ILoveJavascript

Jon D Jones

Contents

I LOVE Javascript

Welcome to #ILOVEJavascript. The aim of this books is to teach you how to write better JavaScript code. Writing bad code is easy. If you look in most peoples repositories you will usually spot some. Writing clean and maintainable JavaScript which clear conveys what it does is difficult. This book is written as hundreds practical code samples you can use within your application.

Data Types

This section will cover code samples for all the primitives that are available within Javascript.

Var, Let and Const

This sections will shows differences between var, let and const. var is function scoped, not blocked scoped like let and const

```
if (true)  {
    var name = 'jon';
}
console.log(name);
```

var is function scoped

```
try {
    const example = function() {
        var scoped = 'test';
    };
;
    console.log(scoped)

} catch(e) {
    console.log('This will be triggered, as var is function scoped');
}
```

var will allow me to redefine cars, this is not a good idea and leads to confusing-code

```
var testThree = 'test';
var testThree = 'test';

// var delations get hosited to the top
// When this code runs it wont throw an exception
```

```
console.log(withVar);
var withVar;

// let does not get hoisted
try {
    console.log(withLet);
    let withLet ;
} catch(e) {
    console.log('This will throw as expeceted');
}
```

Strings

```
const name = 'jon   ';
console.log('Original String = ' + name);
console.log();
console.log('Length = ' + name.length);
console.log('Uppercase = ' + name.toUpperCase());
console.log('Lowercase = ' + name.toLowerCase());
console.log('Include(\'jon\') = ' + name.includes(name));
console.log('Finding a String using indexOf(\'on\') = ' + name.indexOf("on"));
console.log('Searching a String using search(\'jon\') = ' + name.search("jon"));
console.log('Slicing first character = ',name.slice(0, 1));
console.log('Sibstring = ', name.substring(1, 3));
console.log('Replace = ', name.replace('on', 'this part has been replaced'));
console.log('Trim = ', name.trim());
console.log('Concat = ', name.concat(name));
console.log('CharAt(2) = ', name.charAt(2));
console.log('Char[0] = ', name[0]);
console.log('Split(o) = ', name.split('o'));
console.log();
```

Template Strings/String Interpolation

If you need to combine string your code will be easier to read when you use template strings

Template string

```
console.log(`city=${city} country=${country}`);
```

Manually concatenating strings

```
console.log('city=' + city + ' country=' + country);
```

Resources

- Template Literals
- Value Type Vs Reference Type

Booleans

True

```
const imTrue = true;
console.log(imTrue);
```

False

```
const imFalse = false;
console.log(imFalse);
```

Resources

- Primitive
- Value Type Vs Reference Type

Dates

```
const now = new Date();
const year = now.getFullYear();
console.log(year);
```

```
const now = new Date();
const month = now.getMonth();
console.log(month);
```

```
const now = new Date();
const dayofMonth = now.getDate();
console.log(dayofMonth);
```

```
const now = new Date();
const hour = now.getHours();
console.log(hour);
```

```
const now = new Date();
const minute = now.getMinutes();
console.log(minute);
```

```
const now = new Date();
const second = now.getSeconds();
console.log(second);
```

```
const now = new Date();
const timestamp = now.getTime();
const readDataFromString = new Date(timestamp);
console.log(readDataFromString.getFullYear());
```

Resources

- Primitive
- Value Type Vs Reference Type

Undefined

Javascript assigns undefined when no value is explicitly set

```
let implict;
console.log(implict);
```

You can explicitly assign undefined. Explicitly setting undefined isn't recommended, as it hides the intention of the code. If you want to explicitly set something as empty, use null

```
const explicit = undefined;
console.log(explicit);
```

Undefined functions

```
let myFunction = function (paramter) {

    // When no parameter is passed in, this will throw undefinied
    console.log(paramter);
}
const result = myFunction();
```

When no return value is provided, undefined will be thrown

```
console.log(result);
```

Examples

```
if (implict === undefined) {
    console.log('implict is undeinifed');
}
```

Resources

- Primitive
- Undefined

Null

When a variable hasn't been defined correctly, Javascript will assign it undefined If you want to explicitly set something as not being set you should use null Using null instead of undefined gives other developers a better understanding of the intention of your code

```
let name = null;
console.log(name);

// Example
if (name === null) {
    console.log('name is null');
}
```

Resources

- Primitive
- Value Type Vs Reference Type

Numbers

```
const number = 12343.343;

console.log('ToFixed() / round decimal place', number.toFixed(2));
console.log('Math.round()', Math.round(number));
console.log('Math.floor()', Math.floor(number));
console.log('Math.ceil()', Math.ceil(number));
console.log('Math.random()', Math.random() * (100 - 0) + 0);
```

```
const age = 12;
const dogYear = (age + 1) / 7;

console.log(dogYear);
```

```
const numberOfQuestions = 1;
const numberOfCorrectResults = 3;
const percentage = (numberOfQuestions *  100) / numberOfCorrectResults;

console.log(percentage);
```

Resources

- Primitive
- Random()
- Value Type Vs Reference Type

Logic Gates And Conditional Statements

IF & ELSE

IF/ELSE Statement

```
if (true) {
    console.log('true');
} else {
    console.log('false');
}
```

IF/ELSE IF/ELSE Statement

```
if (true) {
    console.log('true');
} else if (false) {
    console.log('false');
} else {
    console.log('NOt Possible');
}
```

Examples

```
const temp = 12;
if (temp <= 32) {
    console.log('Its freezing');
}
```

```
const age = 45;
if (age <= 7) {
    console.log('Is Child');
} else if (age >= 65) {
    console.log('Is Senior');
} else {
```

```
    console.log('Is Adult');
}
```

AND, OR, NOT

```
const optionOne = true;
const optionTwo = true;

if (optionOne && optionTwo) {
    // both conditions are true
    console.log("optionOne AND optionTwo");
}

if (optionOne || optionTwo) {
    // one conditions has to be true
    console.log("optionOne OR optionTwo");
}
```

Example

```
const temp = 65;
if (temp >= 60 && temp <= 90) {
    console.log("It's nice outside");
}

if (temp <= 0 || temp >= 150) {
    console.log("It's crap 'outside");
}
```

Operators

Equality operator

```
const temp = 31;
let isFreezing = temp === 31;
console.log('Is Freezing ' + isFreezing)
```

!== Inequality operator

```
const temp = 31;
isFreezing = temp !== 31;
console.log('Is not Freezing ' + isFreezing)
```

< Less than operator

```
const temp = 31;
isFreezing = temp < 31;
console.log('Is less than Freezing ' + isFreezing)
```

<= Less than or equal operator

```
const temp = 31;
isFreezing = temp <= 31;
console.log('Is less than or equal to Freezing ' + isFreezing)
```

Greater than operator

```
const temp = 31;
isFreezing = temp > 31;
console.log('Is greater than Freezing ' + isFreezing)
```

= Greater than or equal operator

```
const temp = 31;
isFreezing = temp >= 31;
console.log('Is not greater than or equal to Freezing ' + isFreezing)
```

Examples

```
const age = 30;
const isChild = age <= 7;
const isAdult = age > 8 && age < 65;
const isSenior = age >= 65;

console.log('isChild = ' + isChild);
console.log('isAdult = ' + isAdult);
console.log('isSenior = ' + isSenior);
```

Resources

- Double Equals Vs Triple Equals

Ternary Operator

Basic example:

```
let result = (1 + 1) ? 1 : 2;
```

With logic :

```
const isTrue = true;
const result = isTrue ? 0 : 1;
```

Resources

- Ternary Operator

Truthy & Falsey

```
const products = [];
const product = products[0];

// Instead of having to explity compare values, Jaascript will try and inffer it for you
// e.g. comparrison bool example
if (product !== undefined) {
    console.log('Found');
} else {
    console.log('Undefined');
}
```

Truthy - Values that resolves to true in a boolean context
Boolean example

```
if ('truthy') {
    console.log('valid string = truthy value');
}

if ({}) {
    console.log('{} = truthy value');
}

if ([]) {
    console.log('[] = truthy value');
}

if (true) {
    console.log('true = truthy value');
}
```

Falsy - Values that resolves to false in a boolean context. Falsy values are: false, 0, ”, null or defined

```
if (undefined) { }
else {
    console.log('undefined = falsy value');
}

if ('')  { }
else {
    console.log('string empty = falsy value');
}

if (null) { }
else {
    console.log('null = falsy value');
```

```
}

if (NaN) { }
else {
    console.log('Nan = falsy value');
}

if (false) { }
else {
    console.log('false = falsy value');
}
```

Functions 101

Empty Function

```
function SimpleFunction() {
}
SimpleFunction();
```

Basic example with an input and a output

```
function MyFunction(input) {
    console.log(input);

    const output = 'Output';
    return output;
}

MyFunction('Input');
console.log(MyFunction('Input'));
```

Assign a function to a variable

```
const square = function (x) {
    return x * x;
}
console.log(square(3));
```

