



# THE JAVASCRIPT TECHNICAL INTERVIEW WORKBOOK

---

**400+ CODING EXERCISES**

KAURESS

Copyright © 2020 by Jannat Kaur

All rights reserved. No part of this book may be reproduced or used in any manner without written permission of the copyright owner except for the use of quotations in a book review. For more information, address: [kauress@kauress.me](mailto:kauress@kauress.me)

FIRST EDITION

Don't just believe- practice

## Why I wrote this workbook?

As a coding bootcamp instructor for the past 4 years, some of the questions that I've been asked concerning employment are:

1. Will I get a job after this?
2. How many projects do I need to complete to start interviewing?
3. How can I test my JavaScript knowledge?
4. How should I prepare for technical interviews, and what will I be asked?

Circumventing life issues to get employed!

The best way for junior developers to circumvent the issue of not having enough experience or projects, and still get employed is to do plenty of coding exercises (on old and newer ES features) without having done whole projects on them. *This demonstrates a practical understanding of core and advanced JavaScript concepts.*

Often people have to balance multiple things while they learn to code including part-time jobs. Therefore, they're on a time crunch and may not be able to commit to doing projects. Some people leave their current jobs to learn how to code. The more time they spend out of webdev employment status means they dip into their savings and deplete their bank balances.

This workbook focuses on circumventing these issues in a practical and straightforward way.

*To help my own students and others I have filtered, sorted and organized vast amounts of information in a way to make it easier for you to LEARN and PREPARE at the same time.*

- ✓ Practice over 400+ coding exercises. **Answers provided.**
- ✓ Get down to the nitty-gritty of JavaScript. Practice the often neglected parts such as the **this** keyword, hoisting, and more.
- ✓ Answer questions about advanced ES topics without having done whole projects on them.
- ✓ Start preparing to interview ASAP!

# How to use this workbook?



## 1. Want to learn concepts and practice coding exercises?

- Coding exercises are present within each section in each chapter. Therefore, you can learn and practice at the same time
- Each code example is explained line by line in bullet points

## 2. Want to jump straight into the coding exercises?

- All the coding exercises and answers have been compiled separately.
  - You can reference the guide if you get stuck, or want clarity on a particular topic
3. Practice every day, preferably at the same time and place. This will help you form the habit of problem solving daily.
  4. There is a worksheet provided at the end of every chapter. This is a self-assessment worksheet which will help you keep track of your progress.

## Reference styles

- This is how code styles will look:  
This can be used in place of the `concat()` method
- This is how keywords will look:  
let username

- This is how italics will look:

To help your learning of the different ways to **refine** your view.

- This is how syntax is shown:

**Syntax :** `typeof(operand)`

## Content

Why I wrote this workbook? .....	2
How to use this workbook? .....	3
Reference styles .....	3
1.1 What you need to know about ECMA Script.....	15
Exercises.....	16
1.2 – Variables and Data Types .....	17
1.2.1 Scope.....	17
1.2.2 Var .....	19
Var variables in local scope .....	20
Var variables in block scope .....	20
1.2.3 Let .....	21
1.2.4 Const .....	23
1.2.5 Re-assignment and re-declaration .....	24
Var .....	24
Let .....	25
Const.....	25
1.2.6 Advantages of const and let over var .....	27
1.2.7 Data types .....	27
1.2.9 Why can we use methods such as toUpperCase() on primitive data types? .....	30
1.3 – Operators and Comparisons .....	38
1.3.1 Arithmetic operators .....	39
Exercises.....	40
Answers:.....	40
1.3.1.2 Associativity.....	40
Exercises.....	42
Answers.....	42
1.3.2 Assignment operators .....	43
Exercises.....	44
1.3.3 Comparison operators .....	45

1.3.3.1 Strict equality (====) and strict inequality (!==) operators .....	45
1.3.3.2 Loose equality (==) and inequality (!=) operators:.....	46
Answers:.....	47
1.3.4 Logical operators .....	48
1.3.4.2 Logical AND (&&) .....	49
1.3.4.3 Logical OR    .....	51
Exercises.....	53
Answers:.....	55
1.3.5 String operator .....	56
1.3.5.2 Template literals .....	57
1.....	59
1.3.6 Void Operator .....	59
1.3.7 Typeof Operator .....	60
1.3.7.3 Checking for null and undefined .....	62
1.3.7.5 Checking for NaN .....	64
Answers.....	65
1.3.8 Comparisons .....	66
1.3.8.1 Checking for truthy and falsy values .....	67
1.3.8.2 Comparing truthy and falsy values with == .....	68
1.4 – Coercion .....	72
1.4.1 Definition.....	72
1.4.2 String coercion .....	72
1.4.3 Numeric coercion .....	75
1.4.3.2 Explicit number coercion .....	76
Answers:.....	77
1.4.4 Boolean coercion.....	78
1.4.4.1 Implicit boolean coercion.....	78
1.4.4.2 Explicit boolean coercion .....	79
Exercises.....	80
1.5 Iteration .....	80

1.5.1 For statement.....	81
1.5.2 While statement .....	86
Answers.....	88
1.5.3 Do while statement .....	88
Exercises.....	90
1.5.4 Break and continue statements .....	92
Answers.....	94
1.6 – Conditional statements .....	95
1.6.1 If Statements .....	95
1.6.2 Else Statements .....	96
1.6.3 Else If Statements .....	97
1.6.4 Switch statements .....	98
1.6.5 Ternary operator .....	101
Summary .....	107
2.1 Hoisting .....	108
Exercises.....	109
Answers.....	109
2.1.2 Var hoisting.....	109
Exercises.....	111
2.1.3 Let and const hoisting.....	113
Exercises.....	114
2.1.4 Function hoisting .....	115
2.1.5 Precedence for variable and function hoisting .....	115
Answers.....	118
2.2 Error handling with try/catch/throw and finally.....	118
2.2.1 Try/Catch Block .....	118
2.2.2 Throw statement .....	119
2.2.3 Finally statement .....	121
2.3 Types of errors .....	121
2.4 Strict mode .....	123

2.4.1 Enabling strict mode .....	123
2.4.2. Use cases of strict mode.....	124
2.4.3 Advantages of strict mode.....	127
Exercises.....	127
Summary .....	128
3.1 Objects in JavaScript.....	130
3.2 Object creation .....	131
3.2.1 Object literal syntax .....	132
Exercises.....	134
Exercises.....	136
3.2.2 new() keyword .....	139
3.2.2.2 User defined object constructor function.....	140
3.2.3 Object.create() method.....	143
3.2.4 Object.assign() method .....	146
3.2.5 ES6 Classes .....	149
3.3 Object iteration.....	153
3.4 this keyword .....	158
3.4.1 Global (default) binding .....	158
3.4.2 Implicit binding.....	159
3.4.3 Explicit binding .....	161
3.5 Prototype and Inheritance .....	170
3.5.1 Prototype.....	170
3.5.2 Prototype chain .....	173
3.5.3 Prototype Inheritance .....	176
3.6 Classes.....	182
3.6.1 Class declarations .....	182
3.6.2 Constructor method .....	183
3.6.5 Public and private fields .....	190
Answers.....	193
Exercises.....	196

Exercises.....	202
3.6.6 Inheritance with extends .....	203
Answers.....	205
3.6.7 Inheritance with super keyword.....	205
Exercises.....	207
3.6.8 InstanceOf .....	210
3.6.9 Constructor property .....	211
3.7 Accessors: Getters and Setters .....	212
3.7.1 Getters .....	213
1. Default method syntax.....	214
3.8 Copying objects with shallow and deep copies .....	224
Shallow copy.....	225
Deep copy .....	225
3.8.1 Copying objects with Shallow copy .....	226
3.8.1.2 Spread operator .....	227
3.8.1.3 Object.assign .....	230
Exercises.....	233
3.8.2 Copying objects with deep copy .....	237
Answers.....	239
Summary .....	239
4.1 Arrays in JavaScript .....	240
4.1.1 Definition .....	240
4.1.2 Indexing in arrays.....	240
4.1.3 Array values.....	241
Exercises.....	241
1. What is the fundamental difference between an array and an object?.....	241
Answers.....	241
4.2 Array declaration.....	241
Exercises.....	242
Answers.....	242

4.2.2 Array() constructor.....	242
Exercises.....	243
Answers.....	244
Exercises.....	244
Answers.....	244
Exercises.....	245
Answers.....	245
4.3 Array properties .....	245
4.3.1 Length property of an array .....	245
Exercises.....	247
Answers.....	247
4.3.2 Constructor property of an array.....	248
4.3.3 Prototype property of an array .....	248
Exercises.....	249
Answers.....	249
4.4 Array Methods .....	249
4.4.1 Adding an array element .....	250
4.4.1.1 Array.prototype.push() .....	250
4.4.1.2 Array.prototype.unshift() .....	250
4.4.1.3 Array.prototype.splice () .....	251
4.4.1.4 Array.prototype.concat() .....	251
4.4.2 Deleting an array element .....	252
4.4.2.1 Array.prototype.pop() .....	252
4.4.2.2 Array.prototype.shift().....	253
4.4.2.3 Array.prototype.splice().....	253
4.4.2.4 delete keyword .....	254
4.4.2.5 Resetting an array .....	254
4.4.3 Changing array elements .....	255
4.4.4 Slicing an array.....	255
4.4.5 Array.prototype.forEach ().....	256

4.4.5 Array.prototype.some()	257
4.4.6 Array.prototype.find()	258
4.4.6 Array.prototype.every()	260
4.4.7 Difference between some(), find() and every()	261
4.4.8 Array.prototype.sort()	261
Exercises	263
Answers	271
Summary	276
5.1 Destructuring Arrays	277
Exercises	279
Answers	280
5.2 Spread operator	281
5.2.2 Spread operator	281
5.2.2 Merge arrays	281
5.2.3 Copy arrays	282
5.2.4 Convert a string into an array	283
5.2.5 Math object	283
Exercises	284
Answers	284
5.3 Map, Reduce and Filter methods	285
5.3.1 map()	285
5.3.1.2 Map vs for-loop and forEach	286
Exercises	287
Answers	287
5.3.2 Reduce	287
Exercises	290
Answers	290
5.3.3 filter()	291
Exercises	293
Answers	294

5.4 Chaining Map, Reduce and Filter methods .....	296
Exercises.....	300
Answers.....	301
5.5 Multi-dimensional arrays .....	302
5.5.1 Declaring multi-dimensional arrays .....	302
5.5.2 Accessing items in multi-dimensional arrays .....	304
5.5.3 Iteration in multi-dimensional arrays.....	304
5.5.4 Multi-dimensional array methods .....	305
5.5.4.1 pop() .....	305
5.5.4.2 push() .....	305
5.5.4.3 indexOf().....	305
5.5.4.4 splice().....	305
Exercises.....	306
Answers.....	307
Summary .....	308
6.1 Functions as objects .....	309
Answers.....	311
6.2 Functions .....	312
6.3 Function expressions .....	319
Exercises.....	321
Answers.....	323
6.4 Immediately invoked function expressions (IIFE) .....	323
Exercises.....	325
6.5 Anonymous functions .....	327
6.6 Arrow functions .....	329
6.6.1 Syntax.....	329
6.6.1.1 No parameters.....	329
6.6.1.2 Single parameter.....	329
6.6.2 <b>this</b> binding .....	330
6.6.3 When not to use arrow functions .....	331

Exercises.....	331
6.7 Rest syntax .....	333
Exercises.....	335
Answers.....	337
6.8 Callback functions .....	338
6.8.2 Why? .....	339
6.8.4 Anonymous callback function .....	341
6.8.5 Callback function expression.....	341
6.8.6 Asynchronous callback functions.....	343
Summary .....	347
7.1 What is the DOM?.....	349
Exercises.....	351
Answers.....	352
7.2 Properties of the DOM.....	352
Exercises:.....	356
Answers.....	357
7.3 DOM methods .....	358
7.3.1 Accessing elements.....	358
7.3.1.1 getElementById.....	358
7.3.1.2 getElementsByClassName.....	359
7.3.1.3 querySelector .....	359
7.3.1.4 querySelectorAll.....	360
7.3.1.5 getElementsByTagName .....	360
7.3.2 Deleting elements.....	361
7.3.3 Creating DOM elements .....	362
7.3.3.1 createElement() .....	362
7.3.3.2 createTextNode() .....	364
7.3.4 Inserting DOM elements .....	364
7.3.3.2 insertBefore() .....	365
7.3.3. replaceChild() .....	366

Exercises.....	367
Answers.....	369
7.4 Iterating through DOM nodes .....	372
Exercises.....	373
Answers.....	374
Summary .....	374
8.1 What is the call stack? .....	375
8.2 Execution context.....	379
8.2.1 Global execution context.....	379
8.2.3.2 Execution phase .....	381
8.3 Event Loop.....	381
Summary .....	383

# The Basics -1

*“For the things we have to learn before we can do them, we learn by doing them.” — Aristotle*

This chapter lays the foundations for understanding the deeper parts of JavaScript presented in later chapters. It will also give you the advantage of understanding the underlying mechanisms for basic concepts. Practice coding problems are given at the end of each section in each chapter to allow you to solidify your knowledge and get better at coding in JavaScript.

Often the basic aspects of JavaScript are skipped over in favor of the more advanced parts. However, learning the fundamentals well enough gives you the advantage of being able to grasp libraries and frameworks such JQuery, React.js, Vue.js, and others more easily.

You might have read about these concepts and even used them in your projects before. In which case I suggest that you reinforce your skills by practicing the coding exercises at the end of each section. You should use the information in the chapter to guide you. The exercises go beyond the basics; hence a previous background in JavaScript helps.

In this chapter we're going to cover the following main topics:

- What you need to know about ECMAScript
- Data types and variables
- Operators & Comparisons
- Type Coercion
- Iteration
- Conditional statements

## 1.1 What you need to know about ECMA Script

ECMA which is short for **European Computer Manufacturer's Association** is an international organization that creates standards for technologies. One of these standards is called ECMA-262. ECMA-262 is the standard for creating a general-purpose scripting language. This standard is commonly called ECMAScript specification. This specification contains rules and guidelines for creating a scripting language. JavaScript is one of the major implementations of the ECMAScript Specification. Other implementations include JScript

and ActionScript. The committee that works on the ECMAScript specification is called TC39 (Technical Committee number 39).

ES (ECMAScript) followed by a number, references the specific version of ECMAScript, for example, ES1, ES2, etc. So far there are 10 versions of ES from ES1 to ES10. ES.Next references the next upcoming version of ES.

Except for ES1 and ES2, each version of ES introduces a new set of features to the language. For example, regular expressions and try - catch exception handling were added to ES3. Whereas ES6 additions include arrow function expressions, **const** and **let** keywords and more.

Therefore, JavaScript continually evolves, with yearly additions post ES6. We will explore major ES updates of each version in this interview workbook. ***It is important to know what features are available for you to use rather than memorizing what ES version they belong to.***

Yearly ES additions allow browser vendors and developers to implement features at a constant rate.

Each browser uses a different JavaScript engine. V8 is Chrome's JavaScript engine, Firefox uses SpiderMonkey and Chakra is Internet Explorer's JavaScript engine. Each browser adopts the latest ES features at different rates. Therefore, when working with the latest ES features you can:

- Check browser support for a specific feature
- Transpile your code

To check browser support for a particular ES feature you may use

<https://www.caniuse.com>. The website provides a table of browsers and their versions that support a particular feature that is searched for.

To ensure that the code that you write works on all browsers, it is common to transpile your JavaScript code. Transpiling translates your JavaScript code from one version to a previous ES version. This in turn allows you to use the latest ES features that haven't been implemented uniformly across all browsers. Babel is a common transpiler <https://www.babeljs.io>. Babel is used to convert ES6 and onwards code into a backward compatible version so that it may run on older browsers as expected.

## Exercises

1. What are the two ways to ensure your code is compatible in most browsers?
2. What is the difference between ECMAScript and JavaScript?

## Answers

1. To ensure that your code is backward compatible with older browsers, you can transpile your code using a transpiler such as Babel. If developing an application for users of a specific device or browser it is wise to check if a particular ES feature is supported by checking online on <https://www.caniuse.com>.
2. ECMA Script is a general-purpose scripting language. Whereas, JavaScript is a major implementation of the ECMAScript specification

## 1.2 – Variables and Data Types

Variables can be defined with the following keywords:

1. **var**
2. **let**
3. **const**

**let** and **const** are ES6 additions and it is preferable to use them instead of **var**, for reasons discussed later on. The following table displays the difference between the 3.

Name	Scope	Re-assignment
Var	Function & global scope	Yes
let (ES6)	Block scope	Yes (outside current scope)
const (ES6)	Block scope	No

The 3 types of variables declarations will be discussed in reference to their scope and whether values can be re-assigned to them. But before we do so, let's first discuss what "scope" means.

### 1.2.1 Scope

The word "scope" itself means "extent" or "range" to which a subject matter is limited to. In context to coding this means, that whether you can use a variable or not, is determined by where it is declared in your code. There are 4 types of scope in a JavaScript document, for example, `script.js`. These are global, local, block, and lexical scope. Let's go over each one of these.

## 1. Global scope

Inside a JavaScript document, global scope is the area outside any functions and blocks of code. A block of code is denoted by the presence of opening and closing curly braces `{ }`. For example:

```
//Global scope  
function x() {  
}
```

Here, the space outside of `function x` is called global scope.

## 2. Local scope

Local scope refers to variables that are declared and/or assigned a value within a function.

For example:

```
function localScope() {  
    // local scope  
}
```

Anything inside the opening and closing curly braces of `function localScope` has local scope.

## 3. Block scope

Block scope is defined as the area within curly braces `{ }`. For example inside of, if-else conditional statements switch conditions, for and while loops.

```
{  
    // block scope  
}
```

## 4. Lexical scope

Within a function, any code blocks and child functions that have access to the variables defined inside the main parent function's scope are said to have lexical scope. Down below is an example of variable declaration and lexical scope:

```

function parentScope() {
    //variables declared here
    function child() {
        //variables can be accessed here
    }
    child();
}

```

Variables declared within the main `parentScope` function are accessible to the `child()` function.

With that, let's move on to understanding the 3 different types of variable declarations (`var`, `let` and `const`) with context to their differing scopes.

### 1.2.2 Var

Now we will look at variables declared with the `var`, `let` and `const` keywords in the context of global and local scopes. Let's tackle `var` to start with.

#### Var variables in global scope

Global variables are declared outside of any functions or blocks of code and, they are within the scope of the browser's window object. They can be accessed anywhere in your code. For example, in the following code, `var one` is declared globally and it is accessible from within the `if` conditional block in `function consoleOne()`:

```

var one = 1;

function consoleOne() {
    if (one) {
        console.log(one); // 1
    }
}

consoleOne();

```

The number `1` is logged to the console as the global variable `one` can be accessed from inside function `consoleOne()`.

### Var variables in local scope

A **var** variable inside a function has local scope. It cannot be accessed anywhere outside the function within which it is declared. This is also called function scope. Here **var two** is declared inside **function num()**:

```
function num() {  
  var two = 2;  
  console.log(two); // 2  
}  
  
console.log(two); //ReferenceError: two is not defined
```

When we try and log **var two** outside the function, a **ReferenceError** is returned as we're trying to reference a variable outside of its local scope, which is not possible.

### Var variables in block scope

You would think **var** declared inside a block within **curly braces {}**, is accessible only within the curly braces. However, that is not so. If the block of code is not within a function, a block scope variable is accessible throughout your code as **var does not have block scope**.

Analyze the following code carefully. Notice that variable **three** which is defined within block scope is accessible as a global variable to the function **logThree** and the last code block **{ }**.

```
{  
  var three = 3;  
}  
  
console.log(three); // 3  
  
function logThree() {  
  console.log(three); //3  
}  
  
{  
  console.log(three); //3  
}  
  
logThree();
```

The code block demonstrates that variable `three` which is declared in block scope `{ }`, is available in:

- Global scope
- function `logThree()`
- The last code block `{ }`.

**Note:** On analyzing variables declared in the global, local, and block scope, the following is concluded:

1. Global `var` variables can be accessed from anywhere within your code.
2. Variables declared with the `var` keyword inside a function have local scope and can only be accessed within that particular function.
3. `var` variables declared in block scope that are outside functions, can be accessed globally and locally within functions and other blocks of code.

You need not worry about memorizing the above points. We will do exercises to reinforce these concepts. And when we discuss variables declared with `let` and `const`, you will understand why it is better to use those instead.

### 1.2.3 Let

Variables declared with the `let` keyword have block-level scope and are inaccessible outside the block. Let's have a look at `let` in depth by analyzing some code:

Example 1:

```
{  
  let zoo = "Giraffe";  
  console.log(zoo); // "Giraffe"  
}  
  
console.log(zoo); // ReferenceError: zoo is not defined
```

Example 2:

```
function logNum() {  
  if (1 < 2) {  
    let one = 1;
```

```

    console.log(`This ${one} is scoped within the if block`);

    // "This 1 is scoped within the if block"

}

console.log(`This ${one} is scoped within the function
block`); //ReferenceError: one is not defined

}

logNum();

```

From this we can conclude the following:

- Variables declared with **let** are scoped to the immediate enclosing block of code **{ }**. In this case, the **if** block.
- The message logged to the console within the **if** block is **"This 1 is scoped within the if block"**
- Whereas, logging to the console within the function block gives the following message: **ReferenceError: one is not defined**. This is because **let variables cannot be accessed outside the if-block**.

Let's change **let** to **var** and see what happens:

```

function logNum() {

    if (1 < 2) {

        var one = 1;

        console.log(`This ${one} is scoped within the if block`);

    }

    console.log(`This ${one} is scoped within the function
block`);

}

logNum();

```

The two **console.log** statements are:

```

"This 1 is scoped within the if block"
"This 1 is scoped within the function block"

```

This is because `var` is scoped locally to the main function body. Therefore, it is available anywhere within that function. Whereas, `let` is scoped to its immediate enclosing block `{ }` scope, which in this case is an `if` block. Hence, it cannot be accessed outside of the `if` block.

### 1.2.4 Const

`const` like `let` has block scope. `const` refers to “constant” and is a signal that the value of a variable should remain unchanged. Let’s have a look at the block level functioning of `const` in `function fooFighter()`:

```
function fooFighter() {  
  if (1 < 2) {  
    const fooFoo = 'Fire Muncher';  
    console.log(fooFoo); // "Fire Muncher"  
  }  
  console.log(fooFoo);  
  //ReferenceError: fooFoo is not defined  
}  
  
fooFighter();
```

`const fooFoo` is scoped to the `if` block and is inaccessible outside the function. Therefore, the second `console.log` which is within the main body of the function returns a `ReferenceError`.

Now have a look at an example of lexical scoping with `const`:

```
const num = 10;  
  
function firstFunc() {  
  console.log(1 + num)  
}  
  
function secondFunc() {  
  const num = 3  
  firstFunc()  
}
```

```
secondFunc();
```

The following will be logged to the console:

```
11
```

Let's deconstruct what's happening:

- `secondFunc()` is called
- Inside `secondFunc`, `const num` is declared and initialized to a numeric value of `3`. Then `firstFunc()` is called
- Then function `firstFunc()` will run and log the number `11` to the console
- This is because `firstFunc()` looks for `const num` within its own scope and if not found, it will transverse up to the outer global scope. And `const num` within the global is initialized to a value of `10`.

## 1.2.5 Re-assignment and re-declaration

Now that we've discussed scope, we will look at re-assigning values to variables declared with `var`, `let` and `const` keywords.

### Var

Re-assignment of values and re-declaration of `var` variables is possible, we will see in the following examples.

#### Re-assignment

```
var love = 'Rocko';
love = 'Rockster'
console.log(love); // "Rockster"
```

`var love` is re-assigned a new value `'Rockster'` in the above code.

#### Re-declaration

```
var love = 'Chuggli';
var love = 'Muggli';
console.log(love); // "Muggli"
```

Above **var** `love` is re-declared and given a new value. Therefore, demonstrating that the re-declaration and re-assignment of **var** variables is possible.

## Let

**let** variables like **var** variables can also be re-assigned values. However, they can only be re-declared outside their current scope.

### Re-assignment:

The following is an example of re-assignment of a value to a variable declared with **let**:

```
let apple = 'apple';
apple = 'pineapple';
console.log(apple); // "pineapple"
```

### Re-declaration:

**let** variables can only be re-declared outside of their current scope as seen in the following example:

```
{
  let apple = 'apple';
}

let apple = 'pineapple';
console.log(apple); // "pineapple"
```

**let apple** is declared within block scope and then once again re-declared in global scope. The result of logging **apple** to the console reveals that re-declaration was successful, and now the new value of **apple** is the string '**pineapple**'.

## Const

Variables declared with **const** cannot be re-assigned values. However, like **let**, they can be re-declared outside their current scope.

### Re-assignment:

In the following example variable declared with **const** is re-assigned a new value:

```
const veggie = 'spinach';
veggie = 'peas';
console.log(veggie); //Uncaught TypeError: Assignment to
constant variable
```

**const** values cannot be re-assigned values therefore, the above re-assignment of the `const veggie` variable leads to an `Uncaught TypeError`.

Let's digress and look at a use-case of **const** re-assignment concerning objects. While you cannot re-assign values to a variable declared with **const**, you can however, change the value of a property belonging to an object that has been declared with the **const** keyword, as demonstrated in the following code:

```
const user = {
  name: "Kauress",
  age: 100
}
user.name = "Rocko";
console.log(user.name);
// "Rocko"
```

This demonstrates that the properties of objects that have been declared with **const** can be reassigned values. We will cover objects in detail in Chapter 3, *All about Objects*.

#### Re-declaration:

Variables declared with **const**, like **let** can be re-declared outside their current scope.

```
{
  const veggie = 'spinach';
}
const veggie = 'peas';
console.log(veggie); // "pea"
```

In the example, `const veggie` has block scope; its re-declaration is allowed outside in global scope.

## 1.2.6 Advantages of const and let over var

It is preferred to use **const** when you will not be re-assinging values. **let**, which is block-scoped is ideal for functions, loops, conditional statements, and other blocks of code. **var** can induce scoping issues and therefore should be avoided. Additionally **let** and **const** are descriptive and make code more readable. The general consensus is to use **let** over **var** and to use **const** for values that should not or will not change.

Moving on, we will now discuss the different data types in JavaScript.

## 1.2.7 Data types

Data types in software aim to represent or model data in the physical world. For example, a car in real life may be represented as the “object” data type in a web application such as Uber, Ola, or Lyft. JavaScript is loosely typed, which means that you don’t have to enforce a variable’s data type beforehand when declaring it. Instead, the data type is induced at run time. Compare this to other languages such as C++ and Java where the data type of a variable must be stated during variable declaration.

**There are 2 main categories of data types:**

1. Primitives
2. Object

We will explore both types moving on.

### 1.2.7.1 Primitive Data types:

The Mozilla Developers Network ([MDN](#)) defines a primitive data type as:

*“Data that is not an object and has no method”, i.e. no associated functions may be used on them.*

From this we can derive two properties of primitive data types:

1. Immutability
2. Copy by value.

#### 1. Immutability

Values that are primitive data types are immutable like neutrons or electrons of an atom which means they cannot be inherently changed. For example:

```
let universe = 10;  
universe = '10';
```

The variable `universe` is assigned a numeric value of `10` and then re-assigned a string value of `"10"`. The numeric and string values themselves cannot be mutated and for all purposes and are still a number and a string.

Even when you modify primitives like a string with methods, what is returned is in-fact a new string. *The original string value remains intact*. This is demonstrated in the following code.

```
let game = 'Super Mario';
game.toUpperCase() // "SUPER MARIO"
console.log(game) // Super Mario
```

When the `toUpperCase()` method is applied to the `game` variable, the original value remains the same. This is exhibited by when `game` is logged to the console after using the `toUpperCase()` method. Only an instance of the string wrapper object changes. This is covered in depth in section 1.2.9 of this chapter, so if you don't get it right away, *don't worry!*

## 2. Copy by value

Primitive data types are copied by value and not reference. This means a copy of the original value is made in memory and assigned to a new variable. For example, let's deconstruct what is happening in the example:

```
let a = 10;
let b = a;
a = 100;
console.log(b) // 10
```

In the code above:

- Variable `a` is assigned a numeric value of `10`
- Variable `b` is assigned a copy of the original value of `a` which is `10`.
- The value of `a` is then changed to `100`
- `b` still references the original value of variable `a` which was copied over to variable `b` (`10`).

### 1.2.8 Primitive data types

The following table displays the 7 primitive data types:

Data type	Definition	Example
1. String	Characters enclosed with single or double quotation marks	'Rocko' "Rocko"
2. Number	This type represents 64-bit floating numbers. Numbers may be positive or negative integers, decimals, or exponential numbers. Numbers larger than $2^{53}$ cannot be represented by the number data type.	10.89
3. BigInt (ES6)	This data type supports numbers of arbitrary length i.e. they can be larger than $2^{53}$ . A number represented by <b>BigInt</b> has <b>n</b> at the end	9007199254740995n
4. Boolean	A logical data type that can either be represented by true (1) or false (0)	let newUser = true;
5. Null	The null data type can be a bit confusing. It is used to represent the absence of a value. <i>Therefore, in your program, you must intentionally set the value of a variable to be non-existent with null.</i>	let discountForever = null;
6. Undefined	Variables that have been declared but no value has been assigned to them as yet are of the <b>undefined</b> type.	let firstName;
7. Symbol (ES6)	Symbols are unique identifiers that can be created by using the global	const mySymbol = Symbol();

**Symbol()** factory function. Symbols are often used to identify the properties of an object and avoid name clashes between properties.

### 1.2.9 Why can we use methods such as `toUppercase()` on primitive data types?

You may have heard of the very popular saying “Everything in JavaScript is an object”. This is because JavaScript readily coerces between primitives and objects, subsequently allowing you to use methods such as `toUpperCase()` on say a string data type. Except for Null and Undefined data types, all other primitives have an associated object equivalent.

Let me backtrack a bit here and explain what a method is. A method is a function associated with an object. A primitive will momentarily be coerced to an object so that methods such `toUppercase()` can be used on them.

Firstly, let’s examine the statement “*JavaScript readily coerces between primitives and objects*” by analyzing the code below:

```
let num1 = new Number(1);
let num2 = num1 + 2;
console.log(num2); //3
console.log(typeof(num1)); // object
console.log(typeof(num2)); //number
```

The `new Number` constructor function creates a wrapper object around variable `num1` and is coerced into the primitive number type. The value of the object wrapper `num1` is the primitive that it wraps around i.e. `1`. When we use the `typeof` operator to check `num1` and `num2` we get `object` and `number`. Hence, the object coerced to the primitive number data type which is why `num2 = 3`.

Upon applying a method or accessing the property of a primitive for example a string or boolean, a new temporary object is created using the appropriate constructor function. A constructor starts with the `new` keyword followed by the primitive data type. To illustrate this, look at the following code:

```
console.log(typeof("love")); //string
console.log(typeof(new String("universe"))); //object
```

The constructor function creates a wrapper object which inherits the methods and properties of the global string object such as `charAt( )`, `toUpperCase( )`, `length`, etc.

```
let universe = "universe";
universe.toUpperCase(); //UNIVERSE
universe.length; //8
```

The `toUpperCase()` method is applied to the wrapper object. After you apply a method or access the property of a primitive data type, the temporary wrapper object is disposed of to garbage collection and is never re-used. Therefore it is not advisable to add your own properties to the wrapper object.

We will cover constructor methods in Chapter 3, *All about Objects* in greater detail when we uncover objects.

*Important to remember is that primitive data types have methods and properties such available for you to use.*

### 1.2.10 Non-primitive data types

This is the second category of data types in JavaScript. Listed below are the three main properties of non-primitive data types:

- Multiple values and types
- Objects
- Pass by reference

#### 1. Multiple values and types

Unlike primitive types that reference only one value of a singular data type, non-primitive data types can reference multiple types of values of any data type.

#### 2. Objects

Anything that is not a primitive data type in JavaScript is classified as an object. The main non-primitive data types that developers use are listed below, we will review them in detail in the forthcoming chapters.

Objects:

- Objects

- Arrays
- Functions

### 3. Pass by reference

Non-primitive data types are passed by reference compared to primitive data types that are passed by value. As such non-primitive data types are also called reference types.

To understand why they're called reference types, we need to briefly look at how variables are stored in memory. A fixed amount of memory is allocated to a variable after it is declared. For primitive data types, the actual value in memory is copied over because the value assigned to a variable is immutable and known. Subsequently, exactly how much memory it will take is also then known. For example `let user = "Jyoti", is "1001010 1111001 1101111 1110100 1101001"` in binary.

Whereas, for object data types the address in memory of the object is copied rather than the actual value. As objects can have multiple values, which may or may not fit inside fixed memory. Take for example in the following code, the object called `user` which has one property to start with. But with time you can keep adding more properties as you like.

```
let user={  
  role: 'developer'  
};  
  
user.employed = true;  
  
user.name = 'Rocko';
```

When you `console.log` the `user` object, you see it now has all the newly added properties:

```
Object {  
  employed: true,  
  name: "Rocko",  
  role: "Developer"  
}
```

Therefore, properties and their values are added after declaring the object. This illustrates that the memory size of reference types is not known in advance. So objects are copied by reference rather than value.

The difference between primitive and non-primitive data types is shown in the following table:

Primitive Data Types	Non-primitive data types
Immutable	Can be mutated
Reference a single data type	Collection of one or more data types
Pass by value	Pass by reference
In memory value is the actual value	In memory value is an address/reference

In this section, we covered the different data types available for us to use in JavaScript. In the next section, we will discuss manipulating values with operators.

JavaScript has lots of quirks! While you are not expected to memorize anything, by practicing the following exercises you will be more comfortable with the theory and the small quirks of this language. Do not feel defeated, if you get stuck have a peek at the answer and come back to the question and re-try it.

### Exercises

1. When should you use **var**, **let** and **const**?
2. What is logged to the console in the following code block?

```
const game = 'Kings Quest';
game = 'Super Mario';
console.log(game); //What will be logged here?
```

3. What will the two **console.log** statements return here?

```
let num1 = 1;
function foo() {
    let num1 = 10;
    console.log(num1);
}
console.log(num1);
```

```
foo();
```

4. What is the value of **i** inside the for-loop at each iteration and outside the for-loop once iteration has ended?

```
function countI() {  
    for (let i = 0; i <= 5; i++) {  
        console.log(i); // 0 1 2 3 4  
    }  
    console.log("this is " + i); //  
}  
  
countI();
```

5. What is the value of **a** and **b** when function **scoping()** is called and why?

```
function scoping(){  
    let a = 10;  
    if( a <= 10){  
        var b = 5;  
        a = a + 1;  
    }  
    console.log(a);  
    console.log(b);  
}  
  
scoping();
```

6. Analyze the following block of code. What is the value of **book**?

```
{  
let book = 'JavaScript is fun';  
book = 'JavaScript is fun sometimes';  
}  
  
let book = 'Python is fun';  
console.log(book);
```

7. Analyze the following block of code on lexical scope and guess-estimate what will happen:

```
let outer = function() {
    if (1 < 2) {
        var x = 10;
    }
    if (2 < 3) {
        var xSum = 1 + x;
        console.log(xSum);
    }

    function foo() {
        const z = 1000;
        console.log(x);
    }
    foo();
}
outer();
```

8. What is happening here with the wrapper string object?

```
let primitive = 'september';
primitive.vowels = 3;
primitive.vowels; //undefined;
```

9. What will the two `console.log` messages be?

```
console.log(typeof('universe'));
console.log(typeof(new String('universe')));
```

10. What will be logged to the console after applying the `toUpperCase()` method on the primitive string value and why?

```
let game = 'Super Mario';
game.toUpperCase() // "SUPER MARIO"
console.log(` This is a great game! ${game} `); // ?
```

11. Look at the following code. What method should you use to get the desired output?

```
let bigNum = 19393494928383494949505n;
// ***** Some code here ****

// output is: "19393494928383494949505"
```

## Answers

1. It is preferred to use **const** when you will not be re-assinging values. As **const** signifies a constant/unchanging value. **let**, has block-scope and is ideal for functions, loops, conditional statements and other blocks of code. **var** can induce scoping issues and therefore should generally be avoided.
2. You cannot re-assign values to **const** variables, as this induces a **TypeError**:

```
TypeError: invalid assignment to const 'game'
```

3. The following will be logged to the console:

```
1
10
```

We first log the value of **num1** variable and in this case, **num1** has global scope. Therefore, value **1** will be logged. And lastly, we will call function **foo()**. **let** variables are block scoped, therefore inside **function foo()**, the **num1** variable has a value of **10**.

4. Inside the for-loop the value of **i** is:

```
0
1
2
3
4
5
```

Whereas, outside of the for-loop, within the function, the value `i` is:

```
ReferenceError: i is not defined
```

This is because `let` variables are block scoped and cannot be accessed outside of the for-loop block. The for-loop is demarcated by the opening and closing curly braces `{ }`.

5. The following values down below will be logged for `a` and `b` respectively. This is because `a` has function scope. It can be accessed inside the if conditional statement. Therefore its value changes. The variable `b` is declared with `var` and can be accessed anywhere from inside the function as well.

```
11  
5
```

6. The value of the variable `book` is "Python is fun". As `let` variables can be re-assigned values and be re-declared outside of their current scope.
7. The following is logged to the console:

```
11  
10
```

In order to tackle this question, simply look at the 3 blocks (function `foo()` and two `if` conditional statements).

The variable `outer` is declared with the `let` keyword and is assigned to a function. There are two `if` conditional statements and one function nested inside the main `outer` function assignment. Let's concentrate on the `console.log` statements. The `console.log` statement inside the second `if` conditional statement will log the value of variable `xSum` (declared with the `var` keyword), if `1` is less than the number `2`, which indeed it is. The value of `xSum` is assigned to the sum of `1 + x`. In this case variable `x` is declared with the `var` keyword, hence it has function scope and can be accessed anywhere inside the function. The value of `xSum` is `1` added to `10`, which is `11`. Moving on to function `foo()`, we are logging to the console the value of `var x` which is `10`.

8. A primitive temporary string wrapper object is created around the string value. You can add methods and properties of your own. We have added the `vowels` property to the wrapper object. However, after you apply a method or access the property of

a primitive data type, the temporary wrapper object is disposed of to garbage collection and is never re-used. Which is why when trying to access the `vowels` property, `undefined` is returned.

9. The two `console.log` statements are:

```
"string"  
"object"
```

This is because the second is an object created by the constructor function. It is uncommon to instantiate new strings with the constructor function and should be avoided.

10. The following is logged to the console:

```
" This is a great game! Super Mario"
```

The `toUpperCase()` method is applied on the wrapper object. After you apply a method or access the property of a primitive data type, the temporary wrapper object is disposed of to garbage collection and is never re-used.

11. The `toString()` method is used to display a number which is a `BigInt`, as indicated by the '`n`' at the end of the number:

```
console.log(bigNum.toString()); // "19393494928383494949505"
```

## 1.3 – Operators and Comparisons

In this section, we will cover operators in JavaScript. The information is summarized in tables for you to review.

We will review 7 types of operators:

1. Arithmetic
2. Assignment
3. String
4. Comparisons
5. Logical
6. `typeof`

## 7. void

### 1.3.1 Arithmetic operators

The table in this section displays arithmetic operators. You might be well familiarized with them already. However, **associativity and operator precedence are concepts that are often overlooked**, so let's go over them in order for you to prepare yourself better for interviews.

#### 1.3.1.1 Operator Precedence

In an expression operators with higher precedence over others will be evaluated first.

As an example, the multiplication operator has higher precedence than the addition operator. This is demonstrated in the following expression:

```
let x = 2 + 3 * 6;  
x = 2 + 18;  
x = 20;
```

Multiplication takes place first, therefore 3 is multiplied by 6 which equals the number 18. And then 2 is added to 18 which returns 20.

While you are not expected to memorize the precedence order of operators, it is an important consideration while evaluating an expression.

This table displays the precedence orders of operators:

Precedence	Operator	Symbol
1	Parenthesis	()
2	Unary	++, --, -, !, ~, delete, new, typeof,
3	Multiplicative	*, /, %
4	Additive	+, -
5	Shift	<<, >>
6	Relational	<, >, <=, >=
7	Equality	==, !=, ===, !==
8	Bitwise	&,  , ^
9	Logical	&&,

10	Conditional	?:
11	Assignment	=, +=, -=, *=, /=, %=
12	Comma	,

## Exercises

1. Use the rules of operator precedence and associativity to solve the following:

```
let sum = (5 + 8 + 2) * 2;
console.log(sum); // ?
```

## Answers:

1. Using associativity rules, the parenthesis () take precedence so  $(5 + 8 + 2)$  are added and the sum is then multiplied by 2. The final result is 30.

### 1.3.1.2 Associativity

Associativity refers to the direction in which an expression is evaluated i.e. left to right or, right to left.

To demonstrate this concept, have a look at the following block of code and try and work out what the answer will be using the information provided in the precedence order table.

```
let num1 = 100;
let num2 = 30;
let num3 = 80;
let num = num1 - num2 + num3;
```

From the table, we see that the addition (+) and subtraction (-) operators have the same precedence order. And by looking at the following associativity table we see that for **operators of the same precedence/priority level, operations are performed left to right**. Therefore, we are left with:

```
let num = 100 - 30;
num = 70 + 50
num = 150;
```

Listed in this table is the associativity of operators.

Name	Symbol	Associativity
Add	+	Left to right
Parenthesis	( )	Left to right
Subtract	-	Left to right
Multiply	*	Left to right
Divide	/	Left to right
Remainder/modulus	%	Left to right
Exponentiation	**	Left to right
Increment	++	Right to left
Decrement	--	Right to left
Unary plus (+)	+	Right to left
Unary negation (-)	-	Right to left

### 1.3.1.3 Increment and decrement operators

These can be slightly tricky. *Have a good look at what happens when the increment/decrement operator is placed before and after an operand.*

Increment ++	Decrement --
<p>x++: This is called the <b>post -increment</b> operator.</p> <ul style="list-style-type: none"> <li>The original value of x <b>before</b> incrementing is returned</li> <li>1 is added to the variable after</li> </ul> <pre>let x = 0; x++ // 0 x // 1</pre>	<p>x--: This is called the <b>post - decrement</b> operator.</p> <ul style="list-style-type: none"> <li>The original value of x <b>before</b> decrementing is returned</li> <li>The variable is decreased by 1 after</li> </ul> <pre>let x = 10; x-- //10 x //9</pre>

<p><code>++x</code>: This is called the <b>pre - increment</b> operator.</p>	<p><code>--x</code>: This is called the <b>pre - decrement</b> operator.</p>
--	--

- 1 is added to the variable
- The incremented result is returned

```
let x = 0;  
++x // 1  
x // 1
```

- The variable is decreased by 1
- The decremented result is returned.

```
let x = 9;  
--x // 8  
x // 8
```

## Exercises

1. Knowing what you do about post increment operators, what will be logged to the console?

```
let cookies = 10;  
console.log(cookies++); // ?  
console.log(cookies); // ?
```

2. Using your knowledge of pre-decrement operators, what is the value of `cookies` in each of the `console.log` statements?

```
let cookies = 100;  
console.log(--cookies);  
console.log(cookies);
```

3. Try and workout what value of `i` will be logged to the console, using the pre-increment operator:

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

## Answers

- 1.

```
10  
11
```

2.

```
99  
99
```

3. The following is logged to the console:

```
1  
3  
5  
7  
9  
11
```

The pre-increment operator will return the incremented value of `i`. During the first iteration of the for-loop, `i` is 0. One is added to `i` and now the value is 1. After which, `i` is incremented when the final expression of the for-loop is executed (`i++`). The for-loop will continue till `i` is 10, after which `++10` is now 11. The final expression of the for-loop will increment 11 by 1. The conditional statement will evaluate to false as `i` is no longer less or equal to 10.

### 1.3.2 Assignment operators

Assignment operators will assign a value to a variable. Have a look at the examples in the table.

Name	Symbol	Operator associativity	Example	Explanation
Assignment	=	Right to left	let players = 100;	players = 100;
Addition assignment	+=	Right to left	let discount = 50; discount += 90; //140	discount = discount + 90;
Subtraction assignment	-=	Right to left	let guests = 10; guests -= 3; //7	guests = guests - 3;
Multiplication assignment	*=	Right to left	let x = 2; x = x*2;	

			<code>x*= 2; //4</code>	
Division assignment	<code>/=</code>	Right to left	Let <code>y = 10;</code> <code>y /= 3 // 7</code>	<code>y = y/3;</code>
Modulus assignment	<code>%=</code>	Right to left	Let <code>mod = 90</code> <code>mod %= 4 //2</code>	<code>Mod = mod % 4</code>
Exponentiation assignment	<code>**=</code>	Right to left	<code>let toPowerTwo = 2;</code> <code>toPowerTwo *= 2</code> <code>//4</code>	<code>toPowerTwo =</code> <code>toPowerTwo * 2;</code>
Comma	<code>,</code>	Left to Right	<code>let x = 1, y = 2, z = 3;</code>	

## Exercises

1. What will variables `num1`, `num2` and `num3` be?

```
let num1 = 100;
let num2 = 30;
let num3 = 80;
num1 = num2 = num3;
console.log(num3, num2, num1);
```

2. Solve for `x`

```
let x = 10;
let y = 20;
x+= y**= 3;
console.log(x); //?
```

## Answers

1. Following right to left associativity. This is akin to:

```
console.log(num1 = (num2 = num3)); // 80
```

80 | 80 80

2. `+=` and `*=` have right to left associativity. We first multiply (`y**= 3`) which is 60 and then add 10 to the value 60. Therefore, returning 70.

70

### 1.3.3 Comparison operators

A comparison operator will compare 2 operands. The following table displays all the comparison operators that you should be familiar with:

Name	Symbol	Example
Equality	<code>==</code>	<code>"10" == 10</code> //true
Inequality	<code>!=</code>	<code>"100" != 100</code> //false
Strict inequality	<code>!==</code>	<code>"100" !== 100</code> //true
Greater than	<code>&gt;</code>	<code>2 &gt; 1</code> //true
Greater than or equal to	<code>&gt;=</code>	<code>2 &gt;= 1</code> // true
Less than	<code>&lt;</code>	<code>2 &lt; 1</code> //false
Less than or equal to	<code>&lt;=</code>	<code>1 &lt;= 1</code> //true

Special consideration should be taken into account when dealing with the loose equality (`==`), inequality (`!=`), strict equality (`===`), and strict inequality operators (`!==`).

#### 1.3.3.1 Strict equality (===) and strict inequality (!==) operators

The strict equality and strict inequality operators function the following way:

- They will check if the two operands being compared are of the **same type and same value**
- This is called a **strict comparison** as no type coercion is done (i.e. no conversion of data types, covered in detail in *Section 1.4*)
- The `===` operator will return true if both operands are of the same type **and** value
- The `!==` operator will return true if the two operands have different types **or** values.

Let's take for example the comparison of two variables `num1` and `num2`:

```
let num1 = 10;  
let num2 = "10";  
typeof(num1); //number
```

```
typeof(num2); //string  
num1 === num2; //false  
num1 !== num2 //true
```

In this case `num1` refers to the number type and `num2` refers to a string data type. So even though both values (10) are the same, they are of different data types. Therefore, they are **not** strictly equal to one another.

### 1.3.3.2 Loose equality (==) and inequality (!=) operators:

Moving on, let's discuss the loose equality (`==`) and inequality (`!=`) comparison operators by first listing how equality operations are performed.

- If the two operands are of different data types, type coercion is done so that both operands are of the same type. Coercion is the conversion of one data type into another (covered in detail in *section 1.4*)
- Afterward, a strict comparison is performed to check the values of the two operands
- `==` will return true if the two operands have the same values
- `!=` will return true if the two operands *don't* have the same values
- When comparing two objects with the equality operator (`==`) references in memory of each object will be compared

Picking up from the same example as above. `num1` will be coerced into the string data type for comparison purposes.

```
let num1 = 10;  
  
let num2 = "10";  
  
console.log(num1 == num2); //true  
console.log(num1 != num2); //false
```

Coercion and comparisons with equality operators are covered in greater detail moving on in this chapter in *section 1.4*.

### Exercises

1. What are the two main differences between `==` and `===`?
2. Is variable `x` strictly not equal (`!==`) to `y`?

```
let x;
```

```
let y = 10;  
console.log(x !== y); //?
```

3. Compare `x` and `y` and explain why the two `console.log` messages are different?

```
let x = 0;  
let y = false;  
console.log(x === y);  
console.log(x == y);
```

4. What is returned from the comparisons below?

```
let x = true;  
let y = false;  
console.log(x === y);  
console.log(x == y)
```

5. What will be returned, true or false?

```
Symbol() === Symbol();
```

#### Answers:

1. `==` Will check if the two operands being compared are of the *same type and same value*. No type coercion is performed.  
`==` Will check if the two operands being compared are *either the same type or same value*. Type coercion is done.
2. `true`. 0 is strictly not equal to 10.
3. The following is logged to the console:

```
console.log(x === y); //false  
console.log(x == y); //true
```

In the first statement (`x === y`), the number `0` is not equal to the boolean `false`. Hence `false` is logged. Whereas, in the second statement (`x == y`). Type coercion is done and now `x` which is `0` is loosely equal to the boolean `false` value, which is represented by the number `0`. Remember, boolean `true` is represented by `1` and `false` is represented by `0`.

4. The following will be logged:

```
console.log(x === y); false
console.log(x == y); false
```

In the first statement `x` and `y` are of different data types, hence a strict equality comparison returns `false`. The second statement will return `false` as well, since `x`, which is the boolean `true` represented by the number `1` is not loosely equal to the boolean `false(0)` value of `y`.

5. `false` is returned since every time a `new Symbol()` is invoked, a new unique symbol is generated.

### 1.3.4 Logical operators

Logical operators are used to evaluate conditions and make true and false decisions. The table lists the 3 types of logical operators in order of precedence:

Name	Symbol	Associativity
Logical NOT	!	Left to Right
Logical AND	&&	Left to Right
Logical OR		Left to Right

#### 1.3.4.1 Logical NOT(!)

The logical `NOT!` operator acts on only one operand. It works in the following way:

1. It converts an expression into the boolean `true` or `false`
2. Then it inveres the value of the boolean. Therefore, if the expression was evaluated to `true`, it is now `false` and vice versa.

**Syntax:** `!(variable or value)`

Let's take for example the following block of code and compare the number `2` to the string `'two'`:

```
if(!(2 === 'two')) {
  console.log('2 is not strictly equal to two');
}
```

The condition `2 === 'two'` is `false` as both the integer `2` and string `'two'` are of different data types and values. However the **NOT!** operator inverses the false result to true. Consequently the statement "`2 is not strictly equal to two`" will be logged, as the `if` conditional statement evaluates to `true` now.

This table displays the result of the **NOT!** operator on some operands. *Questions on logical operators are often asked during interviews, so be sure to go over them!*

Operand	Result
Object	<code>false</code>
Empty string	<code>true</code>
Non-empty string	<code>false</code>
Non-zero number	<code>false</code>
Infinity	<code>false</code>
Null	<code>true</code>
NaN	<code>true</code>
Undefined	<code>true</code>

The following are a few examples of the **NOT!** operator on values described in the table above

```
console.log(!{}); // false
console.log(!( '' )); //true
console.log(!(null)) //true
```

### 1.3.4.2 Logical AND (**&&**)

The **AND (**&&**)** operator can accept and evaluate multiple operands. It will function in the following way:

1. An expression is evaluated from left to right
2. If the left operand evaluates to `false`:
  - The right-hand side of the expression will not be evaluated. And the original `false` value of the left side operand is returned. This is called **short-circuit** evaluation
3. If the left operand is `true`

- The right-hand side of the expression will be evaluated
4. This will continue onto the rightmost side of the expression
  5. If all conditions are true, then the value of the last operand is returned

**Syntax:** condition 1 **&&** condition 2 **&&** condition 3....

In the following example, evaluating the expression from left to right, the **AND &&** operator returns the first falsy result which is the value of the **user1** variable:

Example:

```
let user1 = ''; //empty strings are falsy values
let user2 = 'TJ';
console.log(user1 && user2); // ""
```

In the following example, the **AND&&** operator is used to evaluate an expression which checks if the value of variable **num1** is less than **num2** **AND&&** if the value of variable **a** is less than **b**:

```
let num1 = 10;
let num2 = 20;
let a = 1;
let b = 2;
if(num1 < num2 && a < b) {
  console.log(true);
}
```

All conditions in the above expression are **true** evaluating from left to right, as **num1** is less than **num2** and **a** is less than **b**. Therefore, **true** is logged to the console.

The following is the truth table for **&&** operand

When evaluating two operands with either true/false values, the results of **x & y** will be as follows:

x(1 <sup>st</sup> operand)	y (2 <sup>nd</sup> operand)	x & y
True	true	true
True	false	false

x(1 <sup>st</sup> operand)	y (2 <sup>nd</sup> operand)	x & y
True	true	true
False	true	false
False	false	false

When evaluating operands that are not boolean values, **x && y** will be as follows:

x(1 <sup>st</sup> operand)	y (2 <sup>nd</sup> operand)	x && y
Object {}	true/false	Returns the 2 <sup>nd</sup> operand i.e. Boolean
True	Object {}	Returns the object
Object {}	Object {}	Returns the 2 <sup>nd</sup> object
Null	Any value (boolean, object etc)	Null
NaN	Any value (boolean, object etc)	NaN
undefined	Any value (boolean, object etc)	Undefined

Example:

```
let user={};  
console.log(user && false); //false
```

As seen from the table the first operand is an object and the second operand is a boolean. Upon comparison using the **AND** operator, the second operand will be returned. In this case, the boolean **false**.

**Again, you need not memorize any of this, only practice!**

#### 1.3.4.3 Logical OR ||

The logical **||** can operate on multiple operands. It functions in the following way:

1. An expression is evaluated from left to right, type coercion is performed during evaluation to coerce an operand into a boolean

2. While evaluating an expression with 2 operands, the operand that evaluates to true is returned.
3. When evaluating multiple operands, the first truthy value is returned.
4. If all operands evaluate to falsy, then the last falsy operand is returned

**Syntax :** condition 1 || condition 2 || condition 3....

Example:

```
let breakfast = true;
let dinner = false;
console.log(breakfast || dinner); //true
```

In the above, true **OR** false returns the first **true** result which is assigned to the variable **breakfast**.

