using Brackets
34|    have the skills necessary to modify and extend the editor. In fact, we use Brackets every day to build
35|    Brackets. To learn more about how to use the key features, read on.
36|</p>
37|
38|<!--
39|    GET STARTED WITH YOUR OWN FILES
40|-->
41|
42|<h3>Projects in Brackets</h3>
43|
44|<p>
45|    In order to edit your own code using Brackets, you can just open the folder containing your files.
46|    Brackets treats the currently open folder as a "project"; features like Code Hints, Live Preview and
```

Yours may look a little different to this, but that is because you can change themes to customize the look of Brackets.

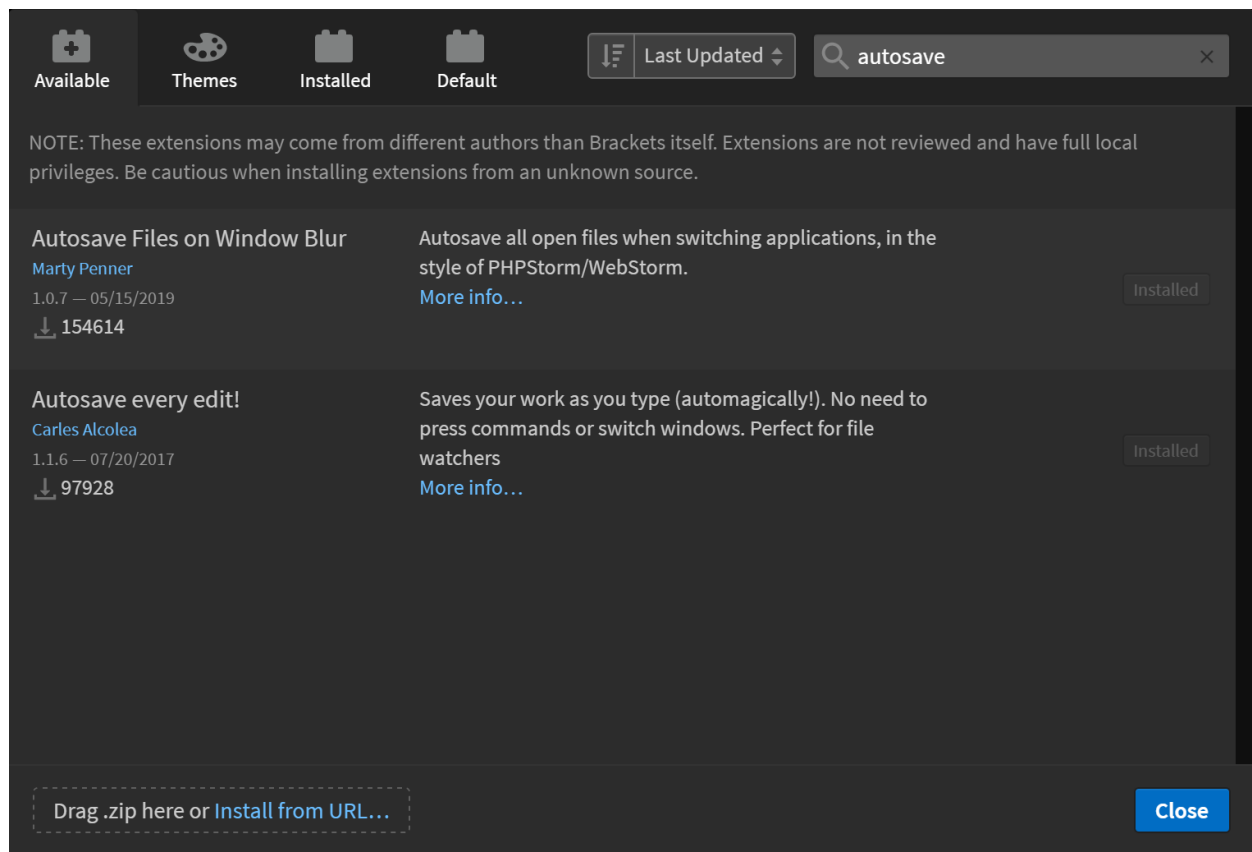
Installing Extensions

You can install extensions in your Brackets application. First, click on the extension manager button on the top right-hand section of the app:



One feature I highly recommend installing is the autosave feature so that you do not have to constantly save your code after making changes. On the extension manager window, type in autosave in the search bar and you should get two options. You can choose any of the autosave

extensions, but I normally use Autosave Files on Window Blur. Once you have installed the autosave extension, you are ready to start coding!



Opening A Folder in Brackets

Before we start coding, you will need to be able to open the folders containing the HTML and JavaScript files which can be downloaded from the course. To open a folder, click on the file button on the top left-hand corner of the application and select open folder. Navigate to the folder you would like to open, highlight it and select open folder. This will open the HTML and JavaScript files that you will need to write your code.

How To Read JavaScript Code in HTML Files

Now that you have your folder open, you will see two files within the folder. The first one will be an HTML file called index and the second will be a JavaScript file (ends in .js) with a name that corresponds to the

section of the course. In the index.html file, you will see a script tag with the filename of the JavaScript file to be used.

```
1 <!DOCTYPE html>|
2 ▼ <html lang="en">
3 ▼   <head>
4     <meta charset="UTF-8">
5     <title>Section 1 - Developers Console, Variables and Data Types</title>
6   </head>
7 ▼   <body>
8     <h1>Developers Console, Variables and Data Types</h1>
9   </body>
10   <script src="section1.js"></script>
11 </html>
```

In this example, the filename is section1.js. This means that any changes you make to the JavaScript file will be visible in this HTML file.

Adding Comments To Your File

Before we get started on some actual coding, there are a couple of things you need to know. You can add comments to your code by typing in two forward slashes (//). You can also add comments by typing in a forward slash and star (/*) and end the commented section by typing in a star and forward slash (*). Here is an example:

```
// This is a comment

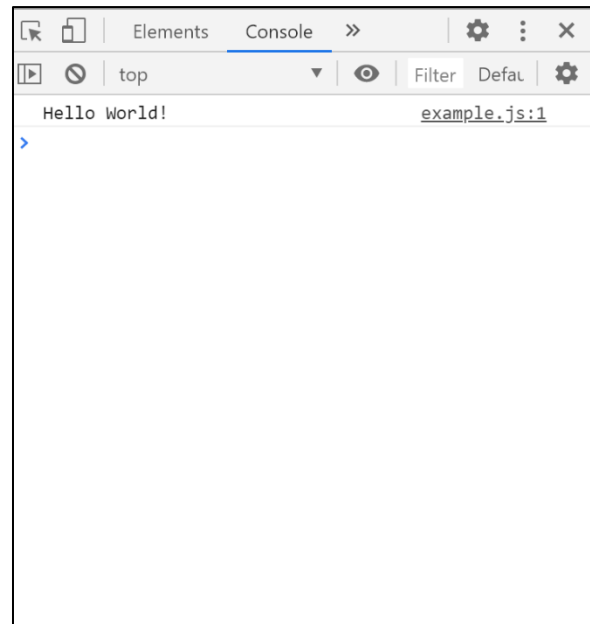
/* This is also a comment */
```

Comments are useful if you need to leave yourself or someone else a note on how a specific piece of codes work. Commenting out codes also stop a selected piece of code from executing, which can be useful when you are trying to find errors. To comment out a piece of code, simply highlight the code and press Ctrl+Shift+/ on a Windows PC and Cmd+Shift+/ on a Mac.

Chapter 2 - Basics of JavaScript

Console Window

The console is a fundamental part of coding in JavaScript, as it provides developers with various pieces of information. The console can display things like error messages to the developer, which would allow them to resolve any issues with the code. You can also log messages to the console by using the `console.log()` function. In this example, I have logged, 'Hello World!' to the console by typing:



```
console.log('Hello World!');
```

Most modern browsers such as Chrome have consoles. You can access the console in Chrome by right-clicking and selecting inspect. You can also access it by holding down Shift+Ctrl+I on Windows and Cmd+Shift+I on a Mac. Throughout your JavaScript coding life, you will use this console a lot, especially when you write more complex pieces of code. To the right of where it says hello world, you'll see some text that says example.js followed by a 1. Example.js is the name of the JavaScript file while 1 is the line number of the code that is being executed.

Data Types

There are 2 data types in JavaScript - primitives and objects. Primitives can be further divided into 6 classifications, which are:

1. Numbers: All numbers, such as 1, 16, 548, 54.71,
2. Booleans: true or false,

3. Strings: Strings are any characters, numbers, spaces etc. inserted between quotation marks, ' ' or " ",
4. Null: The absence of a value,
5. Undefined: The default assigned value for declared variables that have not been assigned a value,
6. Symbols: A feature that has been introduced in the newer versions of JavaScript (ES6). For the purposes of this course, you will not be using these

Objects are collections of related data and are very important in JavaScript. You will be learning more about objects later in the course.

Variables

In JavaScript, variables are like storages or containers that store values. Think of them as a folder or a box that stores important information. The folders or boxes themselves aren't important, but the information stored inside them is. Variables are important because they allow us to store values or pieces of code to make our code more readable and accessible.

There are three keywords we can use to create variables in JavaScript:

1. var
2. let
3. const

Prior to the ES6 version of JavaScript, var was the only way to declare variables. When you use var or let to declare variables, you can leave them undefined and then assign values to them later on. You can also reassign their values at any point. With const variables (short for constant) you must assign a value when they are declared. You also cannot reassign their values as they are constant.

In the example below, I have declared a variable called `name`, and assigned it the value `'Adam'`. If I now log this to the console using `console.log(name)`, Adam will be outputted to the console.

```
1 var name = 'Adam';  
2 console.log(name) // Output: Adam
```

To reassign the value of a variable that has been declared with `var` or `let`, simply type the name of the variable (in this case `name`) followed by the equal sign and then the new value.