Resources

- Functions

Functions Scope

Example of how scope works with functions

```
const convertToFahenheitToVelsuis = function(fahenheit) {
    let celsius = (fahenheit - 32) * 5 / 9;
```

```
    if (celsius <= 0) {
        const isFreezing = true;
    }

    try {

        // This will throw an exception
        // Scope within functions work the same as in conditional statements
        console.log(isFreezing);
    } catch(e) {}

    return celsius;
};

try {

    // Functions create local scope
    // Variables created within that scope are bound inside it and are not globally accessal
    // This will thrown an exception
    console.log(celsius);
} catch(e) {}

const result = convertToFahenheitToVelsuis(32);
console.log(`result=${result}`);
```

Resources

- Functions

Functions With Arguments

Passing multiple parameters into a function

```
const multi = function(a, b) {
    return a + b;
}
console.log(multi(1, 2));
```

Null Arguments Passed in

```
const noValues = function(a, b) {
    return a + b;
}

if (isNaN(noValues()) ) {
    console.log('NaN returned when no parameters provided');
}
```

Default arguments - prevents errors occuring from wrong parameters being passed into a function

```
const defaultParams = function(a = 0, b = 0) {
    return a + b;
}
console.log(defaultParams());
```

Resources

- Functions
- Default Parameters

Arrow Functions

(todo)

Resources

- Arrow Functions

Objects 101

Objets are a way of grouping one or more related properties together

Basic empty object

```
const myObject = {};

// Objects with basic properties
const myObjectWithProps = {
    firstName: 'jon',
    surname: 'jones'
};
```

Using an object

```
console.log(`${myObjectWithProps.firstName} ${myObjectWithProps.surname}`);
```

Updating a property in an object

```
myObjectWithProps.firstName = 'change';
console.log(`Changed Value=${myObjectWithProps.firstName}`);
```

Resources

- Objects
- Object Initializer

Objects And Functions

Dummy Objects Set-up

```
const bookOne = {
    title: '1984',
    author: 'Orwell',
    pageCount: 326
};
```

```
const bookTwo = {
    title: 'Code Complete',
    author: 'Steve McConnell',
    pageCount: 960
};
```

Passing an object into a function

```
const passInFunction = function(book) {
    console.log(`title=${book.title} author=${book.author} pageCount=${book.pageCount}`);
};

passInFunction(bookOne);
passInFunction(bookTwo);
console.log();
```

Returning an object in a function

```
const passOutFunction = function(book) {
    return bookOne;
};

const result = passOutFunction();

console.log(`title=${result.title} author=${result.author} pageCount=${result.pageCount}`);
console.log();
```

Example

```
const convertFahrenheit = function(fahrenheit) {
    return {
        fahrenheit: fahrenheit,
        kelvin: (fahrenheit + 459.67) * (5 / 9),
        celsius: (fahrenheit - 32) * (5 / 9)
    };
};

const convertRsult = convertFahrenheit(45);
console.log(`fahrenheit=${convertRsult.fahrenheit}`);
console.log(`kelvin=${convertRsult.kelvin}`);
console.log(`celsius=${convertRsult.celsius}`);
console.log();
```

Resources

- Objects

- Object Initializer

Objects As Arguments

```
const myAccount = {
    title: 'jon',
    expenses: 0
};

console.log('Example updating passed in variables');

const addExpense = function(account, amount) {
    account.expenses = account.expenses + amount;
    console.log('Object within function', account);
}
```

Original object gets updated. Objects passed into function are done via pointers. Passed in objects DO NOT get created as new objects. Updating one, updates both.

```
addExpense(myAccount, 5);
console.log('Original object', myAccount);
console.log('Passed in objects use pointers, so updating one, updates both');
console.log();
```

Example

```
console.log('Re-assinging the passed in object, what gets updated?');
const accountOne = {
    title: 'jon',
    expenses: 0
};

const passInAndUpdateFunction = function(myAccount) {
    myAccount = {};
    console.log('Object within function', myAccount);
};
```

Scope still works within functions. If you re-assign an object, the object pointer will now be udpated to pass to a new area in memory. The orginal object data will still exist, rather than being overridden

```
passInAndUpdateFunction(accountOne);
console.log('Original object', accountOne);
console.log('Reassinging the passed in object changes the pointer to a new area in memory');
console.log('Reassinging does not override the original object');
console.log();
```

Example

```
console.log('Updating the returned object and comparing it to the original value, what gets
const accountTwo = {
    title: 'jon',
};

const passOutFunction = function(account) {
    account.title = 'Updated';
    return account;
};

let result = passOutFunction(accountTwo);
```

A returned object, creates a new object. Updating the new object does now override the existing object

```
console.log('Returned object re-assigned', result);
console.log('Returned object re-assigned', accountTwo);
console.log('Returned object use pointers, so both objects update');
console.log();
```

Re-assigning the returned value does not override the original object

```
result = {
    title: 'new',
};

console.log('Reassinging passed out object, what happens?');
console.log('Returned object re-assigned', result);
console.log('Returned object re-assigned', accountTwo);
console.log('Reassinging does not override the original object');
console.log();
```

Resources

- Objects
- Object Initializer

Adding To An Object

```
const exampleOne = {
    name: 'Appetite for Destruction',
    releaeYear: 1987,
    artist: 'Guns N Roses',
    amountInStock: 2,

    // Defining a method
```

```
    checkAvailability: function(totalItemsRequested) {

        // This method is just a function within a object
        console.log(totalItemsRequested);
        return true;
    }
};
```

Calling the method is the same as calling a property within an object. You can pass in paramteres and get return values

```
const exampleOneResult = exampleOne.checkAvailability(4);
console.log(exampleOneResult);
```

THIS

```
const exampleTwo = {
    name: 'Appetite for Destruction',
    releaeYear: 1987,
    artist: 'Guns N Roses',
    amountInStock: 2,

    whatIsThis: function() {
        // the 'this' keyword is a special keyword within Javscript that helps you manage s
        // using this' will allow you to access the objects proeprties within your methods
        console.log(this);
    }
};
exampleTwo.whatIsThis();
```

Using the ‘this’ keyword to access object properties within a method

```
const exampleThree = {
    name: 'Appetite for Destruction',
    releaeYear: 1987,
    artist: 'Guns N Roses',
    amountInStock: 2,

    // Defining a method
    checkAvailability: function(totalItemsRequested) {

        // Using this to access child object properties
        return this.amountInStock >= totalItemsRequested;
    }
};

console.log(exampleThree.checkAvailability(3));
```

Resources

- Objects
- Object Initializer

Arrays 101

Example arrays with different types of data

```
const emptyArray = [];
const numberArray = [ 1, 2, 3, 4 ];
const charArray = [ 'a', 'b', 'c', 'd' ];
```

Display array

```
const notes = [ 'note1', 'note2', 'note3'];
console.log('Note array', notes);
```

Get the number of items in the array

```
const notes = [ 'note1', 'note2', 'note3'];
console.log(`Notes length = ${notes.length}`);
```

Display the first item in the array

```
const notes = [ 'note1', 'note2', 'note3'];
console.log(`First Item = ${notes[0]}`);
```

Dynamically decide what item to display in the array, using an operation to determine item place

```
const notes = [ 'note1', 'note2', 'note3'];
console.log(`First Item = = ${notes[notes.length - 3]}`);
```

Undefined items result in an undefined state

```
const notes = [ 'note1', 'note2', 'note3'];
console.log(`Undefinied Item = ${notes[100]}`);
```

Resources

- Array

Adding And Removing items from an array

Adding a note to end of array:

```
const notes = [ 'note1', 'note2', 'note3'];

notes.push('note 4');
console.log('Note array', notes);
```

Remove last node

```
const notes = [ 'note1', 'note2', 'note3'];

const poppedNote = notes.pop();
console.log('Note array', notes);
console.log('Popped Note', poppedNote);
```

Removing first node

```
const notes = [ 'note1', 'note2', 'note3'];

const shiftedNote = notes.shift();
console.log('Note array after shift', notes);
console.log('Shifted Note', shiftedNote);
```

Adding an item to the beginning of the array - unshift

```
const notes = [ 'note1', 'note2', 'note3'];

notes.unshift(shiftedNote);
console.log('Note array after unshift', notes);
```

Getting an item from an array without changing it

```
const notes = [ 'note1', 'note2', 'note3'];

const slicedItem = notes.slice(1, 2);
console.log('Note array doesnt change after slice', notes);
console.log('Slaced() Item = ', slicedItem);
```

Removing an item from a given position in the original array

```
const notes = [ 'note1', 'note2', 'note3'];

const splicedItem = notes.splice(1, 1);
console.log('Note array is changed after splice()', notes);
console.log('Spliaced() Item = ', splicedItem);
```

Removing an item from a given position in the original array

```
const notes = [ 'note1', 'note2', 'note3'];

const splicedItemAdd = notes.splice(1, 1, 'note 2', 'note 3');
console.log('Note array after splice() with an add', notes);
```