Now let's have a look at the evaluation of multiple operands as below.

```
let x = false;
let y = false;
let z= 0;
console.log(x || y|| z); // 0
```

All operands are false in the logical **OR** evaluation, thus the last false result (**z**) is returned as stated above in the functioning of the **OR** operand.

### Truth table for the **||** operand

When evaluating 2 operands with either true or false values **a || b** will be as follows:

A	b	a    b
True	true	true
True	false	true
False	true	true
False	false	false

When evaluating operands that are not boolean values, **x || y** will be as follows:

x (1 <sup>st</sup> operand)	y (2 <sup>nd</sup> operand)	x    y
Object {}	Any value that is not an object	Returns the 1 <sup>st</sup> operand i.e. object
False	Any value that is not the boolean true	Returns the 2 <sup>nd</sup> operand
Null	null	Null
NaN	NaN	NaN
undefined	undefined	Undefined

The following are two examples of the **NOT!** operator used to evaluate expressions:

```
console.log(null || null); //null
console.log(false || 'nice'); // 'nice'
```

As seen from the table above **null** or **null** returns **null**, whereas the boolean **false** **OR** a non – empty string returns the non – empty string (**nice**).

### Exercises

- What will the value of the variable **result** be?

```
let a = true;
let b = 0;
let result = a && b;
console.log(result);
```

- What will be logged to the console?

```
let a = {
  fruit: 'lemon'
}
let b = {
  juice: 'mango'
}
```

```
console.log(a && b) // ?
```

3. Complete function `createUser()` that will check if the username for variable `userName` is equal to or greater than `6` characters and not an empty string. If so return `true` else, return `false`

```
let username = "Kauress";  
  
function createUser(){  
}  
  
createUser();
```

4. What is the inverse of `!('')`  
5. Is `msg1` or `msg2` logged and why?

```
const x = "null";  
  
const y = "Bob";  
  
function logThis() {  
  
    if (y === "Bob" && !(x)) {  
  
        console.log("msg1");  
  
    } else {  
  
        console.log("msg2")  
  
    }  
  
}  
  
logThis();
```

6. What will be the value of `c`?

```
var a = 2 ;  
  
var b = 0;  
  
var c = a || b;  
  
console.log(c);
```

7. What is logged and why?

```
let user1 = {  
    id: 1
```

```

}

let user2 = {
    id:2
};

let user3={
    id: 3
}

let x = 10;

if(x === 10 && typeof(x) === 'number' || !(user1)) {
    console.log( user2 || user3);
} else{
    console.log('nope')
}

```

8. By looking at the following code, what do you think is returned?

```
console.log(( undefined || null || 0 ));
```

### Answers

- 0 is logged to the console. As true and ( && ) false denoted by the number 0 evaluates to the boolean false.
- The following will be logged to the console:

```
Object {
    juice: "mango"
}
```

- 3.

```
let username = "Kauress";

function createUser() {
    if(username.length >= 6 && username.trim(' ')) {
        return(true);
    } else{

```

```

        return(false);
    }
}

createUser();

```

4. An empty string ('') is a falsy value. The inverse of a falsy value is true.
5. msg2 is logged to the console. The inverse of !(x) is false. And a truthy value of non-empty string ("Bob") and (&&) a false value will evaluate to false. Therefore, the condition checks if y which is assigned the value of "Bob" is equal to false. Since it is not, the statement in the else block will execute. If we change the condition to: (y === "Bob" && (x)), then msg1 will be logged to the console.
6. 2. Since 0 is a falsy value, the truthy 2 value will be returned
7. This question takes a bit of flexing. Don't worry. Break down the condition into small parts. At the most basic level, the condition evaluates to:  
`true && true || false.`  
This will then further evaluate to `true || false`, which is true (refer to the truthy table above). The statements in the if conditional statement will execute and in turn will return the first true result of `user2 || user3`. Therefore, returning user2

```

Object {
  id: 2
}

```

8. 0. If all operands evaluate to falsy, then the last falsy operand is returned

### 1.3.5 String operator

In this section, we will cover concatenation of strings with the addition operator + and we'll also discuss template strings introduced in ES6.

#### 1.3.5.1 Concatenation with the addition operator

There is only one string operator, which is symbolized by + and is called the concatenation operator. This operator will join 2 or more string values and return the joined string. For example:

```
let firstName = 'Jyoti';
```

```

let lastName = 'TJ';

let greeting = 'Hello, ' + firstName + ' ' + lastName + ' Welcome';

console.log(greeting); // "Hello, Jyoti TJ Welcome"

```

Take note that white space within quotation marks is a character. Another way of joining strings is to use the addition assignment operator, symbolized by `+=`. Such as in the following example:

```

let firstName = 'Jyoti';

let lastName = 'TJ';

let greeting = 'Hello, Welcome ';

greeting+= firstName + ' and ' + lastName;

console.log(greeting); // "Hello, Welcome Jyoti and TJ"

```

The addition assignment operator adds the value of the right operand to the left operand. In this case, the right operand includes values of both `firstName` and `lastName` variables, and it is added to the left operand (`greeting`).

*Take note, when trying to do a string operation on an integer, with the addition (+) or addition assignment (+=) operators, the integer will be coerced into a string representation:*

```

let num = 30;

let score = "The score is ";

score+= num;

console.log(score); // "The score is 30"

```

### 1.3.5.2 Template literals

Instead of concatenation with `+` we can also use template literals. Template literals were introduced in ES6. They allow us to insert the value of variables into a string, in a relatively shorter way rather than using `+` to concatenate. This is called *interpolation*.

**Syntax :** `` text ${variable} text``

Here is an example of a template string that includes variables and a string value.

```

let firstName = 'Jyoti';

let lastName = 'TJ';

```

```
let greeting = `Hello ${firstName} ${lastName} Welcome `;  
console.log(greeting); // "Hello Jyoti TJ Welcome "
```

With template literals, you can create multi-line strings by just pressing enter. Special characters such as the newline character `\n` are unneeded. White space between the backticks `` `` matters, so be mindful when working with multi-line strings!

The following example demonstrates the use of white space inside backticks `` ``:

```
let fullGreeting = `Hello  
${firstName}  
${lastName}  
Welcome `;  
console.log(fullGreeting);
```

The output will be:

```
"Hello  
Jyoti  
TJ  
Welcome"
```

## Exercises

1. Assign a string to `let desserts` such that the output is as below:

```
"ice-cream  
cupcake  
brownies"
```

2. Fill in the function so that the string is "`Your mortagage is 2000`" is logged.

```
let payment = function() {  
    function calculate() {}  
    calculate();  
}  
payment();
```

## Answers

1.

```
let desserts =  
  `ice-cream  
    cupcake  
    brownies`;
```

2.

```
let payment = function() {  
  function calculate() {  
    console.log(`Your mortagage is 2000`)  
  }  
  calculate();  
}  
payment();
```

### 1.3.6 Void Operator

The only purpose of the ES6 void operator is to evaluate an expression to `undefined`. It is a unary operator so only one expression can be used with it.

**Syntax:** `void expression`

Example of the void operator in use:

```
console.log(void 0); //undefined
```

Any expression with void can be used such as a string, function, and so on, but it is a convention to use the number zero as either `void 0` or `void (0)`.

A useful application of the `void` operator is to display an alert message on a webpage without reloading it, such as in the following code:

```
<a href="javascript:void(0);" onclick="alert('Page won't  
reload.')">Hello!</a>
```

The placement of parenthesis `()` matters when using the `void` operator. To demonstrate this, have a look at the following code:

```
console.log(void 2 == '2'); // false  
console.log(void (2 == '2'));// undefined
```

In the first `console.log` statement, `void 2` will return `undefined`, which is not loosely equal to the number `2`. And therefore, the result of this comparison is `false`. Whereas, in the second `console.log` statement the whole expression `(2 == '2')` is evaluated using the `void` operator. This returns `undefined` as expected.

### 1.3.7 Typeof Operator

Now since we've talked about variables and data types, it is a good time to introduce the `typeof` operator. The `typeof` operator will return a string indicating the data type of its operand.

**Syntax:** `typeof(operand)`

Take as an example the following variable declarations:

```
let users = 10;  
let planet = 'Earth';  
let teamMember = {};
```

Using the `typeof` operator on each operand, returns its data type:

```
console.log(typeof(users)); //"number"  
console.log(typeof(planet)); // "string"  
console.log(typeof(teamMember)); // "object"
```

#### 1.3.7.1 Typeof null

A variable must be assigned a value of `null`, which represents nothing. Take note that the `typeof(null)` is `object` and not `null`. This is a peculiarity from the original specification in the language. You can check specifically for `null` in two ways:

1. In order to check specifically for `null` use the strict equality operator (`==`), or strict inequality operator (`!=`). The following example demonstrates this:

```
let nope = null;  
  
if(nope === null){  
    console.log("null value"); // "null value"  
}
```

2. You can also simply `console.log` the value of the variable to check for a null value. For example:

```
let nope = null;  
console.log(nope); // null
```

### 1.3.7.2 Typeof Undefined

The **undefined** type refers to a variable that has been declared but no value has been assigned to it as yet. There are a couple of ways to check for the **undefined** type.

1. To check for **undefined**, you can check for the presence of a value. The presence of a value assigned to a variable equates to a truthy value (`true`) and absence is a falsy value (`false`):

```
let x;  
  
if (x) {  
    console.log(x)  
} else {  
    console.log('Not assigned a value') // this will be  
    logged  
}
```

2. You can also simply `console.log` the value of the variable to check for **undefined**:

```
let x;  
console.log(x); // undefined
```

3. You may also use the `typeof` operator with a conditional statement like so:

```
let x;  
  
if (typeof(x) === 'undefined') {  
    console.log('x is undefined'); // condition is  
    fulfilled  
} else {  
    console.log('x has a value');  
}
```

Variable `x` is of the `undefined` type, so the condition is fulfilled and `x is undefined` will be logged to the console.

#### 1.3.7.3 Checking for null and undefined

To check for both null and undefined use the `typeof` operator to check if the operand's data type is `undefined`. Followed by checking if the operand's value is strictly equal to `null`.

In the following code, the `typeof` operator is used to determine whether variable is `x` is either of the `undefined` or `null` type:

```
let x;

if(typeof x === 'undefined' || x === null) {
    console.log("x is either undefined or null")
} else{
    console.log("x is neither undefined nor null")
}
```

The following will be logged to the console:

```
"x is either undefined or null"
```

#### Note:

`typeof(null)` returns `null`, therefore it is not advisable to use the `typeof` operator to check for a `null` value.

#### 1.3.7.4 Nullish coalescing operator

As of ES2020 You can use the nullish coalescing operator which will only look for null and undefined values. The nullish coalescing operator is a logical operator denoted by double question marks `??`. The operator only looks like either null or undefined types. Falsy values are accepted by this operator.

Take as an example:

```
const doesExist = 100 ?? null;
console.log(doesExist); // 100
```

The right operand is `null` therefore, the nullish coalescing operator will return the value of the left operand.

This operator will also check for `undefined` values:

```
let num1;  
let num2 = 90;  
console.log(num1 ?? num2); // 90
```

The operator will return the value of `num2` as `num1` is `undefined`.

This operator is short-circuited which means it will stop evaluating an expression when it finds the first instance of `null` or `undefined`.

#### 1.3.7.4 `typeof NaN`:

Querying a `NaN` value with the `typeof` operator returns `number` :

```
console.log(typeof(NaN)); // "number"
```

This is because a `NaN` value is actually a numeric data type whose value cannot be represented by numbers. For example infinity. Each representation of `NaN` is unique. *This is why NaN values are not equal to one another, as they don't necessarily evaluate to the same unrepresented number.*

`NaN` is usually returned when math calculations fail, for example here is a list of operations which will result in `NaN`:

Zero divided by Zero:

```
console.log(0 % 0); //NaN
```

Infinity divided by Infinity:

```
console.log(Infinity % Infinity); //NaN
```

Conversion or operations of non-numeric values into a number:

```
console.log('apple' * 2); //NaN
```

`NaN` as an operand

```
console.log(NaN * 2); //NaN
```

`null` when compared with any number returns false

```
console.log(null > 0); //NaN
```

### 1.3.7.5 Checking for NaN

To check for **NaN**, use the ES6 `Number.isNaN()` method, passing in as an argument the variable or value you want to check. Only values that are of the **number** data type and evaluate to `true` when passed in as arguments to the `isNaN()` method will return as **NaN**.

`Number.isNaN()` does not coerce values to the number data type like the global ES5 `isNaN()` method, this is considered more reliable. It will only return true if the value is of the number data type and is equal to a **NaN** value, for example infinity.

Some use cases of when the `Number.isNaN()` method is true are:

#### **NaN is passed as a parameter:**

```
console.log(Number.isNaN(NaN)); //true
```

Zero divided by zero:

```
console.log(Number.isNaN(0/0)); //true
```

#### **Number.NaN is passed as a parameter:**

```
Number.isNaN(Number.NaN); // true
```

Everything else returns false:

```
Number.isNaN(20); //false as 20 is a number but not a NaN  
value
```

```
Number.isNaN("hello") // false, "hello" isn't a number  
datatype
```

```
Number.isNaN('123');//false as the string '123' is not a  
number, and not the number data type
```

The above examples demonstrate that only values which are of the **Number** data type and evaluate to true when passed in as arguments to the `isNaN()` method will return as **NaN**.

There are other operators such as **new**, **in**, **delete**, **instanceof** and **this**. We will come to them when we work with arrays and objects.

### Exercises

1. What will `x1 === x2` result in?

```
const x1 = 2 * "abc";  
  
const x2 = 2 * "abc";  
  
x1 === x2; //?
```

2. What is logged to the console by both statements and why?

```
console.log(null === undefined); //?  
console.log(null == undefined); //?
```

3. What is logged to the console by both statements and why?

*Hint: This is one of JavaScript's many quirks!*

```
console.log(null > 0);  
console.log(null >= 0);
```

4. Explain your reasoning, why is the result is `false`

```
console.log(undefined > 0); //false
```

5. Will statements inside the `if` block execute? If so, why?

```
let x;  
  
if(x) {  
    //will this return true or false  
}
```

6. What does the `typeof` operator return after the evaluation of the expression `1/10`?

```
console.log(typeof(1/10));
```

7. Does the following expression evaluate to true or false?

```
Number.isNaN(0); // T or F?
```

## Answers

1. `false`. First let's check out the value of `x1` and `x2`:

```
console.log(x1); // NaN  
console.log(x2); //NaN
```

Comparing `x1` and `x2` will return `false` as `NaN` is not equal to anything, not even itself.

2. The statement `console.log(null === undefined)` will return `false` using the strict equality operator. `Null` and `Undefined` are of different types and by definition one indicates the lack of a value (`undefined`) whereas the other indicates a value set to be non-existent on purpose (`null`).

The statement `console.log(null == undefined)` will return `true` using the loose equality operator. `Null` and `undefined` when coerced return `false` and, since `false` is equal to `false`, the comparison returns `true`. According to the specification (<http://www.ecma-international.org/ecma-262/5.1/#sec-11.9.3>) :

*If x is null and y is undefined, return true*

3. Null will be converted to the number 0, therefore, `null > 0` is `false`, since 0 is not greater than 0. Whereas `null >= 0` is `true` since  $0 \geq 0$  is true.
4. The statement `undefined > 0` returns `false`, since `undefined` indicates a lack of a value and 0 is a numeric value.
5. `false`. This is because though `x` has been declared, it has a value of `undefined`. We will often use this in `if-else` conditional to check if an input field has some value, which is to say did it get filled in by a user.
6. The `typeof` operator will return "number" as the result of a mathematical calculation is a number.
7. The expression will return `false` as 0 is a number.

### 1.3.8 Comparisons

During comparisons, a non-boolean value will coerce into its boolean equivalent and, then be evaluated within a boolean context. A **truthy** is a value that is coerced to the boolean `true`. And a **falsy** is a value that is coerced to the boolean `false`. The number `1` represents the boolean `true` whereas boolean `false` is the numeric `0`.

**There are 6 falsy values in JavaScript:**

1. "" (empty string)
2. 0
3. false
4. null
5. undefined
6. NaN

**Everything else will evaluate to a truthy. For example:**

- 'false' (a string containing the text "false")

- '10' (a string containing a number)
- [] (an empty array)
- {} (an empty object)
- function(){} (an empty function)

### 1.3.8.1 Checking for truthy and falsy values

1. Using a simple `if` conditional statement you can check if a variable or value is a falsy:

```
if (!(variable) {  
  //statements will execute as !(false) is true  
}
```

2. To check whether something is true or false, pass it to the `Boolean()` function as an argument:

```
Boolean({}) // true  
Boolean([]) // true  
let foo = ()=>{};  
Boolean(foo); // true
```

An empty object and array return `true` when passed into the Boolean function as arguments. This is because they are of the object type, and what may or may not be defined are their values. A function is also an object, therefore, even though there are no statements of code `within function foo()`, it returns `true` when passed as an argument to the Boolean function.

3. A useful tip to turn any value into its truthy or falsy values is to flip it twice use the `!!` symbol. *Remember the `!(value)` returns the inverse, and `!!(value)` will then inverse the inverse.* In this case, you can check for a variable's actual boolean value, without having to do any mental gymnastics. For example:

```
let a = ""; // false  
let b = 0; // false  
let c = []; // true  
console.log (!! (a)); // false  
console.log (!! (b)); // false
```

```
console.log (!! (c)) ; //true
```

The inverse of falsy values like an empty string, and the number 0 results in true when the NOT (!) operator is used. Therefore, when the double NOT (!! ) operator is used on the operands, the inverse results are flipped. This returns the actual boolean value of the operand.

### 1.3.8.2 Comparing truthy and falsy values with ==

Comparing truthy and falsy values with == can be confusing as the loose equality operator will coerce types. Here are some rules for you to keep in mind:

1. Null and undefined are only loosely equal to themselves and each other.

```
console.log(null == undefined) //true  
console.log(null == null) //true  
console.log(undefined == undefined); //true
```

2. false, "", 0 are all equivalent

```
console.log(0 == false); //true  
console.log("") == 0); //true  
console.log("") == false)//true
```

3. NaN is not equivalent to anything, including itself
4. Empty arrays are truthy values. However it isn't reliable to compare them with true or false as seen in the following code:

```
[] == false; //returns true  
[] == true; //returns false
```

### 1.3.8.3 Comparing truthy and falsy values with ===

*It is much better to use the strict equality comparison operator as no implicit coercion is done.* Therefore both the value and data type of both operands will be compared. For example:

```
console.log(false === "") //false  
console.log("") === false)// false  
console.log(null === undefined)//false
```

```
console.log(Infinity === Infinity)//true  
console.log(false === false);//true  
console.log(true === true);//true
```

The strict equality operator returns the expected result for example `null === undefined` returns `false` as both are different types. `Null` is an indicator of no value, whereas `undefined` is an indication that no value has been assigned as yet.

The loose equality (`==`) operator would have performed coercion, and returned `true` for all the above examples that return `false`. This can cause unexpected errors due to implicit coercion so the strict equality (`===`) operator is preferred. We will explore coercion in the following section.

## Exercises

1. True or false?

```
new Number(0); // T or F?
```

2. Name the 6 values that always evaluate to false/falsy
3. True or false? Explain why?

```
1 == "1"; //T or F?  
1 === "1"; // T or F
```

4. What will be `console.log` display in each case?

```
console.log(false + 1);  
console.log (false == 0);  
console.log(false === 0);
```

5. Change the function by adding one symbol only so that "hello world" is logged:

```
let a = 1;  
let b = () => {  
  if(! (a)){  
    console.log("hello world")  
  }  
}
```

```
b();
```

6. Filter this array of values for only truthy values:

```
const myArr = ["10", 80, true, 0, [], undefined, null, '  
, NaN];
```

7. Is `Nan === Nan` true or false and why?
8. What values of `x` and `y` will be logged to the console?

```
let x = 0;  
  
let y = 1;  
  
if (!(x) == y) {  
  
    console.log(x);  
  
} else {  
  
    console.log(y);  
  
}
```

## Answers

1. `false`. As the number `0`, corresponds to the boolean `false` value. We can check this with:

```
console.log(Boolean(0)); // false
```

2. The 6 falsy values are:
  1. " (empty string)
  2. 0
  3. false
  4. null
  5. undefined
  6. NaN
3. The statement `1 == "1"` returns `true`, as when using the loose equality operator, the string `"1"` is coerced into the number `1`. As a result both numbers are equal to one another. Whereas, `1 === "1"` returns `false`, as both values though the same are of different data types.

4. `1`, `true` and `false`

5. The result `!(a)`, is `false`. We can use the double **NOT (!!)** operator to inverse the result of `!(a)`. Therefore, giving the actual boolean value of the operand, that is `true`

```
let a = 1;
let b = () => {
  if (!!a) {
    console.log("hello world")
  }
}
b();
```

We can also simply remove the **NOT(!)** operator, like so:

```
let a = 1;
let b = () => {
  if(a) {
    console.log("hello world")
  }
}
b();
```

6. We know that the last 3 values are falsy so we remove those first. And then we remove the number `0`

```
let myArr = ["10", 80, true, 0, [], undefined, null, '',
  NaN];
let newArr = myArr.splice(0, 5);
console.log(newArr); // ["10", 80, true, 0, []]
```

Now let's remove the number `0` which is also a falsy value:

```
newArr.splice(3, 1);
console.log(newArr)// ["10", 80, true, []]
```

7. `false`. `NaN` is not equal to anything, not even itself.

8. The following will be logged to the console:

```
0
```

The inverse of `!(x)` is `1`, therefore `1` is loosely equal to the value of `1` is `true` and the statements in the `if` conditional block execute.

## 1.4 – Coercion

Type coercion can be confusing for JavaScript developers. As a result, it is often neglected. This section introduces type coercion in a systematic order. You will go over common use cases in order to help you better prepare for an interview.

The following type coercions will be covered:

- String
- Number
- Boolean

### 1.4.1 Definition

Type coercion and type casting are often used interchangeably. However, there is a slight difference between both.

*Type coercion can be defined as “automatic or implicit conversion of values from one data type to another. For example strings to numbers”. (MDN)*

Whereas, *type casting also called explicit coercion is manual*. It is done by the developer by using functions such as `toString()` which converts a string to a number.

### 1.4.2 String coercion

In this section, we will discuss implicit and explicit string coercion of values. Starting with implicit coercions.

#### 1.4.2.1 Implicit string coercion:

String coercion occurs when a value is converted into the string data type. Take as an example the addition of a string and a number:

```
let numA = 1;  
let numB = '88';
```

```
console.log(numA + numB); // "188"
```

The variable `numA` is assigned the numeric value of `1` whereas; `numB` is assigned a string value of `'88'`. A two-step process will occur:

- *An implicit coercion will convert the value of the number data type into a string data type.* The number `1` now becomes the string `'1'`
- The two strings are concatenated by using the `+` operator
- The result will be the string `"188"`

#### Note:

During implicit coercion, the number data type is coerced into a string, and not vice versa. When comparing a string to an object, the object will coerce into a string.

#### 1.4.2.2 Explicit string coercion:

Let's review manual coercion of the string data type. String coercion via concatenation was covered earlier. The two ways to explicitly coerce strings covered in this section are:

- `String()` function
- `toString()` method
- Concatenation with `+` (covered in section 1.3)

Take for instance a numeric checkout value in a shopping cart application, which is to be displayed as a string to the user:

```
let total = 59.15;  
let result = String(total);  
console.log(result); // "59.15"
```

The `String()` function will take variable `total` as an argument and return the primitive string data type.

Another way to perform explicit coercion is to use the `toString()` method, as shown here:

```
let total = 59.15;  
let result = total.toString();  
console.log(result) // "59.5"
```

### Note:

`toString()` method will not work for the null and undefined types. For either of those use the `String()` function.

For example, when using `toString()` and `String()` to coerce a `null` value into the string "null", the expected result is via `String()`:

```
var x = null;  
x.toString(); // TypeError: x is null  
console.log(String(null)); // "null"
```

Like numbers, booleans can be converted to string values, explicitly as well. For example, here the boolean `false` is coerced into the string "false" using the `String()` method:

```
let likePizza = false;  
console.log(String(likePizza)); // "false"
```

### Exercises

1. What does the following return?

```
console.log('41' == 41);
```

2. Evaluate the expression `12 * 6 + 'a'`. What is its data type?
3. Evaluate the following. Why do you think the answer is different from question 2?

```
let a = 'a' + 4 + 7;  
console.log(a);
```

4. Explicitly coerce the boolean value `false` into a string, using the `String()` function

### Answers

1. `true`. An implicit coercion will convert the value of the number data type into a string data type. The number `41` now becomes the string '`41`'. Which returns `true` during comparison with the `==` operator.
2. '`72a`'. Multiplication takes precedence during evaluation, after which the number `72` is coerced into the string data type, and '`72`' is concatenated with '`a`'.

3. `"a47"`. All the operands have the same operator (+), therefore the expression will be evaluated moving from left to right. The number `4` will be implicitly coerced into a string and concatenated with `"a"`. Then the result `"a4"` will be concatenated with the number `7`.

```
4. console.log(String(false)); // "false"
```

### 1.4.3 Numeric coercion

Like the string data types, values can be implicitly and explicitly coerced into the number data type.

#### 1.4.3.1 Implicit Numeric coercion:

Implicit numeric coercion will convert a non-numeric type such as a string or boolean to a number data type. Since numbers are used for mathematical operations, numeric coercion occurs through operands such as:

- Arithmetic operands (+, \*, %, /)
- Comparison operands (<, >, <=, >=)
- Equality operands == and !==
- Unary +

The following examples demonstrate implicit coercion to the numeric data type:

```
console.log(1+ +'123'); // 124
console.log(2 * '9'); // 18
console.log(1 / 'one'); // NaN
console.log(1 - true); // 0
console.log(1 >= false); // true
console.log(0 == false); // true
```

In the above examples, implicit coercion will convert the string and boolean data types to numbers. *Pay attention to when the unary addition (+) operator is used in front of a string:*

```
console.log(1+ +'123'); // 124
```

The unary operator converts the string `'123'` into a number data type and then `1` is added to the number `123`.

## Note

When a string is an operand, and + is the operator, instead of converting the string to a Number, JavaScript will convert the number to a string.

### 1.4.3.2 Explicit number coercion:

The `Number()` function is used to explicitly coerce an argument into the number data type. Values are typically coerced into numbers manually when it is known that a return value needs to be a number.

For example:

```
console.log(Number("10") + 1); //11  
console.log(Number(true)); // 1  
console.log(Number(false)>= 0); //true
```

Here are a few uses to remember while explicitly coercing to a number:

1. Null will coerce to the number 0

```
let x = null;  
console.log(Number(x)); // 0
```

2. Undefined will coerce to NaN. For example:

```
let y;  
console.log(Number(y)); // NaN
```

3. An empty string is a falsy, therefore will coerce to the number 0

```
let z = '';  
console.log(Number(z)); // 0
```

## Exercises

1. What does the following return? If the return value is a string data type, explicitly coerce it into a number

```
let x = (4 * '4' + '9');
```

2. Are the results in **A** and **B** different? If so, explain why:

A.

```
console.log(800 + ('8'));
```

B.

```
console.log('800' + +'8'));
```

3. Re-write the following expression with numerical values instead of booleans and an empty string.

Hint: Use the `Number()` function to coerce a value into a number.

```
let x = '';
let y = true;
let z = true;
let result = (x || y <= z) ? true: false;
console.log(result) // true
```

**Answers:**

1. `'169'` is returned from the expression. The multiplication operator is being used, therefore the string `'4'` is converted into a number and multiplied with `4` which returns the number `16`. Moving on, the number `16` is then concatenated to the string `'9'`. This is because when using the `+` operator, instead of converting the string to a number, the number is coerced into a string instead.

For the second part, convert the string `'169'` into a number explicitly like so:

```
Number(x); // 169
```

2. No. Notice the use of the unary `+` operator in B. Both return the string `'8008'`. In both A and B, the `+` operator signals the implicit coercion of a number to a string.

In B the unary operator converts the string `'8'` into a number data type. This is then concatenated to the string `'800'`. Therefore, resulting in `'8008'`

- 3.

```
let x = 0;
let y = 1;
let z = 1;
let result = (x || y <= z) ? true: false;
```

```
console.log(Number(result)); // 1
```

*Alternative:*

```
let result = (Number(x) || Number(y) <= Number(z)) ? true:  
false;  
  
console.log(result)
```

## 1.4.4 Boolean coercion

There are two types of boolean coercions: implicit and explicit. In this section we will examine the coercion of values into the boolean true and false.

### 1.4.4.1 Implicit boolean coercion:

Implicit boolean coercions are usually done during comparisons. Use cases of implicit boolean coercion were covered in *section 1.3.9 (Comparisons)* to compare values using `==`, `!=`, `==` and `!=` operators.

Remember that during an implicit coercion, the loose equality and inequality operators (`==`, `!=`) are used. Whereas, the strict equality and inequality (`===`, `!==`) operators are not used as they will **not** coerce values.

**To recap there are 6 falsy values in JavaScript. Everything else is a truthy value.**

1. "empty string"
2. null
3. undefined
4. false
5. 0
6. NaN

Here are some general guidelines for implicit boolean coercion using the `==` operator:

- When a boolean is compared with a non-boolean, the latter will be coerced to a number.
- `true` is coerced to the number 1 and `false` to the number 0.
- `null` and `undefined` are equal to each and themselves, but not to 0, false, or '0'.
- `NaN` is not equal to anything, including itself.

- When comparing a string and a number, the string will be coerced into a number
- When comparing a number to an object, the object will be coerced into a number
- When comparing a string to an object, the object will be coerced into a string

Take for example, the following implicit boolean coercions:

```
console.log('Hello World!' == true); //false
console.log('false' == false); //false
console.log('' == false); // true
```

These examples of implicit boolean coercion performed during comparisons with a boolean. Remember, that when a boolean is compared with a non-boolean, the latter will be coerced to a number.

In the first example, the string 'Hello World!' is being compared with a boolean, which returns false. As 'Hello World!' will first be coerced to a boolean by passing it as an argument to the `Number()` function, which returns `Nan`. Due to `Nan` not being equal to the number `1` (true), `false` is returned. The same logic is applied to '`false' == false`.

An empty string is considered a falsy `(0)` value which when compared to the boolean `false (0)`, returns `true`.

#### **1.4.4.2 Explicit boolean coercion:**

The `Boolean()` function will return the boolean value of the argument passed to it. It can also be used to find the boolean result of an expression.

```
console.log(Boolean('Nintendo')); // true
console.log(Boolean(0)); // false
console.log(Boolean({})); // true
```

The above explicit boolean coercions, pass a string, number and object to the `Boolean()` function, which when logged to the console returns their boolean equivalent.

A boolean result can be used during the evaluation of an expression such as:

```
console.log(Boolean('Nintendo') >= 10);
```

The above expression will return `false` as `Boolean('Nintendo')` returns `true`. And boolean true is represented by the number `1`, which is not greater or equal to the number `10`.

Moving on, we will now discuss iteration in JavaScript, also known as looping, and the different ways to iterate.

### Exercises

1. Write a function called `boolBlazer` which:
  - Takes 2 parameters `x` which is a non-boolean and `y` which is a boolean
  - Coerce the `x` parameter by using `Number()` function
  - Compares `x` and `y` using the equality (`==`) or inequality (`!=`) operator
  - If the comparison returns `true`, log the statement '`Is equal`'
  - Else if the comparison returns `false`, log the statement '`Is not equal`'
  - Pass the arguments '`hello`' and `true` to function `boolBlazer()` and call the function to execute it.

### Answer

1.

```
function boolBlazer(x, y) {  
  if(Number(x) == y) {  
    console.log('Is equal')  
  } else {  
    console.log('Is not equal')  
  }  
}  
  
boolBlazer("hello", true);
```

"Is not equal"

## 1.5 Iteration

Iteration refers to repeatedly performing a task for “x” number of times or until a condition is met. This is useful as you can automate tasks in a program.

Iteration is done by loops and in this section of *Chapter 1*, the following loop statements will be covered:

1. for-loop
2. while
3. do-while

Object and array iteration will be covered in *Chapters 3 and 4*.

### 1.5.1 For statement

A **for** loop will iterate through a block of code and perform a task until a certain condition is met.

The structure of a for – loop is as follows:

```
for (initialization; conditional; increment/decrement) {  
//code statements  
}
```

We can break the structure down into the following:

- Initialization statement: A starting expression will initialize the counter to a starting value such as 0, or any other number that you choose to start with.
- Conditional test statement: This consists of a statement that is evaluated to the boolean true or false. The statement is evaluated before iterating through the loop and increasing the counter.
- Code statements: These are within the body of the for loop within the curly braces **{ }**. The statements are executed if the conditional test clause is true.
- Increment/decrement: This is the last step wherein the counter is incremented or decremented. After which the conditional test clause is evaluated again.
- Once the test clause evaluates to false, iteration will stop.

Take for example, a **for-loop** that iterates through the numbers 10 to 0 and prints out the value of counter **i** at each iteration:

```
for(let i = 10; i >= 0; i--) {  
    console.log(i);  
}
```

The **for** loop is executed in the following manner:

- The starting counter is initialized by variable **i** which has a value of **10**.
- The conditional test clause statement checks if at every iteration, **i** is greater or equal to **0**.
- If so, the value of **i** at the specific iteration will be printed out.
- After which, the counter which started at **10** decreases by **1**.
- In the initialization clause at the beginning **i** will now be **9**, and the same process will carry on until **i** is **0**.

#### Note:

Notice the semi-colon in between each of the statements in the **for** loop. The initialization statement and test statement are separated by semi-colons. The increment/decrement statement has no semi-colon after.

It is possible to have an inner **for** loop inside another outer for loop. **The outer loop will execute first and then the inner loop will follow suit.**

For example:

```
for (let i = 0; i <= 3; i++) {  
    for (let j = 0; j <= 2; j++) {  
        console.log('Inner loop value of j:' + j);  
    }  
    console.log('Outer loop value of i:' + i);  
}
```

The outer loop will first execute and then the inner loop will finish one complete round of execution. To elaborate:

- The outer loop starts its first iteration at **i = 0**
- The body of the outer loop contains the inner loop which will then execute
- The inner loop iterates for **j = 0** and **j <= 2**. Completing rounds 0 – 2. Each time logging the value of **j**, which refers to the specific round of iteration in the inner loop

- Finally, the last statement of the outer block will log the value of `i` at that particular iteration
- Steps 1-5 will repeat, for as long as `i` is less than or equal to three.

The following code example shows this in action:

```
"Inner loop value of j:0"
"Inner loop value of j:1"
"Inner loop value of j:2"
"Outer loop value of i:0"
"Inner loop value of j:0"
"Inner loop value of j:1"
"Inner loop value of j:2"
"Outer loop value of i:1"
"Inner loop value of j:0"
"Inner loop value of j:1"
"Inner loop value of j:2"
"Outer loop value of i:2"
"Inner loop value of j:0"
"Inner loop value of j:1"
"Inner loop value of j:2"
"Outer loop value of i:3"
```

### Exercises

1. Iterate through the numbers 1 to 10, and at each iteration print `i`
2. Iterate through the numbers 1-5, and at each iteration multiply `i` by 3
3. Can you explain what is happening in the code block below? ***Notice the semicolon just before the opening curly brace.***

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

4. Analyze the following code, and explain why the for-loop does not execute after the first iteration:

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

5. A company has a list of employees and their salary in separate arrays. You are tasked with the job of printing out the name of each employee and the employee's salary as a string in the format: 'Employee name: Employee salary:'

```
let employees = ['Lara', 'Sukhi', 'Evee', 'Simi', 'Beno',  
'Jay'];  
  
let employeeSalary = [1000, 1300, 957.89, 3230.14, 750,  
13900];
```

6. Using a nested for-loop print the following pattern:

```
*  
**  
***  
****  
*****  
******
```

7. Solve for num in the following block of code:

```
let num = 0  
  
for(let i = 0; i <= 2; i++) {  
    for(let k = 0; k <= 2; k++) {  
        num++  
    }  
}  
  
console.log(num);
```

## Answers

- 1.

```
for(let i = 0; i <= 10; i++) {  
    console.log(i);
```

```
}
```

2.

```
for(let i = 0; i <= 5; i++) {  
    console.log(i * 3);  
}
```

3. The for-loop has three *expressions*, which are separated by semicolons. Since the semicolons only separate the expressions from each other, only two are required. The extra third semi-colon is interpreted as a separator for a fourth expression. This is improper syntax.
4. Variables declared with **const** cannot be re-assigned values, and an error will be outputted:

```
TypeError: invalid assignment to const 'i'
```

5. The for-loop will iterate over the indices of the first array, and use that to index into the others.

```
let employees = ['Lara', 'Sukhi', 'Evee', 'Simi', 'Beno',  
    'Jay'];  
  
let employeeSalary = [1000, 1300, 957.89, 3230.14, 750,  
    13900];  
  
for(let i = 0; i < employees.length; i++){  
    console.log(employees[i] + employeeSalary[i])  
}
```

This will log the following to the console:

```
"Lara1000"  
"Sukhi1300"  
"Eevee957.89"  
"Simi3230.14"  
"Beno750"  
"Jay13900"
```

6.

```
for(let i=0; i<=5; i++) {  
    for(let j = 0; j <= i; j++) {  
        document.write("*");  
    }  
    document.write('<br>');  
}
```

7. The value of `num` is 9. The inner loop will iterate 3 times (it completes 1 cycle) for each time that the outer loop iterates. At each iteration within the inner loop the value of `num` is incremented by 1, for a total of 3 times

```
console.log(num); // 9
```

### 1.5.2 While statement

The `while` statement executes a block of code while a condition is true. The conditional statement is executed **at-least once** before running. Execution stops once the conditional statement evaluates to false.

#### Syntax:

```
while(conditonal clause) {  
    //statements of code to execute  
}
```

In practice, a while loop looks like this:

```
let x = 0;  
  
while(x <= 3) {  
    console.log(x);  
    x++;  
}
```

This code block will execute as follows:

- Variable `x` is declared, outside of the `while` loop.

- The conditional statement checks if `x` is less than or equal to `3`.
- If so then log the value of `x` to the console.
- Followed by incrementing the value of `x` by `1`.

The while loop continues to evaluate the condition till `x` is `<= 3`. This will result in:

```
0
1
2
3
```

*Be wary of infinite loops, which will keep executing due to the conditional clause never evaluating to false.* Such as in the following example:

```
while(true) {
  console.log('oh no! This loop will crash your browser!');
}
```

The `while` loop will keep executing as the conditional is always true. Therefore, repeatedly logging the following statement to the console. This would result in the following:

```
"oh no! This loop will crash your browser!"
"oh no! This loop will crash your browser!"
"oh no! This loop will crash your browser!"
```

The above infinite `while loop` will slow down the browser drastically or crash it!

## Exercises

1. Is the following statement true or false?:

*"A while loop will execute for as till a conditional statement becomes false."*

2. Write the conditional clause in this while loop that will cause the browser window to crash

```
let x;
while() {
  console.log("Infinite Loop")
}
```

3. Write a function called `magicBall()` that will display 3 random numbers to a user. Use a `while` loop to complete the code.

```
function magicNumber() {  
}  
  
magicNumber();
```

### Answers

1. True

2.

```
let x;  
  
while(true){  
    console.log("Infinite Loop");  
}
```

3.

```
function magicBall(){  
  
let chance = 1;  
  
while(chance <=3){  
  
let magicNumber= Math.floor(Math.random() * 11);  
  
chance++  
  
console.log(magicNumber);  
  
}  
  
}  
  
magicBall();
```

### 1.5.3 Do while statement

A **do-while** loop, will execute statements of code within the **do** block at-least once. It then evaluates the conditional statement in the **while** loop. If true, the **do** block executes again. This is repeated until the condition is false.

### Syntax:

```
do {  
    //code statements;  
} while(condition);
```

Take, for example, a function `ticketCount(x)` that logs the number of tickets available at a local cinema:

```
function ticketCount(x) {  
  
    do {  
  
        console.log(`The number of available tickets left are  
        ${x}`);  
  
        x--  
  
    }  
  
    while (x >= 0);  
  
}  
  
ticketCount(10);
```

Function `ticketCount()` takes `x` as a parameter. The function when called does the following:

- Within the function the `do` block will log a string conveying the number of tickets left and then decrement the value of `x` by 1
- Then the condition in the while statement will be evaluated to boolean `true` or `false`
- The `do-while` loop will iterate once again. Each time the conditional statement: `x >= 0` will be evaluated. Which means till there are no more tickets left to sell

The following will be logged when function `ticketCount()` executes:

```
"The number of available tickets left are 10"  
"The number of available tickets left are 9"  
"The number of available tickets left are 8"  
"The number of available tickets left are 7"  
"The number of available tickets left are 6"
```

```
"The number of available tickets left are 5"  
"The number of available tickets left are 4"  
"The number of available tickets left are 3"  
"The number of available tickets left are 2"  
"The number of available tickets left are 1"  
"The number of available tickets left are 0"
```

### Note: while vs do-while loop

The **do-while** loop will execute at least once, before the conditional statement is evaluated. Code is executed line by line. Therefore, the first time around, the **do** block is read before the **while** condition is evaluated. As a result, code statements will execute at least once, even if the condition is **false**.

### Exercises

1. Code a **do-while** loop which will print the number 0, five times.
2. Write a **do-while** loop that will execute once and log a string 'Love and Peace' with the conditional **x = false**. Do not change the value of **x** inside the while statement.
3. Modify the function in question 2, to infinitely execute a **do-while** loop. You may change the value of **x**.
4. Write a function that takes **x** as a parameter. **x** refers to the number of tickets sold at a local cinema. Within the body of the function use a **do-while** loop to inform the ticket attendant know how many seats and tickets are available, given a fixed number of seats (30).

```
function ticketSold(x) {  
    let availableSeats = 30;  
    let tickets = x;  
    // your code here  
}  
ticketSold(40);
```

## Answers

1.

```
let zero = 0;
let count = 0;
do{
    console.log(zero);
    count++
} while(count < 5);
```

2.

```
let x = false;
do{
    console.log('Love and Peace');
    x = true;
} while (x = false)
```

The **do-while** loop will execute at least once, before the conditional statement is evaluated.

3.

```
let x = true;
do{
    console.log('Love and peace');
}while(x );
```

4.

```
function ticketSold(x) {
let availableSeats = 30;
let tickets = x;
do{
    availableSeats--;
    tickets--
}
```

```

        console.log('available seats:' + availableSeats + " "
tickets left:" + tickets);

    }while(availableSeats >= 1 && tickets >= 1);

}

ticketSold(40);

```

#### 1.5.4 Break and continue statements

The **break** statement is used to exit a loop completely once a condition has been met, or after x number of iterations. As such it is commonly used within an **if** conditional block.

Take as an example the following code:

```

for (let i = 0; i <= 5; i++) {

    console.log(i);

}

```

This will print out the value of variable **i** from 0 to 5. However, what if we only want to print up to but including the number 3 and then exit from the loop?

```

for(let i = 0; i <= 5; i++) {

    if(i === 3) {

        break;

    }

    console.log(i);

}

```

With the **break** statement now included, the loop executes in the following manner:

- The for-loop starts execution
- The **if** conditional block checks for when **i** will be strictly equal to the number **3**
- When it is, then the **break** statement ensures an exit from the loop completely
- As such, only numbers from **0** to **2** are logged as the loop only executes till **i** is **0**, **1** and **2**

See the following code for an example:

```
0  
1  
2
```

The **continue** statement will terminate the current instance of an iteration in a loop. This is better explained with an example, wherein all the numbers from 0 to 5 are logged with except the number 3.

```
for (let i = 1; i <= 5; i++) {  
  if (i === 3) {  
    continue;  
  }  
  console.log(i);  
}
```

The **continue** keyword changes the iteration in the following manner:

- The **if** conditional block checks if the value of **i** at the current iteration is **3**.
- If so, the instance of that current iteration in the loop exits by using the **continue** keyword.
- The rest of the loop executes as is expected starting from the next iteration, with **i** starting at **4**.
- The result will be:

```
1  
2  
4  
5
```

Having covered iteration in JavaScript, the next section of this chapter introduces controlling program flow with conditional statements.

### Exercises

1. What is the key difference between the **break** and **continue** statements?
2. Print out all odd numbers from **0-10** using the **break** statement

3. Iterate over each letter of the 3 letter words in the following array. Logging all the letters in a word on a new line, except for vowels ("a", "e", "i", "o", "u"). Use a **continue** statement.

```
let wordList=['umbrella', 'apple', 'paint', 'never',
'outmost'];
```

### Answers

1. The **break** statement is used to exit a loop completely, once a condition has been met, or after  $x$  number of iterations. Whereas, the **continue** statement will check a condition and terminate the current instance of an iteration in a loop if the condition is true. And then continue iterating.

2.

```
for(let i = 0; i <=10; i++) {

    if(i % 2 === 0) {

        continue;

    }

    console.log(i);

}
```

3.

```
let wordList = [
  'umbrella',
  'apple',
  'paint',
  'never',
  'outmost'
];

for (let i = 0; i < wordList.length; i++) {

  let currentname = (wordList[i]);

  for (let j = 0; j < currentname.length; j++) {

    if (currentname[j] === 'a' ||
      currentname[j] === 'e' ||
      currentname[j] === 'i' ||
      currentname[j] === 'o' ||
      currentname[j] === 'u') {
        continue;
      }

    console.log(currentname[j]);
  }
}
```

```

        currentname[j] === 'i' ||
        currentname[j] === '' ||
        currentname[j] === 'o' ||
        currentname[j] === 'u') {
    continue
}
console.log(currentname[j]);
}
}

```

## 1.6 – Conditional statements

Conditional statements allow you to control program flow. This means that based on specific conditions, different parts of a code block will execute.

There are 4 types of conditional statements:

- If
- Else
- Else if
- Switch

Let's examine each of these statements in turn.

### 1.6.1 If Statements

Statements of code inside an **if** block will only execute when the condition inside the parenthesis **( )** is met.

#### Syntax :

```

if (condition) {
    //code statements
}

```

Take for example, an **if** conditional statement that checks for a blank/empty username in an HTML input element:

```

let username = document.getElementById('user');

if (username.value.trim() === '') {
    alert('Please enter a valid username');
}

```

The following code is executed as follows:

- The `username` variable references an input element in an HTML webpage with an ID of `user`.
- The `if` statement will evaluate the condition inside parenthesis `()`
- The condition will check if the value of variable `username` is an empty string `''` (falsy value). The `trim()` method is used to eliminate any beginning or tail end white spaces. Therefore, we eliminate any possibility of empty white space by pressing the spacebar key
- If the conditional evaluates to `true` i.e. the `username` input field is empty, the user will be alerted with a message “`Please enter a valid username`”

## 1.6.2 Else Statements

An `if` statement is usually accompanied by an `else` statement. In the event a conditional statement in the `if` block is unmet, then statements inside the `else` block will execute.

### Syntax:

```

if (condition) {
    // statement
} else {
    // statements
}

```

Picking up from the same example, let's see what happens if a user does enter a valid username in the input block:

```

let username = document.getElementById('user');

function checkName() {
    if (username.value.trim() === '') {

```

```

        alert('Name must be filled out');

    } else {

        alert(`thanks ${username.value}`);
    }
}

```

Function `checkName()` uses an `if-else` conditional statement to check for a valid username. The function executes the following lines of code:

- The `if` conditional statement will check if the trimmed value of the `username` input value is strictly equal to an empty string
- If the `username` variable is an empty string then alert the message “`Name must be filled out`”
- However, if the condition evaluates to `false` i.e. the `username` input field is not empty then the statement in the `else-block` will execute
- The `else` block alerts the message thanking the user

### 1.6.3 Else If Statements

The evaluation of multiple conditional statements is possible with an `else - if` block.

#### Syntax :

```

else if {
    // statements
}

```

In the example below, an `else-if` block will check if the `username` is longer than 3 characters, in addition to checking for a non-empty value of the input element:

```

let username = document.getElementById('user');

function checkName() {
    if (username.value.trim() === '') {
        alert('Name must be filled out');
    } else if (username.value.length <= 3) {
        alert('Username must be longer than 3 characters');
    }
}

```

```

    } else {
        alert(`thanks ${username.value}`);
    }
}

```

The **else-if** block is inserted between the **if** and **else** blocks. The evaluation of the function will change in the following way:

- In case the **if** conditional statement evaluates to **false**:
- Then the **else-if** conditional statement will be evaluated
- The **else-if** conditional statement checks if the value of the **username** input field is less than or equal to **3**. If **true**:
- Then the alert message '**Username must be longer than 3 characters**' is shown to the user
- If the condition evaluates to **false**, i.e. **username** is longer than three characters:
- The code statements inside the **else** block will execute. And the message in the **else** block is alerted

#### 1.6.4 Switch statements

A switch statement will evaluate an expression against multiple cases. Upon finding a matching case, code for that case will be executed. Multiple **else-if** blocks should be replaced with a **switch** statement, as it improves the readability of your code.

##### Syntax:

```

switch (expression) {
    case a:
        // case a code block executed
        break;
    case b:
        // case b code block executed
        break;
}

```

```
default:  
    // default code block executed  
}
```

The logic for a switch statement is as follows:

- The switch statement first evaluates an expression.
- After which, the first case which is `case a` is tested against the expression. If matching, the code block for `case a` is executed. Subsequently the `break` statement is used to exit the switch block.
- In the event, that `case a` does not match the expression. Then it is tested against `case b`. If `case b` matches the expression then the code for `case b` will execute, and the code block will be exited from using the `break` statement.
- If neither `case a` nor `case b` match the expression, then the default code block will execute.

As an example to further illustrate a switch statement, imagine that a user has three choices from a breakfast menu:

- eggs
- pancakes
- cereal

```
function menuGreeter() {  
  
    let userchoice = document.getElementById('user');  
  
    switch (userchoice.value.toLowerCase()) {  
  
        case 'eggs':  
  
            console.log('Getting ready for the day eh?');  
            break;  
  
        case 'pancakes':  
  
            alert('Great choice, we serve them with Canadian maple  
syrup!');  
            break;  
  
        case 'cereal':
```

```

        alert('On a diet?');

        break;

    default:
        alert('Not serving that today');

    }
}

```

The `menuGreeter` function will execute the following lines of code:

- A function scope variable called `userchoice` is declared with the `let` keyword. It references an HTML input element:
- `<input type = 'text' id = 'user'>`
- The expression inside the switch statement gets the string value of the input field which is converted to lowercase via the `toLowerCase()` method. This ensures that all input is treated the same (lowercase and uppercase values)
- The first case is called `eggs`. If the user input matches against the string value `eggs`, then the custom message contained within the `eggs` case block will be alerted to the user. And the `break` statement will ensure an exit from the `switch` block.
- If the first case does not match, then a match will be tried against the remaining cases
- If no match is found with any of the cases, then the default case will execute. This alerts the user with the message `'Not serving that today'`

#### **1.6.4.1 Multiple case statements:**

In order to execute the same code for multiple case statements, you may group them together. Let's refactor the function `menuGreeter()` to group together all the breakfast choices:

```

function menuGreeter() {
    let user = document.getElementById('user')
    switch (user.value.toLowerCase()) {
        case 'eggs':

```

```

        case 'pancakes':
        case 'cereal':
            console.log('All day breakfast is good!');
            break;
        default:
            console.log('Not serving that today');
    }
}

```

function `menuGreeter()` is changed in the following way:

- The individual cases: `'eggs'`, `'pancakes'` and `'cereal'` have been grouped together
- If the user input matches any of the strings: `'eggs'`, `'pancakes'`, or `'cereal'` then the same message will be alerted for all choices `"All day breakfast is good"`. And then an exit from the switch block is ensues.
- If the user input does not match any of the cases, then the `default` case will execute.

## 1.6.5 Ternary operator

The ternary operator is an alternative to a single if conditional statement. This operator has 3 operands:

1. Condition
2. Value /expression if true
3. Value/expression if false

**Syntax:** `condition? Value if true: value if false;`

The first operand is a conditional statement that is evaluated to either boolean `true` or `false`, followed by a question mark (`?`). The second operand is a value or an expression that will be evaluated if the conditional statement evaluates to `true`. This is followed by a colon. The last operand is a value or expression if the conditional statement evaluates to `false`.

Example:

```
let pet = 'dog';
pet === 'dog'? true: false; //true
```

This is interpreted as follows:

- A variable named `pet` is declared which is equal to the string value '`'dog'`'.
- The condition checks if variable `pet` is strictly equal to (`==`) the string value '`'dog'`'.
- The conditional statement evaluates to `true`. Therefore, the 2<sup>nd</sup> operand return value of `true` is valid.

Let's see what happens when the condition is changed:

```
let pet = 'dog';
console.log(pet === 'giraffe'? true: false); //false
```

The evaluation of the condition will follow like before.

- The condition now checks if variable `pet` is strictly equal to (`==`) the string '`'giraffe '`'.
- Since the expression evaluates to the boolean `false`, the return value of `false` is valid
- Therefore, `false` is logged to the console

#### 1.6.5.1 Variable assignment

The result of a ternary operator may also be assigned to a variable. In the example here, an expression is evaluated and its result is assigned to the variable `greeting`:

```
let userName = 'Shadow';
let greeting = userName !== ''? `Hi ${userName}` : false
console.log(greeting);
```

The following will be logged to the console:

```
"Hi Shadow"
```

The variable `greeting` refers to the result of the conditional statement `userName != ''`. You may now use this variable elsewhere in your code, which allows for greater flexibility.

Additionally, the 2<sup>nd</sup> and 3<sup>rd</sup> operands of a ternary operator can call functions.

```
let userName = 'Shadow';

let greeting = userName.value !== '' ? createUser() : false;

function createUser() {
    //some code here
}
```

Upon evaluation of the conditional statement to `true`, the `createUser()` function will run. This is preferred in the event there is a more complex task to be performed which will take many statements of code.