```
1 var name = 'Adam';  
2 console.log(name) // Output: Adam  
3 name = 'Ben'  
4 console.log(name) // Output: Ben
```

As you can see, you don't even have to type the keyword `var` or `let` again, because this variable name has already been declared! Try to declare a variable with the `const` keyword, then try to reassign its value it and see what happens!

Combining Strings and Variables

You can combine strings and variables and log them to the console. For example, if you want to log a name and age and combine it with a sentence such as, `'My name is x and I am x years old'`, you can simply declare `name` and `age` in separate variables, and then join them together with strings using the `+` symbol. Here is an example:

```
1 var name = 'Adam';  
2 var age = 25  
3 console.log('My name is ' + name + ' and I am ' + age + ' years old.')  
4 // Output: My name is Adam and I am 25 years old.
```

Chapter 2 Quiz

- 1) How do you log a message to the console window?
 - a. `log.console()`;
 - b. `console.log()`;**
 - c. `consolelog()`;
 - d. `logconsole()`;
- 2) What are the two data types in JavaScript?
 - a. Primitives and objects**
 - b. Objects and numbers
 - c. Booleans and strings
 - d. Symbols and primitives
- 3) What are variables?
 - a. One of the data types in JavaScript
 - b. A display window that shows messages
 - c. The name of a text editor
 - d. A container that can store values
- 4) Which of the following methods is NOT a way to declare a variable?
 - a. `var = 'Hello World'`
 - b. `var message = 'Hello World'`
 - c. `let message = 'Hello World'`
 - d. `const message = 'Hello World'`

```
1  var greetings = 'Hello! My name is ';  
2  var name = 'Ben';  
3  console.log(greetings + name);
```

- 5) What will be the output of the code shown above?
 - a. 'Hello! My name is Ben'
 - b. Hello! My name is + Ben
 - c. Hello! My name is Ben

- d. 'Hello! My name is' + 'Ben'
- 6) The default assigned value for variables that have not been assigned a value is:
 - a. Null
 - b. String
 - c. Undefined
 - d. Boolean
- 7) What are the three keywords used to declare variables?
 - a. Var, let and const
 - b. Function, delete and const
 - c. String, number and Boolean
 - d. Boolean, symbol and null
- 8) It is possible to reassign variables declared with the let and var keywords
 - a. True
 - b. False

Chapter 3 - JavaScript Operators

Arithmetic Operators

Operator	Description
Addition (+)	Adds two operands together Example: $2 + 2 = 4$
Subtraction (-)	Subtracts the second operand from the first operand Example: $15 - 5 = 10$
Multiplication (*)	Multiplies two operands together Example: $10 * 10 = 100$
Division (/)	Divides the first operand by the second operand Example: $40 / 8 = 5$
Modulus (%)	Returns the remainder when dividing two integers Example: $12 \% 5 = 2$
Exponentiation (**)	Returns the result of the first operator to the power of the second operator Example: $2 ** 3 = 8$ ($2^3 = 8$) $3 ** 2 = 9$ ($3^2 = 9$)
Increment (++)	Adds one to the operand Example: $x++ = x + 1$
Decrement (--)	Subtracts one from the operand Example: $x-- = x - 1$

Comparison Operators

Operator	Description
Equal (==)	Assesses whether the value of the operands are equal Example: $a == a$ is true $a == 'a'$ is true $a == b$ is false

Not Equal (!=)	Assesses whether the value of the operands are NOT equal Example: a != b is true a != 'a' is false a != a is false
Strict Equal (===)	Assesses whether the value and types of two operands are equal and returns true if they are Example: a === a is true a === b is false a === 'a' is false
Strict Not Equal (!==)	Assesses whether the value and types of two operands are not equal and returns true if they are not Example: a !== b is true a !== 'a' is true a !== a is false
Greater Than (>)	Assesses whether the operand on the left is greater than the operand on the right and returns true if it is Example: 2 > 1 is true 10 > 20 is false
Less Than (<)	Assesses whether the operand on the left is less than the operand on the right and returns true if it is Example: 5 < 10 is true 15 < 3 is false
Greater Than or Equal To (>=)	Assesses whether the operand on the left is greater than or equal to than the operand on the right and returns true if it is Example: 10 >= 10 is true 10 >= 9 is true 10 >= 11 is false
Less Than or Equal To (<=)	Assesses whether the operand on the left is less than or equal to the operand on the right and returns true if it is Example: 15 <= 15 is true 15 <= 16 is true 15 <= 14 is false

Logical Operators

Operator	Description
AND (&&)	Assesses whether two expressions are non-zero, if yes then returns true Example: a = 1, b = 2 (a !== b) && (a < b) returns true (a !== b) && (a > b) returns false

	The second condition returns false because only the first expression is true as 1 is not greater than 2
OR ()	<p>Assesses whether one of two expressions are non-zero, if yes then returns true</p> <p>Example: a = 1, b = 2 (a === b) (a < b) returns true (a === b) (a > b) returns false</p> <p>The first condition returns true because at least one of the expressions is true</p>
NOT (!)	<p>Reverses the boolean value of the operand</p> <p>Example: var a = true !a = false</p>

Assignment Operators

Operator	Description
Assignment (=)	<p>Assigns the operand on the right to the value of the operand on the left</p> <p>Example: x = 5 (x is assigned the value of 5) x = a + b (x is assigned the value of a + b)</p>
Addition Assignment (+=)	<p>Adds the left and right operands and assigns the result to the left operand</p> <p>Example: a += b is the same as a = a + b</p>
Subtraction Assignment (-=)	<p>Subtracts the right operand from the left operand and assigns the result to the left operand</p> <p>Example: a -= b is the same as a = a - b</p>
Multiplication Assignment (*=)	<p>Multiplies the left and right operands and assigns the result to the left operand</p> <p>Example: a *= b is the same as a = a * b</p>

Division Assignment (/=)	Divides the left operand by the right operand and assigns the result to the left operand Example: <code>a /= b</code> is the same as <code>a = a / b</code>
Remainder Assignment (%=)	Divides the left operand by the right operand and assigns the remainder to the left operand Example: <code>a %= b</code> is the same as <code>a = a % b</code>

Typeof Operator

The `typeof` operator is used to determine the data type of the operand, such as a number, string, Boolean etc. For example, let's declare a variable, `test`, and assign it the value 'What type of operand is this?' as shown below.

```
1 var test = 'What type of operand is this?'
```

Now, let's check to see what type of operand this is. To do this, simply insert `typeof` before the variable name on the next line.

```
1 var test = 'What type of operand is this?'
2 typeof test
3 console.log(typeof test) // Output: string
```

In this example, the `typeof` was logged to the console to show you the value. You do not need to declare a variable to check the data type of an operand. Try to log to the console the `typeof` keyword, followed by any primitive and see the results.

Chapter 3 Quiz

- 1) What is the purpose of the addition (+) operator?
 - a. To multiply two operands
 - b. To add two operands
 - c. To assess whether both expressions are non-zero
 - d. To determine the data type of the operand
- 2) What is the purpose of the NOT (!) operator?
 - a. To set the value of the operator to 0
 - b. To assess whether an expression is true
 - c. To reverse the Boolean value of an operand
 - d. To determine the data type of an operand
- 3) Typing `x += 2` is the equivalent of typing:
 - a. `x = x + 2`
 - b. `x = x + x`
 - c. `x = x * 2`
 - d. `x = x - 2`
- 4) What will the following expression return?

```
1  var x = 25
2  var y = 50
3  (x > y || x !== y)
```

- a. True
 - b. False
- 5) What will the following expression return?

```
var x = 10;
var y = 11;
(x === y && x < y)
```

- a. True
 - b. False

6) What is the purpose of the increment (++) operator?

- a. Decrease the operand by 1
- b. Increase the operand by 2
- c. Increase the operand by 1
- d. Decrease the operand by 2

7) What will be the output of the following code?

```
console.log(15 % 2);
```

- a. 15
- b. 2
- c. 1
- d. 3

8) What is the correct way to write x is greater than or equal to y?

- a. $x > y$
- b. $x \geq y$
- c. $x \leq y$
- d. $x < y$

9) The strict equal operator (===) assesses the data type and value of the operands and returns true if they are the same

- a. True
- b. False

10) The not equal operand (!=) reverses the Boolean value of an operand

- a. True
- b. False

11) The decrement operator (--) reduces the operand by 2

- a. True
- b. False

Chapter 4 - Conditional Statements

Conditional Statements

Conditional statements are an important component in coding. Conditional statements are basically statements that checks whether a certain condition is met, then performs a specific task based on the condition. For example, if you're thirsty, drink some water, otherwise if you're hungry, go and eat something, otherwise don't eat or drink anything. There are two types of conditional statements in JavaScript: if... else statements and switch statements. We will be going over both of these in this section of the course.

If... else Statements

```
var thirsty = true;
if (thirsty === true) {
    console.log('Drink some water!');
}
//Output: Drink some water!
```

The screenshot above is an example of an if statement. First, a variable called thirsty was declared and set to true to show that we are thirsty. On the next line, a condition was set that if thirsty is true, log to the console 'Drink some water!'. Notice that we use the *if* keyword, followed by our condition in parentheses (). We then followed that up with a set of curly braces {}, which contains a block of code. If the condition inside the parentheses is met, then the block of code written inside the curly braces will execute. In this case, the condition was met, which is why our string was logged to the console, but what if we want to log something else if the condition is not met? That's where an else statement comes in. Here is an example:

```
var thirsty = true;
if (thirsty === true) {
    console.log('Drink some water!');
} else {
    console.log('Make sure you stay hydrated!')
}
//Output: Drink some water!
```

In the example above, an else statement was added to the line below the block of code in the if statement. So basically what this is saying now is, if you are thirsty, log to the console, 'Drink some water!', but if you are not, log to the console, 'Make sure you stay hydrated!'. So if the value assigned to the thirsty variable was false, the condition for the if statement is not met, and the block of code written inside the else statement will execute instead. Notice how the else keyword is written right after the curly brace and is then followed by another curly brace to signal the start of another block of code. A combination of an if statement and an else statement is known as an if... else statement. The example shown however, only works for true or false, or yes or no questions.

You can also have if... else statements which execute blocks of code for various conditions. Let's take a look at another example:

```
var season = 'spring';

if (season === 'summer') {
    console.log("It's hot outside!") {
} else if (season === 'winter') {
    console.log("It's cold outside!");
} else if (season === 'spring') {
    console.log('The weather is perfect!');
} else {
    console.log("I love this weather!");
}

// Output: The weather is perfect!
```

We can add additional conditions to our if... else statements by adding else if statements. In this example, we have added two additional conditions to our if... else statement. Since the variable season is set to 'spring', it does not satisfy the first two conditions, therefore the lines of code are not executed. However, the third condition is satisfied, so the string, 'The weather is perfect!' is logged to the console. You can add as many else if statements to your if... else statements as you want!

Ternary Operator

A ternary operator can also be used to shorten the if... else statement. In this example below, the if statement has been replaced with a question mark (?). Two expressions then follow the (?) which are separated by a colon (:). If the condition is true, then the first expression will be executed, else the second expression will execute. This is very useful for simple binary or yes/no conditions, as it makes the code look much cleaner.