Resources

- Array

Arrays - Foreach()

iterating through a collection with a callback

```
const notes = [ 'note1', 'note2', 'note3'];
notes.forEach(function(note) {
    console.log(note);
});
console.log();
```

Iterating through a collection with an arrow function

```
const notes = [ 'note1', 'note2', 'note3'];
notes.forEach((note) => {
    console.log(note);
});
console.log();
```

Iterating through a collection with an arrow function

```
const notes = [ 'note1', 'note2', 'note3'];
notes.forEach((note, index) => {
    console.log(`${note} is in position ${index}`);
});
console.log();
```

For()

Basic Example

```
const notes = [ 'note1', 'note2', 'note3'];
for(const positionNumber in notes) {
    console.log(`${notes[positionNumber]} is in position ${positionNumber}`);
}
console.log();
```

For Loop iterating upwards

```
const notes = [ 'note1', 'note2', 'note3'];
for(let count = 0; count < notes.length; count++) {
    console.log(`${notes[count]} is in position ${count}`);
}
console.log();
```

For Loop in reverse order -> iterating backwards

```
const notes = [ 'note1', 'note2', 'note3'];
for(let count = (notes.length - 1); count >= 0; count--) {
    console.log(`${notes[count]} is in position ${count}`);
}
console.log();
```

Resources

- Array
- ForEach()

Map() and Reduce()

(todo)

Resources

- Map, filter and reduce
- Map
- Reduce()
- ReduceRight()

Filter()

Filtering items out of an array

```
const notes = [{
    title: 'note1',
    body: 'body1'},
    {title: 'note2',
    body: 'body2'},
    {title: 'note3',
    body: 'body3'}];

const filteredResult = notes.filter((note) => {
    return note.title.toLowerCase() === query.toLowerCase();
```

```
});

console.log(filteredResults(notes, 'note3'));
```

Searching for an item in an array

```
const notes = [{
    title: 'note1',
    body: 'body1'},
    {title: 'note2',
    body: 'body2'},
    {title: 'note3',
    body: 'body3'}];

// Searching a valid item - will return the index number the match exists
console.log('Basic search with matching item = ', notes.indexOf('note2'));

// Searching an item - will return -1 for non-matching result
console.log('Basic search with non-matching item = ', notes.indexOf('note5'));

//Compaing Objects
const isEqual = {} === {};
console.log('Are two empty objects equal ', isEqual);

const emptyObjectArray = [{}];
console.log('IndexOf on an empty object will return -1= ', notes.indexOf({}));

// Objects are compared if it's the same obejct in memory, rather than  if two objects conte
// Compare two different objects then will always return false, as both obejcts will live i

const index = notes.findIndex((note, index) => {
    return note.title === 'note2'
});

console.log('findIndex on a match = ', index);
```

Resources

- Array
- Filter()

ES6 Features

Rest & Spread

You can pass in dynamic variables and deal with it using ... If you want specific named parameters put them first, otherwise they will be included in the list

```
const sum = (type, ...numbers) => {
    let sum = 0;
    numbers.forEach((num) => sum += num);
    const average = sum / numbers.length;

    return `the average of ${type} is ${average}`
}

console.log(sum('thing', 1,2,3,4,5,6));
console.log(sum('thing', 100,200,300,400));
```

The spread works in an opposite way using ... when you pass in data to a function, will flatten it and pass it as a flat instance in print data we grab the first two passed in as a named value and then print it out. This works when you use the ...

```
const printData = (data, type, letterA, letterB) => {
    console.log(data);
    console.log(data);
    console.log(letterA);
    console.log(letterB);
};

const data = {
    name: 'name',
    type: 'type',
    stuff: ['a','b','c','d','e','f','g']
};

printData(data.name, data.type, ...data.stuff);
```

You can create a new array with old data easily with the spread

```
const cloneOfStuff = ['1', '2', ...data.stuff];
console.log('clone', cloneOfStuff);
```

Also works with objects

```
let myObject = {
    exampleOne: 2,
    exampleTwo: 2,
}
```

This will create a clone

```
let newObject = {
    ...myObject
};

console.log(myObject);
console.log(newObject);
```

If I update myObject now, the new object will not get it

```
myObject.exampleOne = 'updaed';
```

This will print the same

```
console.log(newObject);

let secondObject = {
    exampleThree: 3,
    exampleFour: 4,
}

let combineObject = {
    ...myObject,
    ...secondObject
};

console.log(combineObject);
```

Destructing

```
const numbers = [1, 2, 3, 4, 5];
const [ first, second, ...others ] = numbers;
console.log('first', first);
console.log('second', second);
console.log('others', others);
```

Destructuring

(todo)

Resources

- Destructuring

DOM Manipulation

Selection

Querying DOM for the first element match it finds

```
const p = document.querySelector('p');
p.textContent = h1.textContent + '.';
```

Querying DOM for the first class match it finds

```
const myClass = document.querySelector('.my-class');
myClass.textContent = h1.textContent + '.';
```

Querying DOM for the first id match it finds

```
const myId = document.querySelector('#my-id');
myId.textContent = h1.textContent + '.';
```

Querying DOM for multiple elements

```
const divs = document.querySelectorAll('div');
divs.forEach(function (div) {
    p.textContent = 'some content';
})
```

Removing an element from DOM

```
const h1 = document.querySelector('.heading');
h1.remove();
```

Local Storage

Read item from local storage

```
localStorage.getItem('key');
```

Storing to local storage

```
localStorage.setItemm('key', 'value');
```

Removing item from local storage

```
localStorage.removeItem('key', 'value');
```

Editing item in local storage

```
localStorage.setItem('key', 'updating');
```

Clear everything

```
localStorage.clear();
```

Watch changes in local storage to sync browser tabs

```
window.addEventListener('storage', function (e) {
})
```

Event Listeners

Event Listener - Abort - fired when the loading of a resource has been aborted.

```
document.querySelector('button').addEventListener('abort', (e) => {
    console.log(e);
});
```

Event Listener

```
document.querySelector('button').addEventListener('beforeinput', (e) => {
    console.log(e);
});
```

Event Listener - blur - fired when an element has lost focus. The main difference between this event and focusout is that only the latter bubbles.

```
document.querySelector('button').addEventListener('blur', (e) => {
    console.log(e);
});
```

Event Listener - click - fired when a pointing device button (usually a mouse's primary button) is pressed and released on a single element

```
```javascript
document.querySelector('button').addEventListener('click', (e) => {
    console.log(e);
});
```
```

Event Listener - compositionstart - fired when the composition of a passage of text is prepared (fires with special characters that require a sequence of keys and other inputs such as speech recognition or word suggestion on mobile).

```
document.querySelector('button').addEventListener('compositionstart', (e) => {
    console.log(e);
});
```

Event Listener - compositionupdate - fired when a character is added to a passage of text being composed (fires with special characters that require a sequence of keys and other inputs such as speech recognition or word suggestion on mobile).

```
document.querySelector('button').addEventListener('compositionupdate', (e) => {
    console.log(e);
});
```

Event Listener - compositionend - fired when the composition of a passage of text has been completed or cancelled (fires with special characters that require a sequence of keys and other inputs such as speech recognition or word suggestion on mobile).

```
document.querySelector('button').addEventListener('compositionend', (e) => {
    console.log(e);
});
```

Event Listener - dblclick - fired when a pointing device button is clicked twice on a single element

```
document.querySelector('button').addEventListener('dblclick', (e) => {
    console.log(e);
});
```

Event Listener - error - fired when an error occurred; the exact circumstances vary, events by this name are used from a variety of APIs.

```
document.querySelector('button').addEventListener('error', (e) => {
    console.log(e);
});
```

Event Listener - focus - fired when an element has received focus. The main difference between this event and focusin is that only the latter bubbles

```
document.querySelector('button').addEventListener('focus', (e) => {
    console.log(e);
});
```

Event Listener - focusin - fired when an element is about to receive focus. The main difference between this event and focus is that the latter doesn't bubble

```
document.querySelector('button').addEventListener('focusin', (e) => {
    console.log(e);
});
```

Event Listener - focusout - fired when an element is about to lose focus. The main difference between this event and blur is that the latter doesn't bubble

```
document.querySelector('button').addEventListener('focusout', (e) => {
    console.log(e);
});
```

```
Event Listener - input -  fired synchronously when the value of an <input>, <select>, or <te
```

```javascript
document.querySelector('button').addEventListener('input', (e) => {
    console.log(e);
});
```

Event Listener - keydown - fired when a key is pressed down. Unlike the keypress event, the keydown event is fired for keys that produce a character value and for keys that do not produce a character value

```
document.querySelector('button').addEventListener('keydown', (e) => {
    console.log(e);
});
```

Event Listener - keypress - fired when a key that produces a character value is pressed down. Examples of keys that produce a character value are alphabetic, numeric, and punctuation keys

```
document.querySelector('button').addEventListener('keypress', (e) => {
    console.log(e);
});
```

Event Listener - keyup - fired when a key is released

```
document.querySelector('button').addEventListener('keyup', (e) => {
    console.log(e);
});
```