### Note

The ternary operator is not commonly used. It should preferably be used in place of single if conditional statements, when evaluation of an expression is either true or false.

### Exercises

1. Write the syntax of an `if - else` block
2. Write a conditional statement that will check if:
  1. The `undefined` and `null` data types are strictly equal (`====`) to each other
  2. `Undefined` and `null` are loosely equal (`==`) to each other
3. Code a function that will accept a user's age as input. The function must log different messages depending on the user's age. Use the `if`, `else if`, `else` statements.

Listed below are the conditions:

Ages: 0 – 18: Message: “Please get parental permission”

Ages: 19 – 30: Message: “You will get an email within seven days”

Ages 31 – 40: Message: “Please check your email in 3 days”

Ages 41 – 50: Message: “Please login in a few hours”

Ages 51 – 60 Message: “You are now enrolled”

Ages 60+      Message: “Please continue to login”

*Note: declare a variable called userage and evaluate it against different conditions*

4. Refactor the above question (#3) using a switch statement with various cases
5. What is returned from the following code statement?

```
console.log("0" == false? true: false);
```

6. Refactor the following block of code which will check if **x** is smaller than variables **y** and **z**. Change the function in the following ways:

- Declare the variables in global scope
- Use a ternary operator

*Note: You need not use a function*

```
let kawaii = function() {  
    let x = 10;  
    let y = 20;  
    let z = 30;  
  
    if (x < y) {  
  
        console.log('x is smaller than y');  
    }  
  
    if (x < z) {  
  
        console.log('x is smaller than z');  
    } else {  
  
        Return false;  
    }  
}  
  
kawaii();
```

**Answers:**

1.

```
if (condition) {  
    } else {  
    }
```

2.

```
if(undefined === null) {  
    console.log('Strictly not equal');  
} else {  
    console.log('Loosely equal');  
}
```

3.

```
let userage = 61;  
  
if(userage > 0 && userage <= 18) {  
    console.log('Please get parental permission')  
} else if (userage > 18 && userage <= 30){  
    console.log('You will get an email within seven days')  
} else if (userage > 30 && userage <= 40) {  
    console.log('Please check your email in 3 days')  
} else if (userage > 40 && userage <= 50) {  
    console.log('Please login in a few hours')  
  
} else if (userage > 50 && userage <= 60) {  
    console.log('You are now enrolled')
```

```
    } else {
        console.log("Please continue to login")
    }
}
```

4.

```
let userage = 30;
switch (true) {
    case userage > 0 && userage <= 18:
        console.log('Please get parental permission');
        break;
    case userage > 18 && userage <= 30:
        console.log('You will get an email within seven days');
        break;
    case userage > 30 && userage <= 40:
        console.log('Please check your email in 3 days');
        break;
    case userage > 40 && userage <= 50:
        console.log('Please login in a few hours');
        break;
    case userage > 50 && userage <= 60:
        console.log('Please login in a few hours');
        break;
    default:
        console.log('Please continue to login');
}
```

5. true. The boolean equivalent of string "0" is false.

6.

```
let x = 100
let y = 20;
let z = 30;
x < y ? console.log('x is smaller than y') : false;
x < z ? console.log('x is smaller than z') : false;
```

## Summary

In this chapter, we reinforced the basics of JavaScript. These will form the foundations of the forthcoming chapters. To recap, we learned what ECMAScript means and the different ECMAScript versions. The three different types of variable declarations using **var**, **let**, and **const** were discussed along with their implications on global, local, and block scope. Re-declaration and re-assignment of **var**, **let** and **const** variables were also discussed. Moving on, the two different data types (primitives and objects) were introduced, along with the main differences between them. In this chapter, we focused on primitive data types. We also went through operators, comparisons, and coercion. These are fundamental concepts, a deeper understanding of which will allow you to write code more efficiently. Finally, we finished by discussing iteration and conditional statements.

# The Basics -2

*“All skills are learnable.” — B.Tracy*

This chapter introduces you to advanced basic concepts of JavaScript. Having solidified your understanding in chapter 1, you will be introduced to concepts that will give you a deeper understanding of basic JavaScript mechanisms. You will also be better equipped to handle the newer features of the language such as `let` and `const` by understanding hoisting. It is important not to skip this chapter because practicing the coding exercises provided for this chapter will help you greatly in the next sections. This chapter forms the base for the forthcoming chapters which deal with arrays and objects.

In this chapter we're going to cover the following main topics:

- Hoisting
- Error handling with try/catch/throw
- Types of errors
- Strict mode

## 2.1 Hoisting

During compilation, microseconds before code is executed, variable and function **declarations** are “hoisted” i.e. they are added to memory inside a data structure called “lexical environment”. The official ES6 documentation describes the lexical environment as:

*“Lexical Environment is a specification type used to define the association of Identifiers to specific variables and functions based upon the lexical nesting structure of ECMAScript code.”*

Conceptually, this refers to a data structure containing variable and function declarations that are initialized with pre-defined values:

Name of variable/function	Pre-defined value
---------------------------	-------------------

***Adding variable and function declarations to memory during compilation before they get executed is called “hoisting”.*** All variable declarations and function declarations are scanned for and placed in memory at the start of their enclosing scope `{ }`. This mechanism

[  
of hoisting is what allows you to make forward references to variable and function declarations, as they're saved in memory with pre-defined values. Practical examples are provided as we move on in the next section.

Code gets executed during runtime, which means all declarations, assignments, and function calls will be executed then. We'll explore hoisting in reference to **var**, **let**, **const** and function declarations in the following sections. This will make the concept clearer.

### Note

Only declarations get hoisted, not initializations or assignments

Practice the exercises for this section and then we will delve deeper into hoisting.

### Exercises

1. What is hoisting?
2. Examine the following function assignment. Will it get hoisted?

```
let calculateSurfaceArea = function(h, w, l) {  
    let area = 2 * (h * w) + 2 * (h * l) + 2 * (w * l);  
    return area;  
};
```

### Answers

1. Adding variable and function declarations to memory during compilation before they get executed is called "hoisting". All variable declarations and function declarations are scanned for and placed in memory at the start of their enclosing scope **{ }**.
2. No, it will not get hoisted as assignments are not hoisted. This includes function expressions which are functions assigned to variables.

## 2.1.2 Var hoisting

To understand variable hoisting better, here is a small summary of the variable declaration, initialization and assignment phases. This is commonly known as "making a variable".

## Declaration

Variable declaration refers to the declaration of a variable with a keyword such as `var`. For example:

```
var x;
```

## Initialization

During declaration, variables are automatically assigned a value of `undefined`. This is called initialization:

```
var x = undefined;
```

## Assignment

Assignment refers to assigning a value to a variable with the assignment operator (`=`). Let's assign a numeric value of `100` to a variable called `x`:

```
var x = 100;
```

And this brings us to the next point, how are `var` variables hoisted in memory?

`var` variables are hoisted in memory with a starting value of `undefined`. For example:

```
var icecream = 'Maple Syrup';
```

Conceptually, it will be stored in memory like so:

Variable Name	Value
<code>icecream</code>	<code>undefined</code>

Therefore, which is why when you try and access a `var` variable before its declaration, the `undefined` type is returned:

```
console.log(icecream); // undefined  
var icecream = 'Maple Syrup';
```

`undefined` is logged to the console. The following bullet points will explain why:

- The variable declaration `var icecream` is hoisted to the top of its scope, in this case, global scope. It is initialized with a starting value of `undefined`
- Only the variable declaration is moved, not its assignment (= `'Maple Syrup'`)

- Since only the declaration `var ice-cream` is hoisted to the top, the `console.log` statement will hence return `undefined`.
- When code is executed at runtime, the `undefined` value will be overwritten with the value assigned by us which is '`Maple Syrup`'

The above bullet points can be demonstrated in code format as:

```
var icecream;
console.log(icecream); // undefined
icecream = 'Maple Syrup';
```

After hoisting, during execution, the value of `var` variable will be overwritten with the value assigned by a user.

*Keep in mind that only variable declarations are hoisted, variables assignments are not.*

### Exercises

1. Explain why the two `console.log` statements return different values?

```
console.log(x); // undefined
var x = 100;
console.log(x); // 100
```

2. Examine the following code and analyze what is happening in reference to hoisting:

```
console.log(furniture);
var material = 'Bamboo';
```

3. Analyze the following code what is logged to the console?

```
for(var i = 0; i <= 4; i++) {
  console.log(i);
  setTimeout(function() {
    console.log('The number being logged is ' + i);
  }, 1000);
}
```

## Answers

1. During hoisting `var` variables are stored in memory with a starting value of `undefined`. Therefore, the `console.log` statement will return `undefined`. And during execution the `var` variable's value is overwritten by the user defined value which is `100`. Therefore the second `console.log` statement returns `100`.
2. The variable `furniture` has not been declared, therefore it cannot be referenced. So a reference error is returned.

```
ReferenceError: furniture is not defined
```

3. The `console.log` statement outside the `setTimeout()` function will log numbers from `0–4`. Whereas, for the `console.log` statement inside the `setTimeout()` function the entire loop has already iterated through every iteration that it should. Therefore, by the time (`1` second) that the first `setTimeout()` function will execute, `i` is already `5`. However, there is no way to reference `i`

```
0  
1  
2  
3  
4  
"The number being logged is 5"  
"The number being logged is 5"
```

You can circumvent this issue by using `let` which has block scope:

```
0  
1  
2  
3  
4
```

```
"The number being logged is 0"  
"The number being logged is 1"  
"The number being logged is 2"  
"The number being logged is 3"  
"The number being logged is 4"
```

### 2.1.3 let and const hoisting

`let` and `const` are stored as uninitialized values, in contrast to `var` which is stored as `undefined`. As an example:

```
let city = 'Toronto';  
const apple = 'Granny smith'
```

These will be stored as uninitialized:

Variable Name	Value
<code>let, const</code>	uninitialized

Trying to reference a `let` or `const` variable before its declaration will result in a `ReferenceError` getting thrown:

```
console.log(city);  
console.log(apple);  
let city = 'Toronto';  
const apple = 'Granny smith';
```

```
ReferenceError: can't access lexical declaration 'city' before  
initialization
```

Like `var`, `let` and `const` variables are hoisted. However, unlike `var`, they *cannot be accessed* before they are declared. This is because they lie inside the **Temporal Dead Zone** (TDZ).

`let` and `const` variables are said to lie within the **Temporal Dead Zone** when they're declared, but not as yet initialized in memory. These variables get hoisted, but no value is assigned to them in memory, not even `undefined`.

## Exercises

1. Define the Temporal Dead Zone (**TDZ**)
2. Answer the following questions related to the following block of code:

```
console.log(giraffe);  
  
const giraffe = 'Masai giraffe';  
  
{  
  
const giraffe = 'Nubian giraffe';  
  
}  
  
console.log(giraffe);
```

- What will the first `console.log` statement return?
- After commenting out the first `console.log` statement what will the second `console.log` statement return?

## Answers

1. `let` and `const` variables are said to lie within the **Temporal Dead Zone** when they're declared, but not as yet initialized in memory. These variables get hoisted, but no value is assigned to them in memory, not even `undefined`.

2. Answer the following questions related to the block of code in question #2:

- The first `console.log` statement will return a `ReferenceError` as `const` and `let` variables cannot be accessed before they're declared because they lie within the temporal dead zone (**TDZ**)
- The second `console.log` statement will return "Masai giraffe". As the `const` variable that references this string value is in global scope. Hence, it can be accessed anywhere. The other `const` variable is within block scope and cannot be accessed outside its block scope

## 2.1.4 Function hoisting

Functions are hoisted along with a reference to the body of the entire function. For example, take a function called `myFunc()`:

```
function myFunc() {  
    console.log('This is myFunc');  
}
```

The `myFunc()` function will be stored in memory during compilation like so:

Function name	Value
<code>myFunc</code>	<code>console.log('This is myFunc');</code>

Therefore, referencing a function before its declaration behaves as it would when calling it after its declaration:

```
myFunc();  
  
function myFunc() {  
    console.log('This is myFunc');  
}
```

The function will execute before its declaration and the following will be logged to the console:

```
"This is myFunc"
```

Having understood the hoisting mechanism for `var`, `let`, `const` and function declarations. Let's examine the precedence order for these.

## 2.1.5 Precedence for variable and function hoisting

The order of precedence for hoisting is:

1. Variable assignment
2. Function declaration
3. Variable declaration

For example:

```
function fruit() {  
    console.log('peaches, mandarins, grapes');  
}  
  
var fruit = 'apples';  
  
console.log(typeof(fruit)); // "string"
```

Variable assignment takes precedence over function declaration during hoisting. Hence, the statement `typeof(fruit)` will log "string" to the console.

Moving on, function declarations takes precedence over variable declarations, as we can see from the example below:

```
var fruit;  
  
function fruit() {  
}  
  
console.log(typeof(fruit)); // "function"
```

The `typeof` operator takes `fruit` as an argument and returns "function" due to function declaration taking precedence over variable assignment during hoisting.

An important point to keep in mind is that `let` and `const` cannot be re-declared in the same scope. Let's refactor the above statement to use `let` instead of `var`:

```
let fruit;  
  
function fruit() {  
}  
  
console.log(typeof(fruit)); //
```

You cannot re-declare `let` variables in the same scope, therefore, a `syntaxError` is returned:

```
SyntaxError: redeclaration of let fruit
```

This demonstrates one of the advantages of using `let` and `const` over `var` as discussed in Chapter 1.

While writing bug free code that can be used in production is the ultimate goal, errors can arise on the way due to quirks such as hoisting. Learning to handle errors effectively will greatly increase your productivity as a developer. In the next section of this chapter we will

cover error handling in JavaScript. Before that be sure to practice the coding exercises for this lesson.

## Exercises

1. Have a look at the code snippet and explain why the number 20 is logged to the console:

```
sum(10,10);  
  
function sum(x,y) {  
  
let add = x + y;  
  
console.log(add);  
  
}
```

2. A variable sum has been declared and assigned a value of 100. What will the `console.log` statement return now?

```
function sum(x,y) {  
  
let add = x + y;  
  
console.log(add);  
  
}  
  
sum(1, 2);  
  
var sum = 100;  
  
console.log(typeof(sum));
```

3. Will fruit be hoisted? If so what will the `typeof()` operator return?

```
console.log(typeof(fruit));  
  
var fruit = function() {  
  
let fruitJam = true;  
  
let toast = true;  
  
if (fruitJam && toast) {  
  
console.log(`I like fruit Jam and toast`);  
  
}  
  
}
```

## Answers

1. Functions are hoisted along with a reference to the body of the entire function. Therefore, when the function `sum()` is called before it is declared, the code statements within the function execute and `console.log` statement within the function executes.
2. Variable assignment takes precedence over function declaration. Therefore, the `typeof(sum)` will return "number" as `var sum` is assigned a numeric value of 100. And the variable assignment will be hoisted before the function `sum()`.
3. No, because only function declarations are hoisted. Function assignments are not hoisted. `Undefined`.

## 2.2 Error handling with try/catch/throw and finally

Handling potential errors is more useful than using the `console.log` statement to try and catch potential errors after they occur. By handling your errors you will be able to anticipate potential flaws in your code and as a result write more bug-free code.

In this section we will explore the try, catch and throw statements which are used to handle errors in JavaScript.

### 2.2.1 Try/Catch Block

Usually upon encountering an error, the script stops and an error message is logged to the console. In order to better handle logical and runtime errors, the **try-catch** block should be used. An important point to keep in mind is that syntax errors will not be checked for within a **try-catch** block.

#### Syntax:

```
try {  
    //code to try/test that might contain an error  
} catch (e) {  
    //code to execute in case of an error in the try block  
}
```

The **try block** will contain the code that you are checking for an error in. Whereas, the **catch block** will as the name suggests catch any potential errors. The catch statement takes as a parameter an error object `e`. The error object will have a `name` and `message` property. Let's explore this better with an example:

```

let price = 100;

try {
    price/discount;
} catch (error) {
    console.log(` ${e.name} ${e.message}`);
}

```

The `try` block divides `price` by `discount`. If an error is encountered the `catch` block, which takes the error object as an argument, will execute. Subsequently, within the catch block, a `console.log` statement will log:

- The value of the name property of the error object ("ReferenceError")
  - The value of the message property of the error object ("discount is not defined")
- "ReferenceError discount is not defined"

The `try-catch` block will allow you to make sense of errors in a meaningful way. By understanding your errors, you will be able to write better code. The error object is used extensively in node.js during backend development and is very useful as a way of catching potential errors.

## 2.2.2 Throw statement

You may also create your own custom error messages by using the `throw` statement. This is called *throwing an exception*. The purpose of a `throw` statement within the `try block`, is to intentionally introduce an error and try and contain it. Purposefully introducing an error will allow you to rule out a potential error and handle it. It makes debugging easier as it narrows down the range of potential errors. `throw` is used with the `try` and `catch` statements.

**Syntax:** `throw expression`

Let's take for example, a custom error to let a developer know when an item is discounted at 50% or more, as we don't want to offer free shipping on items less than \$50:

```

let price = 100;

let discount = 0.5;

let result = (price * discount);

try {

```

```

if (result <= 50) {

    throw new Error('The price is 50 and below');

}

} catch (e) {

    console.log(` ${e.name} ${e.message}`);
}

```

The above code block will execute as follows:

1. The `if` conditional statement checks if variable `result` references a value that is less or equal to the number `50`. If this evaluates to `true`, then:
    - Throw a custom error object created with the `new Error` constructor that will let the developer know that "The price is 50 and below"
- `"Error The price is 50 and below"`
2. If there is an error of another type then the code within the `catch block` will execute.

Carrying on from the previous example, the condition has been changed and it now checks if variable `result` is less than `50`. And in this case it is `50` but not less than `50`. Additionally within the `try` block we try to access a variable `x` that is not declared.

```

let price = 100;

let discount = 0.5;

let result = (price * discount);

try {

    if (result < 50) {

        throw new Error('The price is 50 and below');

    }

    console.log(x);

} catch (e) {

    console.log(` ${e.name} ${e.message}`);

}

```

In this instance the code in the `catch` block will be executed, as we have an error of another type:

```
"ReferenceError x is not defined"
```

Moving on, let's examine the `finally` statement.

### 2.2.3 Finally statement

The `finally` statement comes at the end of a `try-catch` statement and will execute regardless of what the result of a `try-catch` block is.

Example:

```
try{
    console.log(x);
} catch(error) {
    console.log(error.name + error.message)
} finally{
    console.log(`This will execute for sure`)
}
```

The statement within the `finally` block will execute, even though there is a `ReferenceError` which occurs as `x` is not declared. The statements that will be shown are:

```
"ReferenceError x is not defined"
"This will execute for sure"
```

Now that we know how to handle errors, let's have a look at the common types of errors that are encountered in JavaScript.

## 2.3 Types of errors

It's worth briefly discussing the common types of errors that you are likely to encounter while coding. Knowing what an error message means will pinpoint you in the right direction to finding a solution.

The common types of errors are:

- **SyntaxError**: Caused by invalid syntax

- **ReferenceError**: Caused by referencing a variable/object/function that doesn't exist.
- **RangeError**: This error is thrown when a value is out of range
- **URIError**: Thrown due to an invalid URI
- **TypeError**: Type errors are thrown when a variable is not of the expected or valid data type
- **Evaluator**: An error with the global `eval()` function throws an `Evaluator`. The current ECMAScript specification does not throw this error at runtime.
- **Internal Error**: Error is thrown when there is an internal error in the JavaScript engine.

We can minimize the probability of errors occurring by using `strict mode`. In the next section we will discuss this ES5 feature.

### Exercises

1. Mrs. Shear is a grade 3 teacher. She wants to ensure that students in her class learn in groups of 3. Write a function called `grouping()` which takes a number as a parameter. Within the function use a `try-catch` block to throw an error if the number passed into the function `grouping()` is not divisible by 3. If it is divisible by 3 then return the number of groupings that are formed. The `finally` block must let Mrs. Shear know that the function is over.
2. Describe 3 types of errors that are encountered while coding
3. Explain what type of error is thrown and why?

```
let num = 1

console.log(s.toUpperCase());
```

### Answers

1. The function `grouping()` is below:

```
function grouping(x) {
try {
if(x % 3 !== 0) {
throw new Error(` Not divisible by 3`)
```

```

} else {
let result = x / 3
console.log(result)
}
} catch(e) {
console.log(e.name + e.message)
} finally {
console.log(`Function is complete`);
}
}
grouping(9);

```

2. ReferenceError, typeError, rangeError.
3. **TypeError** as you are trying to use a string method on the number data type.

## 2.4 Strict mode

Strict mode is an ES5 feature which enforces a semantically stricter version of JavaScript. Strict mode reduces the number of silent errors by adhering to a stricter mode of itself. A better contextual understanding of what strict mode does is provided in the following sections by its use cases.

### 2.4.1 Enabling strict mode:

In order to enable strict mode globally, simply use the below directive, at the top of your script:

```
'use strict';
```

You can also limit the scope of “strict mode” by declaring it within a function:

```
function fun() {
  'use strict';
}
```

#### Note

Strict mode does not work in block scope {}

## 2.4.2. Use cases of strict mode

Listed below are some common use cases comparing to code with and without strict mode.

### 1. Global Variables

Take as an example, the following assignment:

```
x = 6;
```

Without strict mode, this will automatically create a variable/object in the global context:

```
var x = 6;
```

However, with strict mode enabled a global variable/object is not automatically declared.

```
'use strict';
x=6;
```

With strict mode enabled a `ReferenceError` is thrown:

```
ReferenceError: assignment to undeclared variable x
```

### 2. Deletion of variables, functions and objects

In non-strict mode, deletion of variables, objects, and functions is allowed:

```
let x;
function unstrictMode () {
}
let obj = {};
delete(x);
delete(unstrictMode());
delete(obj1);
```

Whereas in strict mode, deleting any one of them is not permissible and will throw an error:

```
'use strict'
let x;
function unstrictMode () {
}
let obj = {};
```

```
delete(x);  
delete(unstrictMode());  
delete(obj1);
```

The following error will be thrown in response to deleting a variable, object and function in strict mode:

```
SyntaxError: applying the 'delete' operator to an unqualified  
name is deprecated
```

### 3. Duplication of a parameter:

In strict mode you cannot duplicate a parameter name:

```
'use strict'  
  
function add(x, x) {  
  
    let sum = x + x;  
  
    console.log(sum);  
  
}
```

The duplication of the parameter `x` in the above code will throw an error:

```
SyntaxError: duplicate formal argument x
```

### 4. Naming of variables

In strict mode, you cannot name your variables “eval” or “arguments”. Also, you may not use any of the following keywords:

- interface
- let
- package
- implements
- private
- protected
- public
- static
- yield

## 5. Writing and deleting properties

Writing to a read-only property while not in strict mode will not throw an error. Take as an example the following code:

```
let friend = {  
    name: 'Lara',  
};  
  
Object.defineProperty(friend, 'job', {  
    value: 'developer',  
    writable: false  
});  
  
friend.job = 'Chef';  
  
console.log(friend.job); // Chef
```

Even though the `job` property of the `friend` object is not-writable, in non-strict mode we can write to read-only properties. Therefore, `Chef` will be logged when the `job` property of the `friend` object is queried.

Compare this to strict mode, wherein we will get the following error:

```
TypeError: "job" is read-only
```

Moving on, writing to a get-only property of an object using 'get' syntax is disallowed as well. We will explore this in much further details in chapter 3, *All about Objects*. For now, keep in mind that `you cannot write to a get-only property of an object in strict mode.`

Consider the following code sample which you have an object `obj`:

```
'use strict'  
  
let obj = {  
    get full() {  
        return 0  
    }  
};  
  
obj.full = 3;  
  
console.log(obj.full);
```

```
TypeError: setting getter-only property "full"
```

In strict mode you cannot write to get only properties. Therefore, when we try to change the `full` property of the object to the number `3`, we encounter a `TypeError`.

In strict mode, you cannot delete un-deletable properties either. For example:

```
'use strict'  
  
delete Object.prototype;
```

The following error is thrown in response to the above block of code:

```
TypeError: property "prototype" is non-configurable and can't  
be deleted
```

## 6. Other use cases

The other use cases of strict mode include the following which are not allowed and will throw an error:

- use of the `with` statement
- use of octal number literals
- use of escape characters

### 2.4.3 Advantages of strict mode

Strict mode will enforce reliability of code by throwing errors to ambiguously written code, for example, assignments to undeclared variables, deletion of variables objects and functions, and the other uses cases as discussed earlier in section 2.4.2.

```
console.log(` ${customer.nom} is a great ${customer.job}`);
```

#### Exercises

1. What is **strict mode** and why should you use it?
2. What is happening here and how can we stop it?

```
x = 6;  
  
console.log(x);
```

3. The following block of code is not in strict mode. What all from this block of code is not allowed while in strict mode?

```

customer = { };

function customerDetails() {

customer.nom = "Shadow Zilla";

customer.id = "808";

customer.job = 'Candy';

return (customer);

}

Object.defineProperty(customer, 'job', {

value: 'Designer',

writable: false

});

console.log(` ${customer.nom} is a great
${customer.job}`);
```

## Answers

1. Strict mode is an ES5 feature which enforces a semantically stricter version of JavaScript. Strict mode reduces the number of silent errors by adhering to a stricter mode of itself.
2. The assignment of `x` to `6` creates a variable in the global scope which is hoisted to the top of its scope irrespective of what line assignment takes place. Use `strict mode` to prevent this
3. The following is not allowed while in strict mode:
  - Assignment to an undeclared variable
  - Writing to a read only property of an object
  - Deleting a property of an object

## Summary

In this chapter we covered advanced basics of JavaScript. In addition to function hoisting, hoisting of variables declared using `var`, `let` and `const` was discussed. Error handling

allows you to handle potential errors more efficiently than relying upon `console.log`. Error handling and the common types of errors were also covered. Additionally when and why you should use `strict mode` an ES5 feature to write more robust JavaScript was discussed. The practice coding exercises in this chapter will make you feel more comfortable with these concepts and I suggest you complete them.

Having covered the basics, the next chapter will be all about objects in JavaScript. Objects are very powerful and are central to coding in JavaScript. The next chapter is jam packed with about 70 practice coding exercises, so keep moving on bit by bit!

# All about objects -3

*Skill comes from consistent and deliberate practice. — S. Allen*

Objects are very useful and central to making your code reusable and organized. In this section, we will go over fundamental and advanced object concepts in JavaScript. By the end of this chapter, you will be able to instantiate and manipulate objects effectively. You will also solve exercises to deepen your understanding of objects in JavaScript. I highly recommend that you do the exercises as the best way to learn coding is to code! Additionally, some Object Oriented Principles of JavaScript will be covered to enable you to write better object-oriented code.

In this chapter we're going to cover the following main topics:

- Objects in JavaScript
- Object creation
- Object iteration
- **this** keyword
- Prototype and Inheritance
- Classes
- Getters and Setters
- Shallow and Deep copy of an object

## 3.1 Objects in JavaScript

Objects are an integral part of JavaScript and a solid grasp of objects will allow you to be more efficient as a JavaScript developer. We briefly covered objects in Chapter 1, as we compared them to primitive data types that are copied by value. Whereas when dealing with object data types, the address in memory of an object is copied over rather than the actual value of the object. More about this later on this in chapter.

In this section, the two key concepts to grasp about objects are that they have:

- **key : value** pairs

- **Methods**

Objects are a compound data type. This means an object is a stand-alone entity (object) with unordered **key: value** pairs. **Key: values** pairs are also called **property: value** pairs. Compared to primitive data types which can reference only one value at a time, objects reference multiple values. Therefore, allowing us to make complicated structures.

The value of a property in JavaScript can be a:

- A Primitive data type such as those that we discussed in Chapter 1, *The Basics*
- A Reference data type such as another object or function.

A function attached to an object is called its **method** and will impart to the object behavior/state of its own.

The grouping of **property: value** pairs represented as a single entity can be powerful and as such objects can be used as the building blocks in an application, for example, you can have a **cart** object with various properties such as a **discount**, **shipping**, **tax**, etc.

Moving on in this chapter, we will go over the important aspects of objects, beginning with object creation.

### Exercises

1. What other **key: value** pairs can you think of (example credit card number and a pin)?
2. What is a method?
3. What is the difference between copy by value and copy by reference?

### Answers

1. Credit card and pin, lock and key, usb and port
2. A method is a function associated with an object
3. When dealing with object data types, the address in memory of an object is copied over rather than the actual value of the object (copy by value).

## 3.2 Object creation

Objects can be created in the following ways:

- Object literal syntax

- Object constructor
- Object.create() method
- assign() method
- ES6 classes

Each one of these object creation methods is discussed below.

### 3.2.1 Object literal syntax

Object literal syntax is similar to declaring a variable, except that the grouping of **property: value** pairs are enclosed within curly {} braces. There is a delimiting comma , in between each **property: value** pair except for the last one.

Let's look at an example of object literal syntax as a way of creating objects.

#### 3.2.1.1 Object creation with Object literal syntax

In this section, we'll explore object creation with an Object literal syntax.

```
let user = {
  id: 1,
  enrolled: true,
  tuition: 1800,
  message: function() {
    console.log(`The student is currently enrolled`);
  }
}
```

In the above example, the `user` variable references an object. It has 3 **property: value** pairs and one method. To re-iterate a **method is a function of an object**.

#### Note

A comma after the last **key : value** pair is called a trailing comma. Trailing commas are part of the ES5 specification. Transpilers such as Babel will remove the trailing comma, therefore ensuring that code works in older browsers.

### 3.2.1.2 Accessing properties of an object

To access any of the properties of an object, the dot notation can be used. The object's name is followed by the dot notation and lastly the name of the property to be accessed:

```
user.id; //1  
user.tuition; //1800  
user.enrolled //true
```

In order to call the method of an object, the object's name must be followed by the dot notation, the name of the method and parenthesis (). Similar to calling a function:

```
user.message(); // "The student is currently enrolled"
```

To check if an object has a property that is non-inherited and its own, you can use the **Object.hasOwnProperty()** method which takes as an argument the name of the property you want to check.

This method will return the boolean **true** value if the property belongs to an object and **false** if the property does not exist or is inherited.

Let's check if the **id** property exists in the **user** object that we declared in this section:

```
console.log(user.hasOwnProperty('id')); //true
```

This will return true as the **id** property belongs to the **user** object.

If we try to check for a non-existent property, then this method will return **false**:

```
console.log(user.hasOwnProperty('snacks')); //false
```

#### Note

The **object.hasOwnProperty()** method will return **false** for object properties that either don't exist or are inherited. We will learn about object inheritance later in this chapter in *section 3.5*.

### 3.2.1.3 Adding properties with the dot notation

It is preferred to assign **property: value** pairs inside the object declaration when the name of the properties and their associated values is known. You can also assign them afterward via the dot notation. The following demonstrates this:

```
let user = {};
```

`user` is an empty object to whom you can assign properties and associated values in the following way:

```
user.id = 1;  
user.enrolled = true;  
user.tuition = 1800;  
user.message = function() {  
    console.log(`The student is currently enrolled`);  
};
```

Besides being added, properties may also be deleted. The following section will explain this.

### 3.2.1.4 Delete a property

To delete a property from an object, the `delete` keyword is used. For example:

```
delete user.enrolled;
```

This will delete the `enrolled` property and its value from the object and the following property values are returned:

```
Object {  
    id: 1,  
    message: function() {  
        console.log(`The student is currently enrolled`);  
    },  
    tuition: 1800  
}
```

To re-add the deleted property from the object, re-assign it:

```
user.enrolled = true;
```

You can also change the values of the individual properties therefore, modifying them:

```
user.tuition = 2000;
```

### Exercises

1. Use the object literal notation to declare a variable called `author`.

- It must have 3 property: values pairs (`name`, `genre`, `year`) and one method called `introduction` which prints out the name of the author and the book genre. You can do this with and without the `this` keyword
- Call the introduction method on the author object using the **dot notation**

## Answers

1.

```
let author = {
    authorname: 'Toilken',
    genre: 'fantasy',
    year: 2000,
    introduction: function() {
        console.log(`name is ${this.authorname} and the genre
is ${this.genre}`)
    }
}
author.introduction();
```

"name is Toilken and the genre is fantasy"

### 3.2.1.5 Object literal enhancement

So far we have seen two ways of initializing an object literal using the object literal syntax. We have also gone over deleting and modifying the properties of an object. What if you want to initialize an object based on the value of a variable? In such a case, **object literal enhancement** can be used.

An object literal enhancement is offered by ES6 which lets you assign variable as properties. Therefore, delegating the value of the variable to be the value of the property. This is demonstrated in the following code:

```
let id = 1;
let enrolled = true;
let tuition = 1500;
let message = function() {
```

```
console.log(`The student is currently enrolled`);  
};  
  
let user = {  
    id,  
    enrolled,  
    tuition,  
    message  
}  
  
console.log(user);
```

The following information about the `user` object will be logged to the console:

```
Object {  
    enrolled: true,  
    id: 1,  
    message: function() {  
        console.log(`The student is currently enrolled`);  
    },  
    tuition: 1500  
}
```

In the above code snippet, the variable names correlate to property names of the object. And the values of the variables are the values of the properties in the `user` object.

We may also access any of the properties individually:

```
console.log(user.id); //1
```

## Exercises

1. Use an object literal enhancement to initialize an object using these variables:

```
let role = 'frontend';  
  
let employed = true;  
  
let vacation = '89';
```

```
let message = function() {
    console.log(` ${role} position has ${vacation} days of
vacation`);
};
```

## Answers

1.

```
let employee = {
    role,
    employed,
    vacation,
    message
}
console.log(employee);
employee.message();
```

### 3.2.1.6 Create dynamic keys in ES6

It is also possible to create dynamic keys with the object literal enhancement. This is better explained with an example:

```
let num1 = 10;
let num2 = 20;
let num3 = 30;

let numObject = {
    [num1+1]: num1,
    [num2+1]: num2,
    [num3+1]: num3,
}
console.log(numObject);
```

The following will be logged to the console:

```
Object {  
  11: 10,  
  21: 20,  
  31: 30  
}
```

The object keys are placed inside square brackets `[]` and dynamically assigned a value by incrementing the existing values by one. Therefore, allowing us to create dynamic keys without having to iterate through the object to update the keys.

Object literal syntax is not the only way to create objects, the `new()` keyword can be used as well to create a new object. We will explore how to do this in the next section.

### Exercises

1. Create an object literal using an ES6 object literal enhancements that allows you to add dynamic keys to an object such that the final object is:

```
Object {  
  Tier-1: "90%",  
  Tier-2: "80%",  
  Tier-3: "70%"  
}
```

### Answers

- 1.

```
let tier = 'Tier';  
  
let i = 1;  
  
let tierRanks = {  
  [tier + '-' + i++]: '90%',  
  [tier + '-' + i++]: '80%',  
  [tier + '-' + i++]: '70%'  
}
```

```
console.log(tierRanks);
```

### 3.2.2 new() keyword

The **new()** operator is used to instantiate objects from a constructor function as we will see in this section.

#### 3.2.2.1 In-built object constructor function

The **new** operator can be used to create an object using the in-built **Object** constructor function. Constructor functions are like regular functions with the exception that:

- They start with an upper case letter
- They are executed with the **new** keyword

For example down below, we create an object using the **new** keyword and **Object** constructor function:

```
let employee = new Object();
```

Querying the **typeof(employee)** results in **object**:

```
console.log(typeof(employee)); // "object"
```

Constructor functions are akin to templates or blueprints from which many objects can be created, using the constructor function as a blueprint. The in-built object constructor function is a function that will initialize and return an empty object. The empty object is then referenced by a variable, which in our case is **let employee**.

Then properties can be added to this object via the dot notation as we saw earlier in *section 3.2.1* of this chapter. For example:

```
employee.id = 100;  
employee.role = 'admin';  
employee.salary = 100000;  
employee.raise = employee.salary * 2;
```

However, what if we want to create many instances of a template object? It would be redundant and tiring to initialize those via the methods that we have learned so far as we would have to add properties to them one by one. In such a case we can utilize a user defined object constructor function.

Before we move on, I want to draw to your attention that you have probably used JavaScript's in-built constructor functions as `new Date()`, `new Array()`.

## Exercises

1. Code the following:

- Use the `new` keyword to instantiate a new `randomNumber` object using the in-built Object constructor function
- Add a `luckydraw` property whose value is the boolean `true`
- Add a method called `value` which will generate a random number between `0` `-10`
- Check if random numbers are generated by calling the `value` method on `randomNumber` object

## Answers

1.

```
let randomNum = new Object();

randomNum.luckydraw = true;

randomNum.value = function() {
    console.log(Math.floor(Math.random() * 11));
}

randomNum.value();
```

### 3.2.2.2 User defined object constructor function

We can also declare and use our own constructor functions. To create a user-defined `employee` object using a constructor function, we will first define the constructor function. For example:

```
function EmployeeConstructor(id, role, salary, raise) {
    this.id = id;
    this.role = role;
```

```
    this.salary = salary;  
    this.raise = this.salary * 2;  
}
```

The above code block can be de-constructed as follows:

- A function called `EmployeeConstructor` takes 3 parameters: `id`, `role` and `salary`
- The value of the `role`, `id` and `salary` properties will be the values referenced by the arguments passed in to the function (`id`, `role` and `salary`) when the function is called
- The value of the `raise` property is `salary` times 2
- The `this` keyword refers to the particular object that is created when the constructor function is called

Now that we have defined a constructor function, let's use it to create three different instances of an employee object using the `new` keyword:

```
let employee1 = new EmployeeConstructor(1, 'admin', 100000);  
let employee2 = new EmployeeConstructor(2, 'dev', 105000);  
let employee3 = new EmployeeConstructor(3, 'manager', 50000);
```

Therefore, using the `EmployeeConstructor` function with the `new` operator, user defined objects can be instantiated easily. They are regular objects. For example we can query the `role` property of `employee2`:

```
console.log(employee2.role);
```

And what we get is the value of the `role` key:

```
"dev"
```

Logging all the property value pairs for `employee2` returns:

```
Object {  
  id: 2,  
  raise: 210000,  
  role: "dev",
```

```
    salary: 105000  
}
```

Next we will look at creating objects with the `Object.create()` method.

### Note

It is standard to name a constructor function starting with an uppercase letter.

### Exercises

1. Code the following constructor function:
  - Declare a constructor function called `Dessert`, which takes 4 parameters `name`, `calories`, `flavor` and `helpings`.
  - Define the value of these parameters within the object using the `this` keyword.  
Add a method called `totalCal` within the constructor function which will `console.log` the number of calories multiplies by the number of helpings
  - Create a `cake` object using the dessert constructor function which has the following properties:
    - `name: Bamboozle`
    - `calories: 1000`
    - `flavour: 'chocolate'`
    - `helpings: 3.5`
  - Call the `totalCal ()` method what is logged?

### Answers

1.

```
function Dessert(name, calories, flavor, helpings) {  
  this.name = name;  
  this.calories = calories;  
  this.flavor = flavor;  
  this.helpings = helpings;  
  this.totalCal = function() {
```

```

        console.log(this.calories * this.helpings);

    }

}

let cake = new Dessert('Bamboozle',1000, 'chocolate',
3.5);

cake.totalCal(); //3500

```

### 3.2.3 Object.create() method

According to the Mozilla Developer Network (MDN):

*The Object.create() method creates a new object, using an existing object as the prototype of the newly created object.*

**Syntax :** `Object.create(prototype [ ,propertiesObject])`

`Object.create()` takes 2 parameters:

- An object which will be the prototype from which another object is created
- An optional properties object which will contain enumerable **property : value pairs** to be added to the newly created child object

This method uses JavaScript's built-in Object type. The built-in Object allows you to make your own object. The built-in object type has many properties and methods which user created objects can access.

Using the `Object.create()` method you can use one or more existing objects to create your own new object, thus implementing inheritance. Let's have a look at this example to elucidate this:

```

let parentObj ={
  descent: 'Scottish descent',
  diabetes: false
};

```

In the above code we have an object called `parentObj`. The object has **two property : value** pairs. This object serves as a prototype to instantiate a new object using the `Object.create()` method.

Let's now use `Object.create()` to create a new child object:

```
let childObj = Object.create(parentObj, {});
```

Here we've created a new `childObj` object using the `Object.create()` method which takes a compulsory object as the first argument, which will be the prototype to base the child object on. And a second optional object which is empty. If we `console.log` the `employee` object we are returned with:

```
console.log(childObj);
```

```
Object {  
  descent: "Scottish descent",  
  diabetes: false  
}
```

We see that `childObj` has inherited the prototype's `parentObj` properties. We can further add properties to the `childObj`:

```
childObj.firstName= 'Lara';  
childObj.age = 29;
```

Upon logging the `childObj` to the console, we get the child object's properties as well its prototype parent's properties:

```
Object {  
  age: 29,  
  descent: "Scottish descent",  
  diabetes: false,  
  firstName: "Lara"  
}
```

Let's now make a `grandchildObj` object using the `Object.create()` method. It will inherit directly from the `childObj`. We will change the values of the `firstName` and `age` properties inherited from the `childObj`:

```
let grandChildObj = Object.create(childObj, {});  
grandChildObj.firstName = 'Mutty';  
grandChildObj.age = 3;
```

The newly created object contains the following property value pairs:

```
Object {  
    age: 3,  
    descent: "Scottish descent",  
    diabetes: false,  
    firstName: "Mutty"  
}
```

Therefore, `grandchildObj` has inherited properties from `childObj`, which in turn has inherited properties from `parentObj`. Inheritance chains such as this one can be set up using the [Object.create\(\)](#) method.

### Deleting inherited properties

Inherited properties cannot be deleted. The property must be deleted from the prototype object instead. Deleting a property from the prototype object will also delete it from child objects.

Therefore the objects created with the [Object.create\(\)](#) method are object literals and are instances of JavaScript's built-in Object type.

Next we will discuss using the ES6 `Object.assign()` method as a way of creating objects.

### Exercises

1. Consider the following object:

```
let department = {  
    name: 'Entertainment',  
    fulltime: true  
}
```

- Create a child object called `musicDepartment` using the [Object.create\(\)](#) method
- Add the following **key : value** pairs to it:

```
employees: 200;  
remote: true;
```

- Delete the `name` property from the `musicDepartment` child object.

- What **key : value** pairs exist inside `musicDepartment` now?
2. What must you do to completely delete the `name` property from the `musicDepartment` child object?

### Answers

1.

```
let musicDepartment = Object.create(department, {});  
  
musicDepartment.employees= 200;  
  
musicDepartment.remote = true;  
  
delete musicDepartment.name;  
  
console.log(musicDepartment);
```

```
Object {  
  employees: 200,  
  fulltime: true,  
  name: "Entertainment",  
  remote: true  
}
```

2. Delete it from the parent object that it inherits from:

```
delete department.name;
```

### 3.2.4 Object.assign() method

This method was introduced in ES6 and will copy one or more source object's **own** enumerable properties to a target object. And then returns the target object.

**Syntax:** `Object.assign(target, ...sources)`

This method takes a target object as the first parameter and an additional unlimited number of source objects as additional parameters. Only the source object's own enumerable properties will be copied over to the target object. This simply means that the properties of the source object must be enumerable i.e. you can iterate through the object's properties that have **not** been inherited from its prototype.

For example:

```

let sourceObj1 = {
  id_1: 1,
  category1: 'home'
}

let sourceObj2 = {
  id_2: 2,
  category2: 'electronics'
}

let sourceObj3 = {
  id_3: 3,
  category3: 'kitchen'
}

let targetObj = Object.assign({}, sourceObj1, sourceObj2,
sourceObj3);

console.log(targetObj);

```

Let's see what is happening here:

- There are three source objects: `sourceObj1`, `sourceObj2` and `sourceObj3` with two properties each.
- We use the `Object.assign()` method passing in an empty object which will be the target object and the three source objects whose own enumerable properties will be copied over to the target object. This is referenced by the `targetObj` variable.

An important point to note here is that the properties of all the source objects have different names. This is because they will be copied over to the target object so there cannot be duplicate property names, as seen when we log `targetObj` to the console:

```

Object {
  category1: "home",
  category2: "electronics",
  category3: "kitchen",
}

```

```
    id_1: 1,  
    id_2: 2,  
    id_3: 3  
}
```

Therefore, the property names must be unique in an object.

### Exercises

1. SuperStore a company competing with Walmart has hired you! They would like to have a central source of truth for all child companies that will be part of their SuperStore. Create a `superstore` object and copy over all the key value pairs from the `food`, `hardware` and `clothing` objects using the `Object.assign()` method:

```
let food = {  
  name: 'Gill SuperStore',  
  locations: ['Albuquerque', 'Orlando', 'Toronto']  
}  
  
let hardware = {  
  name: 'Supermax Hardware',  
  locations: ['Cairo']  
}  
  
let clothing = {  
  name: 'Cloth mania',  
  locations: ['Vietnam', 'Jakarta']  
};
```

**Note:** Keep in mind that that property names of the source objects must have different names, so perhaps you should differentiate between the `name` and `locations` property names for all three source objects by changing them (Hint: `name1, name2..`)

### Answers

- 1.

```

let food = {
    name1: 'Gill SuperStore',
    locations1: ['Albuquerque', 'Orlando', 'Toronto']
}

let hardware = {
    name2: 'Supermax Hardware',
    locations2: ['Cairo']
}

let clothing = {
    name3: 'Cloth mania',
    locations3:
        ['Vietnam', 'Jakarta']
}

let superstore = Object.assign({}, food, hardware,
clothing)

console.log(superstore);

```

```
[object Object] {
    locations1: ["Albuquerque", "Orlando", "Toronto"],
    locations2: ["Cairo"],
    locations3: ["Vietnam", "Jakarta"],
    name1: "Gill SuperStore",
    name2: "Supermax Hardware",
    name3: "Cloth mania"
}
```

### 3.2.5 ES6 Classes

Classes introduced in ES6 allow you to create objects easily from a template function. We will discuss classes in detail later on in this chapter. For now, we will look at how to create classes and then instantiate objects from a class using the **new** keyword.

A class is a special type of function which is declared with the **class** keyword:

```
class Character{  
constructor() {...}  
method() {...}  
}
```

Class declarations are not hoisted; therefore you cannot use them prior to declaration.

Inside the `class Character`, a `constructor()` method is used which will add properties to instances of the `Character` class. A class can only have one constructor `method()`:

```
class Character {  
    constructor(food, energy, points) {  
        this.food = food;  
        this.energy = energy  
        this.points = points;  
    }  
    friendship(){  
        return(this.energy + this.points);  
    }  
}
```

The `constructor()` method takes three parameters `food`, `energy` and `points`. And then inside the constructor method called `friendship()`, the `this` keyword is used to bind the values of the `food`, `energy` and `points` parameters passed to an instance of an object created using a class. `friendship()` is a class method which will return this sum of parameters `energy` and `points` of an object instance:

We can now use the `class Character` template to spawn new objects like so using the `new` keyword and the name of class:

```
let tomNook = new Character('bread',100, 2340);  
let kidCat = new Character('milk', 10, 1340);  
let baaBara = new Character('apples', 40, 1870);
```

In the code example, we have spawned three new object instances using the `Character` class. If we log one of these to the console, a list of `key : value` pairs will be returned for that particular instance. For example:

```
console.log(baaBara);
```

```
Object {  
  energy: 40,  
  food: "apples",  
  points: 1870  
}
```

You need not only use class declarations. Class assignments are possible as well, just like function assignments. For example:

```
let Character = class {  
  constructor(food, energy, points) {  
    this.food = food;  
    this.energy = energy  
    this.points = points;  
  }  
  friendship() {  
    console.log(this.energy + this.points);  
  }  
};
```

This is a brief introduction to classes in JavaScript. They will be covered in detail in *section 3.4*. Next, we will cover iterating over objects in JavaScript.

### Exercises

1. A class is a special type of function declared using the `Class` keyword: True or False?
2. What is the purpose of using a class?

3. Create the following `Player` class:

- Create a class called `Player` whose constructor takes 3 parameters `energy`, `power`, and `level`
  - Add a method called `powerControl()` to the `Player` class which will calculate subtracting `energy` from `power` and return a string  
`'Remaining power left is' + power`
- Create an object called `ryu` from the `Player` class with the following values:

```
energy: 200
power: 500
level: 1
```

- `console.log ryu`, what is displayed?

4. Use the `Object.create()` method to create a new object called `chunLi` from the `ryu` object which acts as the parent object.

## Answers

1. True
2. Classes introduced in ES6 allow you to create objects easily from a template function.
- 3.

```
class Power {
    constructor(energy, power, level) {
        this.energy = energy;
        this.power = power;
        this.level = level;
    }
    powerControl() {
        let powerPoints = this.power - this.energy;
    }
}
```

```
        return (`You have these many power points  
${powerPoints}`);
    }
}

let ryu = new Power(200, 500, 1);
ryu.powerControl();
console.log(ryu);
```

```
Object {  
  energy: 200,  
  level: 1,  
  power: 500  
}
```

4.

```
let chunLi = Object.create(ryu, {});  
console.log(chunLi);
```

```
Object {  
  energy: 200,  
  level: 1,  
  power: 500  
}
```

### 3.3 Object iteration

To iterate through the **key : value** pairs in an object, use the **for-in** statement. This will iterate through all the enumerable properties of an object. Consider the following object:

```
let simpleObject = {  
  id: 1,  
  type: 'simple',  
  level: 10,  
  status: true,  
}
```

In order to iterate through the **key : value** pairs in the `simpleObject` object, use the **for-in** statement:

```
for(key in simpleObject){  
  console.log(simpleObject[key]);  
}
```

This will loop through the `simpleObject` object and log to the console the value of each key in the iteration:

```
1  
"simple"  
10  
true
```

You can also convert an object into an array and then loop through the array using the following three methods:

- `Object.keys`
- `Object.values`
- `Object.entries`

`Object.keys()` takes an object as an argument and it will iterate through an object and return an array of its properties. For example, let's use this `Object.keys()` method on `simpleObject`:

```
console.log(Object.keys(simpleObject));
```

The `Object.keys()` method returns an array of the object's properties. In our case `simpleObject`:

```
["id", "type", "level", "status"]
```

`Object.values()` will iterate through an object and return an array containing the values of the properties in an object. Let's use it on the `simpleObject` object:

```
console.log(Object.values(simpleObject));
```

The values of the keys within the `simpleObject` object are logged to the console:

```
[1, "simple", 10, true]
```

Lastly, the `Object.entries()` method will create an outer array which contains arrays containing the **key: value** pairs as array items. To demonstrate this, let's use this method on `simpleObject`:

```
console.log(Object.entries(simpleObject));
```

This will return an array of arrays. And these inner arrays will contain the **key : value** pairs of the object as array items:

```
[["id", 1], ["type", "simple"], ["level", 10], ["status", true]]
```

These three methods will convert an object into an array and return an array, and then you may iterate through the array.

Next, we will cover a topic in JavaScript that leads to much fear and confusion: **this** keyword.

## Exercises

1. Iterate through the following object using the **for-in** statement and `console.log` the value of each key:

```
let coffee = {  
    roast: 'medium',  
    blend: 'Ethopian',  
    servings: 100,  
    morningMsg: function(){  
        console.log(`Ooh! The smell of an ${this.blend} blend  
in the mornings`)  
    }  
}
```

2. What method will return an array of the properties inside the `coffee` object?

3. What 3 methods can you use to convert an object to an array and then iterate through the converted array?
4. What does the **Object.values()** method do? Apply it on the **coffee** object. What is returned?
5. Consider the **coffee** object modified as below. What does the **Object.entries()** method do? Apply it on the **coffee** object. What will be the output?

```
let coffee = {
    roast: ['light', 'medium', 'dark', 'extra dark'],
    blend: ['Ethopian', 'Columbian', 'American'],
    servings: ['small', 'medium', 'large'],
    morningMsg: function() {
        console.log(`Ooh! The smell of an ${this.blend} blend
in the mornings`)
    },
}
```

6. Check if object **ob** is empty:

```
let ob = {};
```

## Answers

- 1.

```
for(key in coffee) {
    console.log(coffee[key]);
}
```

```
"medium"
"Ethopian"
100
function() {
    console.log(`Ooh! The smell of an ${this.blend} blend in
the mornings`)
```

```
}
```

2. `Object.keys()`

```
console.log(Object.keys(coffee));  
["roast", "blend", "servings", "morningMsg"]
```

3. 3 methods to convert an object to an array:

- `Object.keys`
- `Object.values`
- `Object.entries`

4. `Object.values()` will iterate through an object and return an array containing the values of the properties in an object
5. The `Object.entries()` method returns an array of arrays. Each inner array contains the property value pairs as array items

```
console.log(Object.entries(coffee));
```

```
[["roast", ["light", "medium", "dark", "extra dark"]],  
 ["blend", ["Ethopian", "Columbian", "American"]],  
 ["servings", ["small", "medium", "large"]],  
 ["morningMsg", function(){  
   window.runnerWindow.proxyConsole.log(`Ooh! The smell of  
   an ${this.blend} blend in the mornings`)
```

6. The following will return `true` for an empty object:

```
console.log(Object.keys(ob).length === 0 &&  
ob.constructor === Object)
```

We use the in-built `Object.keys()` method to check if the `length` of the properties is `0`. We also check if the constructor of the object `ob` is `Object` this will check any wrapper instances.

We can also create a function which uses `hasOwnProperty()` method:

```
function emptyObj(object) {  
  for (let key in object) {  
    if (object.hasOwnProperty(key))
```

```
        return false;  
    }  
    return true;  
}  
emptyObj (ob);
```

## 3.4 this keyword

The **this** keyword can cause confusion among beginner and more advanced developers alike. In order to understand **this**, it is imperative to first understand what execution context means.

In JavaScript, **this** is a reference to an object. The object that **this** will refer to depends on whether it is within the global context, inside an object, or within a constructor. We can also explicitly define the value of **this** with function prototype methods such as **apply()**, **call()** and **bind()**.

When a function is called, an execution context is created. The execution context contains information about the environment within which the current block of code is being executed. This information includes where the function has been called from, its parameters, how the function is invoked and then contingent upon this information a binding to **this** is made

*Function declaration does not matter , how a function is called is what matters.*

How a function is called is referred to as its **runtime binding** and this, is what determines the value of the **this** keyword.