```
var thirsty = true;
thirsty ? console.log('Drink some water!') :
console.log('Make sure you stay hydrated!');
// Output: Drink some water!
```

Adding Logical and Comparison Operators To Our Conditions

There will be occasions where we have to compare values and/or add more than one condition to each statement. Please go over the summary sheet from the previous section if you have forgotten the comparison and logical operators. Let's take a look at the example below. We have declared a variable called `temperatureCelcius` and assigned it the value 15. We then wrote our `if... else` statement with conditions that include logical and comparison operators. In the first `if` statement, we have assigned the condition: if the temperature is greater than or equal to 20 AND the temperature is less than or equal to 30, log to the console, 'This weather is perfect!'. In this case, both conditions **must be true** for the block of code to be executed. As the value we assigned to our variable does not meet both of our conditions, the next condition is checked. Since 15 is between 10 and 20, the condition is met and therefore, the block of code is executed.

```
var temperatureCelcius = 15;
if (temperatureCelcius >= 20 && temperatureCelcius <= 30) {
  console.log('This weather is perfect!');
} else if (temperatureCelcius >= 10 && temperatureCelcius < 20) {
  console.log("It's getting a bit chilly outside!");
} else if (temperatureCelcius < 10) {
  console.log("It's freezing outside!");
} else {
  console.log("It's so hot outside!");
}
// Output: It's getting a bit chilly outside!
```

We can also use the OR (`||`) operator in our conditions. In the example below, we have declared a variable called `timeOfDay` and assigned it the

value 'afternoon'. The condition we set for our if statement states that if the time of day is morning, OR the time of day is the afternoon, log to the console, 'I should go and get some exercise!'. In this case, because at least one of the conditions was met, the block of code under the if statement was executed. You will be using logical and comparison operators a lot when you code, so it is a good idea to get used to how each operator works. Play around with a few different operators and see what results you get.

```
var timeOfDay = 'afternoon';
if (timeOfDay === 'morning' || timeOfDay === 'afternoon') {
  console.log('I should go and get some exercise!');
} else {
  console.log('I think it might be time to go to bed');
}
// Output: I should go and get some exercise!
```

Switch Statements

Switch statements are the same as if... else statements, except they are written in a way that is easier to read. Switch statements are especially useful in cases where you have several else if statements. The example below shows how a switch statement is written:

```
var season = 'winter';
switch (season) {
  case 'summer':
    console.log("It's hot outside!");
    break;
  case 'winter':
    console.log("It's cold outside!");
    break;
  case 'spring':
    console.log("The weather is perfect!");
    break;
  default:
    console.log("I love this weather!");
    break;
}
// Output: It's cold outside!
```

So let's break down what is happening in the example. We have rewritten the example we previously used about the season and converted it into a switch statement. We first have to initiate the switch statement with the switch keyword and follow it with the value that we want to compare inside parentheses. The value we are comparing in this case is stored in the season variable, and so we insert that into the parentheses. Inside the curly braces, we insert our multiple cases. The case keyword will compare the value we have written to the value stored in the season variable. If the value matches the value stored in the variable, the block of code for that case would execute. If the values do not match, the next case is checked until one of the values match or the code reaches the default keyword. The break keyword stops the case testing inside the block and breaks out of the switch block. If there is no match, the block of code for the default keyword will execute.

Chapter 4 Quiz

- 1) What are conditional statements?
 - a. Statements that iterate over each other
 - b. Statements that change the data type of a variable
 - c. Statements that help a user know if something is wrong with their code
 - d. Statements that check if a condition is met and performs a specific action based on the condition
- 2) If... else statements can ONLY have 1 condition assigned to each statement:
 - a. True
 - b. False
- 3) You cannot use logical operators in conditions for conditional statements
 - a. True
 - b. False
- 4) If a condition contains the AND (&&) operator:
 - a. Both expressions in that condition must be true for the block of code to execute
 - b. Only one expression in the condition has to be true for the block of code to execute
 - c. None of the expressions have to be true for the block of code to execute
 - d. The block of code in all the statements will execute
- 5) If a condition contains the OR (||) operator:
 - a. Both expressions in that condition must be true for the block of code to execute
 - b. Only one expression in the condition has to be true for the block of code to execute

- c. None of the expressions have to be true for the block of code to execute
 - d. The block of code in all the statements will execute
- 6) What will be the output of the following switch statement?

```
var number = 52;
switch (number) {
  case 1:
    console.log("Any number multiplied by 1 equals that number!");
    break;
  case 2:
    console.log("2 is the only even prime number!");
    break;
  case 3:
    console.log("3 is the atomic number of the element lithium!");
    break;
  default:
    console.log(number + " is an interesting number!");
    break;
}
```

- a. Any number multiplied by 1 equals that number!
 - b. 2 is the only even prime number!
 - c. 3 is the atomic number of the element lithium!
 - d. 52 is an interesting number!
 - e. number + " is an interesting number!"
- 7) What will be the output of the following piece of code?

```
var x = 2;
var y = 10;

if (x > y || x !== y) {
  console.log('This condition is true');
} else {
  console.log('This condition is false');
}
```

- a. This condition is true
- b. This condition is false
- c. Syntax Error
- d. 2, 10

8) How would you refactor the following code to include the ternary operator?

```
var timeOfDay = 'morning';

if (timeOfDay === 'morning') {
    console.log('Have a great morning!');
} else {
    console.log('Have a great day!');
}
```

- a. `timeOfDay === 'morning' ? console.log('Have a great morning!') : console.log('Have a great day!');`
- b. `timeOfDay = 'morning' : console.log('Have a great morning!') ? console.log('Have a great day!');`
- c. `timeOfDay === 'morning' : console.log('Have a great morning!') ? console.log('Have a great day!');`
- d. `timeOfDay === 'morning' ? console.log('Have a great day!') : console.log('Have a great morning!');`

9) What will be the output of the following code?

```
var country = 'Germany';

if (country === 'France') {
    console.log('The capital of France is Paris');
} else if (country === 'Austria') {
    console.log('The capital of Austria is Vienna');
} else if (country === 'Germany') {
    console.log('The capital of Germany is Berlin');
} else {
    console.log('That is a great country!')
}
```

- a. The capital of France is Paris
- b. The capital of Austria is Vienna
- c. The capital of Germany is Berlin
- d. That is a great country!

Chapter 5 - Functions

What Are Functions?

Functions are blocks of code that are designed to perform a specific task. Let's say you need to calculate the distance from one point to another, using the formula: $\text{distance} = \text{speed} * \text{time}$. This would be fairly simple, right? You could simply do something like:

```
var speed = 50;
var time = 2;
var distance = speed * time;
console.log(distance);
// Output: 100
```

So, in the code above, we declared variables for speed, time and distance. We then assigned values for speed and time and then set distance to the value obtained when we multiply speed and time. But what if you had to calculate several different distances? You could repeat the same code in the following manner:

```
var speed1 = 50;
var time1 = 2;
var distance1 = speed1 * time1;
console.log(distance1);
// Output: 100

var speed2 = 25;
var time2 = 1;
var distance2 = speed2 * time2;
console.log(distance2);
// Output: 25

var speed3 = 60;
var time3 = 3;
var distance3 = speed3 * time3;
console.log(distance3);
// Output: 180
```

There is nothing wrong with the codes above. They work and give you the correct answer. But in coding, we do not want to repeat the same code again and again, as it violates something called the Don't Repeat Yourself or DRY principle. We want to try to keep our code as concise as possible to make it easier to maintain, easier to debug (find errors in your code, and save time and effort. Above all, it would be unsustainable to perform these calculations in this manner. Imagine if you had to calculate hundreds of distances! It would just be lines and lines of unnecessary code.

This is where functions come in. We can reuse that block of code we created as many times as we like to perform the same task again and again. In this section, we will learn how to create functions and how to use them efficiently in our code.

Function Declaration

JavaScript has several methods to create a function. One of these methods is known as a function declaration (also known as a function statement or function definition). Let's say we want to create a function receive a reminder to eat our fruits and vegetables. A function declaration for that would look something like this:

```
function reminder() {  
    console.log('Remember to eat your fruits and vegetables!');  
}
```

Let's break down the piece of code shown above. First, we declare a function using the function keyword. We then provide the function with a name or an identifier. In this case, we have given it the name, reminder as that is a sensible name to use in this case. We then have a set of parentheses(). We can set parameters in the parentheses, but we will do that a little later. Within the curly brackets, or the function body, we added a block of code of the task we want the function to perform. In

this case, it's log to the console, 'Remember to eat your fruits and vegetables!'. This function can now be used at any time to display that reminder. So, we have written the function, but how do we actually use it? We need to *call* the function for the code inside the curly brackets to run. To call a function, you simply type in the function name or identifier, followed by a set of parentheses(). Let's give that a try:

```
function reminder() {  
    console.log('Remember to eat your fruits and vegetables!');  
}  
reminder();  
// Output: Remember to eat your fruits and vegetables!
```

This will now output the reminder to your console window! Pretty cool, right? You may have also noticed something else. When you try to log something to the console, you type in console.log(), which is also a function! So you have actually been using the log() function this entire time!

Let's now go back to our first example and write a function to calculate the distance when given the speed and time:

```
function distance(speed, time) {  
    console.log(speed * time);  
}
```

What we have done here is declared a function and given it a name, similar to what we did with the reminder function. Except this time, we have added in two parameters, speed and time, separated by commas inside the parentheses. Adding parameters allows the function to accept input values to perform a specific task using those values. Parameters are sort of like variables; in that they store information. In this case, our speed and time parameters will store the two values that have to be

multiplied together. Let's now call this function to calculate the distance when speed is 50 and time is 2:

```
function distance(speed, time) {  
    console.log(speed * time);  
}  
distance(50, 2);  
//Output: 100
```

We called our function in a similar manner to what we did before, except we added in our two values, 50 and 2. When values are passed when calling the function, they are known as arguments. Notice that we passed in the arguments in the same order that we declared the parameters. Let's now calculate the distances for the other two speed and time values from the first example:

```
function distance(speed, time) {  
    console.log(speed * time);  
}  
distance(50, 2); // Output: 100  
distance(25, 1); // Output: 25  
distance(60, 3); // Output: 180
```

Compare this to the example the was first shown. This is a much cleaner and concise version of the calculations! By doing this, we followed the DRY principal, and our code looks much more readable and is also more sustainable as it is easier to perform several of these calculations!

Return

We are currently able to see our outputs because we are logging them to the console. But let's try to remove the `log()` function from the code block and instead, log the function that call:

```
function distance(speed, time) {  
    speed * time;  
}  
console.log(distance(50, 2)); // Output: undefined
```

When we try to log the function, we call with the arguments 50 and 2, we get an output of undefined. This happened because we did not use the return statement in the function body. The computer did calculate the distance with the arguments provided, it just did not capture the result. When we use the return statement, the execution of the function stops, and a value is returned from that function. If we omit the return statement, undefined is returned by default. Let's add the return statement to the function body and see what happens:

```
function distance(speed, time) {  
    return speed * time;  
}  
console.log(distance(50, 2)); // Output: 100
```

Now that we've added the return statement, the function can produce a result, which we can then store to a variable! Let's try this now:

```
function distance(speed, time) {  
    return speed * time;  
}  
var distance1 = distance(50, 2);  
console.log(distance1); // Output: 100
```

Since we returned the result of the function, we were able to store it into a variable called `distance1`. We can now use this variable later on if we wish to! The return statement is also applicable to other parts of your code where you would need to store certain outputs based on a calculation or a condition.

Function Expressions

In JavaScript, another way to define a function is to use a function expression. One difference between a function declaration and a function expression is that a function expression omits a function name or identifier. When a function does not have a name or identifier, it is known as an *anonymous function*. Let's take a look at an example below:

```
var distance = function(speed, time) {  
    console.log(speed * time);  
}
```

If you compare this to the function declaration in the previous examples, you will notice that this function has not been given a name, but instead it is saved inside a variable. To use this function, we call it by typing the variable name, followed by two parentheses() with the arguments inside. Let's give it a try:

```
var distance = function(speed, time) {  
    console.log(speed * time);  
}  
distance(50, 2); // Output: 100
```

Similar to the function declaration, we typed in the variable name, `distance`, and entered our two arguments in the parentheses. If the result of function declarations and function expressions are the same, why do

we have both? This is because of *hoisting*. We'll find out more about hoisting a little later.

Scope

Scope in JavaScript is an important concept to understand for you to become a good coder. It is quite a difficult concept to grasp, so don't worry if you don't understand it the first time. Scope refers to the current context of the code, or the context at which the declarations are visible or accessible. There are two types of scope in JavaScript: local and global. The global scope is anything declared outside of a block, and the local scope is anything declared within a block. Let's see what this means in terms of variables.

```
var greeting = 'Hi!';

function change() {
  var greeting = 'Hello!'
  console.log(greeting);
}

console.log(greeting); // Output: Hi!
change(); // Output: Hello!
console.log(greeting); // Output: Hi!
```

In the example above, we have declared a variable called `greeting` and assigned it the string, 'Hi!'. We then declared a function called `change` which mutates the variable `greeting` to the string, 'Hello!'. When logging the variable to the console, we can see that even though we have assigned another variable with the same name within the function, the console still logs the string of the original variable. The only time the string changes, is when we call the `change()` function. This is because the

console is logging the variable that was defined in the global scope, and not within the local scope of the function.

Let's go over that again. Any variable that is declared outside a block of code, such as a function, is accessible within the global context. Whereas variables declared within a block of code, are only accessible to that block of code, or the local scope, and not the global scope. Here is another example:

```
//var greeting = 'Hi!';

function change() {
  var greeting = 'Hello!';
  console.log(greeting);
}

console.log(greeting);
// Output: ReferenceError: greeting is not defined
```

The variable declaration has been commented out in the example above and it has only been declared within the function. When we now try to log greeting, we get an error telling us that greeting is not defined, even though it has been declared in the function. This is because the variable declared in the local scope, is only accessible within that scope. Don't worry if you still don't understand it fully. This will make a lot more sense when you become a more experienced coder.

Hoisting

In JavaScript, all variable and function declarations are hoisted to the top of their scope. Let's take a look at one example:

```
distance(50, 2); // Output: 100

function distance(speed, time) {
  console.log(speed * time);
}
```

The distance function was called before the function was declared, and yet the code still worked and gave us the correct output. This is because of hoisting. In this case, the distance function was hoisted to the top of the global scope, which is why the code worked even though the function was called before the code was written.

The same however, does NOT work for function expressions. Take a look at the next example:

```
distance(50, 2); // Output: TypeError: distance is not a function

var distance = function(speed, time) {
  console.log(speed * time);
}
```

The distance function was re-written here as a function declaration. However, unlike the previous example, the code did not work and we instead received an error informing us that distance is not a function. This is the second major different between function declarations and functions expressions. Function declarations are hoisted, whereas function expressions are not.

Just like functions, variables are hoisted to the top of the scope too. However, only the variable name is hoisted to the top, NOT the value stored in the variable.

```
console.log(greeting); // Output: undefined  
  
var greeting = 'Hello!'
```

In the example above, you would expect the console to show an error and not just give an output of undefined. But in fact, the greeting variable has been hoisted to the top of the scope. This means that the computer knows that the variable exists, it just does not know what value it has been assigned. To the computer, the code above looks something like this:

```
var greeting;  
  
console.log(greeting); // Output: undefined  
greeting = 'Hello!'
```

Hoisting is generally considered bad practice and it is better to declare variables and assign them a value before we use them. Function declarations should only be used when you need a function to be available throughout your code. If you need a function to be used in only a specific part of your code, it is advisable to use function expressions.

Function Expression	Stored in variable	Not hoisted – available in local scope
Function Declaration	Not stored in variable	Hoisted to the top of its scope e.g. global scope

Chapter 5 Quiz

- 1) What is a function?
 - a. A piece of code that checks if a condition is met and performs a specific action based on the condition
 - b. A piece of code that performs a specific task
 - c. A piece of code that groups similar data together
 - d. A variable that stores values
- 2) What type of function is shown below?

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

- a. Function expression
 - b. Function declaration
- 3) What is one of the differences between function expressions and function declarations?
 - a. Function expressions can accept parameters as arguments and function declarations cannot
 - b. Function declarations can return results while function declarations cannot
 - c. Function expressions are hoisted to the top of their scope while function declarations are not
 - d. Function expressions are stored in variables while function declarations are not
- 4) Which of the following is a parameter in the function below?

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

- a. function
- b. b

- c. return
 - d. add
- 5) Variables declared in the local scope can be accessed in the global scope
- a. True
 - b. False
- 6) Variables are hoisted to the top of their scope
- a. True
 - b. False
- 7) What will be the output of the following piece of code?

```
var car = 'BMW';

var changeCar = function() {
  car = 'Audi';
}
changeCar();
console.log(car);
```

- a. BMW
 - b. Audi
 - c. Undefined
 - d. Car
- 8) Function declarations can be hoisted to the top of their scope
- a. True
 - b. False
- 9) What will be the output of the following piece of code?

```
var tip = calcTip(100, 0.2);
console.log(tip);

function calcTip(bill, percentage) {
  return bill * percentage
}
```

- a. Undefined

- b. TypeError: calcTip is not a function
- c. 20
- d. Null

10) Hoisting is considered to be good practice and is encouraged

- a. True
- b. False

11) How do you call a function named, 'greeting'?

- a. greeting();
- b. call greeting();
- c. function greeting();
- d. call function greeting();

12) Is the piece of code below a function expression or a function declaration?

```
var greeting = function() {  
    console.log('Good morning!');  
}  
  
greeting();
```

- a. Function Expression
- b. Function Declaration

13) What is the purpose of adding parameters to functions?

- a. They allow the function to be called
- b. They allow the function to return a value
- c. They allow a function to output a value
- d. They allow a function to accept input values

Chapter 6 - Arrays

What Are Arrays?

Arrays are used in JavaScript to store several values into one variable. They are like lists that can store several data types, such as booleans, strings and numbers. Arrays are useful as they allow us to group related data into a single variable instead of storing them separately. Let's take a look at an example:

```
var groceries1 = 'bananas';  
var groceries2 = 'apples';  
var groceries3 = 'oranges';
```

If you have to go to the grocery store, you need to make a list of what you are going to get. You could store them all into separate variables as shown above. But it wouldn't make sense to do it this way because you could have a list of 20-30 different items or more. It would also be very difficult to find each individual item, as they are all stored in different variables. Instead, it would be much easier to store them all in one array:

```
var groceries = ['bananas', 'apples', 'oranges'];
```

By organizing the grocery list into an array, all items can be stored into one variable, and can be accessed easily. To create an array, wrap the items in square brackets [], and separate each item with a comma. Each individual item in an array is known as an *element*.

As mentioned before, arrays can have several data types. Here is an example of an array with two different data types:

```
var person = ['Adam', 'Jackson', 31];
```

This array contains a person's first name, last name and age.

Accessing Elements in an Array

Each individual element in an array can be accessed by its *index number*. In JavaScript, arrays are zero-indexed, which means that the first element in an array starts from 0 instead of 1. The example below demonstrates how to access the first element of the groceries array:

```
groceries[0]; // bananas
```

You can access an element in an array by typing the name of the variable then inputting the number of the element you want to access in square brackets []. In this case, we are accessing element 0 in the groceries array, which is our first grocery item, bananas. If we wanted to access oranges, what index number would we have to put in the square brackets? Also, try making a bigger list and play around with accessing different elements.

Replacing Elements in an Array

It is also possible to replace elements in an array. Let's say we wanted to replace apples with watermelons in our groceries array. Apples has an index number of 1, since it is the second element in our array. So, we would replace apples like this:

```
groceries[1] = 'watermelons';  
console.log(groceries);  
// Output: ['bananas', 'watermelons', 'oranges']
```

Similar to how we accessed the first element, we typed the variable name and inserted the index number of the item we wanted to replace. We then set the value to watermelon and apples automatically gets replaced.

Using the Length Property

In JavaScript, arrays have a built-in property that returns the length (number of elements) in an array. Let's see how this property works:

```
var groceries = ['bananas', 'apples', 'oranges'];  
console.log(groceries.length);  
// Output: 3
```

In the example above, we logged to the console the length, or the number of elements in the groceries array. We received an output of 3, because there are 3 elements in our array. Notice how we typed out the variable name and then added a `.length` at the end. You can use the length property when you want to know how many elements there are in an array. This will be useful when we go through loops in the next section of the course.

Useful Array Methods

Let's go over a few useful array methods: push, pop, shift, unshift, indexOf and splice.

Push Method

The push method allows us to add elements to the end of an array. Let's say we wanted to add peaches and pears to our groceries array. This would be done like this:

```
var groceries = ['bananas', 'apples', 'oranges'];  
groceries.push('peaches', 'pears');  
console.log(groceries);  
// Output: ['bananas', 'apples', 'oranges', 'peaches', 'pears']
```

Notice how we call the push method as a function. We've added peaches and pears, separated by a comma in the middle. You can add as many elements as you want to the end of an array as long as you separate them with commas.

Pop Method

The pop method allows us to remove an element from the end of an array. The pop method can be used like this:

```
var groceries = ['bananas', 'apples', 'oranges'];
groceries.pop();
console.log(groceries);
// Output: ['bananas', 'apples']
```

The pop function removed the final element in our array, oranges. You do not need to enter any arguments into the pop function, as it simply removes the final element.

Shift Method

The shift method allows us to remove elements from the beginning of an array. We can use the shift method like this:

```
var groceries = ['bananas', 'apples', 'oranges'];
groceries.shift();
console.log(groceries);
// Output: ['apples', 'oranges']
```

The shift function removed the first element in our array, bananas. As with the pop function, you do not need to add in any arguments to the function.

Unshift Method

The unshift function allows us to add elements to the beginning of our array. Let's say we want to add papaya and grapes to the beginning of our array. We would do so like this:

```
var groceries = ['bananas', 'apples', 'oranges'];
groceries.unshift('papaya', 'grapes');
console.log(groceries);
// Output: ['papaya', 'grapes', 'bananas', 'apples', 'oranges']
```

The unshift function added elements to the beginning of the array. You can add as many elements into the beginning of the array as you want. Simply add the arguments into the parentheses, separated by commas.

IndexOf Method

The indexOf method allows us to get the index number of a particular element in an array. The indexOf method can be used like this:

```
var groceries = ['bananas', 'apples', 'oranges'];
console.log(groceries.indexOf('apples'));
// Output: 1
```

We typed in the name of our variable, groceries, followed by the .indexOf() function. Inside the indexOf function, we added in our argument, 'apples' to find the index number of that element. As apples is the second element of our index, the output is 1.

Splice Method

The splice method allows us to remove or replace the contents in the middle of an array. This method works slightly differently to the previous methods. Let's say we want to add grapes to our list of groceries, but we want to add it to the middle of the list, right after bananas. We can add grapes like this:

```
var groceries = ['bananas', 'apples', 'oranges'];
groceries.splice(1, 0, 'grapes');
console.log(groceries);
// Output: ['bananas', 'grapes', 'apples', 'oranges']
```

Inside our splice function, we added in three different arguments: 1, 0 and 'grapes'. The first two arguments that the splice function takes in are the index number, and the delete count, in that order. The index number is the start of the index which must be modified. The delete count is the number of elements to be removed, starting with the index number

entered. The next argument after the delete count is the new element we want to add. We can add as many elements as we want after this and just separate them with commas. In our example, because we just wanted to add grapes to our list and not delete anything, we entered the arguments 1 and 0. Now let's say we want to delete apples and replace it with grapes. We would do it like this:

```
var groceries = ['bananas', 'apples', 'oranges'];  
groceries.splice(1, 1, 'grapes');  
console.log(groceries);  
// Output: ['bananas', 'grapes', 'oranges']
```

Notice how we now changed the 0 to 1, because we wanted to delete the apples element and replace it with grapes. Try changing the delete count to 2 and see what happens.

All of these methods, except for the `indexOf` method, are known as *destructive methods* because they alter the original arrays.

Chapter 6 Quiz

- 1) Why do we use arrays in JavaScript?
 - a. To store related information in one variable
 - b. To check if a condition is met and perform a specific action based on the condition
 - c. To change the data type of a variable
 - d. To mutate a variable
- 2) What does the .length property do?
 - a. Returns the index number of an element in an array
 - b. Replaces elements in the middle of an array
 - c. Returns the number of elements in an array
 - d. Removes the final element in an array
- 3) What will be the output of the following code?

```
var animals = ['cats', 'dogs', 'giraffes', 'kangaroos', 'koalas'];  
animals.splice(3, 2, 'lions', 'tigers', 'gorillas');  
console.log(animals);
```

- a. ['cats', 'dogs', 'giraffes', 'kangaroos', 'koalas']
 - b. ['cats', 'dogs', 'giraffes', 'lions', 'tigers', 'gorillas']
 - c. ['cats', 'dogs', 'giraffes', 'kangaroos', 'koalas', 'lions', 'tigers', 'gorillas']
 - d. ['lions', 'tigers', 'gorillas']
- 4) What is the index number of Tokyo in the following array?

```
var cities = ['New York', 'Paris', 'London', 'Tokyo', 'Sydney'];
```

- a. 0
 - b. 3
 - c. 4
 - d. 5
- 5) An array has a length of 134. What will be the index number of the final element in the array?

- a. 134
 - b. 135
 - c. 133
 - d. 132
- 6) Which of the following methods adds one element to the front of an array?
- a. unshift();
 - b. pop();
 - c. slice();
 - d. push();
- 7) Which of the following methods adds one element to the end of an array?
- a. unshift();
 - b. pop();
 - c. slice();
 - d. push();
- 8) Arrays in JavaScript are NOT zero-indexed
- a. True
 - b. False
- 9) Why is the indexOf method not a destructive method?
- a. It is not a method used by arrays
 - b. It allows you to add an element to the end of an array
 - c. It does not alter the original array
 - d. It alters the original array

Chapter 6 - Loops

What Are Loops?

In programming, loops are a tool that programmers use to execute a block of code repeatedly until a certain condition to stop the code is met. Loops are very useful in JavaScript and serve several purposes. Let's take a look at how they work. Let's say we wanted to count from 1 to 10. You could simply log to the console 10 different times, with each log containing a number from 1-10. But that would be extremely tedious and unsustainable! Instead, we could write a loop like this:

```
for (var i = 1; i <= 10; i++) {  
    console.log(i);  
}
```

The example above is known as a *for* loop, where the code *iterates* or repeats itself until the stopping condition is met. Let's break down the different components of the for loop:

- The for loop starts with the `for` keyword, followed by three different expressions in a set of parentheses,
- Each expression is separated by a semi-colon (;),
- The first expression is known as the *initialization expression*. This gives the for loop a starting point and also declares the iterator variable,
- The second expression is known as the *stopping expression*. The code will run as long as the condition evaluates to true. The stopping condition sets the point at which the condition will evaluate to false, thus ending the loop,
- The third expression is known as the *increment expression*. This is the expression by which the iterator variable will update for each loop.

- Inside the curly brackets we have a code block which will execute as long as the condition remains true.

In the for loop above, the output to the console will be:

1
2
3
4
5
6
7
8
9
10

For our for loop above, the iterator variable in the initialization expression is set to $i = 1$, which means our for loop will start at 1. You can use any variable in the initialization expression, however it is common practice use the letter 'i' when using loops. The stopping expression has set the condition to be true as long as the iterator variable is less than or equal to 10. So, our condition will be false, once the iterator variable becomes greater than 10. The increment expression is set to $i++$, which means increase i by 1 for each loop. Finally, the statement in our code block will execute until the condition equals false. In this case, we want to log to the console the value of i , starting at 1 and ending at 10. You should be careful when setting the stopping condition of your for loop, because if you incorrectly set the stopping condition, you could enter something known as an *infinite loop* and your code will never end! For example, if you set your stopping condition to greater than 10, instead of less than 10, there are an infinite amount of numbers greater than 10,

meaning your loop will never end, and you could potentially crash your computer!

Looping Through Arrays

Looping through arrays is a particularly useful feature in programming, because we can easily access and modify the information stored in arrays. Arrays could store information such as inventory in stores and client information. It would be much easier to add, remove or modify information in arrays if we looped through the array instead of modifying each element individually and repeating the same code. Let's create an array and see how we can iterate through it:

```
var hobbies = ['coding', 'sports', 'music', 'hiking'];  
  
for (var i = 0; i < hobbies.length; i++) {  
    console.log(hobbies[i]);  
}
```

coding
sports
music
hiking

In the example above, we stored a list of hobbies inside a variable called hobbies. We looped through our array using a for loop and logged each element in the array to the console. In this case, we set the iterator variable to 0, because arrays are zero-indexed. We set the .length property to set the stopping condition. The stopping condition is set to less than the length of the array, because the final element of an array is always the length of the array minus 1. If you cannot understand this part, make sure you re-read the previous chapter on arrays.

While Loop

Let's now learn about another type of loop, called a while loop. Let's first convert the for loop in the first example to a while loop:

```
// For Loop
for (var i = 1; i <= 10; i++) {
    console.log(i);
}
// While Loop
var i = 1;
while (i <= 10) {
    console.log(i);
    i++
}
```

The while loop will have the exact same output as the for loop. However, in this loop, we start it by typing in the while keyword instead of the for keyword. Notice how we also declared our iterator variable before our while loop, meaning that the variable is in the global scope and can be accessed by our while loop. After the while keyword, we inserted our stopping condition in the parentheses. Finally, we have our code block which will print the value of i to the console and increment the i variable by 1.

While loops are useful in situations where you are unsure of how many times the loop needs to be executed. Whereas for loops are useful in conditions where you know how many times the loop needs to be executed.

Chapter 6 Quiz

- 1) In programming, loops are used to:
 - a. Execute a piece of code repeatedly as long as the condition set is true
 - b. Change the data type of a variable
 - c. Group similar data together
 - d. Declare functions
- 2) In a for loop, the first expression is known as the:
 - a. Stopping Expression
 - b. Initialization Expression
 - c. Increment Expression
 - d. Infinite Expression
- 3) What will be the output of the following for loop?