Event Listener - load - fired when a resource and its dependent resources have finished loading

```
document.querySelector('button').addEventListener('load', (e) => {
    console.log(e);
});
```

Event Listener - mousedown - fired when a pointing device button is pressed on an element.

```
document.querySelector('button').addEventListener('mousedown', (e) => {
    console.log(e);
});
```

Event Listener - mouseenter - fired when a pointing device (usually a mouse) is moved over the element that has the listener attached

```
document.querySelector('button').addEventListener('mouseenter', (e) => {
    console.log(e);
});
```

Event Listener - mouseleave - fired when the pointer of a pointing device (usually a mouse) is moved out of an element that has the listener attached to it

```
document.querySelector('button').addEventListener('mouseleave', (e) => {
    console.log(e);
});
```

Event Listener - mousemove - fired when a pointing device (usually a mouse) is moved while over an element

```
document.querySelector('button').addEventListener('mousemove', (e) => {
    console.log(e);
});
```

Event Listener - mouseout - fired when a pointing device (usually a mouse) is moved off the element that has the listener attached or off one of its children

```
document.querySelector('button').addEventListener('mouseout', (e) => {
    console.log(e);
});
```

Event Listener - mouseover - fired when a pointing device is moved onto the element that has the listener attached or onto one of its children

```
document.querySelector('button').addEventListener('mouseover', (e) => {
    console.log(e);
});
```

Event Listener - mouseup - fired when a pointing device button is released over an element

```
document.querySelector('button').addEventListener('mouseup', (e) => {
    console.log(e);
});
```

Event Listener - resize - fired when the document view has been resized

```
document.querySelector('button').addEventListener('resize', (e) => {
    console.log(e);
});
```

Event Listener - scroll - fired when the document view or an element has been scrolled

```
document.querySelector('button').addEventListener('scroll', (e) => {
    console.log(e);
});
```

Event Listener - select - fired when some text is being selected

```
document.querySelector('button').addEventListener('select', (e) => {
    console.log(e);
});
```

Event Listener - unload - fired when the document or a child resource is being unloaded

```
document.querySelector('button').addEventListener('unload', (e) => {
    console.log(e);
});
```

Event Listener - wheel - fired when a wheel button of a pointing device (usually a mouse) is rotated

```
document.querySelector('button').addEventListener('wheel', (e) => {
    console.log(e);
});
```

Insertation

Adding an new element dynamically to the page

```
const newElement = document.createElement('p');
document.querySelector('body').appendChild(newElement);
```

Async 101

Promises and Callbacks

A promise is a more modern way of dealing with callbacks, using a promise will result in clean and more easy to understand code

Callback example

```
const myCallback = (callback) => {
    setTimeout(() => {
        callback(undefined, 'callback');

        // This accidentl logic issues can occur in callbacks
        // In this example I only want the callback to return once, however, there is no wa
        callback(undefined, 'callback');
    }, 2000);
};
```

Handling a callback

```
myCallback((err, data) => {
    if (err) {
        // Do sad path task
    } else {
        console.log(data);
    };
});
```

Promises - A better approach

```
const myPromise = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve('promise');
    }, 2000);
```

```
    // After resolve is called, the method is finished.
    // In this code reject will never be run
    // Promises gurantee that only one happy path or one unhappypath bit of logic will be co
    setTimeout(() => {
        reject('reject promise');
        reject('this will never be called');
    }, 2000);
});
```

Calling the promise

```
myPromise.then((data) => {
    console.log(data);
}).catch((e) => {
    console.log(e);
});
```

```
const myPromiseWithParameters = (data) => {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(data);
        }, 2000);
    });
};
```

Calling the promise

```
const example = myPromiseWithParameters('print this text');
example.then((data) => {
    console.log(data);
});
```

Resources

- [Async]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function
- Async/Await
- Promises

Uses nodes XHR2 - in a webpage you just use XMLHttpRequest() without importing a library

```
var XMLHttpRequest = require("xhr2");
const basicExample = new XMLHttpRequest();

basicExample.addEventListener('readystatechange', (e) => {
```

```
    if (e.target.readyState === 4) {
        const data = JSON.parse(e.target.responseText)
        data.forEach((element) => {
            console.log(element.name);
        });
    }
});
```

Checking response correct

```
const checkingStatusExample = new XMLHttpRequest();
checkingStatusExample.addEventListener('readystatechange', (e) => {

    // readytate return the state of an xml http request
    // 0 = OPENE
    // 2 = HEADERS_RECEIVED
    // 3 = LOADING
    // 4 = DONE - This is the one we want to call to do something meaningful withthe returne
    if (e.target.readyState === 4 && e.target.status === 200) {
        console.log('success');
    }

    // To check for a For a failed request, you still need to check for 4, minus the staus c
    // If you do not do this the code will return for all state changes, open , loading, hea
    else if (e.target.readyState === 4) {
        console.log('failed');
    };
});
```

Calling the API

```
const apiURL = 'https://restcountries.eu/rest/v2/name/united';
basicExample.open('GET', apiURL);
basicExample.send();

checkingStatusExample.open('GET', apiURL);
checkingStatusExample.send();
```

Resources

- [Async]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function
- Async/Await

Await

To make a function async.. stick the async keyword before it, when you use async it will return a promise

```
const processEmptyData = async () => {
};

const returnValue = processEmptyData();
console.log(returnValue); // this will return promise ( undefinied )
```

Returning data

```
const processDataAsync = async (shouldThrow) => {
    if (shouldThrow) {
        throw 'Error';
    }
    return 12;
};

processDataAsync(false).then((data) => {
    console.log(data);
});

processDataAsync(true).catch((e) => {
    console.log(e);
});
```

This uses async code as expected

```
const processDataWithPromise = (num) => new Promise((resolve, reject) =>  {
    setTimeout(() => {
        resolve(num);
    }, 2000);
});
```

Async await allows you to run your code in a single threaded way the second await statement will not run until the first one has completed this prevents you from having to use then and catch chaining when you want to await

```
const processDataWithAwait = async (text) => {
    let result = await processDataWithPromise(text);
    result = await processDataWithPromise(result + result);
    return result;
}

processDataWithAwait('text').then((data) => {
    console.log(data);
});
```

Resources

- [Async]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function
- Async/Await

Callbacks

Javascript is single treaded. This means that when you make async requests Javascript will just plow on and process statements rather than waiting. Running the code below will display 2 then 1.

```
setTimeout(() => {
 console.log('1');
}, 1);

console.log('2');
```

Console.log will output 2 and then 1 in this order. To prevent these issues from single thread execution from occuring. You shoudl use callback functions, or, Promises to ensure that your code gets called when you want it to

```
const callbackOne = (callback) => {
    console.log('C1');
    callback();
};

const callbackTwo = () => {
    console.log('C2');
};

setTimeout(() => {
    callbackOne(callbackTwo); }, 1);
```

This now renders the numbers in correct sequential order

Resources

- Async
- Async/Await

Fetch

Fetch is a feature that comes with es6. If you run this script with node then you will need to import isomorphic-fetch to get this to work. If you are running this within a browser you do no need to include it

```
var fetch = require("isomorphic-fetch");
var XMLHttpRequest = require("xhr2");

const apiURL = 'https://restcountries.eu/rest/v2/name/united';

fetch(apiURL, {}) {
    .then((response) => {
        console.log(response);
    })
    .catch((e) => {
        console.log(e);
    });
}
```

Fetch is better than doing it the old way like

```
const xmlRequest = new XMLHttpRequest();
xmlRequest.addEventListener('readystatechange', (e) => {
    if (e.target.readyState === 4) {
        console.log('Data');
    }
});

xmlRequest.open('GET', apiURL);
xmlRequest.send();
```

Resources

- [Async]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function
- Async/Await

Promise Chaining

If you want to chain your callbacks (like a recursive function call. Using callback things are ahrd to follow and hard to maintain, this is called callback hell

```
const getCallBack = (num, callback) => {
    setTimeout(() => {
        if (typeof num === 'number') {
            callback(undefined, num * 2);
        } else {
            callback('NUmber must be provided');
        }
    });
};
```

```
getCallBack(2, (err, data) => {
    if (err) {
        console.log(err);
    } else {
        getCallBack(data, (err, data) => {
            console.log(data);
        });
    }
});
```

If this was to become more complex, the code will get harder and harder to run. Promsise chaining fixes this

```
const getPromise = (num) => new Promise((resolve, reject) => {
    setTimeout(() => {
        typeof num === 'number' ? resolve(num * 2) : reject('Number must be provided');
    }, 2000);
});

getPromise(2).then((data) => {
    // return the promise.  This creates the promise chain
    return getPromise(data);
    })
    .then((data) => {
        // log the data
        console.log(data);
    })
    // Catch() is the methodto deal with
    .catch((e) => {
        // deal with any issues
        console.log(e);
    });
```

This snippet of code is much easier to read, modify and maintain. Chaining promise allows us to create complicated sync code, simple!