We will have a look at four different ways of determining the value of **this** by examining its binding.

### 3.4.1 Global (default) binding

**this** refers to the global object (`window`) in the global execution context, when not in strict mode. For example:

```
function globalContext () {  
    console.log(this); // [object Window]  
}  
globalContext();
```

However, when in strict mode, **this** inside a function within the global context will return **undefined**:

```
'use strict';

function globalContext() {
    console.log(this);
}

globalContext(); // undefined
```

### 3.4.2 Implicit binding

Here the value of **this** is determined implicitly by its context. We will discuss the following three contexts:

- Object method
- **new** keyword
- DOM event handler

#### 3.4.2.1 Object method:

When used as part of a method, **this** will refer to its nearest parent object. Take for example the following object:

```
let Tanukichi = {
    airlines: 'Dodo',
    miles: 1000,
    greeting: function() {
        console.log(`Hello, you have ${this.miles} miles`)
    }
}

Tanukichi.greeting(); // "Hello, you have 1000 miles"
```

In the example above, the message **"Hello, you have 1000 miles"** is logged to the console. The call site of the function is within the object and the value of **this** refers to the **miles** **property** of the object **Tanukichi**.

```
let Tanukichi = {
```

```

airlines: 'Dodo',
miles: 1000,
greeting: function() {
    console.log(`Hello, you have ${this.miles} miles`);
}
}

var miles = 50;
let tanukichiGreeting = Tanukichi.greeting;
tanukichiGreeting(); // "Hello, you have 50 miles"

```

In the above example, the call-site of **this** is no longer the object. Instead it is within the global context and the value of **this** is now bestowed by the global **miles** variable whose value is **50**.

### 3.4.2.2 new binding:

When using the **new** keyword as we did in *section 3.2.2* of this chapter, the value of **this** refers to the new object that is being created. Refer to the following example:

```

function tanukichi(miles) {
    this.airlines= 'Dodo'
    this. miles = miles;
    this.greeting = function(){
        console.log(`Hello, you have ${this.miles} miles`)
    }
}

let tom = new tanukichi(50);

```

Calling the **greeting()** method on the newly created **tom** object, will return:

```
tom.greeting();
```

```
"Hello, you have 50 miles"
```

Thus the context of the **this** keyword is the **tom** object.

### 3.4.2.3 DOM event handler:

In the event handler `addEventListener()` `this` refers to the current target of the event, for example a button. This is extremely useful when using vanilla JavaScript to dynamically create elements in the DOM and targeting them. We will go over the DOM in detail later on in this workbook.

Take for example, the dynamic creation of a button in the DOM:

```
let button = document.createElement('button')
button.innerHTML = 'Click';
document.body.append(button)
```

Let's listen for a click event using the DOM `addEventListener()` method and log the value of `this` in the callback function of the method:

```
button.addEventListener('click', function(event) {
  console.log(this)
}) ;
```

The message that is logged to the console will refer to the current newly created button element:

```
"<button>Click</button>"
```

### 3.4.3 Explicit binding

There are 3 methods available for you to use in order to explicitly set the context/value of `this`, independent of how the function is called. These are:

1. `call()`:
2. `bind()`
3. `apply()`

1. `call()`: This method will set the value of `this` inside the function and execute it immediately. Keep in mind that functions are higher order objects which is why we can use methods with functions, passing in arguments, such as objects.

Let's look at a simple example to demonstrate `call()`:

```
function greeting() {
```

```
    console.log(`You are at the ${this.level} level in the
game`);
}
```

The function `greeting`, will log to the console a message indicating a player's level in a game. The `this` keyword is referenced from inside the function to the `level` property, though it has no context. Let's see how we can use this with an object:

```
let merengue = {
  level: 'Intermediate'
}
```

`merengue` is an object with a `level` property whose value is the string '`Intermediate`'. Let's use the `call()` method on function `greeting`. Passing in `merengue` as an argument:

```
greeting.call(merengue);
```

The following message will be logged to the console:

```
"You are at the Intermediate level in the game"
```

The value of `this` is explicitly set in the function itself. Whereas the value/context of `this` is bestowed by the `call()` method.

The `call()` method can take more than one argument. Let's change the function `greeting`, so that we can pass in these arguments to `call()`:

```
function greeting(points, bells) {
  console.log(`You are at the ${this.level} level in the game.
You have ${points} points and ${bells} bells`);
}
```

### Note

There is no reference to `this` when referring to `points` and `bells` in the function declaration. We will pass in these values as arguments when we apply `call()`, further down.

Using the same `merengue` object in order to reflect the new values of `this`:

```
let merengue = {  
    level: 'Intermediate'  
}
```

Passing in `merengue` object with two more properties to the `call()` method will result in the following message:

```
greeting.call(merengue, 5000, 100000);
```

```
"You are at the Intermediate level in the game. You have 5000  
points and 100000 bells"
```

Therefore, using `call()` we are able to connect the `greeting` function and `merengue` object. The value of `this` is inferred by the arguments passed into `call()`.

2. `apply()`: This is similar to `call()` except that it accepts an array of arguments instead of comma separated arguments.

In function `greeting`, there is a `this` reference but the `this` keyword has no context.

```
function greeting(message, mood) {  
  
    console.log(` ${message} You are at the ${this.level} level in  
    the game. You have ${this.points} and ${this.bells} bells.  
    Hopefully you're in a ${mood} mood`);  
}
```

The `merengue` object has three **key : value** pairs:

```
let merengue = {  
  
    level: 'Intermediate',  
  
    points: 50000,  
  
    bells: 100000  
}
```

The function `greeting` and `merengue` object have no connection to each other.

However, by using `apply()` we are able to *invoke the this context of the object on the function*:

```
greeting.apply(merengue, ['Hello there.', 'happy']);
```

The `apply()` method takes two arguments:

- An object, which in this case is `merengue`
- An array of string values , `['Hello there.', 'happy']`

The following message will be logged to the console:

```
"Hello there. You are at the Intermediate level in the game.  
You have 50000 and 100000 bells. Hopefully you're in a happy  
mood"
```

By using the `apply()` method you are able to invoke the `this` context of an object on a function as demonstrated.

3. `bind()`: This method is similar to `call()`, however it will return a new function and set `this` to a specific object. And like `call()` it can accept comma separated arguments.

Carrying on from the previous example, let's use `bind()`:

```
function greeting(points, bells) {  
  
  console.log(`You are at the ${this.level} level in the game.  
  You have ${points} points and ${bel...`);  
  
}
```

There is a `this` reference but the `this` keyword has no context in the function. Nothing has changed in this function, it is the same as we saw in `call()`. We will also leave the `merengue` object as is:

```
let merengue = {  
  
  level: 'Intermediate',  
  
}
```

Now let's use the `bind()` method which takes `merengue` as an argument and will set the context of the `this` keyword to the `merengue` object. Additionally, it will create a new function which is referenced by variable `userMsg`:

```
let userMsg = greeting.bind(merengue, 5000, 100000);
```

Since the `bind()` method creates a new function, which in our case is referenced by variable `userMsg`, let's call the function:

```
userMsg();
```

This will result in the following message logged to the console:

```
"You are at the Intermediate level in the game. You have 5000  
points and 100000 bells"
```

The difference between the `call()` method and `bind()` method is that `bind()` creates a new function. Therefore, it will always return the original context of the `this` keyword.

Moving on we will examine prototype and inheritance in JavaScript, the understanding of which will allow you to write object oriented JavaScript code.

### Exercises

1. Can you explain what `this` is in JavaScript?
2. Consider the following function. What will the value of the `this` keyword be?

```
function log () {  
    console.log(this);  
}  
  
log();
```

3. What will the `this` keyword refer to when in strict mode:

```
function log () {  
    'use strict'  
    console.log(this);  
}  
  
log();
```

4. Declare an object called `HR`. The `HR` object has 3 `key : value` pairs and 1 method. The 1<sup>st</sup> key is called `company` and its value is the string `Zimpak Software`. The second key is called `hiring` whose value is the boolean `false` and the 3<sup>rd</sup> key is called `employees` and its value is the number `100`. The 4<sup>th</sup> key is a method called `message`. The method should return a string which prints the name of the company, the number of employees and the value of the `hiring` key if the number of employees is less than or equal to `100`. Else it should return the string `Zimpack is not hiring`. You must use the `this` keyword
5. Carrying on from #4, change the value of `employees` to `20`. Outside of the object declaration. What does the `message` method return now? Think about what the context of the `this` keyword will be and the call-site of `this`.

6. Consider the following class called `Company`:

```
class Company {  
    constructor(name, hiring, employees) {  
        this.company = name;  
        this.hiring = hiring;  
        this.employees = employees;  
    }  
    message() {  
        if (this.employees >= 400) {  
            console.log(`${this.company} is currently not  
hiring`);  
        } else {  
            console.log(`${this.company} is currently hiring`)  
        }  
    }  
}
```

- Declare a new object called `moogle` from the `Company` class
  - Pass in three arguments for the `name`, `hiring` and `employees` properties
  - Call the `message` method on the your newly created object
  - What is the value of `this` now?
7. What does `call()` do?
8. Consider the following code block:

```
function restaurant(stars, cuisine) {  
    console.log(`Welcome to ${this.name}. We have ${stars}  
stars and serve ${cuisine} cuisine`);  
}
```

- Declare an object and use the `call()` method in such a way that the following message will be logged to the console:

```
"Welcome to Pataya. We have 4 stars and serve Thai  
cuisine"
```

9. Consider the following object:

```
let bretLee = {  
    name: 'Bret Lee',  
    points: 400  
}
```

- Invoke the `apply()` method on a function such that the message that is logged to the screen is

```
"Bret Lee is playing in miniLeague with 400 points for  
this Summer"
```

10. Define the `bind()` method. What does it do?

11. Complete the following:

- Write a function called `gymMembership` that takes a variable `fee` as a parameter. This function will calculate the total remaining amount of a gym membership fee that has been paid in full for the year. So we want to deduct a certain amount per month, which is unknown as yet. Use the `this` keyword which will refer to the total amount remaining. The function should `console.log this.name` and the remaining value after deducting the monthly fees.
- Declare an object called `sukhi` with a name property whose `name` value is '`sukhi`' and a total property whose value is `1000`
- Bind the `gymMembership` method to the `sukhi` object and pass as an argument the number `100` which is the monthly fees for the gym membership
- Assign this to a variable called `getFee`
- Call `getFee()` what is logged to the console?

## Answers

1. In JavaScript, `this` is a reference to an object. The object that `this` will refer to depends on whether it is within the global context, inside an object, or within a constructor function.
2. The global window object
3. `undefined`

4.

```
let HR = {  
    company: 'Zimpak Software',  
    hiring: true,  
    employees: 130,  
    message: function() {  
        if (this.employees <= 100) {  
            console.log(` ${this.company} hiring status is  
${this.hiring} and it currently has ${this.employees}  
employees`);  
        } else {  
            console.log(` ${this.company} is not hiring`);  
        }  
    }  
}  
  
HR.message();
```

5. `HR.employees = 20;`

```
"Zimpak Software hiring status is true and it currently  
has 20 employees"
```

6.

```
let moogle = new Company('Moogle', true, 10)  
moogle.message();
```

```
"Moogle is currently hiring"
```

- the value of `this` refers to the new object (`moogle`) that is created.

7. Using `call()` you can write methods that can be used on different objects

8.

```
function restaurant(stars, cuisine) {
```

```

        console.log(`Welcome to ${this.name}. We have ${stars}
stars and serve ${cuisine} cuisine`);

}

let pataya = {
    name: 'Pataya'
}

restaurant.call(pataya, 4, 'Thai')

```

9.

```

function player(currentLeague, season) {
    console.log(` ${this.name} is playing in
${currentLeague} with ${this.points} points for this
${season} `)
}

let bretLee = {
    name: 'Bret Lee',
    points: 400
}

player.apply(bretLee, ['miniLeague', 'Summer']);

```

10. This method is similar to `call()`, however it will return a new function and set `this` to a specific object. And like `call()` it can accept comma separated arguments

11.

```

function gymMembership(fee) {
    console.log(` ${this.name} has a remaining balance of:
${this.total - fee}`);
}

let sukhi = {
    name:'sukhi',
    total:1000
};

```

```
let getFee = gymMembership.bind(sukhi, 90);
getFee();
"sukhi has a remaining balance of: 910"
```

## 3.5 Prototype and Inheritance

JavaScript is a prototype-based language. This means that that a template object's properties and methods can be cloned and extended onto other objects. Therefore, a prototype is an object instance from which other objects can be created. This is known as **prototypal inheritance**. In this section, we will delve into prototypes and inheritance in JavaScript.

### 3.5.1 Prototype

In JavaScript all objects have an internal property, called `[ [Prototype] ]`. For example let's create a new object called `car`:

```
let car = {};
```

The `car` object has an internal `[ [Prototype] ]` property and we can verify this by using the `Object.getPrototypeOf()` method. This method will return the value of the internal `[ [Prototype] ]` property:

```
console.log(Object.getPrototypeOf(car));
```

This returns many in-built properties and objects of the built-in Object class:

```
▼ {...}
  ► __defineGetter__: function __defineGetter__()
  ► __defineSetter__: function __defineSetter__()
  ► __lookupGetter__: function __lookupGetter__()
  ► __lookupSetter__: function __lookupSetter__()
  ► __proto__: >>
  ► constructor: function Object()
  ► hasOwnProperty: function hasOwnProperty()
  ► isPrototypeOf: function isPrototypeOf()
  ► propertyIsEnumerable: function propertyIsEnumerable()
  ► toLocaleString: function toLocaleString()
  ► toString: function toString()
  ▼ valueOf: valueOf()
    ► length: 0
    ► name: "valueOf"
    ► <prototype>: function ()
  ► <get __proto__():>: function __proto__()
  ► <set __proto__():>: function __proto__()
```

`car` is an instance of JavaScript's built-in Object class. All the properties and methods of the prototype are available for a user created object to use.

Let's declare an object called `auto` which will be the prototype of `car` like so:

```
let auto = {  
    drive: true,  
    wheels: 4,  
    gears: 'automatic'  
}  
  
let car = Object.create(auto);
```

The `Object.create()` method is used to create a new object, using an existing object as a prototype.

And we can verify that `auto` is indeed the prototype of `car` with:

```
console.log(Object.getPrototypeOf(car));
```

And as stated before, `Object.getPrototypeOf()` method will return the properties and methods of the prototype object which in this case is the `auto` object:

```
Object {  
    drive: true,  
    gears: "automatic",  
    wheels: 4  
}
```

### Note

There is no prototype for `Object.prototype()`

Now that we know that a prototype is an object instance from which other objects can be created. Let's have a look at the prototype chain. Before that complete the following exercises.

### Exercises

1. Explain prototype inheritance in JavaScript

2. How will you query the prototype of the `book` object? And what is returned?

```
let book = {};
```

3. Consider the following object:

```
let pizza = {  
    base: 'wheat',  
    sauce: 'tomato',  
    cheese: 'parmesan'  
}
```

- Create a new object called `cheesePizza` whose prototype is the `pizza` object using the `Object.create()` method.
- What will be the output of:  

```
console.log(cheesePizza);
```
- Use the `Object.getPrototypeOf()` method to access the prototype of the `cheesePizza` object

## Answers

1. JavaScript is a prototype based language. This means that a template object's properties and methods can be cloned and extended onto other objects. Therefore, a prototype is an object instance from which other objects can be created. This is known as prototypal inheritance.
2. Using the `Object.getPrototypeOf()` method. It returns the built in Object type as the prototype of the user created object `book`.
- 3.

```
let cheesePizza = Object.create(pizza);  
console.log(cheesePizza);
```

```
Object {  
    base: "wheat",  
    cheese: "parmesan",  
    sauce: "tomato"
```

```
}
```

```
console.log(Object.getPrototypeOf(cheesePizza));
```

```
Object {  
    base: "wheat",  
    cheese: "parmesan",  
    sauce: "tomato"  
}
```

### 3.5.2 Prototype chain

JavaScript provides built in objects such as:

- Object
- Array
- Function
- Date
- RegEx
- String
- Number
- Boolean

Each object has a prototype. When we look up a property of an object, the object itself will be queried. If the property is not found, then the JavaScript engine will look within the object's prototype for the existence of the property. If still not found, then the prototype object's prototype will be queried. Therefore, forming a chain starting at a user-created object and going all the way up to the end of the chain which is **Object.Prototype**. All objects inherit from **Object.Prototype** and attempting to search beyond **Object.Prototype** will result in **null**.

Let's have a look at an example:

```
let x = new Object();
```

**x** is an empty object. Let's query the Prototype of **x** with:

```
let y = Object.getPrototypeOf(x);
```

`y` references JavaScript's `Object.prototype` object and this is demonstrated via:

```
console.log(y);
```

Which returns:

```
Object {}
```

However, what if we attempt to retrieve the prototype of `y`?

```
let z = Object.getPrototypeOf(y);
console.log(z);
```

Attempting to retrieve the prototype `y` will return `null`:

```
null
```

To simplify this, have you ever wondered how we are able to declare objects such as arrays and use methods such as `push()`, `pop()` etc. that we have not coded ourselves? Let's have a look at an example:

```
let numArray = [1,2,3,4];
numArray.push(5);
```

We can use methods such as `pop()`, `push()` etc. on `numArray` as it has access to the methods of `Array.prototype`. We can verify this by querying whether the built in `Array.prototype` object is the prototype of `numArray` by using the `isPrototypeOf()` method:

```
console.log(Array.prototype.isPrototypeOf(numArray));
```

This will log the boolean `true` value to the console as `numArray` inherits from JavaScript's built in array prototype object:

```
true
```

```

[]

  ▶ concat: function concat()
  ▶ constructor: function Array()
  ▶ copyWithin: function copyWithin()
  ▶ entries: function entries()
  ▶ every: function every()
  ▶ fill: function fill()
  ▶ filter: function filter()
  ▶ find: function find()
  ▶ findIndex: function findIndex()
  ▶ flat: function flat()
  ▶ flatMap: function flatMap()
  ▶ forEach: function forEach()
  ▶ includes: function includes()
  ▶ indexOf: function indexOf()
  ▶ join: function join()
  ▶ keys: function keys()
  ▶ lastIndexOf: function lastIndexOf()
  length: 0
  map: function map()
  pop: function pop()
  push: function push()
  reduce: function reduce()
  reduceRight: function reduceRight()
  reverse: function reverse()
  shift: function shift()
  slice: function slice()
  some: function some()
  sort: function sort()
  splice: function splice()
  toLocaleString: function toLocaleString()
  toString: function toString()
  unshift: function unshift()
  values: function values()
  Symbol(Symbol.iterator): function values()
  Symbol(Symbol.unscopables): Object { copyWithin: true, entries: true, fill: true, ... }

<prototype>: {}

  ▶ __defineGetter__: function __defineGetter__()
  ▶ __defineSetter__: function __defineSetter__()
  ▶ __lookupGetter__: function __lookupGetter__()
  ▶ __lookupSetter__: function __lookupSetter__()

  ▶ _proto_: >>
  ▶ constructor: function Object()
  ▶ hasOwnProperty: function hasOwnProperty()
  ▶ isPrototypeOf: function isPrototypeOf()
  ▶ propertyIsEnumerable: function propertyIsEnumerable()
  ▶ toLocaleString: function toLocaleString()
  ▶ toString: function toString()
  ▶ valueOf: function valueOf()
  ▶ <get __proto__(): function __proto__()
  ▶ <set __proto__(): function __proto__()

```

The methods you use, such as `push()`, `pop()`, `reduce()`, `indexOf()`, etc. are methods that are inherited by `numArray` from `Array.Prototype` which is JavaScript's built in Array Object.

Now that we know what prototype and prototype chain means, let's go over prototypal inheritance in JavaScript.

## Exercises

1. Explain the prototype chain

## Answers

1. Each object has a prototype. When we look up a property of an object, the object itself will be queried. If the property is not found, then the JavaScript engine will look within the object's prototype for the existence of the property. If still not found, then the prototype object's prototype will be queried. Therefore, forming a chain starting at a user created object and going all the way up to the end of the chain which is **Object.Prototype**.

### 3.5.3 Prototype Inheritance

Prototypal inheritance put across simply is the ability of a prototype to infer its properties and methods onto a child object.

Take for example two objects, `animal` and `dog`:

```
let animal = {  
    legs: 4,  
    eat: true,  
    nap: true  
}  
  
let dog = {  
    tail: true  
}
```

Dogs have four legs, eat and nap as well so let's set the `dog` object's prototype to be the `animal` object with the `setPrototypeOf()` method which takes two arguments:

- An object
- A prototype

```
Object.setPrototypeOf(dog, animal);
```

We can confirm if the prototype of `dog` is `animal` by logging the object `dog` to the console:

```
console.log(dog);
```

The `dog` object now has its own property as well as the properties of its prototype (`animal`):

```
Object {  
    eat: true,  
    legs: 4,  
    nap: true,  
    tail: true  
}
```

When we try and access a property that is not set on `dog`, JavaScript will search for it in the prototype. If unfound in the prototype, it will move up the chain to the built in **Object.Prototype**.

Let's make another object which inherits properties from the `dog` object:

```
let boston_terrier= {  
  breed: 'Boston Terrier'  
};  
  
Object.setPrototypeOf(boston_terrier, dog)
```

To demonstrate that an object inherits properties from its prototype, let's retrieve the `tail` property from `boston_terrier`:

```
boston_terrier.tail; // true
```

The `boston_terrier` object inherits the `tail` property from the `dog` object. The JavaScript engine will keep going up all the way to **Object.Prototype** in order to find a property. For example let's access the value of the `nap` property from `boston_terrier` object:

```
boston_terrier.nap ;// true
```

The `nap` property is not set in `boston_terrier` or the `dog` object. So the next object up the prototype chain will be queried for the property. It is found in the `animal` object.

## Exercises

1. Consider an object called `book` with the following properties and methods:

```
const book = {  
  educational: true,  
  diagrams: true,  
  author: 'J.K.',  
  discount: 0,  
  sale: function(){  
    if(this.educational){  
      this.discount = 0.5;  
    }  
  }  
}
```

- ```

    }
}

• Declare an object called scienceFictionBook and set its educational key to the boolean value false
• Set the book object to be the prototype of the scienceFictionBook
• What will be the value of scienceFictionBook.discount?

```

### Answers

1.

```

let scienceFictionBook = {
  educational: false,
}

Object.setPrototypeOf(scienceFictionBook, book);
console.log(scienceFictionBook.discount);

```

0

### 3.5.3.2 Constructor functions and inheritance

Let's practice inheritance with constructor functions by example:

```

function SuperPower(invisible, talk, bonus) {
  this.invisible = true;
  this.talk = 'deebble';
  this.bonus = 1000,
  this.dialogue = function() {
    console.log(`My super talk is ${this.talk}`);
  }
}

let invisible_girl = new SuperPower();
console.log(invisible_girl);

```

`SuperPower` is a constructor function and we have created a new instance of `SuperPower` referenced by the `invisible_girl` variable via the `new` operator. We can confirm that `invisible_girl` has inherited the properties of `superPower` by logging it to the console, which displays:

```
Object {  
  bonus: 1000,  
  dialogue = function(){  
    console.log(`My super talk is ${this.talk}`);  
  },  
  invisible: true,  
  talk: "deeble"  
}
```

If we query the `[[Prototype]]` of `invisible_girl` using the `getPrototypeOf()` method, we see that the `constructor` property's value is `SuperPower`.

```
▼ {...}  
  ► constructor: function SuperPower(invisible, talk, bonus)  
  ► <prototype>: Object { ... }
```

Now suppose, we have different super heroes each with their subset of powers. It would be redundant to create them using the `SuperPower()` constructor function. In this case we can code different constructor functions which inherit from the `SuperPower()` constructor.

Let's refactor our code to make this point clear:

```
function SuperPower(talk, bonus){  
  this.talk = 'deeble';  
  this.bonus = 1000  
  this.dialogue = function(){  
    console.log(`My super talk is ${this.talk}`);  
  }  
}  
  
function Invisible(talk) {
```

```

        SuperPower.call(this);

        this.invisible = 'invisible';

    }

function Teleport(talk) {
    SuperPower.call(this);

    this.power = 'teleport';

}

```

The `SuperPower()` constructor function no longer has an `invisible` property, everything else remains the same. There are two new constructor functions:

- function `Invisible()`
- function `Teleport()`

These two constructor functions take as parameters:

- talk

The properties from `SuperPower()` are copied over to `Invisible()` and `Teleport()` constructor functions by using the `call()` method which takes an object instance referenced by the `this` keyword, and `talk` and `bonus` as arguments.

The reference to the value of the `power` parameter is not present in `SuperPower()` constructor and instead is set in `Invisible()` and `Teleport()` constructor function as the functions have different powers (invisibility and teleportation) :

```

this.invisible = 'invisible';

this.power = 'teleport';

```

We can create new instances of `Invisible()` and `Teleport()` with the `new` keyword, like so:

```

let invisible_girl = new Invisible();

let teleport_dog = new Teleport();

```

`invisible_girl` and `teleport_dog` have inherited properties from both the `Invisible()` and `Teleport()` constructor functions respectively. Which in turn contain properties from `SuperPower()` constructor function. We can validate this by checking `teleport_dog`:

```

Object {
  bonus: 1000,
  dialogue: function(){
    console.log(`My super talk is ${this.talk}`);
  },
  power: "teleport",
  talk: "deeble"
}

```

Querying the **[IPrototype]** of `teleport` with the `Object.getPrototypeOf()` method returns:

```
constructor: Teleport(talk, bonus, power)
```

Continuing on we will delve into implementing inheritance with classes in JavaScript.

### Exercises

- Consider the following and answer the questions:

```

function SchoolFranchise(accredited, teachers, online) {
  this.accredited = true;
  this.teachers = false;
  this.online = true;
}

```

- Create a constructor function called `JuniorHigh` which inherits properties from the constructor function `SchoolFranchise()`
- Pass in two parameters to the `JuniorHigh` constructor function: `name` and `type`
- The value of the `type` key will be the string '`Junior High`'
- Create a new object referenced by `let huronPublic` passing in the string '`Huron`' as a parameter which reference the school's name
- `console.log(huronPublic)` what **key : value** pairs are returned?

### Answers

- 

```

function juniorHigh(name, type) {
  SchoolFranchise.call(this);
}

```

```

        this.name = name;
        this.type = 'Junior High'
    }

let huronPublic = new juniorHigh('Huron');
console.log(huronPublic);

Object {
    accredited: true,
    name: "Huron",
    online: true,
    teachers: false,
    type: "Junior High"
}

```

## 3.6 Classes

In object-oriented programming, **class** types are templates for creating objects. As we just learned that in prototypal inheritance, objects inherit properties and methods from a prototype. Classes build upon prototypal inheritance. Classes were introduced in ES6 to mimic the **class** data type found in Java and other object-oriented programming languages. Till now developers used constructor functions to mimic object-oriented design patterns. JavaScript does not have the **class** type, so we create functions in a way that they behave as classes to allow us to easily create objects. A class is a function and when invoked as a constructor will create an instance of that class.

The next few sections explore classes in detail and how can effectively use classes in JavaScript to write object-oriented code.

### 3.6.1 Class declarations

Classes are functions that are declared with the **class** keyword instead of the **function** keyword. The name of a class is capitalized as convention. For example **Person** is a class. Note the similarity in structure to a function:

```
class Person{
```

```
}
```

You can also use a class expression:

```
let Person = class {}
```

Class declarations are not hoisted. Therefore, classes must be defined before being used else it will result in a **ReferenceError**. Also keep in mind that the body of a Class is executed in **strict mode**.

### Exercises

1. What is a class?
2. Declare a class called **Company**

### Answers

1. Classes were introduced in ES6 to mimic the **class** data type found in Java and other object- oriented programming languages JavaScript does not have the **class** type, so we create functions in a way that they behave as classes in order to allow us to easily create objects. A class is a function and when invoked as a constructor will create an instance of that class. Classes are functions except that they are declared with the **class** keyword instead of the function keyword.
2. Class called **Company**:

```
function Company{ }
```

### 3.6.2 Constructor method

The constructor method is a method that is defined within a class declaration using the **constructor** keyword. It will create and initialize the properties of an instance of a class.

We will set the values of the arguments passed in to the constructor method using the **this** keyword:

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

A class can only have one constructor method. More than one constructor method will result in a `SyntaxError`.

To initialize an instance of the `Person` class, use the `new` keyword:

```
let person1 = new Person('Rover', 10, 'snacking');
```

In this example the variable `person1` refers to an instance of the `Person` class. The `person1` instance has its `name` property equal to the string '`Rover`', `age` is set to the numeric value `10` and `hobby` has a value of '`snacking`'. Logging `person1` to the console will return the following:

```
Object {  
  age: 10,  
  hobby: "snacking",  
  name: "Rover"  
}
```

This is just one object instance using the `Person` class. We can instantiate many more object instances from the same class. Therefore, allowing us to implement inheritance in JavaScript by using one parent class.

## Exercises

1. Have a good look at the following block of code. What will be logged to the console:

```
let maplesyrup = new IcecreamFlavor('Maple Syrup',  
false);  
  
class IcecreamFlavor {  
  
  constructor(name, toppings) {  
  
    this.name = name;  
  
    this.toppings = toppings;  
  
  }  
  
}  
  
console.log(maplesyrup);
```

2. Consider the following class `Company`:

```
class Company {
```

```

constructor(name, funding, employees) {
    this.name = name;
    this.funding = funding;
    this.employees = employees;
}
}

```

- Can you add another constructor to this class?
- Instantiate a new object using the `Company` class referenced by `let zimbaPay` that has the following arguments passed in to it: `'Zimba Pay', 1000000, 50`
- Use the `Object.getPrototypeOf()` method to find the prototype of the `zimbaPay` object

## Answers

1. Class declarations are not hoisted. Therefore, classes must be defined before being used else it will result in a `ReferenceError`, as hoisting does not apply here. Also keep in mind that the body of a Class is executed in `strict mode`.

```
ReferenceError: can't access lexical declaration
`IcecreamFlavor' before initialization
```

2. No you cannot add another constructor to the class.

```

let zimbaPay = new Company('Zimba Pay', 1000000, 50);
console.log(zimbaPay);
console.log(Object.getPrototypeOf(zimbaPay));

```

```

Object {
  employees: 50,
  funding: 1000000,
  name: "Zimba Pay"
}

```

```
constructor: class Company { constructor(name, funding,  
employees) }
```

### 3.6.3 Instance method

An instance method is defined within a class similar to a function declaration without the **function** keyword. Instance methods can access and modify the data of an instance class.

For example:

```
class Person {  
  
    constructor(name, age, hobby) {  
  
        this.name = name;  
  
        this.age = age;  
  
        this.hobby = hobby;  
  
    }  
  
    personGreeting() {  
  
        console.log(`hi I am ${this.name} and I like  
${this.hobby}`);  
  
    }  
  
}  
  
let person1 = new Person('Rover', 10, 'coding')
```

The value of **this** will depend on the class instance that is created. We can use **this** to access instance data or even call other methods:

```
console.log(person1.name); // Rover  
person1.personGreeting(); // "hi I am Rover and I like coding"
```

Here the **personGreeting()** instance method will access the data from the instance object **person1** and log the appropriate greeting to the console that is specific to the object.

A class may have more than one instance method. Go ahead and practice these coding exercises.

### Exercises

1. Complete the following:

- Add an instance method called `equity()` to the following `Company` class which will return `10%` of the total `funding` that a company receives
- Instantiate a new object from the `Company` class called `purpleMoon`. And pass in the following parameters to it: `'Purple Moon'`, `5000000`, `50`
- Declare a global variable called `equity` whose value is the `equity()` method called on the `purpleMoon` object

## Answers

1.

```
class Company {
  constructor(name, funding, employees) {
    this.name = name;
    this.funding = funding;
    this.employees = employees;
  }
  equity() {
    return(0.10 * this.funding);
  }
}

let purpleMoonEnterprise = new Company('Purple Moon',
5000000, 50);

let equity = purpleMoonEnterprise.equity();

console.log(equity);
```

**500000**

### 3.6.4 Static method

We can also create a static method within a class. *A static method belongs to a class and can only be accessed by the class and not the object instance created by a class.*

The `static` keyword must be used to declare a static method. For example:

```

class Person {

    constructor(name, age, hobby) {
        this.name = name;
        this.age = age;
        this.hobby = hobby;
    }

    personGreeting() {
        console.log(`hi I am ${this.name} and I like ${this.hobby}`);
    }

    static hello() {
        console.log('Hello how are you')
    }
}

let person1 = new Person('Rover', 10, 'coding');
Person.hello();

```

`hello()` is a static method that is accessible by the `Person` class. Therefore, `Person.hello()` will log to the console:

```
"Hello how are you"
```

If we try to access it via an instance of the `Person` class referenced by variable `person1`, we will get an error:

```
person1.hello();
```

```
TypeError: person1.hello is not a function
```

Therefore, we are returned with a `TypeError` as the static method `hello()` is inaccessible by the `person1` instance. To reiterate a static method can only be accessed by a class and not an object instance created from the class.

Having discussed instance and static methods brings us to the next topic on classes in JavaScript: public and private fields.

## Exercises

1. Define a static method

2. Take as an example the following class:

```
class MusicLabel{  
    constructor(name,genre) {  
        this.name = 'Avocado Label';  
        this.genre = genre;  
    }  
    static labelMotto(){  
        console.log(`Gimme some ${this.name}`)  
    }  
}
```

A new object declared as `artist` is instantiated from this class :

```
let artist = new MusicLabel('Moozic  
Records','Jazz','Richie Zoo');
```

Calling the `labelMotto()` static method on the `artist` object returns an error:

```
artist.labelMotto();  
TypeError: artist.labelMotto is not a function
```

- What can you do so that the `labelMotto()` static method inside the `MusicLabel` class becomes available to the `artist` object?

## Answers

1. *A static method belongs to a class and can only be accessed by the class and not the object instance created by a Class.* The `static` keyword must be used to declare a static method
2. Remove the `static` keyword from in-front of the `labelMotto()` label:

```
class MusicLabel{  
    constructor(name,genre, artist) {  
        this.name = name;  
        this.genre = genre;  
        this.artist = artist;
```

```

    }

    labelMotto() {
        console.log(`Gimme some ${this.name}`);
    }
}

let artist = new MusicLabel('Moozic Records', 'Jazz', 'Richie Zoo');

artist.labelMotto();

```

"Gimme some Moozic Records"

### 3.6.5 Public and private fields

Class fields are variables that hold information. There are 2 types of class fields:

1. Fields on the class instance
2. Fields on the class itself

The public and private fields have two levels of accessibility:

1. Public: the field is accessible anywhere
2. Private: the field is accessible only within the body of the class. We will explore the different types of class fields in the following sections.

#### Note

Both public and private field declarations are an [experimental feature \(stage 3\)](#) proposed at [TC39](#), the JavaScript standards committee (MDN).

#### 3.6.5.1 Public instance fields

Public instance fields exist on every instance created from a class. Public fields can be accessed outside of the class body. Take for example:

```

class Person {

    constructor(name, age, hobby) {
        this.name = name;
        this.age = age;
    }
}

```

```
        this.hobby = hobby;
    }
}

let person1 = new Person('Rover', 10, 'snacking');
console.log(person1.age); //10
```

The `age` property is a public instance field. It can be accessed outside of the `Person` class by using the dot notation to access the value of the property that is on that class instance.

In the previous example, the public fields are inside the body of the constructor. We can also explicitly declare them outside the constructor, but still within the class:

```
class Person {

    name;
    age;
    hobby;

    constructor(name, age, hobby) {

        this.name = name;
        this.age = age;
        this.hobby = hobby
    }
}

let person1 = new Person('Rover', 10, 'snacking')
console.log(person1.age); //10
```

Explicitly setting public fields outside the constructor method makes it easier to decipher the class's data structure and intent. Public fields can be initialized with values during declaration. This ensures that the class instance and constructor have the same fields:

```
class Person {

    name = '';
    age = 10;
    hobby = '';

    constructor() {
```

```
//empty constructor  
}  
}
```

We have initialized the public fields `name`, `age` and `hobby` with starting values. The constructor method remains empty. The `constructor()` method has access to these fields implicitly.

```
let person1 = new Person();
```

We instantiate an instance of the `Person` class and assign its reference to `let person1`. Since public fields can be accessed outside of the class by an instance, the number `10` is logged to the console when we access the value of the `age` property:

```
console.log(person1.age);
```

```
10
```

Public instance class fields can be modified inside the class and outside of the class. In the following code snippet, we have changed the value of the `age` property on the `person1` instance to the number 30:

```
person1.age = 30;  
console.log(person1.age); // 30
```

The public `age` field has been modified outside the class and its value has changed from `10` to `30` on the instance inside itself. Thereby, demonstrating that public instance fields can be modified from within the class and outside of the class on the instance itself.

## Exercises

1. Describe a public instance field
2. Create a class called `Player`. The class has four public instance fields `name`, `score`, `punches`, `throws`. Initialize these fields to starting values of:

```
name='';  
punches = 10;  
throw =3;  
score;
```

- The constructor method returns the sum of `punches` and `throws` and assigns the value to the `score` field

- Code a public method `startMessage()` which will `console.log` the message  
``Are you ready to kung-foo ${this.name}?``
- Instantiate a new object from this class and declare it as `fooFighter`
- `Console.log` the `fooFighter` instance, what do you see logged?
- Call the `startMessage()` method on the `fooFighter` instance what is logged?
- What should we change in this class so that instead of the following message from the `startMessage()` method:  
`"Are you ready to kung-foo Player?"`
- We see a more personalized message, corresponding to a player's actual username, for example:  
`"Are you ready to kung-foo Foo Fighter?"`

## Answers

1. Public instance fields exist on every instance created from a class.

2.

```
class Player{
  name='Player';
  punches = 10;
  throw =3;
  score;
  constructor() {
    this.score = this.punches + this.throw
  }
  startMessage() {
    console.log(`Are you ready to kung-foo
${this.name}?`)
  }
}
let fooFighter = new Player();
console.log(fooFighter);
```

```
fooFighter.startMessage();  
  
Object {  
  name: "Player",  
  punches: 10,  
  score: 13,  
  throw: 3  
}  
"Are you ready to kung-foo Player?"
```

```
class Player{  
  name='';  
  punches = 10;  
  throw =3;  
  score;  
  constructor(name){  
    this.name = name;  
    this.score = this.punches + this.throw  
  }  
  startMessage(){  
    console.log(`Are you ready to kung-foo  
    ${this.name}?`)  
  }  
}  
let fooFighter = new Player('Foo Fighter');  
console.log(fooFighter)  
fooFighter.startMessage();
```

### 3.6.5.2 Private instance fields

Private instance fields are only accessible within the body of a class. That is to say that they are **encapsulated** within that specific class. Private fields cannot be modified outside of the class that they belong to. They can only be read and changed within the class.

In order to declare a private field, pre-fix the name with the `#` symbol. The `#` must be used every time that the private field is declared, modified or read. In this example the `name`, `age` and `hobby` fields are declared as private fields:

```
class Person {  
    #name;  
    #age;  
    #hobby;  
  
    constructor(name, age, hobby) {  
        this.#name = name;  
        this.#age = age;  
        this.#hobby = hobby;  
    }  
  
    getName() {  
        console.log(this.#name);  
    }  
}  
  
const person1 = new Person('Zelda');  
person1.getName(); // Zelda
```

The `#name`, `#age` and `#hobby` private fields cannot be accessed outside of the class. The `getName()` method has access to the private `#name` and `#hobby` fields as it is located inside the class along with the private fields. `getName()` is not a private method which is how we are able to get an output from `person1.getName()`.

Trying to access the `name` field of the `person1` class instance results in an error:

```
person.#name; //SyntaxError: Private field '#name' must be  
declared in an enclosing class
```

Therefore, private fields are encapsulated within a class and they cannot be accessed or modified outside a class.

### Exercises

1. Consider the following code block and answer the questions:

```
class InternalDetails {  
    #gross_profit;  
    #net_profit;  
    #tax;  
    #expenses;  
  
    constructor(gross_profit, expenses, tax, net_profit) {  
        this.#gross_profit = gross_profit;  
        this.#expenses = expenses;  
        this.#tax = tax;  
    }  
  
    getNet() {  
        this.#net_profit= this.#gross_profit - (this.#expenses  
+ this.#tax)  
  
        return(this.#net_profit)  
    }  
}  
  
let mooCompany = new InternalDetails(30,1,1,0);  
mooCompany.getNet();
```

- What will `mooCompany.getNet()` return
- What happens when you try and access `mooCompany.#gross_profit?`
- Can you set the value of the private `#net_profit` field of the `mooCompany` class instance

## Answers

1. `mooCompany.getNet()` returns

28

Cannot access `mooCompany.#gross_profit`:

Uncaught SyntaxError: Private field '#gross\_profit' must be declared in an enclosing class

Setting the value of the private `#net_profit` field will return the following:

Uncaught SyntaxError: Private field '#net\_profit' must be declared in an enclosing class

### 3.6.5.3 Public static fields

Static fields are defined on the class itself and not on the instance of a class as we saw above. Public static fields store information specific to a class which should remain consistent across all instances of a class. To declare a public static field, the **static** keyword is used followed by the name of the field. Let's have a look at an example to illustrate this:

```
class User {  
    static user = 'regular';  
    static superUser = 'management';  
    name;  
    position;  
    constructor(name, position) {  
        this.name = name;  
        this.position = position;  
    }  
}
```

The static fields are `regular` and `superUser`. We have two public fields defined outside of the constructor method, within the class `User`. The public fields are assigned values inside the constructor method using the `this` keyword which corresponds to an instance of the `User` class.

Let's instantiate an instance of the `User` class and assign it to have either a static `user` or `superUser` field:

```
let mrTypo = new User('mrTypo', User.user);
```

We can log the instance called `mrTypo` to the console and check its properties. We can also check whether its `position` property's value is strictly equal to the value of the `User` class's static `user` property.

```
console.log(mrTypo);
console.log(mrTypo.position === User.user); //true
```

The following will be logged to the console:

```
Object {
  name: "mrTypo",
  position: "regular"
}
true
```

An object is logged to the console with two `key : value` pairs (`name` and `position`). The `position` field's value is `User.user`

## Exercises

1. How is a static public field declared?
2. What purpose do static fields have?
3. Declare a class called `WeddingPlanner`. The class has 2 static public fields:

```
llc = 'Wedding Gee LLC'
tax_number = '319000'
```

4. The class has 3 private `instance` fields:

```
company = 'Wedding Gee';
office = '101 Plum Street, Chicago';
planner = 'Keanna Rose';
```

5. The class will have 2 public instance fields:

```
client_name;
client_budget;
```

- Code a constructor function that sets the `client_name` and `client_budget` fields to an instance of an object created from the `WeddingPlanner` class using the `this` keyword
- Code an instance function called `welcomeMessage` that will `console.log` the following message:

```
console.log(`Hi, ${this.client_name}! Welcome to
${this.#company}, I am your planner ${this.#planner}.
Please confirm that your budget is
${this.client_budget}`)
```

- Instantiate a new object which will represent a client using the `WeddingPlanner` class. Call it `missSpadina`. Pass in the parameters (`'J. Spadina', 50000`) to this new instance
- Call the `welcomeMessage()` method on the `missSpadina` instance
- Query the prototype of the `missSpadina` instance

## Answers

1. With the `static` keyword.
2. Static fields reference values that will be consistent across all instances of the class.
3. Answers 3 -5

```
class WeddingPlanner {
    #company = 'Wedding Gee';
    #office = '101 Plum Street, Chicago';
    #planner = 'Keanna Rose';
    static llc = 'Wedding Gee LLC';
    static tax_number = '319000';
    client_name;
    client_budget;
    constructor(client_name, client_budget) {
        this.client_name = client_name;
        this.client_budget = client_budget;
    }
}
```

```

    }

    welcomeMessage(company, planner) {
        console.log(`Hi, ${this.client_name}! Welcome to
${this.#company}, I am your planner ${this.#planner}.
Please confirm that your budget is
${this.client_budget}`)

    }
}

let missSpadina = new WeddingPlanner('J. Spadina',
50000);

missSpadina.welcomeMessage();

```

Hi, J. Spadina! Welcome to Wedding Gee, I am your planner Keanna Rose. Please confirm that your budget is 50000

```
console.log(Object.getPrototypeOf(missSpadina));
```

**constructor: class *WeddingPlanner***

#### 3.6.5.4 Private static fields

Private static fields are useful in order to hide implementation details that you would like to remain unchanged. To declare a field as private, pre-fix the **#** symbol before the field name along with the **static** keyword. The following example demonstrates this:

```

class StartupEmployeeList {

    static #max_instances = 3;
    static #instances = 0;
    name;

    constructor(name) {
        StartupEmployeeList.#instances++;

        if(StartupEmployeeList.#instances >
StartupEmployeeList.#max_instances) {

            throw new Error(
                'Unable to create a new employee instance'
            )
        }
    }
}

```

```

        )
    } else {
        this.name = name;
    }
}
}
}

```

In the `StartupEmployeeList` class we have created a function which has:

- Static private `#max_instances` field which specifies the maximum number of instances the `StartupEmployeeList` class can have. Since it is a class for a tiny startup, the maximum number of employee instances we can create is set to the number **3**
- We also have a static private `#instances` field which is set to a starting value of 0. It serves as a counter to keep track of the number of `StartupEmployeeList` instances created
- There is also a public field called `name`
- The `constructor()` method takes `name` as a parameter. It will check if the number of employee instances created is less than the number of `#max_instances` i.e. **3**. If so it will create a new `StartupEmployeeList` instance and increment the `#instances` counter by 1. If the total number of `#max_instances` exceeds the number **3**, an error will be thrown

Let's create some `StartupEmployeeList` instances and see where we are:

```

new StartupEmployeeList('TJ');

new StartupEmployeeList('Simi');

new StartupEmployeeList('Lara');

```

So far we have created three `StartupEmployeeList` instances.

Attempting to create more new `StartupEmployeeList` instances results in an error:

```

Uncaught Error: Unable to create a new employee instance at
new StartupEmployeeList

```

Private static fields are accessible only within the `StartupEmployeeList` class. Which means that they're encapsulated within that class. No changes can be made to them and that's the benefit of encapsulation.

## Exercises

1. What function do private static fields serve?
2. Using the `DonutGiveaway` class instantiate 2 instances (`donut1` and `donut2`). Log the object instances, to the console what is logged? Also query the `keys` of the `donut1` object

```
class DonutGiveaway {  
    static #max_instances = 5;  
    static #instances = 0;  
    flavor;  
    constructor(flavor) {  
        DonutGiveaway.#instances++;  
        if (DonutGiveaway.#instances >  
            DonutGiveaway.#max_instances) {  
            throw new Error(  
                'Unable to create a new donut instance'  
            )  
        } else {  
            this.flavor = flavor;  
        }  
    }  
}
```

## Answers

1. Private static fields are useful in order to hide implementation details that you would like to remain unchanged.

2.

```
let donut1 = new DonutGiveaway('Java JavaScript');  
let donut2 = new DonutGiveaway('Snappy Semicolon');  
console.log(donut1);  
console.log(donut2);
```

```

console.log(Object.keys(donut1));

▼ DonutGiveaway {flavor: "Java JavaScript"} ⓘ
  flavor: "Java JavaScript"
  ► __proto__: Object

▼ DonutGiveaway {flavor: "Snappy Semicolon"} ⓘ
  flavor: "Snappy Semicolon"
  ► __proto__: Object

["flavor"]

```

### 3.6.6 Inheritance with extends

The **extend** keyword will allow a child class to inherit from a parent class. This is known as **sub-classing**. We learned about inheritance with objects in the previous section. In this section, we will implement inheritance with classes.

Take as an example the following code:

```

class GameCompany {

  company;
  rating;

  constructor(company, rating) {
    this.company = company;
    this.rating = rating
  }
}

```

In this code example, we have:

- A class **GameCompany** with two public fields **company** and **rating** declared outside the **constructor()** method
- The constructor method takes the two public fields as parameters and sets their value using the **this** keyword to the arguments passed when an instance of **Game** is created. Now let's extend this class and create a child class:

```

class AnimalCrossing extends GameCompany {
  scenes = 10;
}

```

Using the **extend** keyword:

- Creates `AnimalCrossing` class as a sub-class of `GameCompany`.
- `AnimalCrossing` has a public field called `scenes` initialized to a starting value of `10`

The `AnimalCrossing` child class inherits from the parent `GameCompany` class. As a result it extends its parent `GameCompany` class. As such, any object created using the `AnimalCrossing` sub-class will exhibits properties from the parent `GameCompany` class as well as the `AnimalCrossing` class. The following code block demonstrates this:

```
let game_1 = new AnimalCrossing( 'Nintendo', 4.5);  
console.log(game_1);
```

We create a new `AnimalCrossing` instance passing in `Nintendo` as the `company` name and `4.5` as the `rating` of the game. This instance is referenced by the `let game_1` variable. Logging this variable to the console returns:

```
Object {  
  company: "Nintendo",  
  rating: 4.5,  
  scenes: 10  
}
```

Therefore we created an instance of the `AnimalCrossing` child class which has properties of its own as well as properties from the `GameCompany` class.

Now practice the following exercise and we will move onto the next topic.

### Exercises

1. Have a look at the following class and answer the questions:

```
class Dog{  
  constructor(legs, tail){  
    this.legs = 4;  
    this.tail = 1;  
  }  
}
```

- Construct child class which extends `Dog`, called `Breed`
- The `Breed` child class has the `breed` property set to the string '`Boston Terrier`'
- Create a new instance of the `Breed` class called `rocko`
- `Console.log` `rocko`, what all `key : value` pairs are logged:

### Answers

1.

```
class Breed extends Dog{
  breed = 'Boston Terrier';
}

let rocko = new Breed();
console.log(rocko);
```

```
Object {
  breed: "Boston Terrier",
  legs: 4,
  tail: 1
}
```

### 3.6.7 Inheritance with super keyword

The `super` keyword allows you to access the parent class from the child class. Therefore, it is used to access the parent class's properties and methods. By being able to do so you avoid duplication of the parts of the constructor present both in the parent and child class. In this example we will see how `super` is used:

```
class Company {
  constructor(brand) {
    this.brand = brand;
  }
}
```

```

class Game extends Company {
    constructor(brand, name) {
        super(brand);
        this.name = name;
    }
    show() {
        console.log(`Game is by ${this.brand} and the name is ${this.name}`);
    }
}

```

In the code example we have:

- A class called `Company` whose constructor method takes the parameter `brand` and sets the value of `brand` tied in to an instance of the class using the `this` keyword
- Using the `extends` keyword we create a child class called `Game`
- The `Game` child class has a `constructor()` method which takes as arguments:
  1. The parent `Company` class's `brand` field
  2. Another parameter called `name`
- The `Game` child class also has an instance method called `show()` which will log to the console a string containing the values of the `brand` and `name` parameters

Let's create an instance of the `Game` class and call the `show()` method on it:

```

let fave_game = new Game("Nintendo", "Animal Crossing");
console.log(fave_game);
fave_game.show();

```

The following will be logged to the console:

```

Object {
  brand: "Nintendo",
  name: "Animal Crossing"
}

```

```
}
```

"Game is by Nintendo, and the name is Animal Crossing"

The code illustrates that we are able to access the parent class's fields using the **super** keyword. In this case we accessed the **brand** field of the parent **Company** class from the child **Game** class and were able to use it in the child's instance **show()** method.

### Note

In a child class constructor, the **this** keyword cannot be used until **super()** is called

Now that we have discussed classes and instantiating object instances, in the next section we will cover how to verify the class of an instance.

### Exercises

1. Consider the following parent class **Dog** and answer the questions:

```
class Dog{  
    constructor(legs, tail){  
        this.legs = 4;  
        this. tail = 1;  
    }  
}
```

- Create a child class called **Breed** which extends the **Dog** class
- Use the appropriate fields such that when an instance is logged to the console, the following is returned:

```
Object {  
    breed: "Boston terrier",  
    legs: 4,  
    tail: 1  
}
```

2. Create a class called **MusicLabel**:

- `MusicLabel` has a constructor method which takes one parameter called `label_name`
- Set the value of the `label_name` field to '`'Avocado Label'`
- The `MusicLabel` class has a static method called `LabelMotto()` which will `console.log` the string:  
`'Hello and welcome!'`
- Create a sub-class called `Jazz` which extends the `MusicLabel` class
- Its `constructor()` method takes `artist`, `label_name` and `genre` as parameters
- Access the `genre` field which is present in the parent `MusicLabel` class
- Using `this` set the values of the `artist` field to the value passed in to the instance
- Set `this.genre` to the string '`'Jazz'`;
- Create an instance of the `Jazz` class which is called `richieZoo`, using the `new` keyword and pass in the string '`'Richie Zoo'` as a parameter which is the artist's name
- `console.log` the `richieZoo` instance

## Answers

1.

```
class Dog{
  constructor(legs, tail) {
    this.legs = 4;
    this.tail = 1;
  }
}

class Breed extends Dog{
  constructor(breed) {
    super();
  }
}
```

```
        this.breed = breed;
    }
}

let doggie = new Breed('Boston terrier');
console.log(doggie);
```

2.

```
class MusicLabel{
    constructor(label_name) {
        this.label_name = 'Avocado Label';
    }
    static LabelMotto() {
        console.log('Hello and welcome!')
    }
}
class Jazz extends MusicLabel {
    constructor(artist, label_name, genre) {
        super(label_name);
        this.artist = artist;
        this.genre = 'Jazz';
    }
}
let richieZoo = new Jazz('Richie Zoo');
console.log(richieZoo);
```

```
Object {
    artist: "Richie Zoo",
```

```
        genre: "Jazz",  
        label_name: "Avocado Label"  
    }
```

### 3.6.8 InstanceOf

In order to check if an object is an instance of a specific class use the **instanceOf** operator. This will check if an object is an instance of that particular class.