```
for (var i = 2; i < 10; i+=2) {  
    console.log(i);  
}
```

- a. 2, 4, 6, 8, 10
 - b. 2, 3, 4, 5, 6, 7, 8, 9, 10
 - c. 0, 2, 4, 6, 8, 10
 - d. 2, 4, 6, 8
- 4) When looping through an array, the initialization variable should be set to 1 because the first element of an array has an index of 1
 - a. True
 - b. False
- 5) When looping through an array, the stopping condition should equal the length of the array
 - a. True
 - b. False

6) What will be the output of the following while loop:

```
var x = 20
while (x < 100) {
  console.log(x);
  x+=25;
}
```

- a. 20, 40, 60, 80, 100
 - b. 20, 45, 70, 95
 - c. 20, 40, 60, 80
 - d. 20, 45, 70, 95, 120
- 7) It is better to use while loops when we are unsure of how many times the loop needs to be executed
- a. True
 - b. False
- 8) What is wrong with the following for loop?

```
for (var i = 0, i <= 10, i++) {
  console.log(i);
}
```

- a. The iterator is incorrect
 - b. The initializing variable should be set to anything other than i
 - c. The stopping condition is invalid
 - d. The commas should be replaced with semi-colons
- 9) What would happen if you ran the following code?

```
for (var i = 0; i > 10; i++) {
  console.log(i);
}
```

- a. The numbers 1-10 would be logged to the console
- b. The loop would return an error
- c. The numbers 1-10 would be logged in reverse
- d. An infinite loop would occur

Chapter 7 - Objects

What Are Objects?

Objects in JavaScript can be compared to objects in real-life. Let's use the example of a car. A car is an object which has several properties. These properties can include things like the car's design, color, model and weight. Similarly, objects in JavaScript have properties associated with it. These properties are stored as variables and is organized into key-value pairs. The variable name can be thought of as the key to a storage that contains values. These values can be anything from the primitive data types to functions. Let's take a look at how we can form an object in JavaScript:

```
var car = {  
  make: 'Audi A8',  
  model: 2020,  
  color: 'black',  
  'fuel type': 'petrol',  
}
```

We declared a variable called car then used the curly brackets {} to designate an object literal. The key-value pair in this case is made up of the key name or identifier and a value, separated by a colon. Each key value pair is then separated using a comma (,). Notice how all the keys except for 'fuel type' is written in quotation marks. This is because fuel type contains a space character.

Accessing Properties of Objects

You can access a property of an object using the dot notation (.). To do this, simply write the name of the object followed by a dot and then the property name:

```
var car = {  
  make: 'Audi A8',  
  model: 2020,  
  color: 'black',  
  'fuel type': 'petrol',  
}  
car.make; // Returns Audi A8
```

You can also access an objects' property through the bracket notation []. The bracket notation must be used when you are accessing a property that has a number, space or special character in it:

```
var car = {  
  make: 'Audi A8',  
  model: 2020,  
  color: 'black',  
  'fuel type': 'petrol',  
}  
  
car['fuel type']; // Returns petrol
```

Assigning Properties

You can assign new properties to objects or mutate them even after they have been created. You can either use the bracket notation or the dot notation to reassign a property or add a new key-value pair. Here is an example:

```
var car = {  
  make: 'Audi A8',  
  model: 2020,  
  color: 'black',  
  'fuel type': 'petrol',  
}  
car.engine = 'V8'; // Adds new key of engine with a value of V8  
car.color = 'silver' // Changes the color property to silver
```

When reassigning properties, any new values assigned to an existing property will be replaced by the new value. If a property does not exist, then the property will be created in that object. In the example above, the engine property did not exist, but we added it to the object and assigned it a value of, 'V8'. We also changed the color of the car from black to silver. It is also possible to delete a property from an object, by using the delete operator:

```
var car = {  
  make: 'Audi A8',  
  model: 2020,  
  color: 'black',  
  'fuel type': 'petrol',  
}  
delete car.model; // Deletes model property
```

Adding Methods to Objects

You can add functions as a property of an object. In this case, functions are called methods. Functions can be added by assigning a property name for the key, followed by a colon and an anonymous function expression as the value:

```
var car = {
  make: 'Audi A8',
  model: 2020,
  color: 'black',
  'fuel type': 'petrol',
  message: function() {
    console.log('This is the latest model with the sleekest design!');
  }
}
```

To invoke an object method, access the method using the dot notation, followed by a set of parentheses ():

```
var car = {
  make: 'Audi A8',
  model: 2020,
  color: 'black',
  'fuel type': 'petrol',
  message: function() {
    console.log('This is the latest model with the sleekest design!');
  }
}
car.message();
// Output: This is the latest model with the sleekest design!
```

Nested Objects

Nested objects are objects that have other objects as properties. Let's modify our car object, so that it includes nested objects:

```
var car = {  
  details: {  
    make: 'Audi A8',  
    model: 2020,  
    color: 'black'  
  },  
  'engine and transmission': {  
    type: 'V8',  
    fuel: 'petrol',  
    transmission: 'automatic'  
  },  
  safety: {  
    locks: 'child safety locks',  
    brakes: 'anti-lock braking system',  
    noOfAirbags: 8  
  }  
}
```

Our car object now includes three objects as properties. First, we have details of the car, which will contain properties such as make, model and color. Then we have the engine and transmission, with properties such as engine type, fuel and transmission. Finally, we have safety features with properties such as locks, brakes and number of airbags.

Now let's say we wanted to add more information on the engine, such as horsepower and torque. We could do it like this:

```

var car = {
  details: {
    make: 'Audi A8',
    model: 2020,
    color: 'black'
  },
  'engine and transmission': {
    engine: {
      type: 'V6',
      horsepower: '335 @ 5000',
      torque: '369 @ 1370',
      fuel: 'petrol'
    },
    transmission: 'automatic'
  },
  safety: {
    locks: 'child safety locks',
    brakes: 'anti-lock braking system',
    noOfAirbags: 8
  }
}

```

We have now nested our objects even further, with the car object have a property of engine and transmission as an object, which has properties of engine and transmission as objects, which then have their own respective properties. To access nested properties, we can chain operators as shown in the example below:

```

console.log(car['engine and transmission'].engine.horsepower);
// Output: 335 @ 5000

```

This Keyword

Let's now create a new object called menu, and enter three properties: appetizer, main and dessert and the following values: caesar salad, steak and cheesecake. Let's then create a method inside the menu object that returns the food we would like to order.

```
var menu = {  
  appetizer: 'caesar salad',  
  main: 'steak',  
  dessert: 'cheesecake',  
  order: function() {  
    console.log('I would like to start with  
    a ' + appetizer + ', followed by a ' +  
    main + ' and finally, ' + dessert + '  
    for dessert');  
  }  
}  
menu.order();  
// Output: Uncaught ReferenceError: appetizer  
is not defined
```

As you can see, we get a reference error that says appetizer is not defined. This happens because inside the scope of the order function, we do not have access to the other properties of the menu object. In order for the order function to gain access to the other properties, we need to use the this keyword:

```
var menu = {  
  appetizer: 'caesar salad',  
  main: 'steak',  
  dessert: 'cheesecake',  
  order: function() {  
    console.log('I would like to start with  
a ' + this.appetizer + ', followed by a  
' + this.main + ' and finally, ' +  
this.dessert + ' for dessert');  
  }  
}  
menu.order();  
// Output: I would like to start with a caesar  
salad, followed by a steak and finally,  
cheesecake for dessert
```

The `this` keyword allows for the calling object's properties to be accessed. In the example shown, the `this` keyword is calling the `menu` object, followed by the `appetizer`, `main` and `dessert` properties of the `menu` object.

Chapter 7 - Quiz

- 1) Objects are stored as key-value pairs
 - a. True
 - b. False
- 2) Which of the following is the correct method to assign a new property of the ship object?

```
var ship = {  
  name: 'The Black Pearl',  
}  
// 1) shipcaptain = 'Jack Sparrow'  
// 2) captain.ship = 'Jack Sparrow'  
// 3) var ship.captain = 'Jack Sparrow'  
// 4) ship.captain = 'Jack Sparrow'
```

- a. 1
 - b. 2
 - c. 3
 - d. 4
- 3) Object properties can be accessed using:
 - a. Bracket notation []
 - b. Dot notation (.)
 - c. Both of these
 - d. None of these
- 4) What are methods in objects?
 - a. A way to change the variable name
 - b. An object that is a property of another object
 - c. A function assigned as a property
 - d. They allow object's properties to be accessed
- 5) The this keyword is used to nest objects inside other objects
 - a. True
 - b. False
- 6) What will be the output of the following code?

```
var Germany = {
  majorCities: {
    capital: 'Berlin',
    financialCapital: 'Frankfurt',
    smallestCity: 'Arnis',
  },
  Population: 83000000,
  currency: 'Euro',
}
console.log(Germany.majorCities.capital);
```

- a. Berlin
- b. Euro
- c. Frankfurt
- d. Arnis

7) In the following code, a favorite city property, will be assigned to the major cities property in the Germany object

```
var Germany = {
  majorCities: {
    capital: 'Berlin',
    financialCapital: 'Frankfurt',
    smallestCity: 'Arnis',
  },
  Population: 83000000,
  currency: 'Euro',
}

Germany.majorCities.favoriteCity = 'Munich';
```

- a. True
- b. False

8) What is the purpose of the this keyword?

- a. To allow the calling properties object to be accessed
- b. To call a method inside an object
- c. To add a new property to an object
- d. To reassign a value to a property

Chapter 8 - ES6 Features

Let and Const

So far, we have only been using `var` to declare our variables. Since ES6 was introduced, there are two other ways to declare variables as well, using the `let` and `const` keywords. These two keywords vary from the `var` keyword, in many ways. The first one is that `let` and `const` cannot be redeclared. Let's take a look at an example:

```
var x = 10;  
var x = 11;  
console.log(x);  
// Output: 11
```

```
let x = 10;  
let x = 11;  
console.log(x);  
// Output: SyntaxError 'x' has already  
been declared
```

```
const x = 10;  
const x = 11;  
console.log(x);  
// Output: SyntaxError 'x' has already  
been declared
```

In this example, we have declared the same variable 'x' twice. When using the `var` keyword, the value of the variable simply mutated and we were given an output of the second value that was assigned. When using the `let` and `const` keywords, we instead received a syntax error saying 'x'

was already defined. This is important, because this means that if we have already declared a variable using the `let` and `const` keywords, we cannot accidentally declare another variable with the same name, unlike if we were using the `var` keyword. Another difference is that while the `var` and `let` keywords can be reassigned, the `const` keyword cannot:

```
let x = 10;  
x = 12;  
console.log(x);  
// Output: 12  
  
var y = 15;  
y = 20;  
console.log(y);  
// Output: 20
```

```
const x = 10;  
x = 12;  
console.log(x);  
// Output: TypeError: Assignment to  
constant variable
```

The `const` keyword is useful if you do not want the value assigned to a variable to be changed. When using the `const` keyword, you must also assign a value to it when declaring a variable. You cannot assign a value to it later on.

Another difference is that the `var` keyword is function scoped while the `let` and `const` keywords are block scoped. Any variables declared with the `var` keyword inside a block of code, within curly brackets, will still be available within the global scope. The only exception to this rule is if they are declared within a function body. Let's see how this works:

```
if (3 > 2) {  
    var message = 'The condition is  
    true';  
}  
console.log(message);  
// Output: The condition is true
```

```
if (3 > 2) {  
    let message = 'The condition is  
    true';  
}  
console.log(message);  
// Output: ReferenceError: message is not  
defined
```

```
if (3 > 2) {  
    const message = 'The condition is  
    true';  
}  
console.log(message);  
// Output: ReferenceError: message is not  
defined
```

In this example, the variable declared with `var` still gave us an output despite it being declared within a set of curly brackets. However, when declaring the same variable using `let` or `const`, we instead get a reference error. Generally, it is considered good practice to use the `const` keyword to assign variables as much as possible, and only use the `let` keyword when you have to reassign variables.

Template Literals

ES6 also introduced a new way to work with strings. Template literals use backticks (``) as a syntax rather than quotation marks. The backticks are located next to the (1) key on the keyboard, below the escape key. Let's see how template literals work:

```
const greeting = `Hello World!`;
// Output: Hello World!
```

In this example, we printed hello world to the console using backticks instead of quotation marks. One of the major benefits of using template literals is string substitution. String substitution allows us to embed variables as if they were part of the string. This works in the following manner:

```
const firstName = `Adam`;
const lastName = `Williams`;
const age = 25;

console.log(`My name is ${firstName}
${lastName} and I am ${age} years old`);
```

Normally, we would have had to use the plus (+) sign and enter the variables separately. By using template literals, we can embed the

variables into the string by using the dollar (\$) sign and a set of curly brackets ({}).

We can also use the `${}` syntax to perform mathematical calculations within the string itself, as shown below:

```
const firstName = `Adam`;
const lastName = `Williams`;
const yearOfBirth = 1995;
const currentYear = 2020;

console.log(`My name is ${firstName} ${lastName}
and I am ${currentYear - yearOfBirth} years
old`);
```

Default Parameters

ES6 allows functions to have default parameters. Default parameters assign a predetermined value to the arguments in a function in the event that there are no arguments passed into the function or if the parameter is undefined. Let's see how default parameters work:

```
function greeting(name = `User`) {
  console.log(`Hello, ${name}`);
}

greeting();
// Output: Hello User
greeting('Adam');
// Output: Hello Adam
```

In this example, we created a function called `greeting` that logs to the console a greeting based on the user's name. However, if an argument is

not passed when calling the function, the default argument is then User. If we did not add a default parameter here, we would get an error when we try to call the function without an argument. This is because function parameters default to undefined if no argument is passed.

Arrow Functions

The arrow function syntax is another new feature in ES6. It allows for functions to be written in a shorter manner. This can be done by typing a set of parentheses (), followed by an arrow (=>). The advantage of using arrow functions is that you do not need to type the function keyword each time a function is created. Let's see how this works:

```
const greeting = () => {  
  console.log(`Good morning!`);  
}  
  
greeting();  
// Output: Good morning!
```

In this example, the function keyword was omitted and instead was replaced by the arrow. If you had to create a function that needed to take in parameters, you could do it like this:

```
const multiply = (a, b) => {  
  console.log(a * b);  
}  
  
multiply(5, 10);  
// Output: 50
```

In this example we have added in two parameters, a and b. This function would work in the exact same way as the other functions would. You will

come across this method of writing functions throughout your JavaScript coding career, so it is important that you know all these methods.

Chapter 8 Quiz

- 1) Variables declared using the let and const keywords can be redeclared
 - a. True
 - b. False
- 2) Variables declared using the const keyword can be reassigned a new value
 - a. True
 - b. False
- 3) The difference between assigning variables using the let and const keywords and the var keyword is/are:
 - a. The var keyword is function scoped while the let and const keywords are block scoped
 - b. The let and const keywords allow variables to be reassigned different values
 - c. Variables declared with the let and const keywords cannot be logged to the console
 - d. All of the above
- 4) Template literals are used to declare variables
 - a. True
 - b. False
- 5) One difference between using template literals and using quotation marks is that
 - a. Assigning values to variables using template literals would allow you to redeclare the variable regardless of the keyword used to declare the variable
 - b. Template literals allow you to log strings to the console while strings in quotation marks cannot be logged to the console
 - c. Template literals allow you to write long strings while quotation marks do not

- d. All of the above
- 6) If a function accepts a parameter, but no arguments are passed when calling a function, then the default function parameter is
 - a. Null
 - b. NaN
 - c. False
 - d. Undefined

7) What will be the output of the following piece of code?

```
function multiply(x = 1, y = 2) {  
    console.log(x * y);  
}  
  
multiply();
```

- a. Reference Error
 - b. Undefined
 - c. 2
 - d. 1
- 8) Arrow functions cannot accept parameters
- a. True
 - b. False
- 9) Arrow functions are longer versions of functions
- a. True
 - b. False

10) What will be the output of the following piece of code?

```
const greeting (name) => {  
  console.log(`Hello ${name}, how are  
    you today?`);  
}  
  
greeting();
```

- a. Hello name, how are you today?
- b. Hello undefined, how are you today?
- c. SyntaxError cannot initialize in const declaration
- d. Hello, how are you today?

Chapter 9 - Debugging

Error Messages

You will have come across error messages at some point when you were coding. New coders might be intimidated by this however, you will find that having errors are actually quite common and even the best coders in the world will come across these errors. Take a look at the piece of code below. Can you identify what the error is?

```
const multiply = (a, b) => {  
  console.log(a * b);  
})  
  
multiply(5, 10);
```

In this example, we are getting something called a **syntax error**. These errors usually occur when a keyword is spelled incorrectly or if a semi-colon, curly bracket, parenthesis or anything similar has been added or omitted incorrectly. In the example above, a closing parenthesis has been incorrectly added right after the closing curly bracket. Take a look at the snip of the error message below. The message informs us about what type of error it is, what the file name is, what line number the error is in and what might be causing the issue. In this case, it is a syntax error, in the file `section7.js`, on line number 93 and we have accidentally added a closing parenthesis.



✖ Uncaught SyntaxError: Unexpected token ')'
section7.js:93

Let's take a look at another type of error, a **reference error**.

```
var greeting = 'Hello World!';  
  
console.log(greetin);
```

Can you see what the issue is in the example above? We have declared a variable called `greeting`, however when we try to log it to the console, we have misspelled the variable name. We are then given a reference error, which tells us that our variable does not exist. If you receive this error, go back and check your variables to make sure they have been declared properly.

```
✖ ▶ Uncaught ReferenceError: section7.js:93  
    greetin is not defined  
    at section7.js:93
```

Another common error is a **type error**.

```
function multiply(a, b) {  
    console.log(a * b);  
}  
  
multiply.push();
```

In this example, we have incorrectly tried to use the `push` method when calling the `multiply` function. We are therefore thrown a type error because it is the wrong operation to use in this case.

✖ ▶ Uncaught TypeError: section7.js:102
multiply.push is not a function
at section7.js:102

How To Debug – Example

Let's go over an example on how we can debug a piece of code with the information we get from the console window. Take a look at the piece of code below:

```
7 ▼ function largerNumber(number1, number2;) {  
8 ▼     if (number1 > number) {  
9         return `${number1} is greater than  
           ${number2}`;  
10 ▼     } else if (number2 > number1) {  
11         return `${number3} is greater than  
           ${number1}`;  
12 ▼     } else {  
13         return `${number1} is the same as  
           ${number2}`;  
14     }  
15 }  
16  
17 console.log(largerNumber(6, 1));
```

You may have already noticed the errors in the code, but let's pretend that you ran the code anyway because you were unaware of the errors. This is the error message we would get:

✖ Uncaught SyntaxError: section8.js:7
Unexpected token ';'

Right away we get a syntax error. This means that we may have a typo somewhere or added an extra symbol in our code. The first thing we need to look at here is what is causing the error. The console tells us that there is an unexpected semi-colon somewhere. Next, look at the name

of the JavaScript file. In this case, it is section8.js. Finally, look at the line number, which is number 7 in this example. So we now know, that in section8.js, we have an unexpected semi-colon on line number 7. If you go back and take a look at the screenshot, you can see that there is a semi-colon right after the second function parameter. Let's remove that and try to run our code again.

```
7 ▼ function largerNumber(number1, number2) {  
8 ▼     if (number1 > number) {  
9         return `${number1} is greater than  
           ${number2}`;  
10 ▼     } else if (number2 > number1) {  
11         return `${number3} is greater than  
           ${number1}`;  
12 ▼     } else {  
13         return `${number1} is the same as  
           ${number2}`;  
14     }  
15 }  
16  
17 console.log(largerNumber(6, 1));
```

```
✖ ▶ Uncaught ReferenceError: section8.js:8  
   number is not defined  
     at largerNumber (section8.js:8)  
     at section8.js:17
```

This time, we have a reference error in our code. This suggests that we may have typed in a variable that does not exist, or that we have misspelled a variable name. The error message tells us that a variable called number has not been defined, within the largerNumber function. The error is now on line 8. So if we take a look at line 8, you can see that in the if condition, the second operand should be number2 and not just number. This is because we do not have an argument called number, so

the computer is trying to find an argument that does not exist. Let's fix that and see what happens.

```
7 ▼ function largerNumber(number1, number2) {  
8 ▼     if (number1 > number2) {  
9         return `${number1} is greater than  
           ${number2}`;  
10 ▼     } else if (number2 > number1) {  
11         return `${number3} is greater than  
           ${number1}`;  
12 ▼     } else {  
13         return `${number1} is the same as  
           ${number2}`;  
14     }  
15 }  
16  
17 console.log(largerNumber(6, 1));
```

6 is greater than 1 [section8.js:17](#)

The code works! We are getting the correct output as 6 is a greater number than 1. Let's now try to enter other arguments when we call our function, so that the second argument will be greater than the first one.