Resources

- [Async]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function
- Async/Await
- Promises

Prototype And Lexical Scoping

Closures and lexical scoping

```
let globalContextDecalaredVariable = 2
function multiplyThis(n) {
    // Within a local function scope you have access to globally scoped variables
    let functionContextDeclaredVariable = n * globalContextDecalaredVariable
    return functionContextDeclaredVariable
};
```

Closure example, counter is created in the global context

```
function createCounter() {
    let counter = 0; // counter is created in a local scoped context
    const myFunction = function() { // myFunction is locally scoped.
        counter = counter + 1 //
        return counter
    }
    return myFunction // local execution context ends, myFunction and counter no longer exis
};

// How you may expect it to work
const increment = createCounter();
```

Global scoped, contains a function definition returned by createCounter(). It is no longer labeled myFunction, it is labeled increment within the global context

```
const c1 = increment();
```

Create a new execution context, try to look up counter in local execution context, it does not exist. Look for counter in global execution context, it does not exist. Javascript will evaluate the code as counter = undefined + 1 and declare a new local variable labeled 'counter' with the number 1. Like truthy and falsey comparisons, undefined is sort of 0, the value 1 gets returned here

```
const c2 = increment(); // repeat above, 1 gets outputted

console.log('example increment', c1, c2); // this console log, outputs 1 and 2.
```

Even though execution has ended and counter is killed, the enginge somehow still gets access to counter, is counter in global sopce, local scope? Somehow it can still access the state of counter, how?

Whenever you declare a new function and assign it to a variable, you store the function definition and a CLOSURE. The closure contains all the variables that are in scope at the time of creation of the function.

```
function createCounterSecondTime() {
    let counter = 0; // counter is created in a local scoped context

    // myFunction is assigin to a function, so a closure is created
    // The closure (counter = 0) is included as part of the function definition
    const myFunction = function() {
        counter = counter + 1
        return counter;
    }
    return myFunction;  // returning the function definition and its closure()
};

const myVariable2 = createCounterSecondTime(); // This creates a closure, e.g. any vars and
backpack and returned with the function definition
```

In this example a var 'counter' is added to the backpack with a value of 0. When myVariable2() is called its a function definition with a closure, counter = 0. When the function runs because of lexial scoping before it tries to find counter in the local execution context, or, global execution context. Javascript looks in the backpack for a closure called 'counter'. In this situation 'counter' is set to 0, the result of the function is 1.

```
const closure1 = myVariable2();  // This will return 1, the backpack closure value 'counter
const closure2 = myVariable2(); // When this runs, the function will look in the backpack f
```

The key to remember is that when a function gets declared, it contains a function definition and a closure. The closure is a collection of all the variables in scope at the time of creation of the function.

Private variables using closures by default Javascript does not provide a private keyword

```
const privatePropertyExample = function() {
    let count = 0;

    return {
        increment() {
            count++;
```

```
        }, decrement() {
            count--;
        }, get() {
            return count;
        }
    }
}

const exampleOne = privatePropertyExample();
exampleOne.increment();
console.log(exampleOne.get());

// Throws undefinied as we don not have access to count
console.log(exampleOne.count);
```

Classes & Subclasses

Instead of having to use constructor functions you can use class. Using class is syntax sugar, under the hood it works exactly the same constructor functions. Using class results in cleaner code

```
class MyClass {
    constructor(argument) {
        this.argument = argument;
    };

    myMethod() {
        console.log('Hi');
    };

    myOverride() {
        console.log('MyClass');
    };
};
```

Use the keyword modifier extends to create a sub class of a class:

```
class MySubClass extends MyClass{
    constructor(argument) {
        super(argument); // Need to call super in sub class to call the base class construct
    };

    myOverride() {
        console.log('MySubClass');
    };
```

```
    getBaseProperty() {
        console.log(this.argument);
    };
};

const myClass = new MyClass('1');
myClass.myMethod();
myClass.myOverride();

const mySubClass = new MySubClass('example text');
mySubClass.myMethod();
mySubClass.myOverride();
mySubClass.getBaseProperty();
```

Resources

- Classes
- Constructor
- New
- Prototype

Get & Set

You can use getters and setters to help you provide better encapsulation within your objects. Using this appraoch means you can define private variables that only the object itself can modify. This is a lot better than making instance variables public, as outside classes shouldn't be able to directly manipulate state within your classes

```
const myObject = {

    innerProperty: 'myProperty',

    // Defining a getter
    get myProperty() {
        return this.innerProperty;
    },

    // Defining a setter
    set myProperty(value) {
        console.log(value);
        this.innerProperty = value;
    }
};

// Getting
```

```
console.log(myObject.myProperty);
```

Setting -> will do a console.log()

```
myObject.myProperty = 'setting';
console.log(myObject.myProperty);
console.log(myObject.innerProperty);
```

hasOwnProperty()

Object is a reference to the global JS. Adding to object will add a method on all objects. Object.prototype.someNewMethod = () => 'blah';. As we attached some method to the JS object class, everything else will be able to access it. This is not something I recommend

```
const different = {};
console.log(different.someNewMethod());

const object = new Object();
```

Calling `__proto__` will display the global objects prototypes, so in our case someNewMethod():

```
console.log('objects prototypes', object.__proto__);
```

Object does not have a prototype as it's the bae object. Trying to view Objects prototype will be null:

```
console.log('objects prototype prototype will be null', object.__proto__.__proto__);
```

hasOwnProperty() on the object will be false as someNewMethod is declared on the prototype:

```
console.log(object.hasOwnProperty('someNewMethod'));
```

hasOwnProperty() on the prototype will be true as someNewMethod is declared on the prototype:

```
console.log(object.__proto__.hasOwnProperty('someNewMethod'));

try {
    object.prototype.hasOwnProperty('someNewMethod');
} catch (e) {
    console.log('This will throw as you cant use prototype to access instance prototypes');
}
```

Custom Object

```
const MyConstructorFunction = function () {
};
```

```
const person = new MyConstructorFunction();
console.log('Calling __proto__.__proto__ on a declared object will show the global js object
console.log('Calling __proto__.__proto__.__proto__ as the global object is always base', per
```

Prototype Chain

int

```
const int = 9;
console.log('int => ', int);
console.log(`${typeof int} =>`, int.__proto__);
console.log(`${typeof int.__proto__} =>`, int.__proto__.__proto__);
```

number

```
const number = new Number(1);;
console.log('number => ', number);
console.log(`${typeof number} =>`, number.__proto__);
console.log(`${typeof number.__proto__} =>`, number.__proto__.__proto__);
```

bool

```
const bool = true;
console.log('bool => ', bool);
console.log(`${typeof bool} =>`, bool.__proto__);
console.log(`${typeof bool.__proto__} =>`, bool.__proto__.__proto__);
```

string

```
const stringExample = 'ji';
console.log('string => ', stringExample);
console.log(`${typeof stringExample} =>`, stringExample.__proto__);
console.log(`${typeof stringExample.__proto__} =>`, stringExample.__proto__.__proto__);
```

null

```
try {
    const nullExample = null;
    console.log('null => ', nullExample);
    console.log(`${typeof nullExample} =>`, nullExample.__proto__);
} catch(e) {
    console.log('null has no prototype chain');
}
```

undefined

```
try {
    const undefinedExample = undefined;
```

```
    console.log('undefined => ', undefinedExample);
    console.log(`${typeof undefinedExample} =>`, undefinedExample.__proto__);
} catch(e) {
    console.log('undefined has no prototype chain');
}
```

array

```
const array = [];
console.log('array => ', array);
console.log(`${typeof array} =>`, array.__proto__);
```

object

```
const object = {};
console.log('object => ', object);
console.log(`${typeof object} =>`, object.__proto__);
```

Prototype Declaration

```
const MyConstructorFunction = function () {
};

const myObject = new MyConstructorFunction();
```

Example() does not exist on creation and throws exception

```
try {
    myObject.Example();
} catch (e) {
    console.log('Example is not definied');
}

MyConstructorFunction.prototype.Example = function () {
    console.log(`Example`);
};
```

Example() now exists and the call runs... even though the object has already been created. Declaring a prototype function/property after object declaration still get associated. Any change made to the prototype at any time of execution will be made to everything:

```
myObject.Example();
```

You can define instance methods/properties to specific objects. The method decoration below, will only be available to this object :

```
myObject.InstanceMethod = function() {
    console.log('InstanceMethod');
};
```

This will work:

```
myObject.InstanceMethod();

try {
    const mySecondObject = new MyConstructorFunction();
    mySecondObject.InstanceMethod();
} catch (e) {
    console.log('This will not work.  Function assocated to instance not the prototype');
}
```

Overriding prototype behavior is also possible:

```
myObject.Example = function() {
    console.log(`Overriding prototype method on instance`);
}
```

This call will now use the instance method declared above. This will not use the global prototype version. In this way you can customize classes based on your needs.

```
myObject.Example();
```