Take as an example, a class called `User` which has a public field called `name` and a `constructor()` method which will set the value of the `name` field to the value passed in while initializing an instance of the `User` class:

```
class User {  
  
    name;  
  
    constructor(name) {  
  
        this.name = name;  
  
    }  
  
    hello() {  
  
        console.log(  
            `Hello, ${this.name}`);  
  
    }  
}
```

Now let's create an instance of `User`:

```
let user_1 = new User('Charles');  
  
const user1 = {};  
  
console.log(user_1 instanceof User); //true  
console.log(user1 instanceof User); // false
```

`user_1` is an instance of the `User` class therefore, `user_1 instanceof User` returns `true`.

Whereas, `user1` is an empty object `{}` which is not an instance of the `User` class, it is an instance of the global Object class, subsequently `user1 instanceof User` is `false`.

Now let's extend the `User` class:

```
class Admin extends User {  
    accesslevel;  
    constructor(name, accesslevel) {  
        super(name);  
        this.accesslevel = accesslevel;  
    }  
}
```

`Admin` is a child class of `User` created using the `extends` keyword. It has a field called `accesslevel` and it accesses the parent class's `name` field using the `super` keyword.

Now let's create an instance of the `Admin` class:

```
let employee = new Admin('Merengue', 1);
```

We can check what class the instance belongs to using the `instanceof` operator:

```
console.log(employee instanceof Admin); //true
```

`employee` is an instance of `Admin` and the `instanceof` operator evaluates to `true`. In turn `Admin` is a subclass of the `User` class. Therefore `Admin` inherits all the properties and methods from the `User` class.

Subsequently the `employee` object inherits the `User` class's properties and methods. Thus, `employee instanceof User` evaluates to `true` as well.

We can also determine the immediate class of an instance by using the `constructor` property, about which we will read and practice next.

### Exercises

1. What does the `instanceof` operator do?

### Answers

1. In order to check if an object is an instance of a specific class use the `instanceOf` operator. This will check if an object is an instance of that particular class.

## 3.6.9 Constructor property

In order to determine the exact class of an object instance you can access the `constructor` property, carrying on from the example in section 3.8:

We can `console.log` the exact class of the `employee` instance as such:

```
console.log(employee.constructor);
```

This informs us that the exact class of the `employee` instance is the `Admin` class:

```
class Admin extends User {  
  accesslevel;  
  constructor(name, accesslevel) {  
    super(name);  
    this.accesslevel = accesslevel;  
  }  
}
```

Keeping in mind that there are two classes that the `employee` instance inherits from, we can also use the strict equality operator to check against each class:

```
console.log(employee.constructor === Admin); //true  
console.log(employee.constructor === User); //false
```

This will return the immediate class of an instance. `Employee` is an instance of `Admin` and therefore, using the `constructor` property on `employee` to strictly check against the `Admin` class returns `true`. Whereas, when checked against `User` it returns `false` because even though the `Admin` class inherits from the parent `User` class, the exact class of the `employee` instance is not `User`.

Next we will go over getters and setters introduced in ES5.

### Exercises

1. What does the `constructor` property of an object indicate?

### Answers

1. It determines the exact class of an object instance

## 3.7 Accessors: Getters and Setters

Getters and Setters are ES5 features. Objects have two types of properties:

1. Static data property
2. Accessor property

So far we have come across static data properties in the objects that we have been making.  
For example:

```
let blackLivesMatter = {  
    status: true,  
    usecase: 'daily'  
}
```

In this object, `status` and `usecase` properties are static. So what then are accessor properties? Accessor properties are functions that will execute on either getting or setting a value.

The two types of accessor properties are:

**Getter property:** Upon getting a property of an object such as `car.color` the value of the property will be generated implicitly by calling a function. The getter works upon reading a property.

**Setter property:** When a property is set for example on an object `car.color = 'red'` a function is executed implicitly passing the value of a property, in this case `'red'` as an argument to the function. The return value of the function is set to the property. The setter works upon assignment of a property.

Let's explore getters and setters more in the upcoming sections by using them.

### 3.7.1 Getters

Getters are functions that retrieve a value from static properties from an external source. You can only access properties and cannot access methods as they are functions of an object or class and are not static.

There are three ways you can use getters and setters:

1. Default method syntax
2. `get` & `set` keyword
3. `Object.defineProperty()` method

We will now discuss the above three in detail.

## 1. Default method syntax

We can define a getter by using the default method syntax in order to retrieve the property of an object. In order to make this clear have a look at the following example:

```
let truck = {  
    color: 'red',  
    make: 'Peterbilt',  
    getDetails: function() {  
        return (this.color)  
    }  
};
```

In the example above, the `truck` object has one getter method `getDetails()`. The `this` keyword references the object that the property belongs to (`truck`). The return statement inside the getter method will return the value of the `color` property.

To access the `color` property of the `truck` object, call the `getDetails()` method on the `truck` object like so:

```
truck.getDetails();
```

The default method syntax can also be used for setters. Setters will set the property of the object to the value passed in to the setter method. For example:

```
let truck = {  
    color: 'red',  
    make: 'Peterbilt',  
    getDetails: function() {  
        console.log(this.color)  
    },  
    setDetails: function(newColor) {  
        this.color = newColor;  
    }  
};
```

```
};
```

The `setDetails()` method takes a parameter `newColor` and sets the value of the `color` property to the parameter passed in. Let's change the `color` of `truck` object:

```
truck.setDetails('blue');
```

Let's check if the setter method worked properly by using the getter method `getDetails()`:

```
truck.getDetails(); //blue
```

The `getDetails()` method will return `blue` therefore indicating that the `setDetails()` method did indeed set the value of the `color` property to `blue`.

Now let's talk about the `get` and `set` keywords.

## 2. Get and Set keywords

The `get` and `set` keywords can be used to explicitly create getters and setters. The following sections will discuss both.

### Get Keyword:

The `get` keyword will explicitly create a getter. It will define an accessor property rather than using a method. This means it cannot have the same name as the data property whose value it is used to get/retrieve. Let's have a look at an example to make this clear:

```
const randomNum = {  
    number1: 1,  
    get number2() {  
        return Math.floor(Math.random() * 100) ;  
    }  
};  
console.log(randomNum.number2);
```

In the method the name of the property `number1` and name of the accessor property defined by the `get` keyword (`number2`) are different. Upon getting `number2`, its value will

be computed to a random number between the numbers zero to hundred using the `Math.random` and `Math.floor` methods.

Therefore, the value of the `number2` property will be generated implicitly by calling a function.

### Set Keyword

The `set` keyword can be used in lieu of the default method syntax for setting the value of a property. The `set` keyword will define an accessor property whose value you can set. For example:

```
const randomNum = {  
    number1: 1,  
    get number2() {  
        return Math.floor(Math.random() * 100);  
    },  
    set num(x) {  
        this.number1 = x;  
    }  
};
```

In the example we have used the `set` keyword to define a `num` accessor property which takes `x` as a parameter. It will set the value of `this.number1` property to `x`.

Let's set the `num` accessor property to a value of `10`.

```
randomNum.num = 10;
```

Upon logging the `number1` property of the `randomNum` object, we see `10` logged to the console:

```
console.log(randomNum.number1);
```

### 3. Object.defineProperty() method

This method is part of ES5. It is used to define properties, getters and setters on an object that has already been declared, you can use the `defineProperty()` method on the

global `Object`. This might seem a bit redundant as properties are defined at the time of an object's declaration. However, this method allows us to define a property's descriptor which is akin to a property's meta data.

**Syntax:** `Object.defineProperty(object, property, descriptor)`

The first argument is the object on which to define a getter or setter. The second argument is the name of the property which is to be defined and, the third argument is the descriptor for the property.

Let's have a look at an example to see how this works. Firstly, we will declare and instantiate a new object called `truck`:

```
let truck = {};
```

Now let's use the `Object.defineProperty()` method to add a data property to the `truck` object:

```
Object.defineProperty(truck, "color", {  
    configurable: false,  
    enumerable: false,  
    writable: false,  
    value: "red"  
});
```

Let's examine the code block above:

- The first argument passed to the `Object.defineProperty()` method is the `truck` object.
- The second argument is the name of the property which we have called `color`.
- The third argument is the `color` property's descriptor. Descriptors allow modification of a property.
  - The `configurable` attribute is set to `false`. Configurable alludes to whether a property can be deleted or changed.
  - The second attribute is `enumerable`. This will dictate whether the `color` property can be enumerated using the `for-in` loop.

- The third descriptor attribute is `writable`. This will dictate whether the `color` property's value can be changed using the assignment `=` operator.
- And lastly the `value` attribute's value will be the `color` property's value which is `"red"`.

Let's check the `color` property of the `truck` object by logging it to the console:

```
console.log(truck.color); // "red"
```

The `Object.defineProperty()` method can also be used to define getters and setters on an existing object. The code below demonstrates a setter method of the `truck` object:

```
Object.defineProperty(truck, "setColor", {
  set: function(newColor) {
    this.color = newColor;
  }
});
```

The property `setColor` is a setter which will set the color of the truck to the new value assigned to it. For example let's change the `color` property of the truck to blue:

```
truck.setColor = ('blue');
```

Logging to the console reveals that the color is still red:

```
console.log(truck.color); // "red"
```

This is because the `writable` descriptor is set to `false`.

Let's add another setter to this object for practice:

```
Object.defineProperty(truck, "setBrand", {
  set: function(brand) {
    this.brand = brand;
  }
});
```

This will create a `brand` property and now let's set its value:

```
truck.setBrand = 'Peterbilt';
```

We can define a getter to get the brand value:

```
Object.defineProperty(truck, "getBrand", {  
  get: function() {  
    return this.brand;  
  }  
});
```

The getter `getBrand` will retrieve the value of the `brand` property:

```
console.log(truck.getBrand); // Peterbilt
```

Getters and setters are useful as they:

1. Secure access to data properties
2. Add some extra logic to properties before getting or setting their values.

Besides getters and setters, so far in this chapter, we have discussed how to declare and use objects. We have also covered using the `this` keyword. Followed by, implementing object-oriented JavaScript with classes, along with an understanding of prototypes and inheritance. In this last section, we will go over copying objects by creating shallow and deep copies

## Exercises

1. What are accessor properties?
2. What are the 2 types of accessor properties:
3. Consider the code below:

```
let donut ={  
  units: 100,  
  flavors:['strawberry','oreo','java'],  
  price: 5.99,  
}
```

- Write a getter called `getDonutFlavor` using the default method syntax
  - Inside the method define a variable called `flavorList` which references the array in `flavors` property
  - For each element in the `flavors` array log to the console the flavor on a new line
  - Call `donut.getDonutFlavor()` what is returned?
4. Using the default method syntax write a setter called `setDonutFlavor` for the same `donut` object in question #3. The `setDonutFlavor` function will receive a string parameter and push it into the array of the `donut.flavors` property. Call the `setDonutFlavor()` method and pass in the string '`'chocolate'`'. The value of `donut.flavors` property should be  
`['strawberry', 'oreo', 'java', 'chocolate']`
5. Replace the `setDonutFlavor` method created in question #4 via the default method syntax with the `set` keyword. And pass in the string '`'chocolate'`' to the `donutFlavor` setter. `Console.log donut.flavors`. It should be:  
`['strawberry', 'oreo', 'java', 'chocolate']`
6. Replace the `getDonutFlavor` method created in question #3 via the default method syntax with the `get` keyword.
7. Consider the following empty `gamingPC` object:

```
let gamingPC = {};
```

The `graphicsCard` property is defined using the `Object.defineProperty()` method:

```
Object.defineProperty(gamingPC, 'graphicsCard', {
  configurable: false,
  enumerable: false,
  writable: true,
  value: 'RTX2060'
});
```

- Using `Object.defineProperty` method, define a setter for the `gamingPC` object called `newCard`. This setter will assign a new value that is assigned to it to the `graphicsCard` property

- Assign 'GeForce RTX 2070 Super' to the `newCard` setter
  - `Console.log(gamingPC.graphicsCard)`. The output should be:  
`"GeForce RTX 2070 Super"`
8. Write a getter function which will retrieve the value of the `graphicsCard` property. Call the getter `getCard`

### Answers

1. Accessor properties are functions that will execute on either getting or setting a value. They are computed *properties*
2. Getters and setters.
- 3.

```
let donut ={
    units: 100,
    flavors:['strawberry','oreo','java'],
    price: 5.99,
    getDonutFlavor: function(){
        let flavorList= donut.flavors;
        flavorList.forEach(function(flavor) {
            console.log(flavor);
        })
    },
};
```

```
"strawberry"
"oreo"
"java"
```

- 4.

```
let donut ={
    units: 100,
```

```
flavors:['strawberry','oreo','java'],
price: 5.99,
setDonutFlavor:
function(x) {
donut.flavors.push(x)
}
};

donut.setDonutFlavor('chocolate');
```

```
["strawberry", "oreo", "java", "chocolate"]
```

5.

```
let donut ={
units: 100,
flavors:['strawberry','oreo','java'],
price: 5.99,
set donutFlavor(x) {
donut.flavors.push(x)
}
};

donut.donutFlavor = 'chocolate';
console.log(donut.flavors);
```

```
["strawberry", "oreo", "java", "chocolate"]
```

6.

```
let donut ={
units: 100,
flavors:['strawberry','oreo','java'],
price: 5.99,
```

```
get flavor() {  
    let flavorList= donut.flavors;  
    flavorList.forEach(function(flavor) {  
        console.log(flavor);  
    })  
}  
;  
donut.flavor;
```

```
"strawberry"  
"oreo"  
"java"
```

7.

```
Object.defineProperty(gamingPC, "newCard", {  
    set: function(newCard) {  
        this.graphicsCard = newCard;  
    }  
});  
gamingPC.newCard ='GeForce RTX 2070 Super';  
console.log(gamingPC.graphicsCard);
```

```
"GeForce RTX 2070 Super"
```

8.

```
Object.defineProperty(gamingPC, "getCard", {  
    get: function() {  
        console.log(this.graphicsCard);  
    }  
});
```

```
gamingPC.getCard;
```

```
"GeForce RTX 2070 Super"
```

## 3.8 Copying objects with shallow and deep copies

Earlier on in Chapter 1, **Basics-1** we discussed that primitive values such as strings, and numbers are copied by value, whereas objects are copied by reference. What do we mean by this? The following code example will explain this practically.

```
let num1 = 10;  
let num2 = num1;  
num1 = 6;  
console.log(num2); //10
```

The value of `num1` which is the number `10` is assigned to the variable `num2`. Which is why even after we change the value of `num1` to the number `6`, the value of `num2` remains the same (`10`). It's because we copied by value of `num1` to `num2` before re-assignment of variable `num1` to `6`.

What then does copy by reference mean? When we copy by reference, the memory address of an object is shared with another object. For example:

```
let num1 = {  
    amount:10  
}  
  
let num2 = num1;  
num1.amount = 6;  
  
console.log(num2); //6
```

Since the memory address of the two objects are the same, when we change the value of the `num1.amount` key from the number `10` to `6`, we are changing the value at the memory location. Therefore, the value of the `amount` key in the `num2` object will also be the number `6`, since it refers to the same memory address.

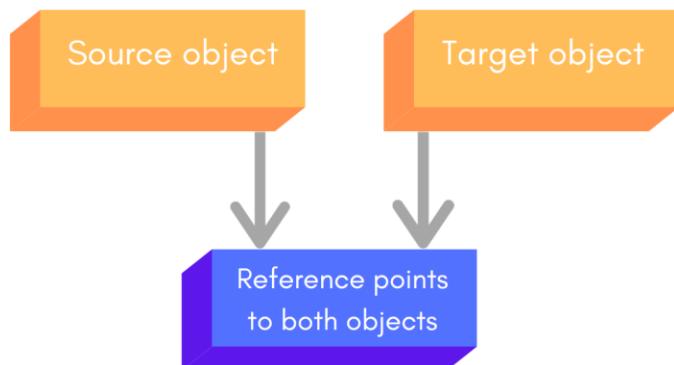
When copy over objects, we cannot simply just use the assignment `=` operator. There are two ways of copying objects:

1. Shallow copying
2. Deep copying

Which gets us to the next point, what do these two terms mean?

## Shallow copy

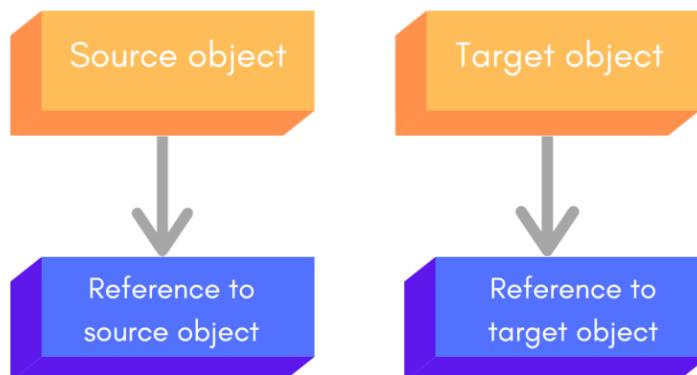
In a shallow copy, a new object is created which has the exact values as the source object. Primitive values are copied by value whereas if a field is of a reference type (for example an array or another object) then the reference/memory address is copied over to the newly created target object. *In a shallow copy, the source object and target object share the same memory address.*



Pictorial representation of a shallow copy

## Deep copy

In a deep copy, all the values (primitive and reference types) are duplicated into a source object and allocated new memory locations. Therefore, *the memory address of the target and source objects is different*. Changing the source object after a deep copy will not affect the target object.



Pictorial representation of a deep copy

## Note

An important note to keep in mind is that in a deep copy, the source and target objects have different memory addresses and are not connected. Whereas, in a shallow copy the source and target objects share a memory address. -

So how do we copy objects using both shallow and deep copy methods? This is discussed next.

### 3.8.1 Copying objects with Shallow copy

Now that you understand what shallow and deep copies mean, the next sections will elaborate on how to create them.

#### 3.8.1.1 Iteration using for key in object

In order to copy the items of an object into another object we can iterate through the **key : value** pairs of an object and assign them to the second object.

Let's copy a course object's **key : value** pairs into a target object using iteration:

```
let console1 = {
    one: 'Nintendo Switch',
    two: 'XBox',
    three: 'PlayStation',
    four: 'Gameboy'
};

let console2 = {};
for(key in console1) {
    //console.log(console1[key])
    console2[key] = console1[key];
}

console.log(console2);
```

```
Object {
```

```
    one: "Nintendo Switch",
    two: "XBox",
    three: "PlayStation",
    four: "Gameboy"
}
```

Looping will create a shallow copy by copying the enumerable properties from `console1` into `console2`.

Let's now change the value of one of the properties in the `console2` object which is a copy of the `console1` object:

```
console2.one = 'Sega';
```

We have assigned the value of the `one` property to '`Sega`' which replaces the value that was copied over for this property from the `console1` object ('`Nintendo Switch`').

Do you think that `console1` will reflect this change? Let's check:

```
console.log(console1);
```

```
[object Object] {
  four: "Gameboy",
  one: 'Nintendo
  Switch'
  three: "PlayStation",
  two: "XBox"
```

The value of `one` has not changed in the source object `console1` since primitive values are copied by value. Whereas if a field is of a reference/object type then the reference/memory address is copied over to the newly created target object. So changing the value at one location will not change the value at another location for a property whose value is a primitive type.

### 3.8.1.2 Spread operator

The spread operator (three dots) introduced in ES6 will copy over the enumerable properties of an iterable source object to the target object.

**Syntax:** `{ ... }`

Let's have a look at `console1` which is an object with four **key : value** pairs:

```
let console1 ={
```

```
    one: 'Nintendo Switch',
    two: 'XBox',
    three: 'PlayStation',
    four: 'Gameboy'
}
```

We will now use the spread operator to make a shallow copy and copy over all the primitive string values over from the target `console1` object to the source `console2` object

```
let console2 = {...console1};
```

Logging `console2` to the console we see that all the values are duplicated and are now inside the target `console2` object:

```
Object {
  one: "Nintendo Switch",
  two: "XBox",
  three: "PlayStation",
  four: "Gameboy"
}
```

The spread operator can also be used to merge two iterable objects. For example the two object literals in the following example:

```
let console1 ={
  one: 'Nintendo Switch',
  two: 'XBox',
  three: 'PlayStation',
  four: 'Gameboy'
}

let console2 = {
  five: 'Sega Saturn',
  six: 'GameCube',
  seven: 'Wii',
```

```
    eight: 'Atari 2600'  
}
```

Using the **spread operator** will merge them into a single object reference by `let consoles:`

```
let consoles = {...console1, ...console2};
```

The `consoles` object now contains the **key : value** pairs from both the `console1` and `console2` objects:

```
Object {  
  eight: "Atari 2600",  
  five: "Sega Saturn",  
  four: "Gameboy",  
  one: "Nintendo Switch",  
  seven: "Wii",  
  six: "GameCube",  
  three: "PlayStation",  
  two: "XBox"  
}
```

Duplicate keys are overwritten so if we have two keys named `1`, then the latter value will prevail:

```
let console1 = {  
  one: 'Nintendo Switch',  
  two: 'XBox',  
};  
  
let console2 = {  
  one: 'Sega Saturn',  
  three: 'GameCube',  
};  
  
let consoles = {...console1, ...console2};
```

```
console.log(consoles);
```

In the preceding object literals, the key 1 is duplicated and the value is over-written by the value of the 2<sup>nd</sup> number 1 key in the console2 object. The consoles object now has the following **key:value** pairs:

```
Object {  
  one: "Sega Saturn",  
  three: "GameCube",  
  two: "XBox"  
}
```

### 3.8.1.3 Object.assign

The ES6 **Object.assign** method will also make a shallow copy by copying enumerable properties from a target object to a source object. This method will take two arguments:

- Target object
- Source object

Take for example an object called box whose own properties will be copied over into a source object:

```
let box = {  
  height: 100,  
  width: 100  
};
```

Using the **Object.assign()** method we can assign the properties from the box object into an empty object denoted by the empty braces {}. The empty object is referenced by **let box\_copy**:

```
let box_copy = Object.assign({}, box);
```

Therefore, when we log **box\_copy** to the console the following is returned:

```
Object {  
  height: 100,  
  width: 100  
}
```

Therefore, properties and their values have been shallow copied.

We can use **Object.assign()** method to merge two objects in the following way:

```
let box = {  
    height: 100,  
    width: 100  
}  
  
let box2 = {  
    border: 20,  
    color: 'green'  
}
```

We have two boxes called box1 and box2, we will be merging the properties of the two boxes using **Object.assign()**:

```
let merged_box = Object.assign({}, box, box2);
```

The method takes 3 arguments:

- An empty object {}
- variable box
- variable box2

The method will assign the **key : value** pairs of objects box and box2 to an empty object referenced by let merged\_box. Therefore, merged\_box now contains the following **key:value** pairs:

```
Object {  
    border: 20,  
    color: "green",  
    height: 100,  
    width: 100  
}
```

If the source objects have the same property name, then the value of the property within the latter object will overwrite the former object.

Primitive values are copied by value whereas for reference values, the memory address is copied over. And in a shallow copy both the source and target objects share a reference/memory address. To understand this, let's modify the `box` object so that the **key : value** pairs are now:

```
let box = {  
    height: 100,  
    width: 100,  
    colors: {  
        one: 'peach',  
        two: 'coral',  
        three: 'orange'  
    }  
};
```

The `box` object now has a property object called `colors` which is an object with three **key : value** pairs. Let's create a shallow copy using the `Object.assign()` method as before:

```
let box_copy = Object.assign({}, box);
```

This newly created object has all the **key : value** pairs copied over from the `box` object:

```
[object Object] {  
    colors: [object Object] {  
        one: "peach",  
        three: "orange",  
        two: "coral"  
    },  
    height: 100,  
    width: 100  
}
```

Let's change the value of the `height` property in `box_copy` to `200` and check whether the `height` property in the source object `box`, is changed:

```
box_copy.height = 200;  
console.log(box);
```

The `console.log` statement will log the following for the source `box` object:

```
[object Object] {
  colors: [object Object] {
    one: "peach",
    three: "orange",
    two: "coral"
  },
  height: 100,
  width: 100
}
```

The `height` property in the original `box` object remains unchanged, even though we assigned a value of `200` in the copy that we created. This is because primitive types are copied by value and not by reference. So changing the value of a primitive type in the target object does not affect the source object.

To check whether the target and source objects share a memory address we will now change the property called `one` of the `colors` property which is a nested object inside the `box_copy` object:

```
box_copy.colors.one = 'red';
```

Since memory addresses are shared and objects are copied by reference, let's log the `box` object to the console and we should see that the value of property `one` of the nested `colors` object is now `red` instead of `peach`:

```
console.log(box);
```

```
[object Object] {
  colors: [object Object] {
    one: "red",
    three: "orange",
    two: "coral"
  },
  height: 100,
  width: 100
}
```

And we see that in both `box` and `box_copy` the value `peach` is replaced with `red`.

Let's move on to the last section of this chapter! But before that let's practice a few coding exercises.

## Exercises

1. What is the difference between shallow and deep copying?
2. What are some ways to create a shallow copy?

3. Create a copy of the following object using the **spread** operator. Name the copied object: `toronto_clone`:

```
let city = {
    name: 'Toronto',
    coordinates: '43.6532° N, 79.3832° W',
    streets: {
        North: 'Bathurst',
        South: 'Queens',
        West: 'Bathurst',
        East: 'Spadina'
    },
    population: 3190000,
};
```

- Once you create a copy of the `city` object, check whether your clone contains all the key: value pairs from the `city` object by `console.logging` it
  - Change the value of the `North` property in the nested `streets` object to the string '`St. George`' in the copied `toronto_clone` object
  - Change the value of `name` property in your copy object to '`Toronto clone`'
  - Console.log both the `city` and `toronto_clone` objects
- Can you explain why the two objects have the same value for the `North` property in nested `streets` object but not for the `name` property?

4. Have a look at the following code and answer the question:

```
let a = {};
let b = {};
console.log(a === b); // true or false
```

5. For the following object, complete the following tasks:

```
let a = {
    one: 'one',
    two: 'two',
    three: 'three'
}
```

- Using `Object.assign()` copy over the contents of object `a` to another object declared as `b`
- Change the value of `b.one` to the string '`zoo`'
- What is `a.one` ?

### Answers

1. In a shallow copy, a new object is created which has the exact values as the source object. Primitive values are copied by value whereas if a field is of a reference type (an array or another object) then the reference/memory address is copied over to the newly created target object. *In a shallow copy the source object and target object share the same memory address.*

In a deep copy, all the values (primitive and reference types) are duplicated into a source object and allocated new memory locations. Therefore, *the memory address of the target and source objects is different*. Changing the source object after a deep copy will not affect the target object.

2. Some ways to shallow copy are:

- Iteration using a for-in loop
- Spread operator
- `Object.assign()`

- 3.

```
let toronto_clone = {...city};
//console.log(toronto_clone);
toronto_clone.streets.North = 'St.George';
toronto_clone.name ='Toronto clone';
console.log(toronto_clone)
console.log(city)
```

`toronto_clone` when cloned (no changes as yet)

```
[object Object] {
  coordinates: "43.6532° N, 79.3832° W",
  name: "Toronto",
  population: 3190000,
  streets: [object Object] {
    East: "Spadina",
    North: "Bathurst",
    South: "Queens",
    West: "Bathurst"
  }
}
```

`toronto_clone` after changes to the `North` and `name` properties

```
[object Object] {
  coordinates: "43.6532° N, 79.3832° W",
  name: "Toronto clone",
  population: 3190000,
  streets: [object Object] {
    East: "Spadina",
    North: "St.George",
    South: "Queens",
    West: "Bathurst"
  }
}
```

`city` object after the changes to the `North` and `name` properties in the `toronto_clone` object

```
[object Object] {
  coordinates: "43.6532° N, 79.3832° W",
  name: "Toronto",
  population: 3190000,
  streets: [object Object] {
    East: "Spadina",
    North: "St.George",
    South: "Queens",
    West: "Bathurst"
  }
}
```

4. `false`. Both objects are empty however, they point to different locations in memory.

5.

```
let b = Object.assign({},a);
b.one = 'zoo';
console.log(a.one);
```

```
"one"
```

### 3.8.2 Copying objects with deep copy

The `JSON.stringify` and `JSON.parse` methods can be used to deep copy an object. In a deep copy, the source and target objects have different memory addresses and are not connected at all. The `JSON.stringify()` method will take a JavaScript object as an argument and transform it into a string. Then the `JSON.parse()` method will parse the JavaScript string and return a JavaScript object.

For example take an object called `object1` which has a nested object called `division`:

```
let object1 = {
  role: 'HR',
  division: {
    management: 'HR Recruiter',
  },
}
```

The object is first transformed into a string and then this string will be parsed and converted into an object before being returned:

```
let object2 = JSON.parse(JSON.stringify(object1));
console.log(object2);
```

`object2` now has the `role` property including the nested `management` property copied over to it:

```
[object Object] {
  division: [object Object] {
    management: "HR Recruiter"
  },
  role: "HR"
}
```

Let's change the value of the management property in the deep clone `object2`:

```
object2.division.management = 'dev';
```

`object2` now has the following **property : value** pairs:

```
Object {  
  division: Object {  
    management: "dev"  
  },  
  role: "HR"  
}
```

We have changed the value of a reference type (object) in `object2`. Do you think that this change will be reflected in `object1`? Let's check:

```
console.log(object1);
```

It is observed that the `object1` object has no changes to it:

```
Object {  
  division: Object {  
    management: "HR Recruiter"  
  },  
  role: "HR"  
}
```

This is because, in a deep copy, the source and target objects have different memory addresses and are not connected at all. Therefore, changing any of the values (primitive or object) in the source or target objects after making a deep clone using the `JSON.stringify()` and `JSON.parse()` methods will have no effect.

This method has the drawback that the Date, RegExp objects, and Infinity cannot be used. Only deeply nested values containing strings, numbers, booleans, and the null data type can be used.

## Exercises

1. Make a deep copy called `truck_copy` of the following object:

```
let peterbilt = {  
  company: 'Peterbilt Motors Company',
```

```
type: 'on-highway',
class_number:8,
load:{
    light: '10 tonne',
    medium: '20 tonne',
    heavy: '30 tonne'
},
}
```

## Answers

1.

```
let truck_copy = JSON.parse(JSON.stringify(peterbilt));
```

## Summary

Good job! This wraps up the chapter on objects. We learned quite a bit here. All the way from the different ways of declaring objects and using them, to the much dreaded **this** keyword. We also covered object oriented programming by going over prototypal inheritance and using ES6 classes. Copying objects by making deep and shallow copies was also discussed. Since this chapter is theory heavy, I do suggest practicing the coding questions included. This will provide you with a practical context and also help you greatly in your preparation for interviews. In the next chapter we will discuss arrays, which are an equally important and powerful object data type.

# Amazing arrays-4

*“The best yardstick for our progress is not other people, but ourselves.” — C. Matakas*

Arrays are extremely useful data structures that store ordered values of multiple data types. In this section, you will go over the fundamentals aspects of arrays. You will read about new array features added to JavaScript from ES5-ES10. Knowing how to manipulate arrays will help you solve complex problems easily. By the end of this section you will be proficient enough to delve deeper into arrays and use them proficiently. Do not skip the coding exercises in this chapter as they provide grounding for the next chapter on advanced array methods.

We're going to cover the following main topics:

- Array declaration
- Array properties
- Array methods

## 4.1 Arrays in JavaScript

### 4.1.1 Definition

Arrays are an object data type used to represent an ordered list of elements. You can traverse a list of items in an array, access an individual item in the array, and use one of the many methods provided by JavaScript to mutate an array. The **typeof** array is **Object**. An object is a collection of unordered **key: value** pairs. Whereas, an array is a collection of ordered values wherein values have an index/position rather than key names.

### 4.1.2 Indexing in arrays

|       |        |        |        |        |
|-------|--------|--------|--------|--------|
| index | 0      | 1      | 2      | 3      |
| item  | item a | item b | item c | item d |

Indexing is number based. In the diagram above items **a - d** are positioned at **0, 1, 2** and **3** respectively. The index of an array starts at zero. This means that the position of the first item in an array will be at indice 0. Moving on, next will be indice 1, indice 2 etc.

### 4.1.3 Array values

Arrays can hold primitive values and reference data types. We will practice problems with primitive and reference data types such as arrays and objects.

#### Exercises

1. What is the fundamental difference between an array and an object?

#### Answers

1. In an array items are ordered by indices. Whereas, objects represent a group of items in unordered **key : value** pairs

## 4.2 Array declaration

An array can be declared in the following two ways:

1. Array literal notation
2. Array() constructor
3. Array.of() method

We will go over these methods of array declaration and finish off by discussing the different types of values that an array can hold. Along the way we will also cover accessing array and inserting values.

### 4.2.1 Array literal notation

You can declare a variable to be a reference to an array by using the `[]` brackets as demonstrated below:

```
let fruitArray = [];
```

The `fruitArray` is empty as it has no items inside it as yet. We can add items in the array by specifying their values. Each item in an array is separated by a comma: Indexes are zero based, therefore the first item in the array which is `'mango'` is at index `0`. The second item `'apple'` is at index `1` and so on. The last item in the array (`'watermelon'`) is at index `4`:

```
fruitArray = ['mango', 'apple', 'grapes', 'blueberry', 'watermelon'];
```

Logging the `fruitArray` to the console we will get:

```
console.log(fruitArray);
```

```
["mango", "apple", "grapes", "blueberry", "watermelon"]
```

Therefore, all the string items of the `fruitArray` are displayed, each separated by a comma except for the last item.

To access an item at a particular index in the array use the `[]` brackets with the index number specified inside. For example let's access the item at index `2`:

```
console.log(fruitArray[2]);
```

The item at index `2` is the third item in the array since remember, an array starts at index `0`:

```
"grapes"
```

You can also use a trailing comma by adding a comma after the last item in the array as such:

```
fruitArray = ['mango', 'apple', 'grapes', 'blueberry',  
'watermelon',];
```

Trailing commas are becoming more common as they make it easier to see version control diffs using Git.

Arrays are objects. And we can verify this by using the `typeof` operator:

```
console.log(typeof(fruitArray)); // "object"
```

There is an ES5 method called `Array.isArray(value)` which will check whether an object passed as an argument to it is an array. If so it will return the boolean `true`. For example:

```
console.log(Array.isArray(fruitArray)); //true
```

The `fruitArray` array is an array therefore, `true` is returned from the method.

## Exercises

1. What are the three ways to create arrays?
2. Create an empty array called `items` and initialize it with values of 5 different data types. What is the length of the array?
3. What method can be used to determine whether a given object is an array?

## Answers

1. Array literal syntax, `Array()` constructor and `Array.of()` method
2. `items` array:

```
let items = ['hello', {id: 1}, true, null, 10];
```

The length of the array is 5 since it has 5 items.

3. Determine whether a given object is an array:

```
Array.isArray(value)
```

### 4.2.2 `Array()` constructor

The `new` keyword used along with an `Array()` constructor function will initialize a new array with a flexible size. For example:

```
let fruitArray = new Array();
```

If the number of array items are known, the number can be passed as a parameter to the `Array()` constructor:

```
fruitArray = new Array(5);
```

This will create an array with five indices/slots logging the `length` property of the `fruitArray`:

```
console.log(fruitArray.length); // 5
```

In order to access an item at a particular index in the array use the `[]` brackets with the index number specified inside. Let's access the first item in the `fruitArray` array which is at index `0`. The `undefined` data type will be logged to the console:

```
console.log(fruitArray[0]);
```

```
undefined
```

Let's add an item at index `0` to the `fruitArray`:

```
fruitArray[0] = 'strawberry'
```

Now when we `console.log(fruitArray)` we are returned:

```
console.log(fruitArray);
```

```
["strawberry", undefined, undefined, undefined]
```

Therefore, the string `strawberry` has been added at index `0`, which makes it the first item in the array since arrays start index `0`. Also note how every other item in the array is of the `undefined` type. This is because those keys do not exist. They are not values set to the data type `undefined`, and we can verify this by using the ES5 `Object.getOwnPropertyNames()` method which will return an array of strings corresponding to all the enumerable and non-enumerable properties that exist on the `fruitArray`. Enumerable properties are those that can be iterated over using a `for-in` loop whereas non-enumerable properties cannot be iterated over.

```
console.log(Object.getOwnPropertyNames(fruitArray));
```

```
["0", "length"]
```

This lets you know that there are two properties of the `fruitArray`:

1. `length`
2. there is a key /index at `0`

Therefore, we see there are only two keys available in the `fruitArray` even though there are four `undefined` keys. As those are not values set to the data type `undefined`.

### Note

Declaring arrays with the array literal syntax is more popular.

### Exercises

1. Answer the following using the array constructor method:

- Using an array constructor initialize an array called `employeeList` which will have `5` values
- Add an item with the string value `'Reno'` in the `employeeList` array at index `0`
- What values does the `employeeList` array contain? Try to guess without console logging

## Answers

```
1. employeeList  
let employeeList = new Array(5);  
  
employeeList[0] = 'Reno';  
  
["Reno", undefined, undefined, undefined, undefined]
```

### 4.2.3 Array.of()

This method is also used to instantiate arrays. It will create a new array instance using the arguments passed to it as array values. The numbers of arguments or type of arguments passed do not matter.

For example:

```
let another = Array.of(1, 2, 3);  
  
console.log(another) // [ 1, 2, 3 ]
```

This instantiates a new array with the numeric values 1, 2 and 3.

## Exercises

1. What is the difference between these two ways of creating arrays:

```
let array1 = Array.of(1, 2, 3);  
  
let array2 = new Array(3);
```

## Answers

1. The `array.of()` method creates a new array with the arguments passed to it as the new array's values. Whereas the `Array()` constructor method will create a new array with 3 indices of `undefined` values:

```
console.log(array1); // [1, 2, 3]  
console.log(array2); // [undefined, undefined, undefined]
```

### 4.2.4 Array values

Arrays can store primitive and reference type values such as objects and functions. For example:

```
let mixedArray = [ '#BlackLivesMatter', { name: 'Rocket' },  
true, function() { alert('Hello World!'); }, null ];
```

`mixedArray` has five values of different data types:

- String (`'#BlackLivesMatter'`)
- Object (`{ name: 'Rocket' }`)
- Boolean (`true`)
- Function (`function() { alert('Hello World!'); }`)
- Null

In order to access the `name` property of the object we first access the item at index 1. Then we can use the dot notation to access the property:

```
mixedArray[1].name; // "Rocket"
```

We can also call the function within `mixedArray` the same way by first accessing the function at the specified index and then calling the function with ():

```
mixedArray[4](); // Hello World!
```

Therefore, arrays can hold primitive and reference data types.

Now let's move on to array properties.

## Exercises

1. Consider the following array of objects:::

```
let arrayObj = [  
    {id:1, enrolled:true},  
    {id:2, enrolled:true},  
    {id:3, enrolled:false},  
    {id:4, enrolled:true},  
    {id:5, enrolled:false}  
];
```

- Access the value of the `enrolled` property of the item at index `4` in the `arrayObj` array
- Change its value to `true`

## Answers

- 1.

```
arrayObj[4].enrolled;
```

## 4.3 Array properties

In this section we will cover the following array properties:

1. Length
2. Constructor
3. Prototype

### 4.3.1 Length property of an array

The length property of an array refers to the *number of items in an array*. Unlike indexing in an array which starts at the number zero. The number of items in an array is counted from the number one onwards. The total number of items an array can hold is  $2^{32}$ .

Use the dot notation followed by the `length` keyword on an array to find the total number of items it contains. For example

```
let flexibleArray = [1, true, 300, 'Snickers', 5];  
  
let flexibleArrayLength = flexibleArray.length;  
  
console.log(flexibleArrayLength); // 5
```

The above following code block can be explained as:

- The array literal syntax is used to declare an array called `flexibleArray`
- `flexibleArray` is initialized with values during its declaration
- We access the number of items in `flexibleArray` using the dot notation followed by the `length` keyword. This number is then assigned to be the value of variable `flexibleArrayLength`
- Logging variable `flexibleArrayLength` lets us know that the array has 5 items

We can add more items to the array and its `length` property will increase. For example:

```
flexibleArray[5] = false;
```

Now the array has six items, therefore `console.log(flexibleArrayLength)` will return 6:

```
console.log(flexibleArrayLength); // 6
```

This brings us to the next topic, how do we iterate through the entirety of an array and access each item at an index? To iterate over an array use a `for` loop. This is demonstrated by the example below:

```
function arrayDetails(arr) {  
    for(let i = 0; i < arr.length; i++) {  
        console.log(arr[i]);  
    }  
}
```

Let's examine what is happening above:

- The function `arrayDetails` takes a parameter called `arr` which is supposed to be a placeholder for an actual array.
- The `for` loop will iterate over the items in the array as indicated by `arr.length`. The iteration will continue for as long as the counter variable `i` is less than the number of items in the array
- And at each iteration, it will `console.log` the item at the specific iteration through the array as indicated by `arr[i]`

Let's declare and initialize an array with a few values and take function `arrayDetails()` for a run:

```
let items = [1,{id: 'admin'}, 300,'cookie', 5];  
arrayDetails(items);
```

The following will be logged to the console, when function `arrayDetails(items)` executed:

```
1
```

```
Object {  
  id: "admin"  
}  
  
300  
"cookie"  
5
```

Therefore, all the items of the `items` array will be logged to the console.

What if we use the `Array()` constructor and initialize an array with fixed value of three. For example:

```
let fixedItems = new Array(3);
```

Moving on, let's add four items to the array, which is one more than the length specified:

```
fixedItems= [1, 2, 3, 4];
```

What do you think happens when the `fixedItems` array is passed as an argument to the `arrayDetails()` function?

```
arrayDetails(fixedItems);
```

The length of an array is flexible, therefore all the items in the array will be logged to the console when the `arrayDetails(fixedItems)` function executes:

```
1  
2  
3  
4
```

## Exercises

1. What is the length of this array:

```
let employeeList = new Array(5);
```

2. Loop over the following array and log to the console the value of the array item at each index:

```
let arrayObj = [{id:1, enrolled:true}, {id:2, enrolled:true},  
{id:3, enrolled:false}, {id:4, enrolled:true}, {id:5,  
enrolled:false}];
```

## Answers

1. 5

```

2.
for(let i =0; i < arrayObj.length; i++){
    console.log(arrayObj[i])
}

```

### 4.3.2 Constructor property of an array

An array's constructor function can be queried by accessing its constructor property. To explain this, have a look at the following array:

```

let characters = ['Tom Nook', 'Isabelle', 'Bubbles', 'Chevre'];
console.log(characters.constructor);

```

The constructor property will return a reference to the constructor function that creates an array's instance. A constructor is a function that will create an instance of an object for example an array. Therefore, down below it is seen that the array `characters` constructor function is in-built `Array()` constructor function:

```

← ▾ Array()
  ▶ from: function from()
  ▶ isArray: function isArray()
  length: 1
  name: "Array"
  ▶ of: function of()
  ▶ prototype: Array []
    Symbol(Symbol.species): undefined
  ▶ <prototype>: function ()

```

You might often see this re-stated in its simpler form as:

```
function Array() { [native code] }
```

When we instantiate a new array such as `characters`, the `Array()` constructor function is used to make an array.

### 4.3.3 Prototype property of an array

The prototype constructor will allow you to add new properties and methods of your own to an Array object. This newly added property or method will be available to all the arrays in your code for use. Let's construct a method which will return the number of characters in string items in an array:

```

Array.prototype.charCount = function() {
    for (i = 0; i < this.length; i++) {
        this[i] = this[i].trim().length;
    }
};

```

The global `Array()` object will now have the `charCount()` method. Therefore any user created array will in turn have access to this method. We can access this property on arrays. Consider a user created array containing strings as below:

```
let characters = ['Isabelle', 'Merengue', 'Coco', 'Bubbles', 'June',  
'Maple'];
```

The user created `characters` array has access to the `charCount()` method created on the prototype Array object. Therefore, we can call the method on the `characters` object:

```
characters.charCount();  
console.log(characters); // [8,8,4,7,4,5]
```

We can also add properties to the global `Array()` object which will then be available to all user created arrays. For example let's add a `name` property:

```
Array.prototype.name = 'Simple array';
```

Now let's create a `numberArr` that has a few numbers:

```
let numberArr = [100,208,345,808,4];
```

The `numberArr` should have access to the `name` property created on the prototype array object:

```
console.log(numberArr.name); // "Simple array"
```

And accessing the `name` property of the `numberArr` array reveals that it has a `name` property whose value is the same value that we set on the array prototype object i.e. the string '`'Simple array'`'.

It is not advisable to add methods and properties to the prototype Array object. As this leaves open the possibility that a modification on the prototype might cause conflict with code from other JavaScript libraries or frameworks.

JavaScript has many in-built methods provided by the prototype Array object that you can use on instance arrays; the next section of this chapter will introduce many of these methods.

## Exercises

1. Is it good practice to add methods and properties to the prototype Array object? If not, why?

## Answers

1. It is not advisable to add methods and properties to the prototype Array object. As this leaves open the possibility that a modification on the prototype might cause conflict with code from other JavaScript libraries or frameworks.

## 4.4 Array Methods

The `Array.prototype` object has many methods that you can use on user created arrays. This section will explain some useful methods including the newer ES5 additions.

When dealing with array methods we want to ask ourselves if the method mutates the original array or returns a copy of the original array with changes caused by using an array method. **Some methods mutate the original array whereas, others return a new array.**

#### 4.4.1 Adding an array element

There are a couple of different ways you can add elements to an array discussed down below.

##### 4.4.1.1 Array.prototype.push()

Consider an array called `studentList`:

```
let studentList = ['Merengue', 'Orion', 'Jannat', 'Ray', 'Amal'];
```

In order to add an array item to the list we can use the `push()` method which pushes an item at the end of an existing array. The method takes as an argument one or more items to be pushed:

```
studentList.push('Sabrina', 'Karan');
```

In this case we have pushed two string values `'Sabrina'` and `'Karan'` into the `studentList` array. Logging the array to the console will display all its contents:

```
console.log(studentList);  
["Merengue", "Orion", "Jannat", "Ray", "Amal", "Sabrina", "Karan"]
```

The array is returned with a new value of the `length` property since items were added to it:

```
console.log(studentList.length); // 7
```

The `push` method is a cleaner alternative to pushing items at specific indices, because if you miss an indice it will create an `undefined` item at that particular indice. For example let's push the string values `'Sabrina'` and `'Karan'` into indice 6 and 7:

```
studentList[6] = 'Sabrina';  
studentList[7] = 'Karan';
```

Now logging the `studentList` array to the console we are displayed:

```
["Merengue", "Orion", "Jannat", "Ray", "Amal", undefined, "Sabrina",  
"Karan"]
```

An `undefined` value exists at index 5, since we missed that one and added the string value `'Sabrina'` at index 6 and `'Karan'` at index 7.

Therefore, in order to avoid unintentional errors like this it is better to use the `push()` method which will add items at the end of an array.

##### 4.4.1.2 Array.prototype.unshift()

The `unshift()` method will add items to the beginning of an array. Carrying on from our previous `studentList` array:

```
let studentList = ['Merengue', 'Orion', 'Jannat', 'Ray', 'Amal'];
```

Let's add items to the beginning of the array using the `unshift()` method which takes as an argument/arguments the items to be added to the `studentList` array:

```
studentList.unshift('Sabrina', 'Karan');
```

Logging the array to the console will return the newly added string values added to the start of the array and a new value for the array `length` property:

```
console.log(studentList);  
console.log(studentList.length);
```

```
["Sabrina", "Karan", "Merengue", "Orion", "Jannat", "Ray", "Amal"]  
7
```

Now that we know how to add an array items let's have a look at deleting array items.

#### 4.4.1.3 Array.prototype.splice ()

The `splice()` method can also be used to insert items to the beginning, middle or end of an existing array. This method takes 3 parameters:

1. The first argument specifies the index at which the items should be added in the array
2. The second argument specifies the number of items that should be removed from the array
3. The third argument specifies the elements to be added to the array

Let's try adding items to the `studentList` array by using `splice()`:

```
studentList.splice(3, 0, 'Sabrina', 'Karan');
```

This will insert the string values `'Sabrina'`, `'Karan'` starting at index `3` and remove `0` elements from the array, therefore returning the `studentList` array:

```
["Merengue", "Orion", "Jannat", "Sabrina", "Karan", "Ray", "Amal"]
```

Therefore, the `studentList` array is modified, and returned with the new items. An array that has been modified is also called a mutated array.

#### 4.4.1.4 Array.prototype.concat()

The `concat()` method will add elements to the end of an array like the `push()` method. It can accept multiple arguments. Have a look at the following code block:

```
let studentList = ['Merengue', 'Orion', 'Jannat', 'Ray', 'Amal'];  
let modifiedStudentList = studentList.concat('Sabrina', 'Karan');  
console.log(modifiedStudentList);
```

The `concat()` method will add two string values to the `studentList` array. The new modified array is referenced by variable `modifiedStudentList`. The items of this new array are:

```
["Merengue", "Orion", "Jannat", "Ray", "Amal", "Sabrina", "Karan"]
```

When adding an array to an existing array its elements are added rather than nesting the array when using `concat()`. Let's take for example an array of numbers:

```
let numArray1 = [100, 200, 300, 400];
```

Now let's concat an array of numbers to the `numArray1` array and declare a new variable to reference the result of the `concat()` method:

```
let numArray2 = numArray1.concat([500,600,700]);
```

The array elements are added to the `numArray1` array and a new array called `numArray2` containing the following element is returned:

```
[100, 200, 300, 400, 500, 600, 700]
```

What happens if we have a nested array whose elements we are trying to add to `numArray1`:

```
let numArray2 = numArray1.concat([500,600,700,[800, 900, 1000]]);
```

Here are using `the concat()` method to add an array which contains another array nested inside it `[500,600,700,[800, 900, 1000]]`. In this case the elements of the array are added only to the 1<sup>st</sup> level of nesting i.e. the elements `[500, 600, 700]`. The 2<sup>nd</sup> level of nested elements `([800,900,1000])` will be added as an array:

```
[100, 200, 300, 400, 500, 600, 700, [800, 900, 1000]]
```

Therefore, only the elements up to the 1<sup>st</sup> level of nesting are added as individual elements.

## Note

The `concat()` method will return a new modified array. The existing array remains unchanged.

### 4.4.2 Deleting an array element

Listed in this section are a few methods to delete array elements.

#### 4.4.2.1 Array.prototype.pop()

In order to delete an element from an array, the `pop()` method is used which will remove the last element in an array. Therefore, decreasing the length of an array. Take for example an array with mixed data type values:

```
let mixedArray = [
  1,
  {id: 1},
  'British Columbia',
  [1,2,3,4],
  null,
];
```

Let's use the `pop()` method to remove the last item in this array:

```
mixedArray.pop();
```

The `pop()` method removes the last item in the `mixedArray` which is the `null` value, and the `mixedArray` is returned with the remaining values:

```
[1, [object Object] {  
  id: 1  
}, "British Columbia", [1, 2, 3, 4]]
```

#### 4.4.2.2 Array.prototype.shift()

The **shift()** method will remove an element from the start of an array and return the altered array.

Let's use this method on the `mixedArray`:

```
let mixedArray = [  
  1,  
  {id: 1},  
  'British Columbia',  
  [1,2,3,4],  
  null,  
];  
  
mixedArray.shift();
```

The **shift()** method will remove the first item in the `mixedArray` which is the number `1`:

```
[ [object Object] {  
  id: 1  
}, "British Columbia", [1, 2, 3, 4], null]
```

#### 4.4.2.3 Array.prototype.splice()

The **splice()** method can also be used to delete items from an array. To recap the **splice()** method takes three arguments:

- The first argument specifies the index at which the items should be deleted in the array
- The second argument specifies the number of items that should be removed from the array
- The third argument specifies the elements to be added to the array

Let's remove the 1<sup>st</sup> two items of the `mixedArray` starting at index 0:

```
mixedArray.splice(0, 2);
```

This will remove the first two items `1, {id: 1}` from the array. We have omitted the 3<sup>rd</sup> parameter as we don't want to add anything to the array. The original array is returned modified with the remaining three elements:

```
[ "British Columbia", [1, 2, 3, 4], null ]
```

In order to delete an item in an array whose value is known but index is unknown, you can use the `indexOf()` method and then the `splice()` method. Take as an example the following array of strings:

```
let alphabet = ['a', 'b', 'c', 'd', 'e', 'f'];
```

We know we would like to remove the letter '`d`' from the above array. In order to do so we must find where it occurs within the array i.e. its index. In order to do so we will use the `indexOf()` method which searches an array for a value that is passed to it as an argument and returns its index:

```
let item = alphabet.indexOf('d')  
console.log(item); // 3
```

Logging `item` to the console lets you know that the string is '`d`' at index 3. Now that we know the index we can remove it from the alphabet array using the `splice` method:

```
alphabet.splice(item, 1);
```

The `splice` method takes as arguments:

1. The index of the element to be removed which is referenced by `let item`
2. The number of items to remove i.e. `1` item

The spliced array will contain the following elements:

```
[ "a", "b", "c", "e", "f" ]
```

#### 4.4.2.4 delete keyword

The `delete` keyword allows you to delete the value of an item whose index is specified, leaving it as `undefined`. It does not alter the length of the array. Take as an example a simple array of string values:

```
let groceryList = ['eggs', 'almond milk', 'bread', 'cereal'];
```

Let's delete the first item in the `groceryList` array at index `0` using the `delete` keyword:

```
delete groceryList[0];
```

Logging the `groceryList` array to the console will return the array with its length intact. However, the string at index `0` will be replaced with an `undefined` value:

```
console.log(groceryList);
```

```
[undefined, "almond milk", "bread", "cereal"]
```

#### 4.4.2.5 Resetting an array

We can reset an array by changing the value of its `length` property. Therefore, as a result emptying its elements. Let's reset the `groceryList` array:

```
groceryList.length = 0;
```

The `groceryList` array now has a `length` property of 0. The `length` property of an array corresponds to the number of items in it. Subsequently, now the `groceryList` array has zero items:

```
[]
```

#### 4.4.3 Changing array elements

In order to swap an array element, access its index and then provide a new element at that particular index in the array:

```
let ids = [208, 937, 13, 450];
ids[0] = 10;
console.log(ids);
```

```
[10, 937, 13, 450]
```

The number 208 at index 0 is swapped with a new value 10.