```
17 console.log(largerNumber(5, 7));
```

✖ ▶ Uncaught ReferenceError: [section8.js:11](#)
number3 is not defined
at largerNumber ([section8.js:11](#))
at [section8.js:17](#)

We now have another error! It seems that we may have misspelled another variable in our code. Looking at the error, we can see that there is a variable named number3 on line 11. Within the code block of our else if statement, we have returned the wrong parameter. That

parameter should be called number2 not number 3. So, let's change that and see what happens.

```
7 ▼ function largerNumber(number1, number2) {  
8 ▼     if (number1 > number2) {  
9         return `${number1} is greater than  
           ${number2}`;  
10 ▼     } else if (number2 > number1) {  
11         return `${number2} is greater than  
           ${number1}`;  
12 ▼     } else {  
13         return `${number1} is the same as  
           ${number2}`;  
14     }  
15 }  
16  
17 console.log(largerNumber(5, 7));
```

7 is greater than 5

section8.js:17

We are now getting the correct output! This example also showed that your code may still have errors, even if it works. You might be wondering why the code worked the first time we ran it. If you will remember from the lecture on conditional statements, each condition will be checked and if that condition is true, then the piece of code will execute. When we ran the code the first time, as the first argument was greater than the second, the first condition was met, and the rest of the code was ignored. This is why testing your code is imperative, because your code may still contain bugs and still work.

Using console.log() To Debug

In the previous example, we saw how codes can run even if they contain bugs. The previous code however, gave us an error when we tried to pass in arguments where the second number was greater than the first. The

code did not run, and we were able to easily figure out what the problem was based on the error that was shown on the console. As you become a more experienced coder, you will find that there will be occasions where the console does not show an error, but the output is still incorrect. When this happens, your code may have something known as a silent bug. These bugs are generally difficult to find, because they are not obvious errors that can cause the code to malfunction. Let's take a look at an example of a silent bug.

```
function greeting(season) {  
    season.toLowerCase();  
    if (season === 'summer') {  
        return `It's so hot outside!`  
    } else if (season === 'winter') {  
        return `It's so cold outside!`  
    } else if (season !== 'spring') {  
        return `The weather is perfect!`  
    } else {  
        return `I love this weather!`  
    }  
}  
  
console.log(greeting('SUMMER'));  
// Output: The weather is perfect!
```

In the example above, we have a function that accepts a parameter called season and will return a statement based on what season is passed as an argument. You will notice that the code contains the toLowerCase function. This is just there to transform the arguments passed in to lowercase. Since JavaScript is case-sensitive, it is important that we match the case to the case of the conditions. Running the code works and the console does not throw any errors. However, we can see that the output is not what we had expected. How do we find out what the

error is? We can use the `console.log` function that you learned about at the start of the book! Let's see how that works. Let's first log to the console the season parameter at the start of the function, so that we know that our function is accepting the arguments we pass in.

```
function greeting(season) {  
  console.log(season);  
  season.toLowerCase();  
  if (season === 'summer') {  
    return `It's so hot outside!`  
  } else if (season === 'winter') {  
    return `It's so cold outside!`  
  } else if (season !== 'spring') {  
    return `The weather is perfect!`  
  } else {  
    return `I love this weather!`  
  }  
}  
  
console.log(greeting('SUMMER'));  
// Output: SUMMER  
// Output: The weather is perfect!
```

We can see that the argument we pass in is accepted by the function. Now let's check if our argument is being correctly transformed to a lower case string. Let's use the `console.log` function and log the season variable after the `toLowerCase` function is called.

```
function greeting(season) {  
    season.toLowerCase();  
    console.log(season);  
    if (season === 'summer') {  
        return `It's so hot outside!`  
    } else if (season === 'winter') {  
        return `It's so cold outside!`  
    } else if (season !== 'spring') {  
        return `The weather is perfect!`  
    } else {  
        return `I love this weather!`  
    }  
}  
  
console.log(greeting('SUMMER'));  
//Output: SUMMER  
// Output: The weather is perfect!
```

Now we are seeing something that is incorrect. In this example, the argument we passed in should have been transformed into lowercase letters. So, we have found a silent bug. We need to actually save the result of the `toLowerCase` function to a variable before we can use it. So let's save it to a variable called `lowercase`. Since we are going to compare our conditions to the value of the variable, we will also have to replace the `season` parameter with the `lowercase` variable name. Let's go ahead and do that.

```
function greeting(season) {  
  var lowercase = season.toLowerCase();  
  console.log(lowercase);  
  if (lowercase === 'summer') {  
    return `It's so hot outside!`  
  } else if (lowercase === 'winter') {  
    return `It's so cold outside!`  
  } else if (lowercase !== 'spring') {  
    return `The weather is perfect!`  
  } else {  
    return `I love this weather!`  
  }  
}  
  
console.log(greeting('SUMMER'));  
// Output: summer  
// Output: The weather is perfect!
```

We've now saved the result of the `toLowerCase` function to a variable and logged the value of the variable to the console. We now get the correct output and the code works! But let's go ahead and test all of the possible conditions.

```
function greeting(season) {  
  var lowercase = season.toLowerCase();  
  if (lowercase === 'summer') {  
    return `It's so hot outside!`  
  } else if (lowercase === 'winter') {  
    return `It's so cold outside!`  
  } else if (lowercase !== 'spring') {  
    return `The weather is perfect!`  
  } else {  
    return `I love this weather!`  
  }  
}  
  
console.log(greeting('spring'));  
// Output: I love this weather!
```

We are now getting another error here! When we enter the value of spring, the output we are receiving is the incorrect one. Let's try to figure this out. The output we are getting is the else statement, which means the condition we set for spring was not met. Within the code block of the else statement that is being executed, let's log to the console the value of the lowercase season.

```
function greeting(season) {  
  var lowercase = season.toLowerCase();  
  if (lowercase === 'summer') {  
    return `It's so hot outside!`  
  } else if (lowercase === 'winter') {  
    return `It's so cold outside!`  
  } else if (lowercase !== 'spring') {  
    return `The weather is perfect!`  
  } else {  
    console.log(lowercase);  
    return `I love this weather!`  
  }  
}  
  
console.log(greeting('spring'));  
// Output: spring  
// Output: I love this weather!
```

The argument we are passing in should have matched the condition where the lowercase variable equals spring. So there must be an issue within that condition. We can clearly see that we set a strict not equals to operator instead of a strict equals to operator! So, we can simply fix that mistake and that should resolve our issue.


```
function greeting(season) {  
  var lowercase = season.toLowerCase();  
  if (lowercase === 'summer') {  
    return `It's so hot outside!`  
  } else if (lowercase === 'winter') {  
    return `It's so cold outside!`  
  } else if (lowercase === 'spring') {  
    return `The weather is perfect!`  
  } else {  
    return `I love this weather!`  
  }  
}  
  
console.log(greeting('spring'));  
// Output: The weather is perfect!
```

The bugs have been fixed and the code now works! So you can now see how powerful the `console.log` function is. Using it at the correct places when debugging can be extremely helpful. As a programmer, you will encounter several errors and using these tools will help you in correcting your code.

Chapter 8 Quiz

- 1) What is a syntax error?
 - a. An error shown when a numeric value or parameter is outside of its valid range
 - b. An error shown when the incorrect operator is used in a specific piece of code
 - c. An error shown when a keyword is misspelled, or a symbol is used incorrectly
 - d. An error shown when an undeclared variable is referenced somewhere in the code
- 2) What is a reference error?
 - a. An error shown when a numeric value or parameter is outside of its valid range
 - b. An error shown when the incorrect operator is used in a specific piece of code
 - c. An error shown when a keyword is misspelled, or a symbol is used incorrectly
 - d. An error shown when an undeclared variable is referenced somewhere in the code
- 3) What is a type error?
 - a. An error shown when a numeric value or parameter is outside of its valid range
 - b. An error shown when the incorrect operator is used in a specific piece of code
 - c. An error shown when a keyword is misspelled, or a symbol is used incorrectly
 - d. An error shown when an undeclared variable is referenced somewhere in the code
- 4) Errors in your code can exist even if the code can be executed
 - a. True

b. False

5) The console.log function can be a powerful tool to use when debugging code

a. True

b. False

6) What is the error in the piece of code below?

```
for (let i = 0; i <= 100; i++) {  
    console.log(i);  
}
```

a. Semi-colons are used instead of commas

b. The opening and closing curly brackets are in the wrong location

c. The var keyword should be used instead of let to declare the initializing variable

d. There are no errors in the code

7) What is the error in the piece of code below?

```
let x = 10;  
let y = 5;  
  
IF (x > y && x !== y) {  
    console.log('The condition is true');  
} else {  
    console.log('The condition is false');  
}
```

a. The variables have been declared incorrectly

b. The opening and closing curly brackets are in the wrong location

c. The if keyword is in uppercase when it should be in lowercase

d. There are no errors in the code

8) What is the error in the piece of code below?

```
function multiply(a, b) {  
    return a * c;  
}
```

- a. There are no errors in the code
- b. The function body should not contain the return statement
- c. The wrong parameter has been entered in the function body
- d. The function name must not be multiply as it is a keyword in JavaScript

Quiz Answers

Chapter 2

- 1) B
- 2) A
- 3) D
- 4) A
- 5) C
- 6) C
- 7) A
- 8) A

Chapter 3

- 1) B
- 2) C
- 3) A
- 4) D
- 5) B
- 6) C
- 7) C
- 8) B
- 9) A
- 10) B
- 11) B

Chapter 4

- 1) D
- 2) B
- 3) B
- 4) A
- 5) B

- 6) D
- 7) A
- 8) A
- 9) C

Chapter 5

- 1) B
- 2) B
- 3) D
- 4) B
- 5) B
- 6) A
- 7) B
- 8) A
- 9) C
- 10) B
- 11) A
- 12) A
- 13) D

Chapter 6

- 1) A
- 2) C
- 3) B
- 4) B
- 5) C
- 6) A
- 7) D
- 8) B
- 9) C

Chapter 7

- 1) A
- 2) B
- 3) D
- 4) B
- 5) B
- 6) B
- 7) A
- 8) D
- 9) D

Chapter 8

- 1) A
- 2) D
- 3) C
- 4) C
- 5) B
- 6) A
- 7) A
- 8) A

Chapter 9

- 1) C
- 2) D
- 3) B
- 4) A
- 5) A
- 6) D
- 7) C
- 8) C