Resources

- Prototype

Prototype Inheritance 101

Constructor function need to use a regular function, instead of an arrow function arrow functions do not bind this

```
const MyConstructorFunction = function (someText) {
    this.someText = someText;
    this.myArray = ['one', 'two'];
};
```

Attaching a method to a constructor function. Attaching methods is done using prototype inheritance. You can attach things to the prototype and then access them using 'this':

```
MyConstructorFunction.prototype.getText = function () {
    return `Using prototype and this, I can access the functions properties = ${this.someTex
};
```

Defining global static text

```
MyConstructorFunction.prototype.globalText = 'Global Text';
```

You can override base properties

```
MyConstructorFunction.prototype.overrideBaseProperties = function(moreText) {
    this.someText = moreText;
};
```

You should use arrow functions to iterate through items in an array arrow functions do not bind this and will use the correct inheritance scope. Create a new function, using function will rebind 'this' and stop you getting access to the prototype properties

```
MyConstructorFunction.prototype.scopeBindingExample = function(moreText) {

    try {
        this.myArray.forEach(function(item) {
            console.log(`prototype property someText = ${this.someText} when decalred in nor
        });
    } catch (e) {
        console.log('Exception thrown when accessing someText, a normal function definition
    }

    this.myArray.forEach( (item) => {
        console.log(`someText =  ${this.someText} and is available because the array was ite
    });
};
```

Invalid way to call a constructor function results in undefined as the new keyword was omitted

```
try {
    const myFunction = MyConstructorFunction();
console.log(myFunction);
} catch(e) {}
```

Correct way to call a constructor function need to use 'new' keyword. Functions that you want to be objects should start with an uppercase letter

```
const myObject = new MyConstructorFunction('text');
console.log(myObject.someText);
console.log(myObject.getText());
console.log(myObject.globalText);

myObject.overrideBaseProperties('new text');
console.log(myObject.getText());

console.log(myObject.scopeBindingExample());
```

Resources

- Prototype

Public & Private Properties

Private variables using closures by default Javascript does not provide a private keyword

```
class PrivateAndPublicProperty {
    constructor(name) {
        const _count = 1  // this is private
        this.count = 2; // this is public
    };
    whatCanIAccess() {
        console.log('whatCanIAccess count = ', this.count); // this will have a value
        console.log('whatCanIAccess _count = ', this._count); // will be undefinied
    }
}
const exampleTwo = new PrivateAndPublicProperty();
```

The constructor scope is private so _count is private using this will make it public

```
console.log('example two accessing _count ', exampleTwo._count);
console.log('example two accessing count ', exampleTwo.count);

exampleTwo.whatCanIAccess();
```

Private variables using closures by default Javascript does not provide a private keyword

```
const privatePropertyExample = function() {
    let count = 0;

    return {
        increment() {
            count++;
        }, decrement() {
            count--;
        }, get() {
            return count;
        }
    }
}

const exampleOne = privatePropertyExample();
```

```
exampleOne.increment();
console.log(exampleOne.get());
```

Lexical Scoping

Global scope -> ANY VARIABLES DEFINIED OUTSIDE OF A CODE BLOCK
WILL BE ADDED TO THE GLOBAL SCOPE
Local scope -> ANY VARIABLE DEFINIED WITHIN A CODE BLOCK

```
try {
    // GLOBAL
    const varOne = 'varOne';

    if (true) {
        console.log(varOne);

        // LOCAL
        const varTwo = 'varTwo';
    }

    // This will throw an exception, as varTwo was definiend within a code block (local scop
    // and not in the global scope
    console.log(varTwo);
}
catch (e) {
    console.log(e);
}
```

Basic Example

```
try {

    // GLOBAL
        // LOCAL
            // LOCAL
        // LOCAL

    const varOne = 'varOne';

    if (true) {
        console.log(varOne);
        const varTwo = 'varTwo';

        if (true) {
            const varFour = 'varFour';
            console.log('varOne = ' + varOne);
```

```
        }

        console.log(varFour);
    }

    if (true) {
        const varThree= 'varThree';
        console.log(varThree);
    }

}
catch (e) {
    console.log(e);
}
```

Variable Shadowing

```
try {
    const name = 'Jon';

    if (true) {

        // Even though the first name is a const,
        // This variable declartion is in a different scope
        let name = 'Jones';

        if (true) {

            // Due to lexical scope, this will never try to override the const name
            name = 'New';
            console.log(name);
        }
    }

    if (true) {
        console.log(name);
    }
}
catch (e) {
    console.log(e);
}
```

Leaked Global

```
try {
```

```
    if (true) {
        // This iresults when you forget to define a variable
        // With no variable declaration, like var, let or const it will create a glodal
        unwantedGlobal = 'Leaked Global';
    }

    console.log(unwantedGlobal);

}
catch (e) {
    console.log(e);
}
```

Correct Scope

```
try {

    if (true) {
        // This will created a scoped variabled
        let scoped = 'Scoped Variabled';
    }

    console.log(scoped);

}
catch (e) {
    // using scoped above will now throw an exception
    console.log(e);
}
```

Resources

- Scope

Hoisting

```
hosited = 10;
console.log(hosited);
var hosited ;
```

~This will console.log 10,, even though it hasn't be defined.

Resources

- Const
- Hoisting

- Let # Function Programming Concepts

I've recently finished reading Functional-Light JavaScript: Pragmatic, Balanced FP in JavaScript Kindle Edition by Kyle Simpson which I highly receommend anyone new to functional programming to read. In the book Kyle lists a number of useful FP functions, however, find the code snippets is pretty time consuming and tricky. This page is a simple list of the more useful functions that are defined in that book for easy access.

Constant

Certain APIs don't let you pass a value directly into a method, but require you to pass in a function

```
function constant(v) {
    return function value(){
        return v;
    };
}
```

Compose

```
function compose(...fns) {
    return fns.reduceRight( function reducer(fn1,fn2){
        return function composed(...args){
            return fn2( fn1( ...args ) );
        };
    } );
}
```

Currying

Currying is an architectural approach where you split you functions, into smaller components. In a curried method, a function only accepts one argument and returns one function. Using closures and currying you cycle down the chain until no all arguments have been processed

```
function curry(fn,arity = fn.length) {
    return (function nextCurried(prevArgs){
        return function curried(nextArg){
            var args = [ ...prevArgs, nextArg ];

            if (args.length >= arity) {
                return fn( ...args );
            }
            else {
                return nextCurried( args );
```

```
            }
        };
    })( [] );
}
```

Another example

```
function add(a, b) {
    return a + b;
}
console.log(add(1, 2));
```

Curried - One Level

```
function curryOneAdd(a) {
      return (b) => {
        return a + b;
      }
}

console.log(curryOneAdd(1)(2));
```

Or another way of writing it

```
const add10 = curryOneAdd(10);
console.log(add10(7));
```

Curried - One Level

```
function curryTwoAdd(a) {
    return (b) => {
        return (c) => {
            return a + b + c;
      }
    }
}

console.log(curryTwoAdd(1)(2)(3));
```

Resources

- Curring

Filter

```
function filter(predicateFn,arr) {
    var newList = [];

    for (let [idx,v] of arr.entries()) {
```

```
        if (predicateFn( v, idx, arr )) {
            newList.push( v );
        }
    }

    return newList;
}
```

Filter Out

```
function filterOut(predicateFn,arr) {
    return filterIn( not( predicateFn ), arr );
}
```

Flatten

```
var flatten =
    arr =>
        arr.reduce(
            (list,v) =>
                list.concat( Array.isArray( v ) ? flatten( v ) : v )
        , [] );
```

Guard

```
var guard =
    fn =>
        arg =>
            arg != null ? fn( arg ) : arg;
```

Identity

Takes a argument and returns the value untouched. Useful for chaining, using with filter etc...

```
function identity(v) {
    return v;
}
```

IIFE Example

```
var sum = (function IIFE(){

    return function sum(...nums) {
        return sumRec( /*initialResult=*/0, ...nums );
    }
```

```
    function sumRec(result,num1,...nums) {
        result = result + num1;
        if (nums.length == 0) return result;
        return sumRec( result, ...nums );
    }

})();

sum( 3, 1, 17, 94, 8 );
```

Make

```
function makeObjProp(name,value) {
    return setProp( name, {}, value );
}
```

Map

```
function map(mapperFn,arr) {
    var newList = [];

    for (let [idx,v] of arr.entries()) {
        newList.push(
            mapperFn( v, idx, arr )
        );
    }

    return newList;
}
```

Merge

```
function mergeLists(arr1,arr2) {
    var merged = [];
    arr1 = [...arr1];
    arr2 = [...arr2];

    while (arr1.length > 0 || arr2.length > 0) {
        if (arr1.length > 0) {
            merged.push( arr1.shift() );
        }
        if (arr2.length > 0) {
            merged.push( arr2.shift() );
        }
    }
```

```
    return merged;
}
```

Not

```
function not(predicate) {
    return function negated(...args){
        return !predicate( ...args );
    };
}
```