#### 4.4.4 Slicing an array

The `slice()` method will slice the array values at the specified index and return a shallow copy of the sliced values. It does not mutate the original array. Instead it returns a new array containing the sliced values. This method takes two arguments:

1. The start index at which to slice from
2. The end index at which to end the slice at. The value at the end index will not be included

Take as an example an array of prime numbers:

```
let prime = [2, 3, 5, 7, 11, 13, 17, 19, 23];
```

Let's slice the first five prime numbers from the array with the `slice()` method:

```
let fivePrimeNumbers = prime.slice(0,5);
```

This will slice the array starting from index 0 up to but not including index 5. Therefore, returning:

```
[2, 3, 5, 7, 11]
```

If you do not specify an end index, the `slice()` method will slice the array from the starting index all the way up to the end of the array. For example let's slice the array from index 4 onwards:

```
let primeNumbers2= prime.slice(4);
```

This will slice the `prime` array starting from and including index 4 all the way up to the end of the array. Logging `primeNumbers2` to the console will return:

```
[11, 13, 17, 19, 23]
```

In order to start from the end of a string you can use a negative number. For example:

```
let primeNumbers3= prime.slice(-4, -1);
```

The start position will be position number **4** starting from the end of the array. You will count from the number **1** and not **0**. The `slice()` method will end at but not including position **1** (from the end of the array). The result will be:

```
[13, 17, 19]
```

#### 4.4.5 Array.prototype.forEach ()

This is a very useful method introduced in ES5 that executes a function for each element in a given array that has a value. For example:

**Syntax:** `function(currentValue, index, arr)`

The `forEach()` method will take a callback function as an argument, which in turn has the following parameters:

1. `currentValue`: The value of the current element. Required parameter
2. `index`: The index of the current element. Optional parameter
3. `arr`: The array that the element belongs to. Optional parameter

Let's take the following array as an example:

```
let amazingPrices = [9.99, 14.75, 13.99, 99.99, 79.98];
```

Using the `forEach()` method we can print out the value of each array item and its index:

```
amazingPrices.forEach(function(item, index, arr) {  
  console.log(`item: ${item} index: ${index}`);  
})
```

The following will be logged to the console:

```
"item: 9.99index: 0"  
"item: 14.75index: 1"  
"item: 13.99index: 2"  
"item: 99.99index: 3"  
"item: 79.98index: 4"
```

The `forEach()` method will execute a function for each element in the array. The function logs to the console the value of each array item and its index. This method can only be used on arrays, maps and sets.

The `forEach()` method is useful when computing values for every array item. For example let's return the discounted prices after a 15% discount:

```
amazingPrices.forEach(function(item, index, arr) {  
  let discount = 0.15  
  item = item - (item * discount);  
})
```

```
    console.log(item.toFixed(2))  
})
```

Each item in the array will be multiplied by 0.15 (15%). Then this amount will be subtracted from the price of the item in order to give the final total price rounded off to two decimal places using the `toFixed()` method.

#### 4.4.5 Array.prototype.some()

The `some()` method will determine whether a queried value exists in an array by executing a callback function for every element in an array. If the queried value is found, the boolean `true` is returned, else boolean `false` is returned. Let's try this out, consider the following the array:

```
let hackathonTeamA = [  
  {  
    name: 'Kenna',  
    role: 'frontend'  
  },  
  {  
    name: 'Ray',  
    role: 'backend'  
  },  
  {  
    name: 'Jim',  
    role: 'security'  
  },  
  {  
    name: 'Buztos',  
    role: 'designer'  
  }  
];
```

`HackathonTeamA` is an array of objects with two **key:value** pairs each (`name` and `role`). We will now query `hackathonTeamA`, for a `designer` role. Each team must have a designer. Instead of looping through every item of the array and querying each **key:value** pair in the object, we will use the `some()` method:

```
let designer = hackathonTeamA.some(function(item) {  
  return(item.role === 'designer');  
});
```

Let's deconstruct what is happening here:

- The `some()` method is called on the `hackathonTeamA` array
- The method calls back a function which takes in an argument called `item`
- `item` refers to each array item
- Within the body of the function value of the `role` key in each item is queried to check if it is strictly equal to the string '`'designer'`'
- The result of this is assigned to `let designer`

We can log the result of the `designer` variable to verify if `true` or `false` is returned from the `some()` method on the `hackathonTeamA` array:

```
console.log(designer); // true
```

`true` is logged since one of the values of the `role` property in the objects in the `hackathonTeamA` array is `'designer'`.

The `some()` method will not mutate the original array and it will not work for array elements that do not have a value.

#### 4.4.6 Array.prototype.find()

The `find()` method will return the first occurrence of a queried value within an array. Picking up from the `hackathonTeamA` example used in the `some()` method in *section 4.4.5*. Let's use the `find()` method to check if any of the roles in hackathon team is a designer:

```
let designer = hackathonTeamA.find(function(item) {  
  return(item.role === 'designer');  
});
```

In the above code snippet, we are using `find()` instead of `some()`. And this method should return the first match that it finds for the string `'designer'`. And we can verify this by logging the variable `designer` to the console:

```
console.log(designer);
```

```
Object {  
  name: "Buztos",  
  role: "designer"  
}
```

Let's add another designer (2<sup>nd</sup> last object in the array) to the `hackathonTeamA` array of objects:

```
let hackathonTeamA = [  
  {  
    name: 'Kenna',  
  },  
  {  
    name: 'Liam',  
  },  
  {  
    name: 'Buztos',  
    role: 'designer'  
  },  
  {  
    name: 'Kenna',  
  }]
```

```

        role: 'frontend'
    },
{
    name: 'Ray',
    role: 'backend'
},
{
    name: 'Jim',
    role: 'security'
},
{
    name: 'Merengue',
    role: 'designer'
},
{
    name: 'Buztos',
    role: 'designer'
}
]

```

Now let's use the `find()` method once again:

```

let designer = hackathonTeamA.find(function(item) {
  return(item.role === 'designer');
});

```

What do you think will be returned when the following statement is executed?:

```
console.log(designer);
```

Since the `find()` method returns the first occurrence of the queried value within an array, the following object will be returned:

```

Object {
  name: "Merengue",
  role: "designer"
}

```

The first occurrence of the string `designer` occurs in the newly added object to the `hackathonTeamA` array. Therefore, it will be the return value assigned to `let designer`.

The `find()` method does not execute on empty array elements.

#### 4.4.6 Array.prototype.every()

The `every()` method will execute a function for every element in an array. This method will check if every element in an array passes a test. If so, then the boolean `true` will be returned. Else, if an element does not pass the test, the boolean `false` will be returned.

In the following example, let's modify the array `hackathonTeamA` so that now every object has an `enrolled` property with either a `true` or `false` value:

```
let hackathonTeamA = [  
  {  
    name: 'Kenna',  
    role: 'frontend',  
    enrolled: true  
  },  
  {  
    name: 'Ray',  
    role: 'backend',  
    enrolled: true  
  },  
  {  
    name: 'Jim',  
    role: 'security',  
    enrolled: true  
  },  
  {  
    name: 'Buztos',  
    role: 'designer',  
    enrolled: false  
  }  
]
```

The first three objects in the array have an `enrolled` property whose value is the boolean `true`. The last object's `enrolled` property is set to the value `false`.

Every team member must be enrolled in the hackathon to take part. Therefore, the `enrolled` property for each member must have a value of `true`. To check if it does, we can use the `every()` method which executes a function for each array item to see if it passes a set test:

```
let enrolled = hackathonTeamA.every(function(item) {  
    return(item.enrolled === true);  
});
```

The code above can be read as such:

- Use the `every()` method on the `hackathonTeamA` array
- The `every()` method takes as an argument a callback function which will execute for all the array items therefore, it takes `item` as an argument
- The function will check if the `enrolled` property of each object `item` in the array has a value of `true`. If so then assign `true` to be the value of `let enrolled`.
- If any of the objects item has a value of `false`, then the test shall fail and `false` will be returned and assigned to `let enrolled`.

We can verify the result by logging the value of the `enrolled` variable:

```
console.log(enrolled);
```

Since the last object in the array had a value of `false` assigned to the `enrolled` property, `false` will be returned:

```
false
```

The `every()` method will not execute the callback function for array items without values. And it also does not mutate the original array.

#### 4.4.7 Difference between `some()`, `find()` and `every()`

The `every()` method will only return true if each and every array element passes a set test. Whereas `some()` will return true if there is at least one occurrence which passes the test. The `find()` method will find the first occurrence of a queried element's value

#### 4.4.8 `Array.prototype.sort()`

A sorting algorithm will sort elements in the order specified (numerically or alphabetically). We don't need to write our own sort algorithm, JavaScript provides a `sort()` method to do this. This method will sort array items in alphabetical, numeric, ascending or descending order. This method will cast the array values to strings and then compare them. By default the `sort()` method will arrange elements in ascending and alphabetical order after being applied to an array and return the changed array.

**Syntax:** `Array.sort([compare function])`

The compare function is optional and will specify the sort order when passed to the `sort()` method.

Let's take a look at an example to understand how `sort()` works:

```
let numbers = [5, 0, 3, 1, 2];
```

We can sort the numbers in the numbers array using `sort()`:

```
numbers.sort();
```

This will sort the numbers in the array in ascending order:

```
[0, 1, 2, 3, 5]
```

Now let's try this a bit differently, we will change the values in the numbers array:

```
let numbers = [0, 1, 10, 20, 3];
```

Applying the `sort` method, gives us an unexpected result:

```
numbers.sort();
```

```
console.log(numbers); // [0, 1, 10, 20, 3]
```

The reason why the numbers are not in ascending order is because the `sort` method will cast array items to strings and then compare them. This means that the strings '`'0'`', '`'1'`', '`'10'`', '`'20'`' and '`'3'`' are compared. The string '`'10'`' is compared to '`'3'`' and placed before `3` as `1` is less than `3`.

In order to fix this, you will use a compare function to specify the order of the sort like so:

```
numbers.sort(function(a,b) {  
    return(a - b)  
});
```

The compare function in the `sort()` method takes two parameters `a` and `b`. The compare function always returns either a positive, negative number or 0. The two parameters will be compared:

- If  $a - b$  results in a negative number then `a` will be placed before `b`.
- If  $a - b$  results in a positive number then `b` is placed after `a`.
- If 0 is returned then `a` and `b` will not be re-arranged.

So now using the compare function `10(a)` will be compared to `3(b)`. And since  $10 - 3$  returns a positive number (7), 10 will be placed after 3. Therefore returning the sorted array:

```
[0, 1, 3, 10, 20]
```

We can also sort the array in descending order by passing in a compare function that takes two parameters `a` and `b`. Let's sort the items in the numbers array in descending order:

```
numbers.sort(function(a,b) {
```

```

        return(b - a)
    });

console.log(numbers); [20, 10, 3, 1, 0]

```

Therefore, the numbers in the array are now in descending order. Instead of returning `a - b`, we are now returning `b - a`. So we are doing the opposite to sort items in descending order.

What about string values? The `sort()` method will also sort an array alphabetically:

```

let characters = ['Coco', 'Merengue', 'Drago', 'Flip', 'Hazel',
'Rocket'];

characters.sort(); // ["Coco", "Drago", "Flip", "Hazel", "Merengue",
"Rocket"]

```

The `characters` array is returned sorted alphabetically in ascending order. What about alphabetical descending order? Using a compare function that can be done as well:

```

characters.sort(function(a, b) {
    return(a < b)
})

```

The compare function now checks if `a` is smaller than `b`, if so `a` will be placed after.

```

console.log(characters);
["Rocket", "Merengue", "Hazel", "Flip", "Drago", "Coco"]

```

And now the array items are in descending alphabetical order (Z-A).

Besides these methods mentioned in these chapters, there are lots of other methods! It is advised to use the methods provided by `Array.prototype` object when possible rather than code your own.

Some other methods that you can explore are `some()`, `find()`, `fill()`.

## Exercises

- Consider the following array and answer the question:

```
let myAlphabet = ['A', 'B', 'C', 'D', 'E', 'F', 'G'];
```

- What is the length of the array?
- Write a function called `myAlphabetLength` which `console.logs` the length of the array
- Within the function also use an `if-conditional` statement that checks if the number of items within the array is less than 4

- Declare and initialize an array called '`Planets`' with 5 string values

- `console.log` each item in the array

- Also `console.log` the index in each iteration
3. Declare and initialize an array called `wowDataTypes`
- The array must have 5 different data types
  - Iterate over the array and `console.log` each item in the array, its index and data type in the array
4. `console.log` each item in this array WITHOUT using a for loop
- ```
let myArr = [ 1, 2, 'One', true];
```
5. Loop over the 2 arrays and if there are any common occurrences/elements. If so `console.log` them
- ```
let student1Courses = ['Math', 'English', 'Programming'];
let student2Courses = ['Geography', 'Spanish', 'Programming'];
```
6. Compare the 2 arrays and find common food if any:
- ```
let food = ['Noodle', 'Pasta', 'Ice-cream'];
let food1 = ['Fries', 'Ice-cream', 'Pizza'];
```
7. Compare the 3 arrays and find any common elements:
- ```
let values1= ['Apple', 1, false];
let values2 = ['Fries', 2 ,true];
let values3 = ['Mars', 9, 'Apple'];
```
8. For each item in this array `console.log` the letters in each item
- ```
let furniture = ['Table', 'Chairs','Couch'];
```
9. Does the `push()` method modify/mutate the original array?
10. What will be logged?
- ```
let villagers =
['Coco','Merengue','Drago','Flip','Hazel','Rocket'];
villagers[2] = 'Poppy';
console.log(villagers);
```
11. Remove the letter `e` from the string `'icecream'`
12. `pop()` will mutate the original array's length and return the array. True or False?
13. Using the `concat()` method, concat these 2 arrays and remove any extra occurrences of an element
- ```
let arr = [1,2,3];
```

```
let arr1 = [1,5,6];
```

14. Consider the following array of objects for a small indie bakery:

```
let bakery = [
  {
    cookie: 'oreo',
    calories:350
  },
  {
    cookie: 'fudge',
    calories:450
  },
  {
    cookie: 'butter',
    calories:700
  },
]
```

- Write some code so that when the `bakery` array is queried or `console.logged` it is now:

```
[ [object Object] {
  calories: 350,
  cookie: "oreo",
  twoHelpings: 700
}, [object Object] {
  calories: 450,
  cookie: "fudge",
  twoHelpings: 900
}, [object Object] {
  calories: 700,
  cookie: "butter",
  twoHelpings: 1400
}]
```

15. Remove the first element from this array:

```
let amazingArray = [ {}, null, undefined, '' ];
```

16. What is the difference between `splice()` and `slice()`?

17. Splice the following array and remove the item '`'new code'`' from the `companies` array:

```
let companies = ['cultivating coders', 'purple moon',
'enterprise skills', 'new code', 'soft code'];
```

- Slice the item 'cultivating coders' and assign it to a variable declared as `techX`
- Convert `techX` into an object (hint you will need the spread operator). This object is referenced by a variable called `techZ`
- Assign a `students` property to `techZ` whose value is the number 8200
- `Console.log` the `techZ` object it should look like this:

```
Object {
  0: "cultivating coders",
  students: 8200
}
```

18. Have a look at the following code for `insects` and find if the `insects` object has an insect called 'Diptera'

```
let insects = [
  {
    taxonomy: 'insecta',
    name: 'Archaeognatha',
    species: 513,
    exoskeleton: true
  },
  {
    taxonomy: 'insecta',
    name: 'Plecoptera',
    species: 3743,
    exoskeleton: true
  },
  {
    taxonomy: 'insecta',
    name: 'Thysanoptera',
    species: 5864,
    exoskeleton: true
  },
  {
    taxonomy: 'insecta',
```

```

        name: 'Trichoptera',
        species: 14391,
        exoskeleton: true

    },
    {
        taxonomy: 'insecta',
        name: 'Diptera',
        species: 155477,
        exoskeleton: true
    },
];

```

19. Add the follow object to `insects` object in question #18:

```

        taxonomy: 'insecta',
        name: 'Mantodea',
        species: 2400 ,
        exoskeleton: true

```

20. Carrying on from question #19, the `insects` object which is as below. Query whether the `species` property in each object of the insect array has a value of greater than `1000`

```

let insects = [
    {
        taxonomy: 'insecta',
        name: 'Archaeognatha',
        species: 513,
        exoskeleton: true
    },
    {
        taxonomy: 'insecta',
        name: 'Plecoptera',
        species: 3743,
        exoskeleton: true
    },
    {

```

```

        taxonomy: 'insecta',
        name: 'Thysanoptera',
        species: 5864,
        exoskeleton: true
    },
    {
        taxonomy: 'insecta',
        name: 'Trichoptera',
        species: 14391,
        exoskeleton: true

    },
    {
        taxonomy: 'insecta',
        name: 'Diptera',
        species:155477,
        exoskeleton: true
    },
    {
        taxonomy: 'insecta',
        name: 'Mantodea',
        species:2400 ,
        exoskeleton: true
    },
];

```

21. Consider the following array referenced by `let eshoppe` which is an array of objects for a single page shopping cart application being built:

```

let eshoppe = [
    {
        name: 'Pens',
        units: 403,
        price: '$1.99'
    },

```

```

    {
      name: 'Cotton socks',
      units: 432,
      price: '$3.99'
    },
    {
      name: 'Shirts',
      units: 1010,
      price: '$12.99'
    },
    {
      name: 'Stickers',
      units: 8200,
      price: '$1.99'
    },
    {
      name: 'Coffee mug',
      units: 2140,
      price: '$10.99'
    },
  ],
];

```

- Sort the `eshoppe` array by price in ascending order (from the lowest to the highest price) as some users would like to see items sorted by price.
- `Console.log` the `eshoppe` array, you should see that the objects in the array are sorted by ascending order:

```

[[object Object] {
  name: "Pens",
  price: 1.99,
  units: 403
}, [object Object] {
  name: "Stickers",
  price: 1.99,
  units: 8200
}, [object Object] {
  name: "Cotton socks",
  price: 3.99,
  units: 432
]
];

```

```

}, [object Object] {
  name: "Coffee mug",
  price: 10.99,
  units: 2140
}, [object Object] {
  name: "Shirts",
  price: 12.99,
  units: 1010
}
]

```

- Now sort the `eshoppe` array by alphabetical order (from A - Z). As sometimes users like to see items in alphabetical order.
- Console.log the `eshoppe` array, you should see that the objects in alphabetical order, like so:

```

[[object Object] {
  name: "Coffee mug",
  price: "$10.99",
  units: 2140
}, [object Object] {
  name: "Cotton socks",
  price: "$3.99",
  units: 432
}, [object Object] {
  name: "Pens",
  price: "$1.99",
  units: 403
}, [object Object] {
  name: "Shirts",
  price: "$12.99",
  units: 1010
}, [object Object] {
  name: "Stickers",
  price: "$1.99",
  units: 8200
}
]

```

22. For the following array use the `fill()` method such that the array returned is `[199.99, 89.75, 10, 10, 8200.99, 79.95]`

```
let prices = [199.99, 89.75, 62.25, 13.99, 8200.99, 79.95];
```

23. For the same prices array in #22, find the index of the `8200.99` using the `findIndex()` method:

```
let prices = [199.99, 89.75, 62.25, 13.99, 8200.99, 79.95];
```

- Once you find the index, replace the value `8200.99` with the value `9900`

24. First `sort()` and then `reverse()` the following array:

```
let items = ['Calculator', 'Laptop',
'Console', 'USB', 'Keyboard'];
```

25. Consider the following question from Toptal and try to work out what will be console.logged:

```
var arr1 = "john".split('');
var arr2 = arr1.reverse();
var arr3 = "jones".split('');
arr2.push(arr3);

console.log("array 1: length= " + arr1.length + " last=" +
arr1.slice(-1));
console.log("array 2: length=" + arr2.length + " last=" +
arr2.slice(-1));
```

26. The `reverse()` method reverses the contents of an array , thereby mutating the original array. Can you come up with a way of reversing the contents of the original array called `num1` without mutating it?

```
let num1 = [100, 818, 319000, 79];
```

## Answers

1.

- The length of the `myAlphabet` array is 7, as there are 7 items in the array
- Function to log the length of the array:
- If conditional to check if the number of items in the array is less than 4

```
function myAlphabetLength(arr) {
    console.log(arr.length);
}

myAlphabetLength(myAlphabet); //7
```

```
function myAlphabetLength(arr) {
    console.log(arr.length);
    if(arr.length < 4) {
        console.log('less than 4 items')
    } else {
        console.log('more than 4 items')
    }
}

myAlphabetLength(myAlphabet); //7 "more than 4 items"
```

2.

```
let planets = ['Earth', 'Mars', 'Jupiter', 'Venus', 'Pluto'];

for(let i = 0; i < planets.length; i++) {
    console.log(planets[i] + ' ' + i);
}
```

3.

```
let wowDataTypes = [true, 'a', 100, {greeting: 'hello'}, null];

for(let i = 0; i < wowDataTypes.length; i++) {
    console.log(` ${wowDataTypes[i]} ${i}
${typeof(wowDataTypes[i])}`)
}
```

4.

```
console.log(myArr); // [1, 2, "One", true]
```

5.

```
let student1Courses = ['Math', 'English', 'Programming'];

let student2Courses = ['Geography', 'Spanish', 'Programming'];

for(let i = 0; i < student1Courses.length; i++) {
    for(let k = 0; k < student2Courses.length; k++) {
        if(student1Courses[i] === student2Courses[k]) {
            console.log(student1Courses[i])
        }
    }
}

"Programming"
```

6.

```
let food = ['Noodle', 'Pasta', 'Ice-cream'];

let food1 = ['Fries', 'Ice-cream', 'Pizza'];

for(let i = 0; i < food.length; i++) {
    for(let k = 0; k < food1.length; k++) {
```

```

        if(food[i] === food1[k]){
            console.log(food[i])
        }
    }
}

```

"Ice-cream"

7.

```
let combinedValues = [values1, values2, values3].flat();
```

8.

```

for(let i = 0; i < furniture.length; i++){
    for(let k = 0; k < furniture[i].length; k++) {
        console.log(furniture[i][k])
    }
}

```

9. Yes

10. The value at `villagers[2]` is replaced with the new value.

`["Coco", "Merengue", "Poppy", "Flip", "Hazel", "Rocket"]`

11.

```

let icecream = "icecream";
let newIcecream = icecream.split('e').join('');

```

`"iccramp"`

`split()` method will truncate the characters in the string:

```
let newIcecream = icecream.split('e') // ["ic", "cr", "am"]
```

And then `join()` will join the characters in the array without any separator:

```
let newIcecream = icecream.split('e').join('');
```

12. True

13.

```

let arr2 = arr.concat(arr1);

let noDuplicate =[];
arr2.forEach(function(item){
    if(!noDuplicate.includes(item)){
        noDuplicate.push(item)
    }
})

```

```
}) ;  
console.log(noDuplicate) ;
```

14.

```
bakery.forEach(function(item) {  
    return(item.twoHelpings = item.calories * 2);  
}) ;  
console.log(bakery) ;
```

15.

```
amazingArray.shift() ;
```

```
[null, undefined, ""]
```

16. The `slice()` method will slice the array values at the specified index and return the sliced values. It does not mutate the original array. Instead it returns a new array containing the sliced values. This method takes two arguments:

- The start index at which to slice from
- The end index at which to end the slice at. The value at the end index will not be included
- The `splice()` method can also be used to delete items from an array. The method takes three arguments:

The first argument specifies the index at which the items should be deleted in the array

The second argument specifies the number of items that should be removed from the array

The third argument specifies the elements to be added to the array

17.

```
let companies = ['cultivating coders', 'purple moon',  
'enterprise skills', 'new code', 'soft code'];  
companies.splice(3, 1);  
let techX = companies.slice(0, 1)  
let techZ = {...techX}  
techZ.students = 8200;  
console.log(techZ) ;
```

18.

```
let diptera = insects.some(function(item) {  
    return(item.name === 'Diptera')  
}) ;
```

```
console.log(diptera); //true
```

19.

```
insects.push({  
    taxonomy: 'insecta',  
    name: 'Mantodea',  
    species: 2400,  
    exoskeleton: true  
});
```

20.

```
let speciesNumberCheck = insects.every(function(item) {  
    return(item.species > 1000);  
});  
console.log(speciesNumberCheck); //false
```

21. Sort by price

```
eshoppe.forEach(function(item) {  
    item.price = item.price.substring(1);  
    item.price = parseFloat(item.price)  
    console.log(item.price)  
});  
eshoppe.sort(function(a,b) {  
    return(a.price - b.price)  
});
```

Sort by alphabetical order:

```
eshoppe.sort(function(a,b) {  
    return(a.name > b.name)  
});
```

22.

```
// use the fill() method  
prices.fill(10, 2, 4);
```

23. Find the index of 8200.99

```
let index = prices.findIndex(function(item) {  
    return(item === 8200.99);  
});
```

```
console.log(index); //4
```

Replace 8200.99 with 9900

```
prices[index] = 9900;  
console.log(prices);
```

```
[199.99, 89.75, 62.25, 13.99, 8900, 79.95]
```

24.

```
items.sort();  
items.reverse();  
console.log(items);
```

```
["USB", "Laptop", "Keyboard", "Console", "Calculator"]
```

25.

```
"array 1: length= 5 last=j,o,n,e,s"  
"array 2: length=5 last=j,o,n,e,s"
```

<https://www.toptal.com/javascript/interview-questions>

26.

```
let num2 = num1.slice().reverse();  
console.log(num2);  
console.log(num1);
```

```
[79, 319000, 818, 100]
```

```
[100, 818, 319000, 79]
```

## Summary

In this chapter, we covered the fundamental aspects of arrays. We also learned the different ways of declaring arrays. Followed by the properties available on the **Array.prototype** object from which all user-created arrays inherit from. Then we manipulated arrays via **Array.prototype** methods available for user-created arrays. Arrays are extremely useful data structures and we combined those with objects to do a few **Array.prototype** exercises. To practice the fundamental aspects of arrays I highly suggest you do the exercises provided in this chapter. In the next chapter, we will cover some ES6 methods that use the **spread** syntax, and newer methods like **filter()**, **map()** and **reduce()**.

# Advanced Amazing arrays - 5

*"Strive for continuous improvement, instead of perfection" – K. Collins*

In the last chapter you were introduced to arrays and a few array methods. In this chapter we will pick up from where we left and continue on to some more advanced array methods.

We're going to cover the following main topics:

- Destructuring arrays
- Spread operator
- map(), reduce() and filter() methods
- Chaining map(), reduce() and filter()
- Multi-dimensional arrays in JavaScript

## 5.1 Destructuring Arrays

Destructuring is an ES6 feature. Destructuring arrays allow you to unpack the values from arrays and assign them to individual variables. This allows for a more direct way of accessing array items. Let's have a look at what destructuring manually means:

```
let alphabet = ['a', 'b', 'c', 'd', 'e', 'f'];
```

In order to assign the value of the first array element from `alphabet` to a variable, this is what we would do:

```
let firstEl = alphabet[0];
```

And then to assign the second array element from `alphabet` to a variable, we would access its value at index `1`:

```
let secondEl = alphabet[1];
```

Logging the `firstEl` and `secondEl` variables we get:

```
console.log(firstEl, secondEl);
```

```
"a" "b"
```

Imagine that we would like to do this for all the array elements in the `alphabet` array; it would be a bit tiresome to do this for each one of the array elements. In this case, destructuring will be useful as it will allow you to unpack the values from the array into individual variables. And this is how we would do it:

```
let [firstEl, secondEl, thirdEl, fourthEl, fifthEl, sixthEl] =  
alphabet;
```

The variables follow an order from left to right. Therefore, the first variable (`firstEl`) will be assigned the value of the array item at index `0`, `secondEl` will be assigned the value of `alphabet[1]` and so on.

Destructuring assignment allows for a much faster and easier way of assigning array values to variables. We can also skip array elements if we wish. For example, let's skip the letter '`d`' which is at `alphabet[3]` by omitting the variable name `fourthEl` and replacing it with a comma :

```
let [firstEl, secondEl, thirdEl, , fifthEl, sixthEl] = alphabet;
```

There is no reference to the variable `fourthEl` whose value would correspond to the letter '`d`' at the third index in the `alphabet` array. So when we log `fourthEl` to the console a `ReferenceError` occurs:

```
console.log(fourthEl);
```

```
ReferenceError: fourthEl is not defined
```

A comma separator is used to skip values in an array. For example let's skip the first 4 values in the `alphabet` array:

```
let [,,, fifthEl, sixthEl] = alphabet;
```

The variable names have been replaced by commas and will be skipped over. Only `fifthEl` and `sixthEl` variables will be assigned values corresponding to their position moving left to right in the `alphabet` array. For example let's check the `fifthEl` variable:

```
console.log(fifthEl); // "e"
```

Nested arrays can be destructured as well. Take as an example a nested array called `arrNest`:

```
let arrNest = ['a', 'b', 'c', [1,2,3]];
```

In order to use destructured assignment, we will declare variables like so:

```
let [a,b,c,[one, two, three]] = arrNest;
```

The variables are declared moving from left to right corresponding to the array values from index `0` onwards.

```
console.log(a, b, c, one, two, three);
```

```
"a"  
"b"  
"c"  
1  
2  
3
```

Besides nested arrays, it is also possible to destructure arrays returned from functions. For example consider the following simple function:

```
function values() {  
    return(['This', 'is', 'a', 'message']);  
}
```

We can destructure the array as we have been doing earlier on:

```
let [one, two, three, four] = values();
```

The variables `one`, `two`, `three` and `four` correspond to the values of the array items returned from function `values()`. And we can confirm this by logging the variables to the console:

```
console.log(one, two, three, four);
```

```
"This"  
"is"  
"a"  
"message"
```

You can also swap the values of variables via destructuring. This is a handy feature and is succinct. In the following example we will swap the values of `author1` and `author2` variables:

```
let author1 = 'Herman Hesse';  
let author2 = 'Roald Dahl';
```

Using destructuring:

```
[author1, author2] = [author2, author1];
```

The value of variable `author1` on the left side will correspond with the value of `author2` on the right side. Similarly the value of `author2` on left side will be swapped with the value of `author1` on the right side. And we can confirm this by either using `alert()` or `console.log()` to check either one of the variables:

```
console.log(author1); // "Roald Dahl"
```

Destructuring allows for a quick and succinct way to extract values from arrays easily. Next up is the spread operator, which is an equally handy ES6 feature.

## Exercises

1. Destructure the following array:

```
let pokemonArray = [{name:'Bulbasaur'}, {name:'Squirtle'},  
{name:'Pikachu'}, {name: 'Pokemon'}, {name:'Eevee'}];
```

*Note: you may use whatever variable names that you like*

- Log the fourth variable to the console. What do you get?

2. Destructure the following string so that only the last letter 's' is returned:

```
let myName = 'SleepySaurus';
```

3. Destructure the following nested array:

```
let array1 = ['hello', 'world', [100,200]];
```

*Note: you may use whatever variable names that you like*

4. Swap the values of these 2 arrays. What is the value of `array2`?

```
let array1 = [{role: 'I.T'}, {role: 'game dev'}];
```

```
let array2 = ['thin crust', 'medium crust', 'pan crust'];
```

## Answers

1.

```
let [one, two, three, four, five] = pokemonArray;  
console.log(four);
```

```
Object {  
  name: "Pokemon"  
}
```

2.

```
let [,,,,,,,,,, last] = myName;  
console.log(last);
```

```
"s"
```

3.

```
let [greeting1, greeting2, [num1, num2]] = array1;  
console.log(greeting1, greeting2, num1, num2);
```

```
"hello" "world" 100 200
```

4.

```
[array1, array2] = [array2, array1];  
console.log(array2);
```

```
[ [object Object] {  
  role: "I.T"  
}, [object Object] {  
  role: "game dev"  
} ]
```

## 5.2 Spread operator

The spread and rest operators introduced in ES6 look familiar but work in different contexts. In this section, we will go over the spread operator and its use cases. We will come to the rest operator in *Chapter 5, Fun functions*.

### 5.2.2 Spread operator

The spread operator will spread or expand an array into its set of items. The definition provided by MDN is:

*Spread syntax allows an iterable such as an array expression or string to be expanded in places where zero or more arguments (for function calls) or elements (for array literals) are expected*

What this means is that the spread operator will allow an expression to be expanded in places where multiple elements/arguments are expected.

#### Syntax: ...

The spread operator spreads out the items of the inner array in the outer array. So it starts with [] – which then expands to [1] and then, [1, 2]. After which it becomes [1, 2, 3] etc.

To understand this better let's take a look at an example:

```
let array1 = [1, 2, 3, ...[4, 5]]
```

Nested within `array1` is another array containing the numeric values 4 and 5. We would like to expand on the items in the nested array. Therefore, we use the spread operator denoted by the triple dots (...) just before the nested array.

Let's view the output on logging `array1` to the console:

```
console.log(array1);
```

```
[1, 2, 3, 4, 5]
```

The spread operator spreads out the items of the inner array in the outer array. The spread operator will not mutate the original array. The goal should be to keep data structures such as arrays and objects in their unchanged states i.e. immutable. Fewer states/things changing in your code means that there will be less things to track and worry about.

Down below are a few uses cases of the spread operator

### 5.2.2 Merge arrays

The spread operator can be used to merge the values of two arrays. This can be used in place of the `concat()` method. For example:

```
let greeting = ['Hey', 'there'];
let devGreeting = ['Hello', 'world!'];
```

We can use the spread operator to spread the individual items of the `greeting` and `devGreeting` arrays in one array as such:

```
let greetingFull = [...greeting, ...devGreeting];
```

The `greetingFull` array now contains the following:

```
["Hey", "there", "Hello", "world!"]
```

However, take note that for larger arrays with more items the spread syntax is slower. Therefore, when dealing with larger data sets it is better use the `concat()` method.

### 5.2.3 Copy arrays

We can copy arrays with the spread operator without mutating the original array. For example consider copying the elements from `arrayNum` into the array `copyNum`:

```
let arrayNum = [1, 1.89, 319000, 30000];
let copyNum = [...arrayNum];
```

Now `copyNum` contains the following array values:

```
console.log(copyNum);
```

```
[1, 1.89, 319000, 30000]
```

Let's add another numeric value to the `arrayNum` array by using the `push()` method:

```
arrayNum.push(8200);
```

What do you think the value of the `copyNum` array will be? `copyNum` contains all the elements copied over from the `arrayNum`. Do you think it will contain the new value `8200` pushed to the `arrayNum` array?

Upto now, this is what our code looks like:

```
let arrayNum = [1, 1.89, 319000, 30000];
let copyNum = [...arrayNum];
arrayNum.push(8200);
console.log(copyNum);
```

```
[1, 1.89, 319000, 30000]
```

Now let's log `arrayNum` to the console and check its values:

```
console.log(arrayNum);
```

```
[1, 1.89, 319000, 30000, 8200]
```

Therefore, upon mutating the original array afterwards, the contents of the array copy have not changed. This is different from doing something like this:

```
let arrayNum =[1, 1.89, 319000, 30000];
let copyNum = arrayNum;
arrayNum.push(8200);
```

Upon mutating the original array, the `copyNum` array is mutated as well and as a result, both arrays now have the newly pushed numeric value `8200`:

```
console.log(copyNum);
console.log(arrayNum);
```

```
[1, 1.89, 319000, 30000, 8200]
[1, 1.89, 319000, 30000, 8200]
```

The `copyNum` array is assigned a reference to the `arrayNum` array, since object types are copied by reference, changing the value of the reference array (`arrayNum`) changes the copied array's (`copyArray`) value as well.

Whereas mutating the original array by pushing an element to it does not change the copied array when using the spread operator.

#### 5.2.4 Convert a string into an array

Using the spread operator also allows us to convert a string into an array of the string's characters. This is much faster and more convenient than looping through the characters in a string and accessing them at each index. This example demonstrates this:

```
let game ='Animal Crossing New Horizons';
let chars = [...game];
```

The spread operator will return an array of characters, including the whitespace contained in the string. The result of this operation is:

```
["A", "n", "i", "m", "a", "l", " ", "C", "r", "o", "s", "s", "i",
 "n", "g", " ", "N", "e", "w", " ", "H", "o", "r", "i", "z", "o",
 "n", "s"]
```

Now it becomes much easier to manipulate the characters in the string. For example we can use the `filter()` method which will filter the array elements representing the whitespace (`" "`) from the array. We will come to the `filter()` method in this chapter moving on.

#### 5.2.5 Math object

The Math object in JavaScript provides many methods such as `Math.min()` which will return the smallest integer from the arguments provided to the method and `Math.max()` which will return the largest integer. Take for example:

```
console.log(Math.max(319000, 30000, 8200)); //319000
```

Let's try this on an array of integers:

```
console.log(Math.max([319000, 30000, 8200])); //NaN
```

The result of the operation will return `Nan`. Let's use the spread operator which will spread the elements into a list of arguments and then apply the `Math.max()` method:

```
console.log(Math.max(...[319000, 30000, 8200])); //319000
```

Using the spread operator we are able to apply the `Math.max` method to pick out the largest number from a list of arguments created by the spread operator.

## Exercises

1. What will be the output of the following code?:

```
let a =[1,2,3];
let b =[4,5,6, ...a];
let c =[...a,...b];
console.log(c);
```

2. Join these two arrays using the spread operator and reference the result via let

```
resultArray
```

```
let arrayA = [{num: 1, max: 10}, {num:2, max:100}, {num3: 3, max:450}];
let arrayB = [{fruit: 'apple'}, {fruit: 'mango'}, {fruit: 'blueberry'}];
```

- What other array method can be used to join arrays? What is the main difference between this method and using the spread operator?
- Loop over `resultArray` and return true if the value "mango" is within the array

## Answers

- 1.

```
[1, 2, 3, 4, 5, 6, 1, 2, 3]
```

- 2.

```
let resultArray = [...arrayA, ...arrayB];

resultArray.forEach(function (arrayItem) {
  if(arrayItem.fruit === 'mango')
    //console.log('true');
    return(true);
});
```

`concat()` can be used to merge two or more arrays. `concat()` will return a new array.  
`concat()` is preferred when working with a larger array of items

## 5.3 Map, Reduce and Filter methods

ES5 introduced map, reduce, and filter methods. These are powerful features of the JavaScript language that will allow you to use and maneuver arrays to a greater extent. Additionally, these methods allow you to take a functional programming approach. In a gist, functional programming is a programming paradigm that avoids changing state and mutating data. Programs are treated as mathematical functions whose execution will always produce identical and reliable results when the same arguments are passed to them. Let's have a look at each one of these methods starting with `map()`.

### 5.3.1 map()

The `map()` method will transform every element of an array by executing a callback function for each array item. This method does not mutate/change the original array instead, it returns a new array with the mapped/changed array items.

**Syntax:** `array.map (callback, thisArg);`

The first parameter is a function that will be executed on each element in the array. This is called a callback function as it is executed when the method is called on the array. The callback function in turn can take 3 arguments:

- Current Item: The current item in the array being processed. Required parameter.
- Index: The index of the current item in the array. Optional parameter. For example index 0.
- Array: The array on which `map()` will enact. Optional parameter. There is already a reference to the original array tied in to the `map()` method when it is called.

The 2nd parameter that the `map()` method can take is:

`thisArg`: The value to use as `this` while executing the callback function. Optional parameter.

The example preceding shows the transformation of an existing array of numbers using `map()`:

```
let numberArray = [10,100,1000,10000];
let mappedArray = numberArray.map(function multiplySix(item) {
    return(item * 6);
});
```

In this example, the value of each array item in the `numberArray` is multiplied by 6, by executing the callback function named `multiplySix`. The first argument of the callback function is `item` which refers to the current item in the array being processed. It is a required parameter. Once the callback function has finished executing for each item in the array, the newly transformed values of array items are returned and assigned to `let mappedArray`.

We may also pass an anonymous callback function, which means you don't have to name it:

```
let numberArray = [10,100,1000,10000];  
  
let mappedArray = numberArray.map(function(item) {  
  
    return(item * 6);  
  
});
```

Therefore, the callback function is anonymous as it has no name. We still have to pass in the required parameter which we have called `item`. *This represents the current item that the callback function is processing.*

On querying the contents of `mappedArray` by using `console.log` we see the transformed values:

```
console.log(mappedArray);
```

```
[60, 600, 6000, 60000]
```

We can also achieve the same result by using a `for` loop:

```
let mapArr = [];  
  
for(let i = 0; i < numberArray.length; i ++){  
  
    mapArr[i] = numberArray[i] * 6  
  
}
```

Here we are using a `for` loop which will iterate over the `numberArray` for the length of the `numberArray`. And remember that the length of an array corresponds to the total number of items in it. At each iteration, the value of the `numberArray` item is multiplied by `6` and assigned to the corresponding index in the `mapArr` array.

`mapArr` now has the following values:

```
[60, 600, 6000, 60000]
```

This brings us to the next point, why should `map()` be used instead of a `for loop` or even the `forEach()` method?

### 5.3.1.2 Map vs for-loop and forEach

Utility methods such as `map()` allow for greater readability of code and are intentional. The meaning inferred by the `map()` utility method is to map/assign new values to existing array items. Whereas a `for-loop` and `the forEach()` method can be used to log the items in an array in addition to transforming them like `map()`.

It is preferable to use `forEach()` and the `for loop` when all you want to do is log data or save it to a database. Also, `for loops` can be used to iterate through array-like objects such as an HTML collection and therefore are useful in DOM manipulation.

Whereas, when it is your intention to change/transform the values of array items then `map()` is the better choice. `Map()` will return a new array with changed values whereas the return value of the callback function of `forEach()` is `undefined`. `forEach()` will execute a function for each array item

but it doesn't return anything. `Map()` can be chained with other methods such as `filter()` and `reduce()`. However since `undefined` is returned by the `forEach()` method, this method cannot be chained along with others.

In terms of speed the `for-loop` is faster than both `map()` and `forEach()`. This is because of the callback function which is an overhead expense. When speed and optimization is a concern consider using a `for loop`. When readability takes precedence consider using utility methods such as `map()` and `forEach()`.

## Exercises

1. Consider the following code and use the `map()` method to achieve the output of `array2` logged to the console:

```
let array1 = [10,20,30,40];  
  
let array2 = [];  
  
//Your awesome code here!  
  
console.log(array2); // [80, 60, 40, 20];
```

2. Using the `map()` method, transform the values of the following array by tripling them. Make sure that the return values are rounded off to 2 decimal places:

```
let numbersArray = [1.01, 89.99, 4562.79, 120];
```

## Answers

- 1.

```
let array2 = array1.map(function(item) {  
  
    return(item * 2);  
  
}).reverse();
```

- 2.

```
let tripledArray = numbersArray.map(function(item) {  
  
    return (item * 3).toFixed(2);  
  
});  
  
console.log(tripledArray);
```

```
["3.03", "269.97", "13688.37", "360.00"]
```

## 5.3.2 Reduce

Introduced in ES5 the `reduce()` method will execute a function on each element of an array and reduce the array to a single value. The reduced value is returned and assigned to a new array.

**Syntax:** `array.reduce(callback, initialValue);`

`callback` is a function that will be executed for each element of the array that `reduce()` is being applied on. And `initialValue` is an optional parameter passed to the `reduce()` method. It is the initial starting value, if needed.

The callback function passed to the `reduce()` method can in turn take four arguments:

- accumulator: this is the total of all of the returned values from the callback functions. The value of the accumulator changes after each iteration
- value: the current value that is being processed
- index: the current index of the value being processed
- array: the original array that the `reduce()` method is being applied to

Take as an example an array of four numbers:

```
let numberArr = [319000, 820, 10];  
  
let reducedArr = numberArr.reduce(function( accumulator, value){  
    return(accumulator + value);  
});  
  
console.log(reducedArr);
```

319830

The `reduce()` method applied on the `numberArr` will execute a callback function on each element in the array and add the items resulting in a total of 319830. This value is referenced by `reducedArr`. Therefore, as demonstrated `reduce()` can be thought of as an abstraction over looping operations.

We can also specify an optional starting initial value. Let's use the `reduce()` method with a starting initial value of 100 and add all the elements in the `numberArr`:

```
let reducedArr = numberArr.reduce(function( accumulator, value){  
    return(accumulator + value);  
}, 100);  
  
console.log(reducedArr);
```

319930

This will add an initial value of 100 to the accumulated total of 319830. Therefore, resulting in a total value of 319930 which is assigned to the variable `reducedArr`.

Besides addition, we can perform other mathematical operations such as subtraction, multiplication etc.

Let's have some fun! With your understanding of objects, arrays and the `reduce()` function, add all the values of the `points` property in the `kawaiidata` object:

```
let kawaiidata =[  
    {  
        username: 'Melodrama',  
        village: 'Apples landing',  
        points: 1000  
    },  
    {  
        username: 'Kawaii',  
        village: 'Kawaii village',  
        points: 1500  
    }]
```

```

    points: 1450
  },
{
  username:'Nook',
  village: 'Zeldander',
  points: 100
},
{
  username:'Tango',
  village: 'Cactiizilla',
  points: 7000
},
]

```

Let's analyse `kawaiidata`:

- It is an array of 3 objects
- Each object has 3 **property : value** pairs (`username`, `village` and `points`)
- The goal is to sum the values of the `points` property present in each object

To complete this task we will be using the `reduce()` method on the `kawaiidata` object:

```

let kawaii_points = kawaiidata.reduce(function(acc, value) {
  return acc + value.points;
}, 0);

```

Here is a brief explanation of what's happening:

- The `reduce()` method is called on the `kawaiidata` object
- The callback function takes two arguments: `acc` (accumulator) which represents the sum total as we iterate through the array. The second argument, `value` represents the current value of the array which is inferred by `value.points`
- The optional initial starting value is `0` when the reduction starts
- For each object in the array, `value.points` will be summed and assigned to the variable `kawaii_points`

```
console.log(kawaii_points); //8550
```

The total sum of all the values of the `points` properties in each object accumulates to `8550`.

Now let's move on to talking about an equally powerful and useful method called `filter()`.

## Exercises

1. Multiply all the values in the following array with each other and then double the result

```
let arr = [1,10,10, 2];
```

2. For the following array, calculate the total population in all the cities:

```
let cities= [  
    {  
        name: 'Albuquerque',  
        population: 2304004  
    },  
    {  
        name:'Toronto',  
        population: 547569284  
    },  
    {  
        name:'Kuwait city',  
        population: 39302848  
    },  
    {  
        name:'Vancouver',  
        population: 7834751  
    }  
];
```

## Answers

- 1.

```
let multiplierArr = arr.reduce(function(acc, value){  
    return acc * value;  
},2);  
console.log(multiplierArr);
```

400

- 2.

```
let total_population = cities.reduce(function(acc, value){
```

```

        return acc + value.population;
    }, 0);
console.log(total_population);

```

597010887

### 5.3.3 filter()

The ES6 **filter()** method also acts upon arrays in JavaScript. It will filter an array based on some criteria and return a **new array** with the filtered elements. A callback function is passed as an argument to the filter method and is executed for each array item. The callback function will either return true or false based on the whether the array elements pass the test or not. If an element satisfies the filter criteria it will be included in the resultant array. If no elements satisfy the filter criteria then an empty array is returned.

**Syntax:** `array.filter (callback, thisArg);`

Let's examine the first value of the **filter()** method. The first parameter is a function that will be executed on elements in the array. This is called a callback function and it in turn can take 3 arguments:

1. Current Item: The current item in the array. Required parameter.
2. Index: The index of the current item in the array. Optional parameter. For example index 0.
3. Array: The array on which **filter()** will enact. Optional parameter.

The 2nd parameter that the filter method can take is:

`thisArg`: The value to use as **this** while executing callback. Optional parameter.

In the following example `yummies` is an array with a duplicate item ('Cupcake'):

```
const yummies = ['Ice-cream', 'Cupcake', 'Donut', 'Cupcake'];
```

Using **filter()** we are able to remove the duplicate items of the `yummies` array:

```

const filteredYummies = yummies.filter(function(currentItem, index) {
    if (yummies.indexOf(currentItem) === index) {
        return currentItem;
    }
});
console.log(filteredYummies);

```

["Ice-cream", "Cupcake", "Donut"]

Now let's examine what is happening in the example:

- The `indexOf()` method returns the position of the first occurrence of a specified string value. For example position 0, 1, 2 etc.
- The `filter()` method filters through each index and applies the `indexOf` method on an array item at a particular index
- We check if the index of the `currentItem` in the `yummies` array is its first occurrence if so then it is part of the `filteredYummies` array
- During the 4th iteration, index or `i` is 3 and the first occurrence of 'Cupcake' using `yummies.indexOf('Cupcake')` was 1. Therefore `3 === 1` is `false`. Therefore, the second occurrence of 'Cupcake' is not included in the `filteredYummies` array
- Note: the `indexOf`/position of the item in the array should be equal to `i`

As another, example let's filter through an array of objects called `games` to filter only PC games:

```
let games = [
  {
    name: 'Nintendo',
    game: 'Animal Crossing'
  },
  {
    name: 'Nintendo',
    game: 'Super Mario'
  },
  {
    name: 'PC',
    game: 'Caesar IV'
  },
  {
    name: 'Playstation',
    game: 'Grand Theft Auto V'
  },
  {
    name: 'Playstation',
    game: 'The Sims 4'
  },
];
;
```

We only want to filter through this array for PC games, therefore, we will use the `filter()` method:

```
let onlyPCGames = games.filter(function(item) {
```

```
    return(item.name === 'PC');
});
```

The following can be surmised from this example:

- The `games` array is filtered for any objects whose `name` property is strictly equal to the string `'PC'`
- If any such object is found, which means that the result is `true`, then the value that is returned will be assigned to the `onlyPCGames` variable

Querying the length and content of the `onlyPCGames` array confirms that the `filter()` method filtered the correct object:

```
console.log(onlyPCGames.length);
console.log(onlyPCGames);
```

```
1
[ [object Object] {
  game: "Caesar IV",
  name: "PC"
} ]
```

In the event no elements satisfy the filter criteria then an empty array is returned. For example suppose we are filtering an array declared as `desserts` for the string `'apple pie'`:

```
let desserts = ['ice-cream', 'gelato', 'brownies', 'cake',
'pudding'];
let applePie = desserts.filter(function(item) {
  return(item === 'apple pie');
});
```

Since no element in the `desserts` array has the string value `'apple pie'`, the filter criteria is not satisfied and therefore, the `applePie` array remains empty:

```
console.log(applePie); // []
```

**Map()**, **reduce()** and **filter()** are useful utility methods that can greatly increase your productivity as a developer. In the next section will learn how to chain these methods and use them together. Before that solve these coding exercises to get a head start on the next section.

## Exercises

1. Filter the following data and return users that are of the `'admin'` type

```
let userData = [
  {
    email: 'user1@hello.com',
    name: 'user 1',
    type: 'regular'
```

```

},
{
  email: 'user2@hello.com',
  name: 'user 2',
  type: 'admin'
},
{
  email: 'user3@hello.com',
  name: 'user 3',
  type: 'admin'
},
{
  email: 'user4@hello.com',
  name: 'user 4',
  type: 'regular'
},
];
let keyword = 'admin';

```

2. For the following array, filter the words which have the characters 'er' in them:

```
let wordArray= ['dog', 'pineapple', 'letter', 'technology',
'chatter', 'donut'];
```

3. In the following array, filter the prime numbers. *A prime number is not divisible by any number other than 1 and by itself.*

```
let numbers = [2, 4, 5, 7, 12, 13, 17, 19, 24, 29, 31, 33, 41,
43, 47, 53];
```

## Answers

- 1.

```
let adminUsers = userData.filter(function(item) {
  return(item.type === keyword);
});
console.log(adminUsers);
```

```
[ [object Object] {
  email: "user2@hello.com",
```

```
    name: "user 2",
    type: "admin"
}, [object Object] {
  email: "user3@hello.com",
  name: "user 3",
  type: "admin"
} ]
```

2.

```
function contains_char(wordArray) {
  return wordArray.indexOf('er') !== -1;
}

let filteredArray = wordArray.filter(contains_char);

console.log(filteredArray);

["letter", "chatter"]
```

Note: *-1 indicates that no words were found. This is how you test for something NOT being in an array in JavaScript:*

3. Code a function to check for non-prime numbers:

```
function isPrime(num) {
  for (let i = 2; num > i; i++) {
    if (num % i === 0) {
      return num ;
    }
  }
  return false;
}
```

Pass in the `isPrime()` function as an argument to the `filter()` method called upon the `numbers` array:

```
let nonPrimeNumbers = numbers.filter(isPrime);
```

Log the `nonPrimeNumbers` array to the console:

```
console.log(nonPrimeNumbers);
```

## 5.4 Chaining Map, Reduce and Filter methods

Practically the calls to `map()`, `reduce()` and the `filter()` methods on an array can be chained with the dot notation. Chaining these methods allows you to do functional programming. The preceding example will make chaining clearer.

A company has received a few resumes for an in-house web developer opening. They are looking to hire JavaScript developers with a cumulative experience of at least 12 years. It does not matter if a single developer possess 12 years of experience or two or more developers have a total of 12 years experience:

```
let webdevResumes = [
  {
    name: 'Sanrio',
    langs: 'Python',
    experience: 8
  },
  {
    name: 'Prince',
    langs: 'Python',
    experience: 1
  },
  {
    name: 'Amy',
    langs: 'JavaScript',
    experience: 5
  },
  {
    name: 'Karan',
    langs: 'JavaScript',
    experience: 7
  },
];
```

It is observed that there are two objects whose `langs` property has a value of '`JavaScript`'. Since we are looking for JavaScript developers, we first have to filter out the objects whose `langs`

property is strictly equal to the string '`JavaScript`'. And for this we will use the `filter()` method:

```
let developers = webdevResumes.filter(function(item) {  
  return(item.langs === 'JavaScript')  
});
```

This returns the following array of objects assigned to `let developers`:

```
[ [object Object] {  
  experience: 5,  
  langs: "JavaScript",  
  name: "Amy"  
}, [object Object] {  
  experience: 7,  
  langs: "JavaScript",  
  name: "Karan"  
} ]
```

Now, we want to extract the cumulative experience from both these objects since the company is looking for at least 12 years in total. And for this, we will use the `reduce()` method, which will return the numbers of years of experience reduced to a single number:

```
let developers = webdevResumes.filter(function(item) {  
  return(item.langs === 'JavaScript')  
}).reduce(function(acc, item) {  
  return( acc + item.experience);  
}, 0);
```

Let's deconstruct that is happening:

- The `reduce()` method is chained to the `filter()` method right where the call to the `filter()` method ends with the dot notation.
- The `reduce()` method is tied in to the result produced by the `filter()` method referenced by variable `developers`.
- The `reduce()` method will take an accumulator as an argument. This is the total of all of the returned values from the callback function. Additionally the current `item` being executed by the `reduce()` method and an initial value of `0` is also passed in as an argument.

Let's query the `developers` variable which references the result of the `filter()` and `reduce()` methods:

```
console.log(developers);
```

This will return the number `12` as the answer:

Therefore, we are able to write clean and functional code by chaining utility methods on our instance arrays provided by the **Array.Prototype** object.

Let's go over another example which involves chaining **filter()**, **map()** and **reduce()**. In this example, we will be working with an array of objects declared as **players**:

```
const players = [  
  {  
    name: 'Superwomanx13',  
    points: 100,  
    bonus: 30  
  },  
  {  
    name: 'Rocko',  
    points: 1350,  
    bonus: 300  
  },  
  {  
    name: 'Mario10',  
    points: 790,  
    bonus: 91  
  },  
  {  
    name: 'Misssel',  
    points: 902,  
    bonus: 101  
  },  
  {  
    name: 'Arrayful',  
    points: 1200,  
    bonus: 45  
  },  
];
```

Our goal is to:

- Filter the players whose value of the `points` property is greater than 1000, for this we will use the `filter()` method
- Then we would like to add the values of the `points` and `bonus` properties of each filtered object. To achieve this we will use `map()`
- Lastly we then want to reduce the values to a single digit and this will be done using the `reduce()` method

First let's go ahead and filter objects whose `points` value is greater than 1000:

```
let topScore = players.filter(function(item) {  
    return(item.points > 1000);  
});
```

Logging `topScore` to the console returns two objects:

```
[ [object Object] {  
    bonus: 300,  
    name: "Rocko",  
    points: 1350  
, [object Object] {  
    bonus: 45,  
    name: "Arrayful",  
    points: 1200  
}] ]
```

Now we would like to use the `map()` method to sum the values of the `points` and `bonus` properties of each object returned by `filter()`. To achieve this we will chain the `map()` method after the `filter()` method call:

```
let topScore = players.filter(function(item) {  
    return(item.points > 1000);  
}).map(function(item) {  
    return(item.points + item.bonus)  
});
```

This will return the values mapped to the `topScore` array:

```
[1650, 1245]
```

Lastly, let's reduce these two mapped values to a single digit by chaining the `reduce()` method:

```
let topScore = players.filter(function(item) {  
    return(item.points > 1000);  
}).map(function(item) {  
    return(item.points + item.bonus)  
}).reduce(function(acc, item) {
```

```
    return(acc + item);  
}, 0);
```

The contents of the `topScore` variable are now reduced to a single digit:

```
2895
```

Therefore, by chaining these three utility methods, we are able to efficiently filter, map and reduce array values.

## Exercises

1. Consider the following array:

```
let words = ['ae', 'baed', 'led', 'ce', 'kaede'];  
  
• Filter the words that have the characters 'ae'  
• Map the filtered words to a length > 3
```

2. Let's make our own planner for switching over into freelancing using the preceding array.

The goal is to find out how much money freelancing makes per week, after all we would want to know that before quitting our day job! :

```
let tasks = [  
  {  
    day: 'Monday',  
    minutes: 480,  
    tasks: 'client work, coding'  
  },  
  {  
    day: 'Tuesday',  
    minutes: 80,  
    tasks: 'reading, coding'  
  },  
  {  
    day: 'Wednesday',  
    minutes: 300,  
    tasks: 'writing, working out'  
  },  
  {  
    day: 'Thursday',  
    minutes: 280,
```

```

    tasks: 'client work, coding'
},
{
  day: 'Friday',
  minutes: 380,
  tasks: 'client work'
},
{
  day: 'Saturday',
  minutes: 180,
  tasks: 'coding'
},
{
  day: 'Sunday',
  minutes: 40,
  tasks: 'reading, working out'
},
];

```

- Filter out all the days that have a 'client work' task
- Calculate the number of hours worked on these days
- Filter out the days in which you've worked more than 5 hours
- For days that you worked more than 5 hours calculate the total \$ amount earned by multiplying the number of hours worked by your hourly rate which is \$35
- Return the accumulated \$ sum of the hours worked in total

## Answers

1.

```

let words = ['ae', 'baed', 'led', 'ce', 'kaede'];
let words1 = words.filter(function(item) {
  return(item.indexOf('ae') !== -1)
}) .map(function(item) {
  return(item.length > 3)
})

```

```
});  
console.log(words1);
```

```
[false, true, true]
```

2.

```
let clientBilling = tasks.filter(function(item){  
    return(item.tasks.indexOf('client work') != -1);  
}).map(function(item){  
    return(item.minutes / 60);  
}).filter(function(item){  
    return(item > 5);  
}).map(function(item){  
    return(item * 35);  
}).reduce(function(accumalator, item){  
    return(accumalator + item);  
},0);
```

console.log the clientBilling variable:

```
console.log(clientBilling.toFixed(2));
```

```
"501.67"
```

## 5.5 Multi-dimensional arrays

A multi-dimensional array is defined as an array containing other arrays. JavaScript natively does not support multi-dimensional arrays, so in order to create a multi-dimensional array in JavaScript we insert an array inside of another array to make it multi-dimensional. Therefore, the items will be organized as a matrix of rows and columns.

### 5.5.1 Declaring multi-dimensional arrays

Multi-dimensional arrays can be created in JavaScript by nesting one array inside another. Multi-dimensional arrays are defined in the same way that a one dimensional array is defined:

```
let twoDimensionalArray = [];
```

Now let's declare two other arrays:

```
let infoArray1 = ['a', 'b', 'p', 's'];  
let infoArray2 = ['apple', 'banana', 'pineapple', 'strawberry'];
```

In order to make a multi-dimensional array, we will nest these two 1D arrays into another array:

```
twoDimensionalArray = [infoArray1, infoArray2];
```

We had declared an array earlier called `twoDimensionalArray` which was empty. And then we inserted 2 arrays into `twoDimensionalArray`.

Therefore, when we `console.log twoDimensionalArray` we see that `infoArray1` and `infoArray2` are nested inside:

```
[["a", "b", "p", "s"], ["apple", "banana", "pineapple",  
"strawberry"]]
```

We can also define a two dimensional array in the following way:

```
let twoDimensionalArray = [  
    ['a','b','p','s'],  
    ['apple', 'banana', 'pineapple', 'strawberry']  
];
```

Instead of declaring two separate one dimensional arrays, and then inserting them inside the outer array. We can insert a 1D array during declaration as we have done here. When we log `twoDimensionalArray` to the console we are returned with a two arrays nested inside the outer array:

```
console.log(twoDimensionalArray);
```

```
[["a", "b", "p", "s"], ["apple", "banana", "pineapple",  
"strawberry"]]
```

	1	2	3	4
1	'a'	'b'	'p'	's'
2				

The `length` property of a multi-dimensional array can be accessed via the dot notation. This will return the number of rows in the multi-dimensional:

```
twoDimensionalArray.length; //2
```

In order to query the data type use either the ES5 `Array.isArray()` method which will return the boolean `true` or `false`. You can also use the `typeof` operator:

```
Array.isArray(twoDimensionalArray); // true
```

```
typeof(twoDimensionalArray); // "object"
```

The `Array.isArray()` method takes `multiDimensionalArray` as an argument and returns `true` as it is an array. The `typeof` operator will return `object` as arrays are of the object data type.

### 5.5.2 Accessing items in multi-dimensional arrays

In order to access items in a multi-dimensional array, we use the square bracket notation, indicating both the row and column number. Keep in mind that indexing starts at 0. Let's access the first item in the first row from the `multiDimensionalArray` array:

```
let twoDimensionalArray = [
    ['a', 'b', 'p', 's'],
    ['apple', 'banana', 'pineapple', 'strawberry']
];
console.log(twoDimensionalArray[0][0])
```

The first item in row number 0, and column number 0 is the letter '`a`':

```
"a"
```

Let's access the string '`pineapple`' in the array:

```
console.log(twoDimensionalArray[1][2]);
```

The string '`pineapple`' is located at column 2 in row 1, therefore, logged to the console is:

```
"pineapple"
```

### 5.5.3 Iteration in multi-dimensional arrays

In order to iterate through a multi-dimensional array, use a nested for-loop. In the following example we will loop through the `multiDimensionalArray` array and print out all the items in the array on a new line:

```
for(let i = 0; i < twoDimensionalArray.length; i++) {
    let item = twoDimensionalArray[i];
    for(let k = 0; k < item.length; k++) {
        console.log(item[k]);
    }
}
```

Let's see what is happening here:

- The outer for loop will iterate through the nested arrays inside the multi-dimensional array.
- Inside the first for loop we declare a variable called `item` which references an individual nested array.
- The second for loop will iterate through each item in the individual nested array and log the item

## 5.5.4 Multi-dimensional array methods

We can use methods such as `pop()`, `push()`, `indexOf()`, `splice()`, `shift()`, `reverse()`, and `sort()` on multi-dimensional arrays. Let's try a few of these, it's not very much different from one dimensional arrays.

### 5.5.4.1 `pop()`

This will remove the last item in the multi-dimensional array, thereby decreasing the length of the array. Let's remove the last item in the `twoDimensionalArray` array:

```
twoDimensionalArray.pop();
```

This will remove the last nested array in the main outer array. Therefore, now the `twoDimensionalArray` array has only 1 nested array in it:

```
[["a", "b", "p", "s"]]
```

### 5.5.4.2 `push()`

We can also add items to a multi-dimensional array which will insert an array at the last index. For example:

```
twoDimensionalArray.push([1, 2, 3, 4]);
```

Our two dimensional array now contains the following:

```
[["a", "b", "p", "s"], ["apple", "banana", "pineapple", "strawberry"], [1, 2, 3, 4]]
```

### 5.5.4.3 `indexOf()`

The `indexOf()` method will return the index of an argument that is being queried for within an array. We can either iterate through a multi-dimensional array or flatten it and use the `indexOf()` method. Let's flatten the array and find the index of the string `"apple"`:

```
let flattenedArray = twoDimensionalArray.flat();
```

This will return a flattened array with the following items:

```
["a", "b", "p", "s", "apple", "banana", "pineapple", "strawberry"]
```

Now that we have everything in single array, we can use the `indexOf()` method which will return the index of the string `apple` if found. If the queried item is not found, then this method returns -1:

```
let index = flattenedArray.indexOf('apple');
```

The index of the string `'apple'` is referenced by variable `index`. We can log the `index` variable to console to verify its value:

```
console.log(index); // 4
```

### 5.5.4.4 `splice()`

The `splice()` method alters an array by adding or removing an item from the array. This alters the original array. It takes the specified index of the item you want to remove from/add to and how

many items you want to remove as arguments. Additionally you can also pass as an argument the items you want to add to the array.

Let's add another array to our two dimensional array at index 2:

```
twoDimensionalArray.splice(2,1, ['koffee',  
'kafe','kaffeine','kaafe']);
```

We are adding at index 2, 1 nested array ([ 'koffee', 'kafe', 'kaffeine', 'kaafe' ])

Now our two dimensional array contains an extra nested array added at index 2 (row 2) in the twoDimensionalArray:

```
[["a", "b", "p", "s"], ["apple", "banana", "pineapple",  
"strawberry"], ["koffee", "kafe", "kaffeine", "kaafe"]]
```

We can also use the `splice()` method to remove items from the two dimensional array. In the following example, we are going to remove 1 item from the first row (index 0):

```
twoDimensionalArray.splice(0,1);
```

Now the `twoDimensionalArray` array has the following items:

```
[["apple", "banana", "pineapple", "strawberry"], ["koffee", "kafe",  
"kaffeine", "kaafe"]]
```

Therefore, we have removed the first item [ "a", "b", "p", "s" ] which is at index 0 in the two dimensional array.

## Exercises

1. Flatten the following array down to 1 level using the `flat()` method:

```
let nestedArray = [[0,[1, 2]],3, [9], [10, 12]];
```

2. Using the data below, declare a two dimensional array called `weather`, which will contain 4 nested arrays representing the weeks of a month. Each array contains the temperature from Monday to Friday.

```
[35, 30, 32, 31, 33]  
[30, 32, 33, 31, 30]  
[29, 30, 31, 29, 30]  
[29, 28, 30, 29, 30]
```

3. Consider the following two dimensional array called `weather`. It contains 4 nested arrays with 5 values each representing the temperatures from Monday to Friday:

```
let weather = [  
    [35, 30, 32, 31, 33],  
    [30, 32, 34, 31, 30],  
    [30, 30, 31, 29, 30],  
    [29, 30, 31, 29, 30],
```

```
[29, 28, 30, 29, 30]  
];
```

- Filter the number of days that had a temperature equal to or greater than 33
- Return the sum of the filtered values

## Answers

1.

```
let flattenedArray= nestedArray.flat(1);  
  
console.log(flattenedArray);//[0, [1, 2], 3, 9, 10, 12]
```

2.

```
let weather = [  
  
    [35, 30, 32, 31, 33],  
  
    [30, 32, 33, 31, 30],  
  
    [30, 30, 31, 29, 30],  
  
    [29, 30, 31, 29, 30],  
  
    [29, 28, 30, 29, 30]  
];
```

```
[[35, 30, 32, 31, 33], [30, 32, 33, 31, 30], [30, 30, 31, 29,  
30], [29, 30, 31, 29, 30], [29, 28, 30, 29, 30]]
```

3. Flatten the array

```
let weatherFlat = weather.flat();  
  
console.log(weatherFlat);
```

```
[35, 30, 32, 31, 33, 30, 32, 34, 31, 30, 30, 30, 31, 29, 30,  
29, 30, 31, 29, 30, 29, 28, 30, 29, 30]
```

Use the `filter()` method on the flattened array:

```
let weatherHigh = weatherFlat.filter(function(item) {  
  
    return(item >= 33);  
  
});  
  
console.log(weatherHigh);
```

```
[35, 33, 34]
```

Use the `reduce()` method to find the sum of the filtered values:

```
let weatherSum = weatherHigh.reduce(function(acc, value) {  
    return (acc + value);  
}, 0);  
console.log(weatherSum);
```

102

## Summary

Good job! We covered many advanced concepts based on arrays in JavaScript. Starting with destructuring arrays, followed by using the spread operator. We also covered the very popular filter(), reduce() and map() methods, which are essential tools of any serious web developer. We also learnt how to use these methods together by chaining them. And lastly we finished the chapter by discussing multi – dimensional arrays in JavaScript. The practice exercises in this chapter will give you enough confidence to use these methods in your own code and also enable you to answer questions about them confidently.

# Fun functions - 6

*“Example is the best precept.” — Aesop*

Functions are fun! They are central to making your code more organized and functional. What this means is that functions are blocks of code assigned to certain functionality in your applications. So far we have covered objects and arrays which are of the object data type in JavaScript. Functions are considered first-class objects in JavaScript, as functions too can have methods and properties. We will discuss this in detail later on in this chapter. We will learn the basics and newer ES function features such as arrow functions. There are lots of coding exercises in this chapter to aid your learning.

We're going to cover the following main topics:

- Functions as objects
- Function expressions
- Immediately invoked function expressions (IIFE)
- Anonymous functions
- Arrow functions
- Rest syntax
- Callback functions

## 6.1 Functions as objects

In JavaScript functions are a type of object known as **function object**. You can work with function objects as though they were objects. Functions may be assigned to objects, passed in as arguments to other functions, returned from other functions. Additionally a function can be passed into an array. And this gets me to Chapter 1, *The Basics* wherein I stated “*almost everything in JavaScript is an object*”. Let's explore this in further detail.

Take as an example the preceding function:

```
function simple() {  
    return true;  
}
```

Let's query the type of `simple()` using the `typeof` operator:

```
console.log(typeof(simple)); // "function"
```

The `typeof()` operator returns **function**, as functions are a special type of object called **function object**. The **function object** inherits from the **Function.prototype.object** which in turn inherits from the **Object.prototype** object. The built in function object has pre-defined properties such as name, length and methods such as call() and bind(). Let's have a look at a few in the upcoming paragraphs.

We can query the value of the `name` property of a function using the dot notation. For example, using the `simple()` function:

```
function simple() {  
    return true;  
}  
  
console.log(simple.name); // "simple"
```

This lets us know that the function's `name` property has a value of `"simple"`. This correlates to the name of the function.

We can also query the number of arguments passed into a function with the `length` property like so:

```
console.log(simple.length); // 0
```

The function `simple()` has no arguments therefore, the `length` property returns `0`. User-created properties may also be defined on a function however, that is not recommended. Let's add a property to the `simple()` function for fun:

```
simple.category = 'basic';
```

Just as we can add properties to an object, we can do the same to functions. We have added a `category` property to the `simple()` function whose value is the string `basic`. Accessing the `category` property on the function will display its value:

```
console.log(simple.category); // "basic"
```

Existing methods on the function object are `call()`, `bind()` and `apply()`. The discussion of these is out of the scope of this chapter, but it is worthwhile to keep in mind that the `function object` has these pre-defined methods. We can also add user-created methods on the function object. For example, let's add a rather simple method on the `simple()` function which will return to us a random number when called:

```
simple.numGen = function(){  
    let randomNum = Math.random();  
    return randomNum;  
}
```

The `numGen` method of the `simple()` function returns a random number generated using `Math.random()`. Calling it will return a random number:

```
simple.numGen(); //0.8845385071782436
```

Equipped with this background knowledge on JavaScript functions, let's forge ahead and learn how to use functions in the upcoming sections of this chapter.

## Exercises

1. What type of object is a function?
2. What are some properties of the in-built function object?

3. Answer the following about function expression `employees`:

- Access the `name` property of `employees`. What is the value of the `name` property of `employees`?
- Access the `length` property of `employees`. What is the value of the `length` property of `employees`?
- Add a `num` property to the function whose value is equal the parameter `x`
- Add a method to the function expression called `salary`. In the company 'Dev shop foo' everyone is paid \$89,000 per year regardless of their position in the company. Calculate the total cost when there are 79 employees

```
let employees = function information(x) {  
  let company = 'Dev shop foo';  
  employees.num = x;  
  if (x > 100) {  
    console.log(` ${company} is a mid sized company`);  
  } else if (x < 100) {  
    console.log(` ${company} is a small sized company`);  
  } else {  
    console.log(` ${company} is a large sized company`);  
  }  
}
```

4. Why are functions called objects?

#### Answers

1. A function is a special kind of object called a function object
2. `length`, `name`, `caller`
3. function expression `employees`:

The value of the `name` property is:

```
console.log(employees.name); // "information"
```

The value of the `length` property is:

```
console.log(employees.length); // 1
```

`salary()` method which calculates the total salary of all employees

```
employees.salary = function() {  
  let salary = employees.num * 89000;  
  console.log(salary)
```

```
}

employees(79);

employees.salary(); //7031000
```

4. This is because you can work with function objects as though they were objects. Functions may be assigned to objects, passed in as arguments to other functions, returned from other functions

## 6.2 Functions

Functions are a block of code assigned to certain functionality in your program. The functionality afforded by these blocks of code can be used repeatedly without needing to re-write them repeatedly. You might be accustomed to using methods that are in-built functions of the window object such as `alert()` and `confirm()`.

### 6.2.1 Function declaration

Making a function is known as a function declaration. Just as making a variable is known as a variable declaration. To declare a function, use the `function` keyword followed by the name of the function, followed by parenthesis `()`. Any statements of code inside a function are enclosed within curly braces `{ }`.

#### Syntax:

```
function functionName() {
    //statements of code inside the function block
}
```

The above is known as a function statement. The content of the function is compiled but not executed. The function will be executed when we call the function.

### 6.2.2 Function call

A function is called by its name and parenthesis `()`. Let's declare and call a function:

```
function sayHello() {
    console.log('Hello!');
}
```

The `sayHello()` function will log the string `'Hello!'` once the function is called. Call the function by its name to execute it:

```
sayHello();
```

```
"Hello!"
```

### 6.2.3 Function parameters and arguments

Data that is to be used by a function can be passed to it within the parenthesis after the function's name. These are called parameters. Parameters are placeholders used to represent data. One or

more comma-separated parameters can be passed to a function. For example, let's pass in a parameter to the `sayHello()` function which is supposed to represent the name of a person:

```
function sayHello(person) {  
  console.log(`Hello! ${person}`);  
}
```

Now let's call the function as we did earlier:

```
sayHello();
```

This time we get the following message logged to the console:

```
"Hello! undefined"
```

Remember that a parameter is a placeholder for some real value which we will pass to the function when it is called. Since we did not pass in any value to the function call, the default return value of a function is returned which is `undefined`. Now let's pass in a string value to the function:

```
sayHello('Rocko');
```

Now we get the following message logged to the screen:

```
"Hello! Rocko"
```

And this brings us to the next topic, what is an argument? An argument is a real value which is passed either in the function definition or function call. In this case the string '`'Rocko'`' is an argument which is passed to the function call `sayHello()`.

The number of arguments need not be the same as the number of parameters defined in a function. If extra arguments are passed into a function, any extra arguments will be ignored. For example:

```
function sum(x, y, z) {  
  return(x + y + z);  
}
```

Function `sum()` has three parameters `x`, `y` and `z`. This function will return the sum of its parameters. Now let's call the function with an extra argument:

```
sum(100, 80, 90, 100); // 270
```

The return value of the function is `270`, as `100 + 80 + 90` is `270`. The last argument `100` is ignored as it won't be assigned to any parameters.

Now let's see what happens if we pass in one argument less than the number of parameters:

```
sum(100, 80); //NaN
```

Instead of three arguments, we pass in two arguments. The parameters that have no corresponding arguments are set to `undefined`. The result of the sum of `100`, `80` and `undefined` is `NaN`. For example, taking the following code snippet to illustrate that a number added to an undefined value returns `NaN`:

```
let x = 1;
```

```
let y = 2;  
let z;  
console.log(x + y + z); // NaN
```

## 6.2.4 Arguments parameter

The arguments parameter is implicitly passed to a function just like the `this` parameter. It is available inside all functions. It is an array like object that is used to access the arguments passed into a function. Though rarely used, it can be useful to query the number of arguments passed to a function.

Consider the following function:

```
function game(x, y) {  
    console.log(arguments.length);  
}
```

We can query the arguments parameter from within the function with this line of code:

```
console.log(arguments.length);
```

Therefore, when we call the function like so:

```
game('WOW', 'Angry Birds');
```

This will log the number `2` to the console as there are two arguments passed into the function:

```
2
```

## Exercises

1. Declare a function called `oddNum`. The function takes 1 parameter called `x`. The function does nothing as yet.
2. In the body of the function `oddNum()`, do the following:
  - The purpose of the function is to determine whether a passed in argument `x` is an odd number.
  - If a passed in argument is an odd number `console.log` the statement (`x + "is an odd number"`)
  - If a passed in argument is not an odd number `console.log` the statement (`x + "is not an odd number"`)
3. Declare a function called `prime`. The function takes 1 parameter called `x`. The function is empty and does nothing as yet.
4. In the body of the `prime()` function, do the following:
  - Evaluate whether the passed in argument is a prime number or not.
  - The function will `console.log` the string `x + „is a prime number“` if a passed in argument is a prime number

- If the argument is not a prime number, it will console.log the string `x + "is not a prime number"`.
- **Note:** A prime number is a natural number greater than 1 that has no divisors other than 1 and itself.

5. Determine the value of `this` in the following function:

```
function outer() {
  'use strict';

  let x = 10;

  function inner(x) {
    let y = x + 10;
    console.log(this);
  }

  inner();
}

outer();
```

6. Code a function which will calculate the factorial of any number passed to it as an argument
7. Write a function which accepts an array as a parameter. The function should do two things:
  - Check if the parameter passed to it strictly an Array
  - Remove duplicates from the array if any
8. Write a function to swap the values of two variables `a` and `b`
9. What is the difference between a function parameter and argument?
10. Analyze the following block of code, what do you think will be logged to the console?:

```
function outer() {
  let inner = function(x) {
    console.log(x)
  }

  inner();
}

outer();
```

11. How do you think we should rectify the code in question #10 (above) such that the `console.log` statement should log the value of `x`

12. Code a function called `stringVal()` with the following requirements:

- The function takes 3 parameters `a`, `b` and `c`
- The function will log to the console the values of `a`, `b` and `c`
- The function should take in the following string arguments: `('Hello', 'there', 'world', '!')`
- What do you think is logged to the console and why?

13. For the preceding function, write a statement of code that will `console.log` the number of arguments passed to a function:

```
function clients([a, b, c], [x, y]) {  
    // write your code here  
}  
  
clients([1,2,3],['a','b']);
```

14. Have a look at the following function `someMath()` and try and gauge what is returned:

```
function someMath(a,b,c) {  
    return(a * b + c);  
}  
  
someMath(10,13);
```

## Answers

1. function `oddNum(x)`

```
function oddNum(x) {  
  
}
```

2. function `oddNum(x)`

```
function oddNum(x) {  
    if(x % 2 === 0) {  
        console.log(x + ' is not an odd number');  
    } else {  
        console.log(x + ' is an odd number');  
    }  
}  
  
oddNum(3);
```

```
"3 is an odd number"
```

3. **function prime(x)**

```
function prime(x) {  
}  
}
```

4. **function prime(x)**

```
function findPrime(x) {  
    let num = 0;  
    //check if the x is divisible by itself and 1  
    for(i = 1; i <= x; i++) {  
        if(x % i == 0) {  
            //increment value of num  
            num++;  
        }  
    }  
    if(num == 2) {  
        console.log(x + ' is a Prime number');  
    } else {  
        console.log(x + ' is NOT a Prime number');  
    }  
}  
findPrime(5);
```

5. In strict mode, the value of **this** is **undefined**

```
undefined
```

6. **function factorial(num)**

```
function factorial(num) {  
    let result = num;  
    if(num < 0) {  
        return false;  
    } else if(num === 1 || num === 0) {  
        return 1;  
    } else {
```

```

        while (num >= 2) {
            result = result * (num - 1);
            num--;
        }
        return(result);
    }
}

factorial(5);

```

7. function `checkArray(x)`

```

function checkArray(x) {
    if(Array.isArray([x])) {
        return x.filter((a, b) => x.indexOf(a) === b)
    }
}
console.log(checkArray([1,1,1,20,3,3])); // [1, 20, 3]

```

The `indexof()` method finds the first index of an item in an array. The `filter()` method will return an array of all the elements that pass a test.

8. function `swap (num1, num2)`

```

function swap(num1, num2) {
    let temp = num1;
    num1 = num2;
    num2 = temp;
    console.log(num1, num2);
}
swap(10, 30); // 30, 10

```

9. A parameter is passed into the function declaration. It is a placeholder for a real value to be passed into the function later on when the function is called. A function argument is value that is passed into the function call.

10. Arguments that are not passed in will have a value of `undefined`

11. We can pass in an argument to the function call of `inner()`:

```

function outer() {
    let inner = function(x) {
        console.log(x)
    }
}

```

```
inner(10);  
}  
  
outer();
```

This will log to the console the numeric value 10:

```
10
```

12.

```
function stringVal(a,b,c) {  
    console.log(` ${a} ${b} ${c}`);  
}  
  
stringVal('Hello', 'there', 'world', '!');
```

The following is logged to the console:

```
"Hello there world"
```

When the number of arguments are not the same as the number of parameters defined in a function, the extra arguments will be ignored

13.

```
function clients([a, b, c], [x, y]) {  
    console.log(arguments.length);  
}  
  
clients([1,2,3], ['a','b']);
```

The following will be logged to the console, as there are two array parameters passed to the function:

```
2
```

14. The answer is `NaN`. if we pass in one argument less than the number of parameters The parameters that have no corresponding arguments are set to `undefined`. The mathematical operation `10 * 13 + undefined` is equal to `NaN`.

## 6.3 Function expressions

Another way of declaring functions is through function expressions. Assigning a function to a variable is known as a function expression. Function expressions are defined during parse time whereas; function declarations are defined during run-time. Let's have a look at an example:

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

```
let total = sum(10, 100);
console.log(total); //110
```

In this example, we assign a function without a name to the `let sum`. Functions without names are known as anonymous functions. The function takes two parameters `a` and `b`. Within the body of the function is the code statement (`return(a + b);`) which we want should be executed when the function is called. ***The return value of a function expression is the function.***

Further on, we declare a variable called `total` which is assigned the value of the result that we get by calling the anonymous function referenced by variable `sum`. The numbers `10` and `100` are passed in as arguments.

Upon logging the variable `total` to the console, we are returned with the number `110`.

Important to note is that function statements i.e. functions declared with the **function** keyword are hoisted whereas function expressions are not. Therefore if you need to call a function before it is declared use a function statement. As you can only use a function expression after it is defined. To demonstrate this, have a look at the following code snippet:

```
function hello(){
  alert('hello');
}

let hi = function(){
  alert('hi');
};
```

The function `hello` is a function statement, and the function expression is referenced by variable `hi`. When we try and access the `hello()` function before its declaration, the string `'hello'` will be alerted as function statements are hoisted to the top of the scope in which they are defined. This is demonstrated below:

```
Hello(); // "hello"

function hello(){
  alert('hello')
}
```

However, function expressions are not hoisted. Therefore, you cannot use a function expression before defining it. And this is demonstrated in the following block of code:

```
hi();
let hi = function(){
  alert('hi')
};
```

Attempting to access the function expression `hi()` results in a `ReferenceError`:

```
ReferenceError: can't access lexical declaration 'hi' before initialization
```

Therefore, function expressions cannot be used before they are declared. Whereas, function statements are hoisted to the top of their scope and can be used before they are declared.

### Exercises

1. A function expression cannot be used unless it is defined. True/False
2. Function expressions are hoisted. True/False
3. Write a function expression such that:
  - The function is referenced by a variable declared as `multiply`
  - The function takes 3 parameters `x`, `y` and `z`
  - The function will return the value of `x`, `y` and `z` multiplied with each other
  - Call the function expression with the following arguments: `2, 20, 10`
4. Have a look at the following code block and then answer the questions:

```
let multiply = function(x, y, z) {  
    return(x * y * z);  
}  
  
multiply(2,20,10);  
  
  
function collection() {  
    let fruit ={  
        a: 'apple',  
        b: 'banana',  
        c: 'cantaloupe',  
    }  
    console.log(fruit);  
}
```

Calling the function `collection()` before its declaration. What is logged and why?

```
collection();  
  
let multiply = function(x, y, z) {  
    return(x * y * z);  
}  
  
multiply(2,20,10);
```

```

function collection() {
  let fruit ={
    a: 'apple',
    b: 'banana',
    c: 'cantaloupe',
  }
  console.log(fruit);
}

```

Calling the function `multiply()` before it's declaration. What is logged and why?

```

multiply();

let multiply = function(x, y, z) {
  return(x * y * z);
}

multiply(2,20,10);

function collection() {
  let fruit ={
    a: 'apple',
    b: 'banana',
    c: 'cantaloupe',
  }
  console.log(fruit);
}

```

5. In this exercise we will make our own timer using the `setInterval()` method and Date constructor:

- Declare a function expression called `timer`
- The function should log the current local time to the console.
- ***Hint:*** use the `new Date()` constructor and from the date, then get the time with the `toLocaleTimeString()` method
- Pass in the `timer` function expression to the `setInterval()` method as an argument
- This should log to the console the time at intervals of 1 second

## Answers

1. True.
2. False
- 3.

```
let multiply = function(x, y, z) {  
    return(x * y * z);  
}  
  
multiply(2,20,10); //400
```

4. Calling the function `collection()` before its declaration will log to the console the following, as function declarations are hoisted:

```
Object {  
    a: "apple",  
    b: "banana",  
    c: "cantaloupe"  
}
```

Calling the function `multiply()` before its declaration will log the following `ReferenceError` to the console, as function expressions are not hoisted:

```
ReferenceError: can't access lexical declaration 'multiply'  
before initialization
```

- 5.

```
let timer = function(){  
    let date = new Date();  
    let timeNow = date.toLocaleTimeString();  
    console.log(timeNow)  
}  
  
setInterval(timer, 1000);
```

## 6.4 Immediately invoked function expressions (IIFE)

Usually a function is executed after it is called. However you can also execute functions immediately after being defined with immediately invoked function expressions. These do not require a function call.

### Syntax:

```
(function() {  
    //function logic
```

```
    console.log('run immediately');
})();
```

*Looking carefully at the syntax it is noticeable that the function statement is enclosed within () parenthesis and then immediately called by appending a pair of parenthesis () at the end.*

The first enclosing parenthesis `()` make the function an expression. The last set of parenthesis `()` will immediately signal the execution of the function. Code statements within an IIFE are within their own lexical scope. Any variables enclosed within an IIFE cannot be accessed outside of it.

Thus from the example above, we see that there are two steps in to creating an IIFE:

1. Creating a function expression
2. Invoking the function expression immediately with parenthesis `()`

Upon loading the page, the function is invoked immediately and the following will be logged to the console:

```
"run immediately"
```

The return value of an IFFE is an executable function. You can also use a named IIFE, for example:

```
(function IIFE() {
    console.log('run immediately');
})();
```

Named and unnamed immediately invoked function expressions do not leak into the global space and cannot be called again. Additionally, an IIFE can take parameters as well, for example:

```
(function IIFE(a, b) {
    console.log(a + b);
})(100, 1500);
```

This will log to the console, `1600` immediately:

```
1600
```

A lot of JavaScript libraries will use immediately invoked function expressions so as to avoid naming conflicts within the global namespace between the library and programs that use the library. For example analyze the preceding code block:

```
let num1 = 10;

(function() {
    let num1 = 100;
    console.log(num1);
})();
console.log(num1);
```

Within the IIFE, the `num1` variable will be logged to the console immediately when the IIFE is executed. Whereas, outside of the IIFE, the value of `num1` in the global namespace will be logged to

the console. This is because JavaScript has function-level scope, the variables that are in the function (IIFE) are local variables and cannot be accessed outside of the function. Outside of the function, the value of `num1` which is `10` will be logged to the console:

```
100  
10
```

We can also achieve the same result by using the `let` keyword. If you remember from Chapter 1, `let` is block scoped. Therefore, the code block can be refactored to:

```
let num1 = 10;  
  
{  
  
    let num1 = 100;  
  
    console.log(num1);  
  
}  
  
console.log(num1);
```

This will achieve the same result. The `num1` variable declared within the curly {} set of braces has block scope and therefore the value of `num1` within that block is `100`. Whereas, the variable `num1` in global scope has a value of `10`.

```
100  
10
```

The advantage of using an IIFE is that the function executes immediately, therefore if that is what is needed in a web application, the IIFE construct is available. Also, any variables declared inside an IIFE cannot be accessed outside. This prevents polluting the global namespace and avoiding name conflicts. Immediately invoked functions also helps free up memory, this is because as soon as an IIFE is invoked, the variables and IIFE itself are available for garbage collection, therefore freeing memory.

## Exercises

1. What is an IIFE and what is a use case for using IIFEs?
2. Code an IIFE that will `console.log` the string '`'hello world'`
3. Why are IIFE's used?
4. What is the context of `this` in the following function:

```
(function () {  
  
    console.log(this);  
  
})();
```

5. Have a look at the following code and do the following tasks:

- Convert the function `logPerson` to an IIFE
- What will be logged to the console first?

```
let person = 'Rena'

function logPerson() {
    let person = 'Robbie'
    console.log(person);
}

logPerson();
console.log(person);
```

The following is logged to the console currently:

```
"Rena"
"Robbie"
```

6. Have a look at the following block of code. What will be logged to the console inside the IIFE and outside the IIFE?

```
let x = 10;

(function(x) {
    x++;
    console.log(x);
}) (x);

console.log(x);
```

## Answers

1. An IIFE is an immediately invoked function expression. It is executed immediately. Use cases of IIFEs are:
  - Loading a user's preferences, shopping cart when they return to website/web application by using an IIFE
  - Showing the user a message as soon as they visit a website

2.

```
(function() {
    console.log('hello world');
}) ();
```

3. To avoid polluting the global namespace, as all the variables used inside the IIF are not visible outside its scope.
4. The context of **this** will be the global window object.
- 5.

```
let person = 'Rena';
```

```
(function() {  
    let person = 'Robbie'  
    console.log(person);  
})();  
console.log(person);
```

The following will be logged to the console in this order:

```
"Robbie"  
"Rena"
```

6. The number 11 is logged to the console inside the IIFE and the number 10 is logged to the console outside of the IIFE. Any variables declared or changed inside an IIFE cannot be accessed outside. Hence, there are two different values logged to the console for x.

```
11  
10
```

## 6.5 Anonymous functions

Anonymous functions are functions that do not have names and are dynamically declared at runtime. These functions are useful in creating a temporary/private scope. We have seen examples of anonymous functions in both function expressions and immediately invoked function expressions (IIFE). For example:

```
(function() {  
    console.log('This is an immediately invoked function');  
})();
```

The above code demonstrates an IIFE, that is executed immediately as indicated by the presence of the last pair of parenthesis (). The function has no name, therefore it is an anonymous function. Anonymous functions are used commonly in libraries, frameworks, and in methods of the window object such as **setTimeout()**:

```
setTimeout(function () {  
    console.log('Execute later after 1 second')  
, 1000);
```

This method takes an anonymous function as an argument. The body of the function will log to the console the statement 'Execute later after 1 second' at an interval of a 1 second (1000 milliseconds).

Arrow functions, typically have no name therefore, they are anonymous functions as well. For example:

```
let msg = () => console.log('This is an anonymous function');
```

If you quite don't understand the syntax of the arrow function, don't worry as it is covered in the upcoming section.

## Exercises

1. What is an anonymous function and can you think of any popular/common examples where anonymous functions are used in JavaScript?
2. Declare a function expression referenced by a variable called `fooScore` and do the following:
  - The function takes one parameter `x`
  - Inside the function raise `x` to the power of `5`
  - Invoke the function and pass in the number `2` as an argument

Hint: use `Math.pow()` to raise `x` to the power of `5`
3. Let's code our own count down timer. Use the `setInterval()` method to countdown from 10 to 0 at intervals of 2 seconds.
4. Use the `window.onload()` method to execute a function which will alert the string '`Hello User`' when a page loads

## Answers

1. A function that does not have a name is an anonymous function. IIFEs, `setTimeout()` functions, and a lot of DOM event handlers use anonymous callback functions.
- 2.

```
let fooScore = function(x) {  
    return(Math.pow(x, 5));  
};  
fooScore(2); //32
```

- 3.

```
let count = 10  
  
setInterval(function() {  
    if(count >= 0){  
        console.log(count);  
    }  
    count--  
  
, 2000)
```

4.

```
window.onload = function() {
    alert('Hello User!')
}
```

## 6.6 Arrow functions

Arrow functions also called fat arrow functions are part of the ES6 specification. They allow you to write function expressions with a shorter and more concise syntax. It can take some time to wrap your head around these, as the syntax can be a bit unusual at first. However, with time and practice, the syntax gets familiar.

While many people prefer using arrow functions, it is my experience that a beginner should not necessarily get hung up on replacing every function declaration or expression with an arrow function.

### 6.6.1 Syntax

Let's first have a look at the different ways to write function expressions before we start using them.

**Syntax:** `let functionName = (arg1, arg2...argN) => statement`

Arguments are within the parenthesis `()` and, following an arrow (`=>`) symbol is the code statement that a function will execute.

#### 6.6.1.1 No parameters

When there are no parameters, the function statement will look like this:

```
let greeting = () => ('Hello World!');
```

There will be nothing inside the parenthesis, and therefore, if we log the `greeting` expression, we should see its value:

```
console.log(greeting()); // "Hello World!"
```

#### 6.6.1.2 Single parameter

If you only have 1 parameter, then the parenthesis `()` around the parameter can be omitted. This is a personal style choice. Let's take a look at an example:

```
let singleVal = a => a + 1;
```

The `singleVal()` function takes a single parameter `a`, increments it by 1 and returns it. We can call the `singleVal()` function expression and query its value by logging it to the console:

```
console.log(singleVal(1));
```

### 6.6.1.3 Multiple arguments

Take as an example the following function:

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

This can be re-written using the shorter arrow function syntax:

```
let multiply = (a, b) => {return a * b}
```

The **function** keyword is deleted. Instead we use a function expression. We pass in parameters **a** and **b** within the parenthesis **()**. Then we have inserted a fat arrow (**=>**) to the right of the function parameters. The function will return **a** multiplied by **b**. We can call the function expression like so:

```
multiply(2, 3); // 6
```

### 6.6.1.4 Return values

Arrow functions implicitly return values and there is no need to use the **return** keyword. Let's have a look at an example to demonstrate this:

```
let nums = (x, y, z) => x + y + z
```

**nums** is a function expression using fat arrows. The function takes **x**, **y** and **z** as parameters. Notice that there is no **return** keyword and no enclosing curly braces **{ }{ }**. The absence of curly braces indicates the return of a value.

When using the **return** keyword or multiple statements in an arrow function, the curly braces are compulsory. For example:

```
let sumMultiply = (x, y) => {  
    let z = 5;  
    return (x * y) + z;  
};
```

We can then call the **sumMultiply** function expression:

```
sumMultiply(2,3); // 11
```

Arrow functions are concise and they implicitly return values. However, there are use cases when you should not be using arrow functions. We will discuss these now.

## 6.6.2 **this** binding

One of the major differences between a regular function and arrow function is how the **this** keyword will behave. Remember in Chapter 2, *The Basics -2* we covered the **this** keyword in detail. In summary, how **this** behaves is determined by how a function is called. The **this** keyword is either implicitly or explicitly bound.

An arrow function does not have its own binding of the **this** keyword. Instead the **this** keyword is lexically bound. Which means that arrow functions use **this** from their own enclosing scope.

Let's have a look at an example to demonstrate this.

**HTML:**

```
<button id="btn" type="button">Click me!</button>
```

**JavaScript:**

```
// event listener with a standard callback function
document.getElementById("btn").addEventListener('click', function()
{
    this.innerText = 'Clicked'
});
```

The above code attaches an event listener to a button element. The **this** keyword references the object (HTML element) on which the event will occur. This in our case is the button element.

Now let's factor our code to use an arrow function:

```
document.getElementById('button1').addEventListener('click', () =>
this.innerText = "Clicked!" ; )
```

In the arrow function, the **this** keyword is inherited from its parent scope which in this case is the global window object. Re-stated, the arrow function will use the **this** keyword from the code that contains the arrow function, which is the window object.

Therefore, the **this** keyword is lexically bound in arrow functions. Next let's wrap up this topic by briefly discussing when not to use arrow functions.

### 6.6.3 When not to use arrow functions

Arrow functions should be avoided when writing constructor functions, and any prototype extensions. Instead rely on the ES6 class syntax to write object oriented code. Also keep in mind that arrow functions can be confusing as they usually are anonymous (have no name). Therefore, making code less readable. Additionally the lack of function names can also make debugging harder. It is better to use arrow functions along with array methods such as `filter()`, `map()` as these methods are self-explanatory and indicate the purpose of the callback function. Let's put this to practice by practicing some coding exercises based on arrow functions.

#### Exercises

1. Convert the following function to an arrow function

```
function employee(hours) {
    return hours;
}
employee(10);
```

2. Convert the following function to an arrow function:

```

function employee(hours) {
  hours *= 1.5;
  return hours;
}

```

3. When should you use arrow functions?
4. Fix this function so that the the `setInterval()` method logs numbers from 0 onwards:

```

function timer() {
  seconds = 0;
  setInterval(()=>{
    console.log(this.seconds++);
  }, 1000)
}
timer(); NaN

```

5. Analyze the following code, what do you think will be returned when the method `productDetails()` is called?

```

let headset = {
  model: 'Oculus Quest',
  company: 'Facebook',
  productDetails: () => {
    return `${this.company}${this.model}`;
  }
};

console.log(headset.productDetails());

```

## Answers

1.

```

let employee = (hours) => hours;
employee(10); // 10

```

2.

```

let employee = (hours) => {
  hours *= 1.5;
}

```

```
    return hours;  
}  
  
employee(190);
```

3. Arrow functions should be used when you're aiming for shorter concise syntax. Arrow functions use, **this** from their own enclosing scope. Therefore, **this** context within the arrowfunctions is lexically or statically scoped .This provides a narrower and safer scope.

4.

```
function timer() {  
  seconds = 0;  
  setInterval(()=>{  
    console.log(this.seconds++);  
  }, 1000)  
}  
  
timer();
```

Remember that an arrow function does not have its own binding of **this**. Instead **this** is lexically bound. Which means arrow functions use **this** from their own enclosing scope.

5. **undefined**

## 6.7 Rest syntax

Before we delve into what the rest parameter does and why it's useful, remember that earlier on we discussed what happens when arguments are in access of parameters. For example:

```
function sum(x, y, z) {  
  return(x + y + z);  
}
```

Function **sum()** has three parameters **x**, **y** and **z**. This function will return the sum of its parameters. Now let's call the function with an extra argument:

```
sum(100, 80, 90, 100); // 270
```

The return value of the function is **270**, as **100 + 80 + 90** is **270**. The last argument **100** is ignored as it won't be assigned to any parameters.

The rest parameter syntax introduced in ES6 is used to pass in an indefinite number of parameters to a function. These parameters are represented as an array, therefore you can use array methods on them.

### Syntax:

```
function numbers(...parameters) {
```

```
//statements  
}
```

The three dots ... are followed by the name of an array used to represent the values. To demonstrate that the parameters are represented as an array, have a look the following code snippet.

```
function example(a,b,c,...d) {  
    console.log(a);  
    console.log(b);  
    console.log(c);  
    console.log(d);  
}
```

In this example, we will pass in 3 parameters (a, b, and c) and then using the rest syntax pass in the fourth parameter d. Keep in mind that the rest parameter must be included at the very end. The function will log to the console, the value of each argument passed into the function. So let's go ahead and try this:

```
example(1,2,3,4);
```

The following is logged to the console:

```
1  
2  
3  
[ 4 ]
```

Therefore, this demonstrates that all the first three arguments are logged as numbers whereas the last argument is an array.

If we do not pass in anything as the fourth parameter, an empty array is returned. Let's call the function example() and this time only pass in three arguments:

```
example(1,2,3);
```

The following is logged to the console:

```
1  
2  
3  
[ ]
```

An empty array is returned as the fourth argument was omitted in the function call.

We can also use the arguments parameter from section 6.2.4, however it does not support array methods such as forEach(). It is an array like object that is iterable, not quite an array. As seen in the preceding example.

Take for an instance a function that will log the sum of all its parameters. The number of parameters is unknown:

```
function sumNums(...params) {  
  let sum = 0;  
  params.forEach(function(item) {  
    sum += item;  
  });  
  console.log(sum);  
}
```

Let's deconstruct what is happening here:

- The rest parameter is used to represent an unknown indefinite number of parameters passed to the function `sumNums`
- Within the function variable `sum` is declared with the `let` keyword
- We use the array `forEach()` method that executes a callback function for each parameter passed into the function
- The callback function will add the parameter values and assign them to the `sum` variable during each iteration using the `+=` addition assignment operator
- Lastly, we will log to the console the value of the `sum` variable

Let's try this out:

```
sumNums(1, 2, 3, 4, 5); // 15
```

Therefore, you can use array methods along with the rest syntax, which is not possible using the `arguments` object.

### Note

When three dots (...) are used with a function it is the rest syntax. Whereas when they occur in an array it called the spread syntax. The rest syntax is used to gather an indefinite amount of function parameters into an array. Whereas, the spread operator will allow an expression to be expanded in places where multiple arguments are expected in an array.

The rest syntax is extremely useful when the number of arguments to be passed into a function is unknown. Next let's go over callback functions and why and how they're used in JavaScript.

### Exercises

1. What is the purpose of the rest parameter?
2. Have a look at the following function with 9 parameters. Re-factor the function using rest syntax:

```
function adder(a,b,c,d,e,f,g,h,i){  
  console.log(arguments.length)  
  for(let i = 0; i < arguments.length; i++) {
```

```

        console.log(arguments[i] + arguments[i])
    }
}

adder(1, 2, 3, 4, 5, 6, 7, 8,9);

```

The following is logged to the console, thereby indicating that each argument is doubled.

```

2
4
6
8
10
12
14
16
18

```

**Note:** You can use array methods as the parameters are converted into an array when using the rest syntax

3. Have a look at the following function with 9 parameters. Re-factor the function using rest syntax. The function sums up all arguments passed to it:

```

function sum(a, b, c, d, e, f, g, h, i) {
    console.log(a + b + c + d + e + f + g + h + i);
}

sum(1, 2, 3, 4, 5, 6, 7, 8, 9);

```

The function will log the following sum:

```
45
```

4. Analyze the following function what is returned? If an error is thrown, how will you fix it?

```

function workshopAttendees(a,...b, c) {
    return true;
}

```

5. Can you think of any differences between a rest parameter and the arguments object which is inherently present in all functions?
6. What is the difference between rest and spread parameters?
7. Write a function that will multiply any amount of arguments passed to it by 8

## Answers

1. The rest parameter introduced in ES6 is used to pass in an indefinite number of parameters to a function. These parameters are represented as an array; therefore you can use array methods on them.

2.

```
function adder(...a) {  
  return a.forEach(function(item) {  
    console.log(item + item);  
  })  
}  
  
adder(1, 2, 3, 4, 5, 6, 7, 8, 9);
```

3.

```
function sum(...params) {  
  return params.reduce((previousIndex, currentIndex) => {  
    return previousIndex + currentIndex;  
  });  
}  
  
console.log(sum(1, 2, 3, 4, 5, 6, 7, 8, 9)); // 45
```

4. An error is thrown as only last parameter is allowed to be defined with the rest syntax:

```
SyntaxError: parameter after rest parameter
```

In order to fix it, change the rest parameter to be the last parameter passed to the function:

```
function workshopAttendees(a, c, ...b) {  
  return true;  
}  
  
//we can then call the function:  
workshopAttendees(1, 2, 3, 4, 5, 6, 7);
```

5. There are a couple of differences between the two:

- The arguments object is array like but not an array. Whereas the rest parameters are array instances which is why you can use array methods such as sort(), map(), filter() etc. on them

- The arguments object contains all the arguments passed to an object. Whereas rest parameters are a reference to arguments to be passed later on in the function and are not formally defined in a function

## 6. Example of the rest parameter:

```
function rest(x,y,...z) {
    console.log(x); // 1
    console.log(y); // 2
    console.log(z); // [3,4,5]
}

rest(1,2,3,4,5);
```

Example of the spread operator:

```
let array1 = ['a', 'b', 'c'];
let array2 = ['d', 'e', 'f'];
let combinedArray = [...array1, ...array2];
console.log(combinedArray); // ["a", "b", "c", "d", "e", "f"]
```

The most obvious difference is that the rest operator is used with functions, and the spread operator is used with arrays. The rest operator collects all the remaining elements into an array. Whereas the spread operator allows arrays/objects/strings to be expanded into single arguments/elements.

## 7.

```
function multiplier(...a){
    a.forEach(function( item){
        return(item * 8);
    });
}
multiplier(1, 8, 2, 10, 9); // 8, 64, 16, 80, 72
```

## 6.8 Callback functions

### 6.8.1 Introduction

JavaScript is considered to be synchronous i.e. single-threaded. This means one line of code is executed at a time. It cannot multi-task and move on to the next line or block of code until the current line or block has been executed. Therefore, if you have a function that has to perform some

complex computations, it may well be possible that during that time other parts of the application freeze. To make our code behave asynchronously, we can use callback functions.

Before we continue, let's understand functions are objects this means that they can be used in an object like manner:

- Functions can be stored in variables
- You can pass in a function as an argument to another function
- You can return a function, from a function
- One function can be created inside another

These are aspects of what is known as asynchronous programming, of which callback functions are a part.

A function that is passed as an argument to another function is called a **callback** function. The function that accepts another function as an argument is called a **higher-order** function and contains the logic for when the callback function will be executed.

MDN describes **callback** functions as:

*A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.*

### 6.8.2 Why?

Callback functions are efficient as a function doesn't have to wait for something to finish or return a value to complete a routine or task. Callback functions are often used in DOM events and the array methods that we saw in the previous chapter (`filter()`, `map()`, etc.). Callback functions can be asynchronous or synchronous.

In a normal function, some value is returned and stored in a variable to presumably perform some other task. However, with callback functions the callback function is passed as an argument to the main function and, the value returned from the main function is provided to the callback function.

It is also a way of making sure that an event for example a `click` event has taken place before something else runs. Therefore, the callback function is declared but won't run till the `click` event occurs.

Let's have a look at a simple callback function in the preceding example:

```
//mainFunc accepts a callback function as a parameter

function mainFunc(item1, item2, callback) {
    console.log(`I like eating ${item1} and ${item2}`);
    //callback function is called after the above lines have finished
    // executing
    callback();
}
```

```
}

mainFunc('chocolate', 'cheese', function() {
  console.log('Do you have some?');
});
```

Let's examine what's happening:

- When we pass a function as an argument to another function, we are treating it like a variable
- Only the function definition is passed. The function is not executed in the parameter
- The callback function is executed when the `mainFun()` is called
- The code/functions are still executed in the order given. The only change is that the JavaScript engine doesn't wait for a response before moving on.

#### Note

When a callback is passed into another function, only the reference to the function is passed. You call it without executing it, thus the absence of the parenthesis () .