Partial

```
function partial(fn,...presetArgs) {
    return function partiallyApplied(...laterArgs){
        return fn( ...presetArgs, ...laterArgs );
    };
}
```

Partial Props

```
function partialProps(fn,presetArgsObj) {
    return function partiallyApplied(laterArgsObj){
        return fn( Object.assign( {}, presetArgsObj, laterArgsObj ) );
    };
}
```

Partial Right

```
function partialRight(fn,...presetArgs) {
    return function partiallyApplied(...laterArgs){
        return fn( ...laterArgs, ...presetArgs );
    };
}
```

Pipe

```
function pipe(...fns) {
    return function piped(result){
        var list = [...fns];

        while (list.length > 0) {
            // take the first function from the list
            // and execute it
            result = list.shift()( result );
        }
```

```
        return result;
    };
}
```

Prop

```
function prop(name,obj) {
    return obj[name];
}
```

Reduce

```
function reduce(reducerFn,initialValue,arr) {
    var acc, startIdx;

    if (arguments.length == 3) {
        acc = initialValue;
        startIdx = 0;
    }
    else if (arr.length > 0) {
        acc = arr[0];
        startIdx = 1;
    }
    else {
        throw new Error( "Must provide at least one value." );
    }

    for (let idx = startIdx; idx < arr.length; idx++) {
        acc = reducerFn( acc, arr[idx], idx, arr );
    }

    return acc;
}
```

Reverse Args

```
function reverseArgs(fn) {
    return function argsReversed(...args){
        return fn( ...args.reverse() );
    };
}
```

SetProp

```
function setProp(name,obj,val) {
    var o = Object.assign( {}, obj );
    o[name] = val;
```

```
    return o;
}
```

Spred Args

```
function spreadArgs(fn) {
    return function spreadFn(argsArr){
        return fn( ...argsArr );
    };
}
```

When

```
function when(predicate,fn) {
    return function conditional(...args){
        if (predicate( ...args )) {
            return fn( ...args );
        }
    };
}
```

Unary

Pass a single argument to a function

```
function unary(fn) {
    return function onlyOneArg(arg){
        return fn( arg );
    };
}
```

Uncurry

```
function uncurry(fn) {
    return function uncurried(...args){
        var ret = fn;

        for (let i = 0; i < args.length; i++) {
            ret = ret( args[i] );
        }

        return ret;
    };
}
```

Unique

```
function unique(list) {
    var uniqList = [];

    for (let v of list) {
        if (uniqList.indexOf( v ) === -1 ) {
            uniqList.push( v );
        }
    }

    return uniqList;
}
```

```
var unique =
    arr =>
        arr.filter(
            (v,idx) =>
                arr.indexOf( v ) == idx
        );
```

Zip

```
function zip(arr1,arr2) {
    var zipped = [];
    arr1 = [...arr1];
    arr2 = [...arr2];

    while (arr1.length > 0 && arr2.length > 0) {
        zipped.push( [ arr1.shift(), arr2.shift() ] );
    }

    return zipped;
}
```

More information can be found here.

Design Patterns

Decorator

A wrapper can be thought of as a wrapper which extends the functionality of an object/function while maintaining the interface.

```
function Drink(type) {
  this._cost = 2.50;
  this._type = type;
}

Drink.prototype.cost = function () {
  return this._cost;
}

function DrinkDecorator(drink) {
  Drink.call(this);
    this.drink = drink;
}
DrinkDecorator.prototype = Object.create(Drink.prototype);
DrinkDecorator.prototype.cost = function () {
  return this._cost + this.drink.cost();
}

function WithSugar(sandwich) {
  DrinkDecorator.call(this, sandwich);
    this._cost = 0.50;
}
WithSugar.prototype = Object.create(DrinkDecorator.prototype);

function Coffee() {
  Drink.call(this);
    this._cost = 5;
}
Coffee.prototype = Object.create(Drink.prototype);
```

```
var coffee = new Coffee();
coffee = new WithSugar(coffee);
console.log(coffee.cost()); // 3
```

Facade

```
class Albumns {
  get resources() {
    return [
      { id: 1, title: "Ride The Lightening" },
    ];
  }

  fetch(id) {
    return this.resources.find(item => item.id === id);
  }
}

class GetMovie {
  constructor(id) {
    return this.resources.find(item => item.id === id);
  }

  get resources() {
    return [
      { id: 1, title: "Lord Of The Rings" }
    ];
  }
}

class Facade {
  constructor(type) {
    this.type = type;
  }

  get(id) {
    switch (this.type) {
      case "music": {
        const db = new FetchMusic();
        return db.fetch(id);
      }

      case "movie": {
        const db = new GetMovie();
```

```
        return db.fetch(id);
      }
    }
  }
}
```

Proxy Pattern

Control access to a resource:

```
var proxied = jQuery.ajax;
jQuery.ajax = function() {
    jQuery("#loading").dialog({modal: true});
    return proxied.apply(this, arguments);
}
```

Adapter Pattern

An adapter allows two incompatible interfaces to work together.

```
var LoggerOne = (log) => console.log(log);
var LoggerTwo = (log, log2) => console.log(`${log} ${log2}`);

function LoggerAdapter(loggerObj) {
    if(loggerObj.getType() === "LoggerOne") {
        LoggerOne(loggerObj.text);
    }
    if(loggerObj.getType() === "LoggerTwo") {
        LoggerOne(loggerObj.text, loggerObj.text);
    }
}
```

Singleton

Restricts the instantiation of a class to one object.

```
const myFucntion = (function(){
    const name = 'example'
    const getName = () => name

    return {
        getName
    }
}())
```

```
myFucntion.getName() // example
president.name // undefined
```

Factory

Creates a pre-populated object for you

```
class MyObject {
  constructor(name){
    this.name = name
  }

  getName(){
    return this.name
  }
}

const Factory = {
  Object : (name) => new MyObject(name)
}
const door = Factory.Object("Name")
```

flyweight

An object that minimizes memory use by sharing as much data as possible with other similar objects

```
// Anything that will be cached is flyweight
class Drink {
}

// Acts as a factory and saves the tea
class Server {
    constructor(){
        this.availableItems = {}
    }

    make(preference) {
        this.availableItems[preference] = this.availableItems[preference] || (new Drink())
        return this.availableItems[preference]
    }
}

class Shop {
    constructor(server) {
```

```
        this.server = server
        this.orders = []
    }

    takeOrder(type, table) {
        this.orders[table] = this.server.make(availableItems)
    }

    serve() {
        this.orders.forEach((order, index) => {
            console.log('Serving table #' + index)
        })
    }
}
```

Proxy

Restricts Access To Something

```
class Validator {
    login() {
        console.log('loged in')
    }

    logout() {
        console.log('logged out')
    }
}

class Security {
    constructor(validator) {
        this.door = door
    }

    login(password) {
        if (this.authenticate(password)) {
            return this.validator.login()
        }

        console.log('Invalid Login Attempt')
    }

    authenticate(password) {
        return // add logic to authenicate here
    }

    logout() {
        this.validator.logout()
    }
}
```

Bridge

Decouple an abstraction from its implementation so that the two can vary independently, .e.g move from inheritance to composition!

```
class Tea{
    constructor(extras) {
        this.extras = extras
    }

    getDrink() {
        return "Tea with " + this.extras.get()
    }
}

class Sugar{
    get() {
        return 'Sugar'
    }
}

const tea = new Tea()
const about = new Sugar(tea)
```

Composite

Composite pattern lets clients to treat the individual objects in a uniform manner.

```
class Developer {

    constructor(name, salary) {
        this.name = name
        this.salary = salary
    }

    getName() {
```

```
        return this.name
    }

    setSalary(salary) {
        this.salary = salary
    }

    getSalary() {
        return this.salary
    }
}

class Designer {

    constructor(name, salary) {
        this.name = name
        this.salary = salary
    }

    getName() {
        return this.name
    }

    setSalary(salary) {
        this.salary = salary
    }

    getSalary() {
        return this.salary
    }
}

class School {
    constructor(){
        this.pupils = []
    }

    addPupil(pupils) {
        this.pupils.push(pupil)
    }

    getSalaries() {
        let salary = 0
        this.pupils.forEach(pupils => {
            salary += employee.getSalary()
        })
```

```
        return netSalary
    }
}

const dev = new Developer('Jon', 12000)
const designer = new Designer('Ana', 10000)

const school = new School()
school.addPupil(dev)
school.addPupil(designer)

console.log("Net salaries: " , organization.getNetSalaries()) // Net Salaries: 22000
```

Misc Features

Strict Mode

```
'use strict'
```

Strict-mode will enforce stricter JS parsing by the compiler. You should use this to increase the odds of you not making silly mistakes. This will cause an error in strict mode. This line will compile OK in normal mode

```
asdf = 'error';
```

You should use strict-mode to prevent accidental global variables from being created