Let's have a look at three different ways of making a callback function:

1. Named callback function
2. Anonymous callback function
3. Named variable

#### 6.8.3 Named callback function

Callback functions can be declared and then passed as arguments to other functions which will call them. Take the preceding block of code as an example:

```
function mainFunction(callback) {
  callback();
}

function namedCallbackFunction() {
  console.log('This is from within the named function passed as a
callback');
}

mainFunction(namedCallbackFunction);
```

Here is a breakdown of the code:

- We have a function called `mainFunction()` that takes a callback as a parameter. The callback function is called from within `mainFunction()`.

- The callback function is conveniently declared as `namedCallbackFunction()` it will log a string 'This is from within the named function passed as a callback' to the console.
- The `mainFunction()` function is then called. We pass in the named callback function which we had declared.

The following statement will be logged to the console:

```
"This is from within the named function passed as a callback"
```

#### 6.8.4 Anonymous callback function

Anonymous callback functions have no names and we often observe them in DOM methods and methods of the `Array.prototype` object such as `filter()`, `reduce()`, and others.

Take as an example the following `addEventListener()` method:

```
<button id = 'myButton'>click</button>
```

Let's suppose we have a button in our HTML document with an ID equal to `myButton`

```
document.getElementById('myButton').addEventListener('click',function()
{
  console.log('Button clicked');
});
```

In this example, the `addEventListener()` method will listen for a click event. After a user clicks on the button, the anonymous callback function will be triggered and the statement '`Button clicked`' is logged to the console.

#### 6.8.5 Callback function expression

We discussed function expressions earlier in this chapter, now we will briefly go over callback function expressions. To re-iterate assigning a function to a variable is known as a function expression. In the following `setTimeout()` function the callback function passed in is a reference to a function expression:

```
let callbackFunction = function() {
  console.log('Complete!');
};

setTimeout(callbackFunction, 3000);
```

Below is an explanation of the code snippet:

- The variable `callbackFunction` is a function expression. The function logs a single statement to the console.
- We pass in the `callbackFunction` variable to the `setTimeout()` method which will execute it after three seconds. Therefore logging to the console the statement '`Complete!`'

Let's take another example for a spin. In this example we have declared a function called `addScore()` which takes three parameters: `x`, `y` and a `callback` function. And our callback function is assigned to the `message` variable:

```
function addScore(x, y, callback) {  
  let sum = x + y;  
  if (sum > 100) {  
    callback();  
  } else {  
    return false;  
  }  
}  
  
// callback function expression  
let message = function() {  
  console.log('You can go to the next level now');  
};  
addScore(1, 100, message);
```

Let's deconstruct what is happening in this example with the function expression:

- Function `addScore()` takes three parameters `x`, `y` and a callback function
- A variable called `sum` is assigned to the sum of `x` and `y`
- An `if-else` conditional statement is used to check whether the value of the `sum` variable is greater than `100`
- If the conditional statement is true then the callback function `message()` is called
- Else return `false`
- `message` is a function expression which when called will log the statement '`You can go to the next level now`'
- We then call the function `addScore()` and pass in the number `1` (`x`), the number `100` (`y`) and `message` (callback function) as arguments.

In this section we looked at the three different ways of declaring callback functions. In the next two sections we will go over the two broad categories of callback function: synchronous and asynchronous callback functions.

### 6.8.5 Synchronous callback functions

In synchronous callback functions, code is executed line by line from top to bottom. Let's have a look at an example:

```
function mainFunction(callback, param) {  
  callback(param);  
}
```

```

function details(developer) {
  if(developer) {
    console.log(`Hi, my name is ${developer.name}`);
  }
}

mainFunction(details, {
  'name': 'Kauress'
});

```

In this example a few points to note are:

- `mainFunction()` takes two parameters, a callback function and a parameter. The callback function is called from within this function, taking the parameter as an argument
- The function `details()` is the callback function declaration that is passed to `mainFunction()` as an argument.
- Also passed to the `mainFunction()` as an argument is an object with a `name` property whose value is `Kauress`

The following will be logged to the console:

```
"Hi, my name is Kauress"
```

Some common implementations of synchronous callback functions are methods of `Array.Prototype` object. For example the `forEach()`, `filter()`, `map()` and `sort()` methods. The following code example uses the `sort()` method:

```

let numbers = [10, 100, 9, 3];
const sortNumbers = numbers.sort(function(a, b) {
  return a - b
});
console.log(sortNumbers);

```

In this example, the `sort` method will finish first, executing a function for each element of the array. The `sort()` method provides a partial implementation that is swapped in a larger implementation. Callbacks in array methods such as as `sort()` are an implementation of a design pattern. Lastly the `console.log` executes to display the following:

```
[3, 9, 10, 100]
```

Therefore synchronous callback functions are executed in a top down manner. In the next section we will go over asynchronous callback functions.

### 6.8.6 Asynchronous callback functions

Callback functions allow JavaScript to behave asynchronously. This means that multiple things can occur at the same time. Compared to a synchronous model wherein things happen one at a time. This is easier to understand with an example, so have a look at the following example:

```
setTimeout(callbackFunc, 3000);
```

The `setTimeout()` function takes a callback function called `callbackFunc` as a parameter which it will execute 3 seconds later.

```
function callbackFunc() {  
    console.log('hello')  
}
```

We also have another statement which will be logged to the console:

```
console.log('execute this first');
```

Therefore, the full code snipped looks like this:

```
setTimeout(callbackFunc, 3000);  
  
function callbackFunc() {  
    console.log('hello')  
}  
  
console.log('execute this first');
```

The `setTimeout()` function is asynchronous and it will execute the rest of the code while waiting. In the meantime the `console.log` statement will be executed. Three seconds later, the callback function passed into the `setTimeout()` function is executed. Therefore, the order of the statements logged to the console is:

```
"execute this first"  
"hello"
```

Many web APIs are asynchronous in nature which means code can execute regardless of the main program flow. For example an event handler which registers a click event:

```
<button id = 'button'> click </button>
```

Here we have an HTML button element with an ID equal to '`'button'`'. We will listen for a click event attached to this button by using the `addEventListener()` method:

```
document.getElementById('button').addEventListener('click',  
function() {  
  
    console.log('Button clicked');  
});
```

The `addEventListener()` method accepts a callback function which will execute only when the click event is triggered. Therefore, the callback function is not executed immediately. It only runs in response to a click event which will occur whenever a user clicks on the button.

Asynchronous callback functions are more common than synchronous callback functions therefore, you're likely to see them more. Callback functions can often be confusing, so I suggest you practice the questions provided in this section to solidify your understanding of callback functions.

## Exercises

1. What is a callback function?
2. Finish the following code snippet. The declarations for `onSuccess` and `onFailure` callback functions are missing. You may add anything inside the body of the callback functions

```
function respond(onSuccess, onFailure) {  
  let error = true;  
  
  if (error) {  
    onFailure(error)  
  } else {  
    onSuccess()  
  }  
}  
  
respond(onSuccess, onFailure);
```

3. Have a look at the following code and explain what the order of the `console.log` statements will be:

```
console.log(1);  
console.log(2);  
setTimeout(() => {  
  console.log(3);  
}, 0);  
console.log(4);
```

4. Can you think of a callback function used in the DOM?
5. Callback functions can be confusing, nested callbacks are often called callback hell. Can you explain what is happening here?

```
function a(x) {  
  b(c);  
}  
  
function b(y) {  
  c();  
}
```

```

}

function c() {
    console.log('hi')
}

a(b);

```

6. Code a function called `input` which takes a parameter called `text` and a callback function as a second parameter. When the function `input()` is called it should return a string argument passed to it reversed. For example the string '`'hello world!'`' is returned as '`'!dlrow olleh'`'

### Answers

1. A function that is passed as an argument to another function is called a **callback** function. The function that accepts another function as an argument is called a **higher-order** function and contains the logic for when the callback function will be executed. This outer function undertakes the asynchronous task

2.

```

function respond(onSuccess, onFailure) {

    let error = true;

    if (error) {
        onFailure(error)
    } else {
        onSuccess()
    }
}

function onFailure(err) {
    console.log(err)
}

function onSuccess(err) {
    console.log('Success!')
}

respond(onSuccess, onFailure);

```

3. The following is the order of the statements logged to the console:

```

1
2
4
3

```

Notice that the `setTimeout()` function has a `0` second delay. However, the `setTimeout()` function does not execute immediately. This is because callback functions are treated as second class citizens. They are executed after regular functions (first class citizens) finish executing first.

4. The `addEventListener()` method is an example of a callback function that is called in response to a click event:

```
document.getElementById('button').addEventListener('click',
function() {
    console.log('Button clicked');
});
```

5. Function `a()` takes a parameter `x` and function `b()` is called from inside this function. Similarly function `b()` takes a parameter called `y` and function `c()` is called from inside function `b()`. Lastly, function `a(b)` is called passing in function `b` as an argument.

- 6.

```
function input(text, reverse) {
    reverse(text);
}

function reverse(string) {
    let reverse = string.split('').reverse().join("");
    //console.log(reverse);
    return reverse;
}

input('hello world!', reverse);
```

## Summary

In this chapter we covered key topics based on functions starting with the premise that functions are objects. We then learnt about the different ways of declaring functions as functions expressions, immediately invoked functions, anonymous functions and arrow functions. We also learnt about the rest parameter passed to functions which is a new ES6 feature. And lastly we covered callback functions which are an often dreaded topic.

In order to solidify your knowledge it is highly recommended that you practice all the exercises provided in this chapter. It will help you put the theory to practice and in addition cement the concepts in your memory better.

# DOM POWER-7

*"Your calling should be a pointer for your skill development" — S. Adeleja*

DOM stands for **Document Object Model**. It is extremely important when it comes to accessing HTML elements and manipulating them via JavaScript. This is a relatively shorter chapter; however don't let that fool you, as there are lots of practice exercises. Knowledge and practical experience of DOM manipulation are important for frontend web developers. As it will allow you to make more dynamic frontend web applications for your portfolio. Additionally, it'll also allow you to learn and appreciate frameworks such as React.js and Vue.js which make state management much easier.

We're going to cover the following main topics:

- What is the DOM?
- Properties of the DOM
- DOM methods
- Iterating through DOM nodes

## 7.1 What is the DOM?

The DOM can often be a source of confusion for beginners. This is because beginners know the basics of JavaScript but they cannot quite fill in the dots on how to use JavaScript to build functionality into their websites or web applications using the DOM API. To understand the DOM let's first understand its purpose. Put across simply the DOM allows you to access and manipulate HTML elements on webpages. You can add, delete, and change HTML elements via the DOM API using JavaScript. Therefore, knowing what DOM methods and properties to use will allow you to build functionalities such as adding items to a shopping cart, toggling signup/login buttons, etc. All these dynamic functionalities are part of what is known as state management. State management refers to the state of the components (elements) on a webpage. For example, a signup button can have a visible state when the user is not logged in. The same signup button can have an invisible state when the user logs in and now is hidden. Now that we know what the purpose of the DOM, this brings us to the next question what is the DOM?

The DOM broadly has two purposes:

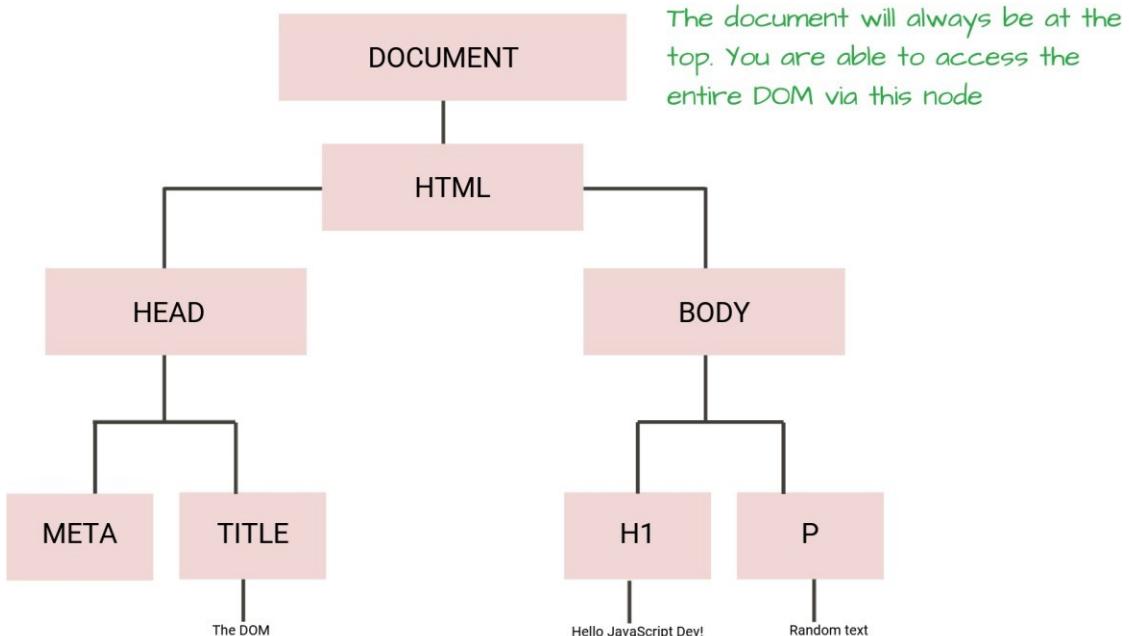
1. It defines the logical structure of an HTML document for example index.html
2. Dictates how the elements within an HTML document can be accessed and modified

The Document Object Model is a web API that allows you to access and manipulate a webpage's content, structure, and styles. It is an object-based representation of a source document's HTML elements, created by the browser when it parses a webpage. All the elements in an HTML document are represented in a family tree-like manner called a "node tree". It is called a node tree because we can visualize it as a tree with a single root and many branches and leaves.

Let's have a look at what this means. Take as an example the following HTML document (`index.html`):

```
<!DOCTYPE html>
<html>
<head>
    <title>The DOM</title>
</head>
<body>
    <h1>Hello JavaScript Dev!</h1>
    <p>Random text</p>
</body>
</html>
```

Each HTML element inside `index.html` will be represented in the DOM as a node:



The DOM includes the content of a webpage in addition to all the elements. However, we normally leave out drawing/representing the content.

#### Pictorial representation of the DOM

An important point to keep in mind is that these nodes are objects. The document that is parsed by the browser becomes a `document` object, this is the root node. All other nodes such as `HTML`, `head`, `body`, etc. are child nodes. And from *Chapter 3*, we know that objects have properties and methods. The same way these node objects can be manipulated via DOM properties and methods. And this is how you can perform CRUD (create, read, update, and delete) operations on DOM elements.

If you'd like to visualize the DOM you can do so by using the Live DOM Viewer at <https://software.hixie.ch/utilities/js/live-dom-viewer/>. In the next section, we will cover some essential DOM properties.

#### Exercises

1. Can you explain the DOM?
2. What are the 2 purposes of the DOM?

3. The DOM API can only be used with JavaScript. True or False?

4. How are objects in the DOM assembled? Choose one option:

- Queue
- Hierarchy
- Stack

### Answers

1. The DOM API allows you to access and manipulate HTML elements on webpages.

You can add, delete and change HTML elements via the DOM API using JavaScript.

2. The two purposes of the DOM API are:

- It defines the logical structure of an HTML document for example index.html
- Dictates how the elements within an HTML document can be accessed and modified

3. False. The DOM API is language neutral programming API.

4. Hierarchy. Objects in the DOM are constructed as a hierarchical tree of objects.

## 7.2 Properties of the DOM

The nodes represented in the DOM are objects that have properties such **style**, **innerHTML**, and many more. This is akin to objects in JavaScript that have properties and methods associated with them. In this section, we will go over some of the more common DOM properties by first understanding what a DOM property is.

A DOM property is the value of an HTML element such as its ID, value or class.

Take as an example the following HTML:

```
<p id='myPara'>Twitter Bootstarry</p>
```

From the previous section we know that the root **document** object provides access to all the properties and methods of node objects in the DOM. Using the root **document** object there are two steps to accessing the property of an element in the DOM:

1. Access the node using the dot notation (.)
2. Access the property of the node using the dot notation (.) again

Therefore, we can access the `p` element in the document by first referencing the `document` object followed by the dot notation (.):

```
document.
```

Next, we will access the `p` element by the `id` assigned to it. To undertake this action we use the DOM `getElementById()` method which takes as an argument that value of the `id` within parenthesis ():

```
document.getElementById('myPara');
```

To verify that we have retrieved the `<p>` element we can log this statement to the console to get a full reference to the HTML element:

```
console.log(document.getElementById('myPara'));
```

```
"<p id='myPara'>Twitter Bootstarry</p>"
```

Next, we would like to access the `innerHTML` property of the `p` element. The `innerHTML` property refers to the text value of an element. We can accomplish this by chaining the `innerHTML` property using the dot notation (.):

```
document.getElementById('myPara').innerHTML;
```

This will retrieve the text inside the `p` node with the `id` of 'myPara' in the DOM which is:

```
"Twitter Bootstarry"
```

We can also modify the value of the `innerHTML` property by assigning a new string value to it:

```
document.getElementById('myPara').innerHTML = 'Cake';
```

And now the `<p>` element with an `id` of 'myPara' will be updated in the DOM and this change will be reflected on the webpage.

### Note

The `innerHTML` property's value will be updated in the DOM. Your HTML code in your index.html file or wherever you have written your HTML will **not** change.

Therefore, by using the root `document` object and then using the dot notation (.) to retrieve an element by its ID we were able to access and modify the `innerHTML` property.

It is important to note that the attributes of HTML elements do not have a 1 : 1 mapping to DOM properties. Attributes are defined by HTML whereas; properties are defined by the

DOM. Only standard HTML attributes are converted to DOM properties. To examine this in detail, have a look at the following example:

```
<input type='text' id='userId' value=1 status='false'>
```

The above input element has three attributes:

1. `type` (standard HTML attribute)
2. `id` (standard HTML attribute)
3. `status` (user defined HTML attribute)

We can verify that only standard HTML attributes are mapped to DOM properties by querying each attribute of the input element:

```
console.log(document.getElementById('userId').type); // "text"  
console.log(document.getElementById('userId').id); // "userId"  
console.log(document.getElementById('userId').value); // "1"
```

The standard HTML attributes are converted to DOM properties hence when we query them we are returned with their values. The property `status` is user-defined and is not a standard HTML attribute therefore, let's see what happens when we try and access it:

```
console.log(document.getElementById('userId').status); // undefined
```

`undefined` is returned as you cannot access and modify non-standard HTML attributes in the DOM.

Listed below are some common DOM properties:

1. `nodeName`: This will return the name of the node
2. `parentNode`: This retrieves the parent of a queried HTML element
3. `childNodes`: Returns the child nodes of an HTML element
4. `style`: references the style object which in turn can be used to modify style attributes

Let's see how these are used with an example:

```
<div id='myDiv'>  
  <p id='para1'>Hello!</p>  
</div>
```

For the above HTML, let's access the `style` property of the `div` element and change the font color of the text to red. To do so, we first access the `div` itself using the root `document` object:

```
document.getElementById('myDiv')
```

Now let's access its `style` property by chaining it to the `myDiv` element using the dot notation:

```
document.getElementById('myDiv').style
```

We would like to change the font color of text inside this `div` to `red`. A point to keep in mind about the `style` property is that it is an object. Even though most DOM properties are strings, the `style` property is an object. The different properties of a `style` object correlate to CSS properties such as `color`, `background-color`, `width`, `height`, etc.

Now let's change the color of any text inside the `myDiv` element:

```
document.getElementById('myDiv').style.color = 'red';
```

hello!

We can also modify other style properties such as the font size:

```
document.getElementById('para1').style.fontSize = "2em";
```

This will increase the font size of the `p` element that has an `id` of `para1` to `2em`

# hello!

### Output on the webpage with the font-size increased

Similarly if we wanted to get a node's name, we can use the `nodeName` property:

```
document.getElementById('para1').nodeName; // "P"
```

And this lets us know that the name of the node that has an `id` 'para1' is "P". This might seem a bit futile since the id 'para1' insinuates that it is a paragraph. However, this is useful in cases where elements have ambiguous names.

Next, we will learn and practice DOM methods. These are more exciting than DOM properties and are truly what make the DOM powerful.

### Exercises:

1. What root DOM node provides access to all the other DOM nodes?
2. Access the text content in between the opening and closing `<p>` tags:

```
<p id='myP'>The DOM is super handy for making webpages  
interactive</p>
```

3. Change the text content in between the `<h1>`tags to Enterprise Solutions  
`<h1>WebTech Solutions</h1>`
4. Map the following input attributes to DOM properties

```
<input type='number' id='productVal' value= 100  
status='available'>
```

5. Return the length of the children of the following `<ul>` element

```
<ul>  
    <li>ID: 2183</li>  
    <li>ID: 1030</li>  
    <li>ID: 105</li>  
    <li>ID: 8294</li>  
</ul>
```

6. Change the color of the text within the first `<p>` tag to #b19cd9

```
<p>id quod maxime placeat facere possimus, omnis volupt</p>  
<p>Sed ut perspiciatis unde omnis iste natus error sit  
voluptatem accusantium doloremque laudantium, totam rem  
aperiam, eaque ipsa quae ab illo inventore veritatis et quasi  
architecto beatae vitae dicta sunt explicabo. </p>  
<p>Ut enim ad minima veniam, quis nostrum exercitationem  
ullam corporis suscipit laboriosam, nisi ut aliquid ex ea  
commodi consequatur?</p>
```

7. Change the color, background color and width properties of the following `div` element. The color should be `#77dd7`, background color should be `#fdfd96` and the width should be set to `100px`.

```
<div id='div1'>  
  <p>Arrays rock</p>  
</div>
```

## Answers

1. `document` is the root object/node which will allow you to access all the other DOM nodes.
2. We can verify that we've accessed the text content of the `myP` element by retrieving the element and then accessing the value of its `innerHTML` property. This is assigned to a variable `paraText` and we can `console.log paraText` to verify its value:

```
let paraText = document.getElementById('myP').innerHTML;  
console.log(paraText);
```

"The DOM is super handy for making webpages interactive"

- 3.

```
let heading1 = document.querySelector('h1').innerHTML =  
'Enterprise Solutions';
```

4. Only the standard HTML attributes are converted to DOM properties.

```
console.log(document.getElementById('productVal').type); //  
"number"  
  
console.log(document.getElementById('productVal').id);  
//productVal  
  
console.log(document.getElementById('productVal').value); //  
"100"  
  
console.log(document.getElementById('productVal').status); //
```

- 5.

```
let children =  
document.querySelector('ul').getElementsByName('li');
```

```
console.log(children.length); //4
```

6. The `querySelector()` method returns the 1<sup>st</sup> instance of a query:

```
let firstP = document.querySelector('p');  
firstP.style.color = "#b19cd9";
```

7. Retrieve the `div` element and use the `style` property to modify the `div`:

```
let div = document.getElementById('div1');  
div.style.color = '#77dd77';  
div.style.width = '100px';  
div.style.backgroundColor = '#fdfd96'
```

## 7.3 DOM methods

As we discussed earlier, nodes in the DOM are objects. Having discussed objects earlier in *Chapter 3*, we know that objects have properties and methods associated with them. Just as object methods are actionable, in the same manner, DOM methods allow you to perform actions on elements on a webpage via the DOM. For example, using DOM methods you can perform CRUD actions such as access an element on a webpage, create, delete, replace, and update an element. This is known as DOM manipulation and is one of the reasons that makes JavaScript so popular. Let's examine a few popular DOM methods

### 7.3.1 Accessing elements

There are a couple of ways to access elements using DOM methods. We will cover the most popular ones.

#### 7.3.1.1 `getElementById`

This method will retrieve an element by its ID. The ID is passed to the method within the parenthesis `()`. For example take for example an `h1` element on a webpage:

```
<h1 id='headingUnique'>hello</h1>
```

In order to retrieve this element using JavaScript, we will use the `getElementById()` DOM method:

```
document.getElementById('headingUnique');
```

We first reference the `document` object followed by the dot notation and the name of the method which is `getElementById()`. The ID of the element that we are retrieving is passed within strings inside the parenthesis `()`.

This will retrieve the following `h1` element

```
<h1 id='headingUnique'>hello</h1>
```

### 7.3.1.2 getElementsByClassName

This method will retrieve all the elements of a certain class. For example consider the following HTML:

```
<h1 class='classA'>hello</h1>
<p class='classA'>world</p>
<h1 class='classA'>!</h1>
```

In order to retrieve all these elements from the `classA` class, we will use the DOM `getElementsByClassName()` method:

```
document.getElementsByClassName('classA');
```

This will get all the elements in our HTML file that have a class of `classA`. Let's verify this in the following way:

```
let elements = document.getElementsByClassName('classA');
```

The variable `elements` is a reference to all the elements retrieved by the `getElementsByClassName()` method. This is represented as a `NodeList`. We will explore NodeLists in detail further on in this chapter in *section 7.4*

### 7.3.1.3 querySelector

This method will return the first element that matches a specific CSS selector

For example let's take the following HTML as an example:

```
<h1 class='classA'>hello</h1>
<p class='classA'>world</p>
<li class='classA'>!</li>
```

We can query the first element that has the class `classA` with:

```
document.querySelector('.classA');
```

We can verify that only the first element with the `classA` class was retrieved by logging to the console what the method returns:

```
console.log(document.querySelector('.classA'));
```

This will return the element retrieved the `querySelector()` method:

```
"<h1 class='classA'>hello</h1>"
```

And we see that only the first element that occurs on a webpage with the specified class is returned.

#### 7.3.1.4 `querySelectorAll`

This method will retrieve all the elements that match a specific CSS selector as a static **NodeList** object. Taking on from the example above in *section 7.3.1.3 (querySelector)*, let's retrieve all elements that match the `classA` class:

```
document.querySelectorAll('.classA');
```

We can verify that the `querySelectorAll()` method returned all the elements with the specified class, by querying the `NodeList`'s `length` property:

```
let elements = document.querySelectorAll('.classA');
console.log(elements.length); //3
```

3 is returned as there are three HTML elements with the `classA` class. If you are confused about what a `NodeList` is, don't worry it is covered in the upcoming section. For now keep in mind that a `NodeList` as the name suggests is a list of nodes. Since nodes are objects, they have a `length` property.

#### 7.3.1.5 `getElementsByTagName`

This method will retrieve all elements of a specific tag. For example let's retrieve all the HTML elements that are `p` elements:

```
<p>1</p>
<p>2</p>
<p>3</p>
<p>4</p>
```

By using the `getElementsByTagName()` let's retrieve all the `p` elements:

```
document.getElementsByTagName("p");
```

Once again we can verify that all the four `p` elements were retrieved in the following way:

```
let paras = document.getElementsByTagName("p");  
let parasLength = paras.length;  
console.log(parasLength); // 4
```

By accessing the `length` property of the `NodeList` object that is returned, we are able to verify that indeed all the `4 p` elements were returned.

Moving on, can also delete nodes in the DOM. The following section will discuss deletion of nodes in the DOM.

### 7.3.2 Deleting elements

In order to remove a node in the DOM, use the `remove()` method. For example take the following HTML:

```
<ul>  
  <li>apple</li>  
  <li>strawberry</li>  
  <li>coconut</li>  
</ul>
```

We can remove the entire `ul` element with:

```
document.querySelector('ul').remove();
```

This will entirely remove the `ul` node from the DOM. *Take note that it does not modify your HTML in your file. The webpage view will be updated but your HTML is not.*

We can also remove a child of the `ul` element by using the `removeChild()` method. Let's remove the first `li` element with the `ul` tag:

```
let parent = document.querySelector('ul');  
let children = document.querySelectorAll('li');  
parent.removeChild(children[0]);
```

Let's discuss what's happening here:

- We first access the parent `ul` element with the `querySelector()` method which is referenced by the variable `parent`.
- And similarly, we retrieve all the children `li` elements by the `querySelectorAll()` method and assign all the nodes which are returned to

variable `children`. Remember, that the `querySelectorAll()` method returns a `NodeList` object

- We then call the `removeChild()` method on the `parent`. The `removeChild()` method takes as an argument the element to remove (`children`) and its index `0` in the `NodeList` object. This will remove the first child `li` element.

Another way that implicitly removes an element from the DOM is to set the `innerHTML` property of the element to an empty string `''`. For example:

```
let parent = document.querySelector('ul');  
parent.innerHTML = '';
```

This will hide all the text content of the `ul` element, which includes all the list items.

However, this is less explicit in its intention and should be avoided.

### 7.3.3 Creating DOM elements

You can also create nodes and dynamically insert them into the DOM by using methods that create elements. Let's take a brief look at some of these.

#### 7.3.3.1 createElement()

In order to create an element and insert it into a webpage there are two steps:

1. Create the element
2. Append it to the body of the document

Take as an example the following HTML page:

```
<!DOCTYPE html>  
  
<html>  
  
<head>  
  
  <meta charset="utf-8">  
  
  <meta name="viewport" content="width=device-width">  
  
  <title>The DOM</title>  
  
</head>  
  
<body>  
  
</body>  
  
</html>
```

Right now there is nothing within the body tags. Therefore, nothing is shown on the webpage. Let's insert an `h1` node via the DOM into our webpage with the following code:

```
let heading = document.createElement('h1');  
heading.innerHTML = 'Hello JavaScript';  
document.body.appendChild(heading);
```

Let's examine what's happening here:

- We declare a variable called `heading` and its value is assigned to an `h1` element created by the `createElement()` method
- We then set the `innerHTML` property of the newly created `h1` element to '`Hello JavaScript`'
- And lastly we insert it into the webpage by appending it to the `body` of the `document` by using the `appendChild()` method which takes as an argument the element that you want to insert into the DOM

Therefore, we have created a new `h1` element, set its text content, and inserted it into our webpage.

# Hello JavaScript

## Output on the webpage

We can also use the `textContent` property instead of the `innerHTML` property, like so:

```
heading.textContent = 'Hello JavaScript';
```

This will have the same effect. Therefore, using the `createElement()` method and `textContent` property you can set up an entire basic webpage using only JavaScript!

### Note

`innerHTML` is not considered to be secure as it is risk of XSS (cross-site scripting) attacks. Therefore, it is recommended to use the `textContent` property.

In the upcoming section we will have a look at a method called `createTextNode()`.

### 7.3.3.2 `createTextNode()`

This method creates a text node. A text node refers to a string, it correlates to the information that you see between opening and closing tags. For example

```
<p>This is some text between the p tags</p>
```

Instead of using the `innerHTML` property, we can use the `createTextNode()` method to create some text. The only difference here is that we will have to append the text to its parent element.

Let's refactor the example above in *Section 7.3.3.1* to use the `createTextNode()` method:

```
let heading = document.createElement('h1');
let text = document.createTextNode('Hello JavaScript');
heading.appendChild(text);
document.body.appendChild(heading);
```

A line by line explanation of what is happening is:

- We declare a variable called `heading` that is a reference to an `h1` element created by the `createElement()` method
- We also declare a variable called `text` and assign its value to be the text `'HelloJavaScript'` which is created by the `createTextNode()` method
- Then we append the text to its parent element with the `appendChild()` method called on the `heading`. The method takes variable `text` as an argument
- Lastly, we insert the heading into the webpage by appending it to the body of the document by using the `appendChild()` method which takes as an argument the `heading` element

### 7.3.4 Inserting DOM elements

We have seen a DOM method called `appendChild()` which will insert a newly created element into the DOM. This method adds a node as the last child of its parent. Let's look at inserting a node as the first child of its parent with the `insertBefore()` method.

### 7.3.3.2 insertBefore()

This method will insert a node into the parent element before a specific sibling node. This is better explained with the following example:

```
<ul>
  <li>Eggs</li>
  <li>Milk</li>
  <li>Bread</li>
</ul>
```

- Eggs
- Milk
- Bread

On our webpage we have three list items nested inside a `ul` tag. We would like to insert a list item before `Eggs` on our webpage. To do this we will use the `insertBefore()` method:

```
let ul = document.querySelector('ul');
let li = document.createElement('li');
li.textContent = 'Cheese';
ul.insertBefore(li, ul.firstChild);
```

Let's examine what we are doing:

- We first use the `querySelector()` method to retrieve the `ul` element and assign its reference to `let ul`
- Then we create a `li` element using the `createElement()` method. This newly created DOM node is referenced by `let li`
- We then set the `textContent` property of the `li` element to the string `'Cheese'`
- Lastly, we call the `insertBefore()` method on the `ul` element which is referenced by `let ul` in our code. We pass two arguments to this method. The first argument is the node that we want to insert into the DOM which is the `li`

node that we created. The second argument specifies at what position in the parent element we want to insert the node into. The position is assigned by the `firstElementChild` property which gets the first child of the parent `ul` node

- Cheese
- Eggs
- Milk
- Bread

Therefore, we have inserted a new `li` node into the DOM as the first child of its `ul` parent. You can also replace a node with another by using the `replaceChild()` method.

### 7.3.3. `replaceChild()`

Let's replace the third item of our list which is `Bread` with a new list item. The HTML will remain the same:

```
<ul>
  <li>Eggs</li>
  <li>Milk</li>
  <li>Bread</li>
</ul>
```

Our JavaScript code so far looks like this:

```
let ul = document.querySelector('ul');
let li = document.createElement('li');
li.textContent = 'Cheese';
ul.insertBefore(li, ul.firstElementChild);
```

Now let's create a new list node which will replace the third list item on the webpage:

```
let ul = document.querySelector('ul');
let li = document.createElement('li');
li.textContent = 'Cheese';
ul.insertBefore(li, ul.firstElementChild);

let newLi = document.createElement('li');
```

```
newLi.textContent = 'Chocolate';  
ul.replaceChild(newLi, ul.children[3]);
```

We create a new `li` node and set its `textContent` property to be the string '`Chocolate`'. And then we use the `replaceChild()` method on `ul` by passing in as an argument, the new `li` node. The second argument of the `replaceChild()` method is the node in the DOM which is to be replaced. In this case it is the third child of the `ul` node. Keep in mind that numbering starts from 0, therefore, the last item is

```
<li>Bread</li>
```

- Cheese
- Eggs
- Milk
- Chocolate

Therefore, by using the `replaceChild()` method we can replace specific nodes in the DOM.

In this section, we covered DOM methods for performing CRUD (create, read, update, and delete) actions. These methods are what make webpages interactive. As depending on certain conditions you can apply these DOM methods. For example, you can display a cart item on an e-commerce website once someone clicks on the 'Add to cart' button.

In the next section, we will discuss how to iterate through DOM nodes.

## Exercises

1. What are 3 DOM methods used to retrieve DOM nodes/HTML elements
2. What is the difference between `querySelector()` and `querySelectorAll()`?
3. For the following HTML, retrieve the number of `<p>` elements

*Hint: use the `length` property:*

```
<ul>  
  <p>foo</p>  
  <p>bar</p>  
  <p>zingga</p>
```

```
<p>nee</p>  
</ul>
```

4. For the above HTML (#3) retrieve the `innerHTML` of the fourth `<p>`
5. Carrying on from question #3 and #4, remove the third `<p>`
6. Create a `<div>` and `<p>`. Assign the following text to the `<p>` and append it to the `<div>`. Make sure that both elements are present on a webpage:

```
„Cras ultrices, risus vitae tristique malesuada, metus  
tortor pulvinar eros, id finibus ex neque id magna. In  
lectus tortor, sagittis et purus in, vulputate viverra  
massa. Nullam in erat diam. Sed eleifend, eros ac  
venenatis convalli'.
```

7. From question #6, replace the `<p>` node with a `<li>` node. Set the text content of the `<li>` to 'Magnus Operandi'. Don't delete anything.
8. For the following HTML insert the number 100 before 200:

```
<ul id='myUl'>  
  <li>200</li>  
  <li>300</li>  
  <li>400</li>  
</ul>
```

9. `innerHTML` is preferred over the `textContent` property? True or False?
10. `console.log` the value of the third input element ('Pinky') using the `querySelectorAll()` method.

```
<label for='username'>Username:</label>  
<input type='text' id='username' name='username' value=' '>  
<button id='myButton'>Submit</button>  
  
<label for='username1'>Username:</label>  
<input type='text' id='username1'  
name='username1' value='Gigi'>
```

```

<button id='myButton'>Submit</button>

<label for='username2'>Username:</label>
<input type='text' id='username2'
name='username2' value='Pinky'>
<button id='myButton'>Submit</button>

```

11. Create a button and insert it into your document. Set the text content of the button to 'click'
12. Carrying on from question #11, set the `id` attribute of the button to 'myButton' and set its `onclick` attribute to 'clickFunc()'. Log the button to the console, what do you get back?
13. Attach an `addEventListener()` method to the button so that when the button is clicked, the user is alerted with the greeting, 'Hi there junior dev'
14. Get the `onclick` attribute of the button that you just created with the `getAttribute()` method
15. What is the difference between the document and window object?
16. What is the difference between `remove()` and `removeChild()`?

## Answers

1. 3 DOM methods used to retrieve DOM nodes/HTML elements are:
  - `getElementById()`
  - `querySelector()`
  - `getElementsByName`
2. The `querySelector()` methods returns the first query. Whereas the `querySelectorAll()` method will return all the queried nodes that match a specific CSS selector.
- 3.

```

let pFourth = document.querySelectorAll('p');
console.log(pFourth.length); // 4

```

4.

```
let pEl = document.querySelectorAll('p');
console.log(pEl[3].innerHTML); // "nee"
```

5.

```
let pEl = document.querySelectorAll('p');
let ul = document.querySelector('ul');
ul.removeChild(pEl[2]);
```

6.

```
let div = document.createElement("div");
let p = document.createElement("p");

p.textContent = 'Cras ultrices, risus vitae tristique
malesuada, metus tortor pulvinar eros, id finibus ex neque id
magna. In lectus tortor, sagittis et purus in, vulputate
viverra massa. Nullam in erat diam. Sed eleifend, eros ac
venenatis convallim';
div.appendChild(p);
document.body.appendChild(div);
```

7.

```
let div = document.createElement("div");
let p = document.createElement("p");

p.textContent = 'Cras ultrices, risus vitae tristique
malesuada, metus tortor pulvinar eros, id finibus ex neque id
magna. In lectus tortor, sagittis et purus in, vulputate
viverra massa. Nullam in erat diam. Sed eleifend, eros ac
venenatis convallim';
div.appendChild(p);

let li = document.createElement('li');
li.textContent = 'Magnus Operandi';
div.replaceChild(li, div.children[0]);
document.body.appendChild(div);
```

8.

```
let ul = document.getElementById('myUl');

let li = document.createElement('li');

li.textContent = '100';

ul.insertBefore(li, ul.firstChild);
```

9. False. `textContent` is preferred as `innerHTML` is at risk of XSS attacks.

10. The `querySelectorAll()` method returns a node list therefore, you must specify the index of the input element you want:

```
let input = document.querySelectorAll('input');

console.log(input[2].value);
```

11.

```
let button = document.createElement('button');

button.textContent = 'click';

document.body.appendChild(button);
```

12.

```
let button = document.createElement('button');

button.textContent = 'click';

button.setAttribute('id', 'myButton');

button.setAttribute('onclick', 'myFunc()');

document.body.appendChild(button);

console.log(button);
```

```
"<button id='myButton' onclick='myFunc()'>click</button>"
```

13.

```
let button = document.createElement('button');

button.textContent = 'click';

button.setAttribute('id', 'myButton');

button.setAttribute('onclick', 'myFunc()');
```

```
button.addEventListener('click', function() {
    alert('Hi there junior dev')
})
```

14.

```
button.getAttribute('onclick')
```

15. The global `window` object is loaded into your browser, it has methods and properties such as `window.onload()` etc. The document is a document containing your code which will be loaded inside the window.
16. `remove()` will remove the entire parent element. `removeChild()` will remove the first child node of a specified parent.

## 7.4 Iterating through DOM nodes

Besides performing CRUD tasks on nodes in a document you can also iterate through what is called a **NodeList**. A **NodeList** is a collection of nodes retrieved from a document. It is an array-like object therefore, you can iterate through it. However, it is not an array and array methods cannot be used.

A NodeList has a `length` property by which you can access the number of nodes in the list. The `length` property also allows you to iterate through a node list and access a particular node at an index, starting at 0. Let's have a look at how to iterate through a NodeList object:

```
<ul>
    <li>1</li>
    <li>2</li>
    <li>3</li>
    <li>4</li>
</ul>
```

There are four `li` elements in the above HTML snippet. We will access them and modify each one of them by iterating through the NodeList that is returned:

```
let listItems = document.querySelectorAll('li');
```

We first retrieve all the `li` nodes in the DOM. Next, let's log to the console the number of `li` nodes through its `length` property:

```
console.log(listItems.length); //4
```

This lets us know that there are 4 `li` nodes. Next, we will iterate through this NodeList and add the string ' `apples`' to each one of the `li` nodes:

```
for(var i =0; i < listItems.length; i++) {  
    listItems[i].innerHTML+= ' apples';  
}
```

We have used a regular for-loop to iterate through the length of the NodeList. At each iteration we use the addition assignment `+=` operator to add the string ' `apples`' to the existing text content of the list item. And the output as shown below:

- 1 apples
- 2 apples
- 3 apples
- 4 apples

#### Output on the webpage

Iterating through DOM nodes is useful when you're doing extensive DOM manipulation and need to modify a node list.

A point to keep in mind about NodeLists is that there are two types:

1. Static NodeList: Changes in the DOM will have no effect on the NodeList. The `querySelectorAll()` method returns a static list
2. Live NodeList: Any changes in the DOM will update the NodeList

### Exercises

1. What is **NodeList**?
2. Iterate through the following `<p>` elements and append the string '`Bonus Player`' to each one

```
<div class='container'>  
<div class='row'>  
    <div class='col-sm-12'>  
        <p>BaaBara</p>
```

```
<p>Poco</p>  
<p>Stardew</p>  
<p>LikeK</p>  
</div>  
</div>  
<div>
```

3. Remove the 2<sup>nd</sup> <p> from the div

## Answers

1. A **NodeList** is a collection of nodes retrieved from a document. It is an array-like object therefore, you can iterate through it. However, it is not an array and array methods cannot be used.

2.

```
let pEl = document.querySelectorAll('p');  
pEl.forEach(function(item) {  
    item.textContent += ' Bonus Player';  
})
```

3.

```
let div = document.getElementsByClassName('col-sm-12')[0];  
div.removeChild(pEl[1]);
```

## Summary

In this chapter, we covered key topics on the DOM API. This includes properties and various methods of the DOM. We also discussed iterating though a **NodeList** as a way to access and modify each node in a node list. Using JavaScript to manipulate the DOM is a good entry point for frontend developers looking to build interactive websites and web applications.

# Theory Thor - 8

*“Mastery lies on an infinite continuum” — C. Matakas*

In this chapter, we will go over some important theoretical concepts. Though these concepts are intermediate in nature, an understanding of these will greatly help you along your journey in honing your JavaScript expertise. I recommend that you keep practicing the coding exercises from other chapters as you go along. This will motivate you to keep coding as you form the habit of reading other topics.

We're going to cover the following main topics:

- What is the call stack?
- Execution context
- Event loop

## 8.1 What is the call stack?

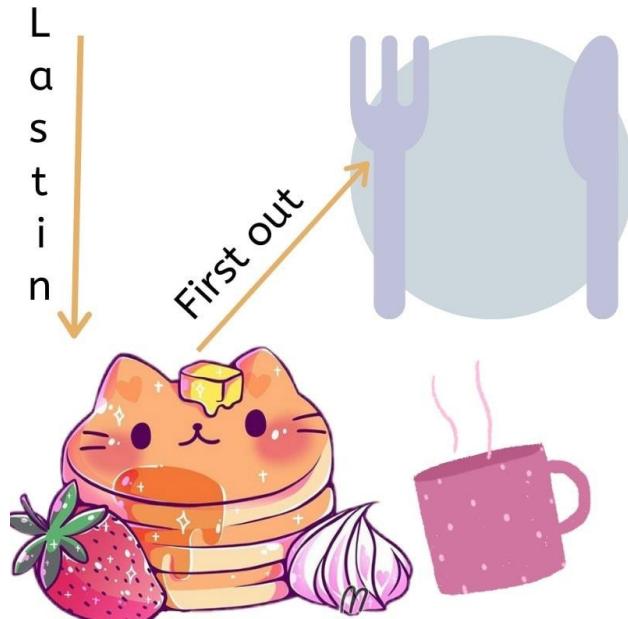
An understanding of the call stack will allow you to understand how execution order in JavaScript works. JavaScript is a single-threaded language; this means that it handles one task at a time. JavaScript code is executed by the JavaScript engine. Google's V8 engine is an example of a popular JavaScript engine, there are others such as the SpiderMonkey and Rhino. The most popular host environment for the JavaScript engine to run in is the browser.

The V8 engine is written in C++. It is an open-source, high-performance JavaScript and Web Assembly engine. The V8 engine has two main parts:

- Heap: The heap is an unstructured memory that is used for memory allocation of variables and objects
- Stack: The stack is used for function calls and it records where we currently are in the program

Since JavaScript only handles one task at a time, as such it has a single call stack. The call stack is mainly used for function calls. Functions are executed from top to bottom, one at a time. A call stack is a dynamic data structure that follows the Last-In Last-Out (LIFO) principle. Its goal is to record where in the program we currently are. The LIFO principle temporarily stores and manages function invocation.

When we say that the call stack applies the LIFO principle to function invocation, this means that the last function to be pushed into the stack is the first to be popped out from the stack, when the function returns.



Last pancake on the stack is the first one on the plate

Let's have a look at an example:

```
function functionOne() {  
    throw new Error('Stack Trace Error');  
}  
  
function functionTwo() {  
    functionOne();  
}  
  
function functionThree() {  
    functionTwo();  
}  
  
functionThree();
```

The following will be returned:

```
✖ ► Uncaught Error: Stack Trace Error
    at functionOne (<anonymous>:2:9)
    at functionTwo (<anonymous>:6:2)
    at functionThree (<anonymous>:10:3)
    at <anonymous>:13:1
```

Notice that the order of the functions logged to the console:

- functionOne
- functionTwo
- functionThree

functionOne which is the last function to be called from within functionTwo is the first function out and logs an error. This is followed by functionTwo. Lastly, functionThree is logged, which is the first function to be pushed onto the stack during execution. This demonstrates the LIFO principle. The last function to be pushed into the stack (functionOne) is the first one to be popped from the stack. And the first function to be pushed into the stack is the last one to be popped from the stack.

When a function is invoked, its arguments and variables within the function are pushed into the stack. An entry in the call stack is called a stack frame. A stack frame is a memory location within the call stack. Once the function is popped off the stack, the memory is cleared.

The call stack manages function invocation as it keeps a record of the position of each function call (stack frame) as functions get pushed and popped from the call stack. This record-keeping is what makes code execution synchronous. As the position of the next function to execute is known in advance.

Let's have a look at another example:

```
function function1() {
  console.log('function 1');
}

function function2() {
  function1();
  console.log('End of function 2');
}
```

```
function2();
```

The order of the messages logged to the console is:

```
"function 1"  
"End of function 2"
```

Let's deconstruct what is happening:

- `function2()` is called
- A stack frame for `function2` is constructed
- `function1()` is called from within `function2()`, at which point it is added to the call stack
- The `console.log` statement within `function1()` is executed and '`function 1`' is logged to the console
- `function1()` which was the last function to be added to call stack is the first to be popped from the call stack
- Then the execution order points to `function2()`, the `console.log` statement within `function2()` is executed and is logged
- `function2()` which was the first function to be added to the call stack is then popped from the call stack
- Memory is cleared

Thus the call stack by managing function invocation knows the execution order of functions and code is executed line by line, synchronously.

You might have heard of the term stack overflow. The stack is said to overflow when a function calls itself repeatedly from within itself (a recursive function). For example:

```
function hello(){  
    hello();  
}  
  
hello();
```

The following will return:

```
✖ ▾Uncaught RangeError: Maximum call stack size exceeded
    at hello (<anonymous>:2:3)
    at hello (<anonymous>:2:3)
```

Recursive functions as `hello()` will cause the stack to overflow. This happens when there is no exit condition and a function repeatedly calls itself. The browser's call stack has a maximum call stack size that it can accommodate before it throws an error, as seen above in the example.

## 8.2 Execution context

Execution context is an abstract concept describing the environment within which JavaScript code is executed. The execution context will dictate if a variable, object, or another block of code is accessible or not. There are two types of execution contexts: global and function.

Let's examine the global execution context first.

### 8.2.1 Global execution context

The global execution context is the default execution context. Variables and other blocks of code that are not within a function are executed inside the global execution context. The global context is attached to the `window` object and there is only one global execution context throughout code execution. For example:

```
let greeting = 'hello';
console.log(greeting === window.greeting); true
```

The global variable `greeting` is a property of the global `window` object. Therefore, `true` is returned.

The global execution context therefore, does the following:

- Creates the `window` object as the global execution context
- Creates a reference to the `window` object using `this`

## 8.2.2 Function execution context

The code within a function is executed within a function execution context. There are two phases concerning the function execution context:

1. Creation phase

2. Execution phase

Let's discuss these starting with the creation phase.

### 8.2.3.1 Creation phase

When a function call is encountered, the creation phase begins. This phase will setup up everything for execution. At this point the JavaScript engine has called a function but not yet executed it. A couple of things happen during the creation phase:

- An activation object is setup: Memory is set up in the activation object. This is a special object which contains all the variables defined within the function, the arguments object for the function, function arguments and any inner functions declaration inside the main function
- Scope chain is created: The scope chain is a list of all the variables within the function. This also includes all the variables in the global scope. It forms a connection to the outer environment.
- Create a **this** context: The value of **this** is initialized after the creation of the scope chain

Hoisting is an important part of the creation phase. During the creation phase memory is setup for all variable and function declarations to be saved. Therefore, before execution the JavaScript engine already has a reference to variable and function declarations. This is how you have access to variables before they are declared. For example:

```
console.log(a); //undefined  
var a = 5;
```

The variables are initialized in memory with a default value of **undefined** during the creation phase. This is why you are able to access **a** before its declaration. Variables declared with **let** and **const** are not assigned any initial values and trying to access them before declaration results in a **ReferenceError**:

```
console.log(a); //ReferenceError: can't access lexical  
declaration 'a' before initialization  
let a = 5;
```

The entire body of a function statement is hoisted, therefore you have access to the entire function statement before its declaration:

```
sum(1);

function sum(x) {
    console.log(x + Math.floor(Math.random() * 10));
}
```

The entire body of function `sum(x)` is available before it is declared. Now moving on let's briefly discuss the execution phase of the function execution context.

### 8.2.3.2 Execution phase

During this phase, code is parsed line by line by the interpreter. The variable object is updated with the values assigned to the variable by a user. And function calls are executed. If a new function call is encountered, it will stop executing the current function and the creation of a new execution context for that function will begin. A new outer reference to the immediate execution context will be setup.

## 8.3 Event Loop

JavaScript is single-threaded this means that the main thread on which code is executed runs one line at a time. There is no possibility of doing anything in parallel. Therefore, the run time is blocked until a task is completed. An example of blocking the run time is an infinite loop. However, the DOM API, `setTimeout()`, `setInterval()` methods are provided by the web API to allow us to write asynchronous non-blocking code.

Code execution flow is based on a concept called an event loop. While there are tasks (code to be executed) the JavaScript engine will execute the tasks starting with the oldest task first. Event handlers, loading external scripts, `setInterval()`, and `setTimeout()` methods are a few examples of such tasks.

The event loop is a constantly running process that will check if the call stack is empty. We learned about the call stack in detail in *section 8.1*. Let's understand this with an example:

```
function funcHello() {
    console.log('hello');
}

setTimeout(funcHello, 3000);

console.log('hi');
```

The order of the `console.log` statements will be:

```
"hi"  
"hello"
```

Function `funcHello()` is declared within global scope. After which the `setTimeout()` method is encountered. This method is provided by the browser API. It will delay tasks without blocking the main thread. The method accepts a callback function as an argument (`funcHello()`) and a second parameter (3 seconds) after which the callback function should be added to the event queue. `funcHello()` is added to the event queue after 3 seconds. The event queue is also called the callback queue, as the name suggests it is a list/queue of callback functions. It will remain there till the call stack is empty. When all the functions in the call stack are empty i.e. previously invoked functions have been popped off and the statement `console.log('hi')` has finished executing, then the first item in the event queue (`funcHello()`) will be added to the top of the call stack. The event queue follows the FIFO principle (First In First Out). The first item in the event queue is pushed to the call stack. The callback function is then invoked, returns a value, and gets popped off the call stack.

This process of monitoring the call stack and moving tasks from the event queue to the call stack is the event loop.

Let's have a look at another example:

```
function func1() {  
    console.log('function 1');  
}  
  
function func2() {  
    console.log('function 2');  
}  
  
setTimeout(func1, 5000);  
func2();
```

In the above example we have two functions, `func1()` and `func2()`. The function `func1()` is passed to the `setTimeout()` method which will execute the `func1()` function after 5 seconds. Let's examine how the function calls will execute:

- Initially the call stack and queue are empty
- The `setTimeout()` method is invoked and a timer runs in the browser

- The callback function `func1` passed to the `setTimeout()` method is added to the event queue after `5` seconds, where it will wait until the call stack is empty.
- In the meantime the function call `func2()` is added to the call stack first. And the statement `console.log('function 2')` is added to the top of the previous frame. The JavaScript engine executes the top frame (`console.log('function 2')`) and `func2()` is popped from the call stack
- Then the first item in the event queue which is `func1()` is pushed and popped from
- the call stack and the statement `console.log('function 1')` will be executed

The order of the statements logged to the console are:

```
"function 2"
"function 1"
```

Therefore the event loop monitors the call stack and event queue. When the call stack is empty, it will take an event (a callback function) from the event queue and push it to the call stack. Then the call stack executes the event/function and pops it off the call stack. Each cycle is called a tick.

## Summary

In this chapter, we learned about the call stack, execution context, and the event loop. Since JavaScript only handles one task at a time, the call stack which follows the LIFO (Last In Last Out) principle determines execution order. Moving on, we discussed execution context which describes the environment within which JavaScript code is executed. It will dictate if a variable, object, or another block of code is accessible or not. We also learned about the two types of execution contexts: global and function. Lastly, we learned about the event loop which is the process of monitoring the call stack and moving tasks from the event queue to the call stack.