Try/Catch

try/catch is a way to catch errors thrown in your code so you can handle them throw - exposes a message you can pass

```
try {
    throw('error');
} catch (e) {
    console.log(e);
}
```

throw new Error - exposes an error event with two params name & message

```
try {
    throw new Error('An error', 'message');
} catch (e) {
    console.log(e);
}
```

Type Coercion

In Javascript it is possible to add/compare two different data-types. When you do this you will get odd results and this should be avoided. This displays 55

```
console.log('5' + 5);
```

This displays 0

```
console.log('5' - 5);
```

As you can see, type coercion can result in odd behaviors. The additional sign concatenates the numbers together. The minus operator subtracts two values. When comparing values, you can use the non-strict equality operator. However with type coercion this can lead to unexpected errors. The code below will equal type, as the data type will be ignore and type-coercion kicks in. This is a bad coding practice

```
console.log('5' == 5);
```

Using strict equality will show false. This is why should always use ===

```
console.log('5' === 5);
```

If you want to compare different types. It is better practice to use the typeof operator instead:

```
const bool = true;
console.log('Boolean type = ', typeof bool);

const int = 1;
console.log('Int type = ', typeof int);

const array = [];
console.log('Array type = ', typeof array);

const object = {};
console.log('Object type = ', typeof object);
```

JSON Stringify

Serialize an object to a string

```
const myObjects = {
    name: 'test',
    example: 'testone'
};

const asString = JSON.stringify(myObjects);
console.log(asString);
```

De-serialize a string to JSON

```
const myObjects = {
    name: 'test',
    example: 'testone'
};

const toObject = JSON.parse(asString);
console.log(toObject);
```

Useful Functions

- Bind()
- Comma Operator()
- Freeze()
- Locale Compare()
- Includes()
- parseInt()
- Random()

Data Structures

- Mutator Methods
- Filter()
- ForEach()
- From()
- Reduce()
- ReduceRight()
- Reverse()
- Sort() # Misc Code Samples

Algorithm Examples

Random Number

This code generates a random number

```
const min = 0;
const max = 100;

const random = Math.random() * (max - min) + min;
console.log(random);
```

Binary Tree

```
BinaryTree.map = function map(mapperFn,node){
    if (node) {
        let newNode = mapperFn( node );
        newNode.parent = node.parent;
        newNode.left = node.left ?
            map( mapperFn, node.left ) : undefined;
        newNode.right = node.right ?
            map( mapperFn, node.right ): undefined;

        if (newNode.left) {
            newNode.left.parent = newNode;
        }
        if (newNode.right) {
            newNode.right.parent = newNode;
        }

        return newNode;
    }
};
```

Temperature Conversion

This code converts Fahrenheit to celsius

```
const fahrenheit = 33;

console.log('Fahrenheit = ' + fahrenheit);

const celsius = (fahrenheit - 32) * 5 / 9;

console.log('Celsius = ' + Math.round(celsius * 100) / 100);

const kelvin = (fahrenheit + 459.67) *  5 / 9;
console.log('Kelvin = ' + Math.round(kelvin * 100) / 100);
```

GetCounty()

Get Countries from restcountries.eu

```
var fetch = require("isomorphic-fetch");

const getCountry = async (countryCode) => {
    const response = await fetch('http://restcountries.eu/rest/v2/all')
```

```
    if (response.status === 200) {
        const data = await response.json()
        return data.find((country) => country.alpha2Code === countryCode)
    } else {
        throw new Error('Unable to fetch the country')
    }
}

getCountry('GB').then((data) => {
    console.log(data.nativeName);
}).catch((e) => {
    console.log(e);
});
```

Resources

- Fetch

Get IP Loction

Get IP Loction

```
var fetch = require("isomorphic-fetch");
const api = 'https://ipinfo.io/json?token=e4db2aebef0663';

const getLocation = async () => {
    const response = await fetch(api);
    if (response.status === 200) {
        const data = await response.json();
        return data.region;
    } else {
        throw 'some error';
    };
};

getLocation().then((data) => {
    console.log(data);
})
```

Resources

- Fetch

AXIOS Example

Fetch data from an API

```
var axios = require('axios');

axios.get('http://restcountries.eu/rest/v2/all')
  .then((response) => {
    console.log(response);
  })
  .catch((error) => {
    console.log(error);
});
```

Isomorphic Fetch

Isomorphic Fetch

```
var fetch = require("isomorphic-fetch");

const getCountry = async (countryCode) => {
    const response = await fetch('http://restcountries.eu/rest/v2/all')

    if (response.status === 200) {
        const data = await response.json()
        return data.find((country) => country.alpha2Code === countryCode)
    } else {
        throw new Error('Unable to fetch the country')
    }
}

getCountry('GB').then((data) => {
    console.log(data.nativeName);
}).catch((e) => {
    console.log(e);
});
```

Resources

This section contains a list of useful resources.

Javascript Stack

You maybe new to web development, or maybe you haven't heard of some of the technologies we use. Below are different technologies we use to build our stack.

Beginner

These are the technologies you will need to know to get you off the ground working with JS. With these you will be able to build components and see them on the website.

- **NVM** - https://github.com/nvm-sh/nvm We try to ensure our node versions are in-sync but to make sure you can run multiple versions we recommend using NVM
- **NodeJS** - https://nodejs.org/en/ NodeJS is used to build our presentation layer
- **YARN** - https://yarnpkg.com For more performant and reliable package-management we use Yarn
- **TypeScript** - https://www.typescriptlang.org/ To make our code more stable we want to use static typing: .
- **Express** - https://expressjs.com/ A web server framework for node used in our presentation layer:
- **React** - https://reactjs.org/ React is used to build our component library:
- **Redux** - https://redux.js.org/ Redux is how we handle state management:
- **BEM** http://getbem.com/ BEM (Block Element Module) is a methodology we use to style our components:
- **PostCSS** - https://postcss.org/ Like Babel PostCSS allows us to use modern CSS with less overheads
- **Jest** - https://jestjs.io/ We use Jest as our unit test runner, assertions, coverage and mocking:
- **Cypress** - https://www.cypress.io/ A modern functional test runner, we use to test components and presentation layers like a customer: .

- **YAML** - https://yaml.org/ We use YAML for configuration files, it's a standard that allows for readable data serialization:
- **Commitizen** - https://commitizen.github.io/cz-cli/ We have to have consistent commit messages so we can automate documentation, alerts and pretty much everything we can:

Intermediate

Once you are comfortable with the basic features, these next technologies will help you debug and improve the ecosystem.

- **Lerna** - https://github.com/lerna/lerna A monorepo manager
- **Babel** - https://babeljs.io/ We want to use the latest JavaScript syntax but we need to support customers with older browsers so we use babel.
- **Webpack** - https://https://webpack.js.org/ We want to make sure our website is performant and our developer experience as smooth as possible, for this we use web-pack
- **Make** - https://www.gnu.org/software/make/ Make is a build automation tool that we use to run commands in our CI/CD pipeline

Advanced

Once you are comfortable with the intermediate features, these will help you learn about our infrastructure and run times.

- **Docker** - https://www.docker.com/ So we can have consistent builds and deploys we use Docker as our container platform
- **Terraform** - https://www.terraform.io/ Write, Plan, and Create Infrastructure as Code. A platform agnostic approach to infrastructure provisionment
- **Kubernetes** - https://kubernetes.io/ Orchestrating, deploying and scaling out containers
- **Helm** - https://kubernetes.io/ Similar to Yarn however Helm is package management

Packages

- Axios
- Babel
- CSS Modules
- Classnames
- Create React App
- Enzyme
- Heroku
- Isomorphic Fetch
- Jest
- Lodash

- Mobx
- Mobx React
- Node
- NPM
- NVM
- Overreacted
- React Alternative Class Component Syntax
- React Devtools
- React Native
- React Profiler
- React Redux
- React Router
- Redux
- Styled Components
- Yarn
- Essential React Libraries in 2018

Cheatsheets

- Easings
- Enzyme
- Emmit Cheat Sheet
- ES6
- Complete list of Github markdown emoji markup
- Jest
- Markdown
- React
- ReduxForm

Interesting Reads

- A guide to setting up Vim for JavaScript development
- Best Practices for Using Modern JavaScript Syntax
- Clean Code in Javascript
- How I wrote the world's fastest JavaScript memoization library
- JavaScript Clean Coding Best Practices
- Keeping your code clean
- 7 Useful JavaScript Tricks

Interesting Videos

- What's new in JavaScript Google I/O 19
- Keep Betting on JavaScript - Kyle Simpson

Important Programming Concepts

- Call Stack
- Call Stack, Event Loop , Tasks - Understanding Javascript Function Executions
- Understanding the JavaScript call stack
- Event Loop
- Event loop - How JavaScript works
- Execution Context and Execution Stack
- How JavaScript works
- Inheritance In Javascript
- Inheritance with JavaScript
- Hoisting
- Proxy

Functional Programming

- Why Functional Programming Matters
- Professor Frisby's Mostly Adequate Guide to Functional Programming
- Pure Functions
- Higher Order Functions
- Partial Application
- Recursion

Design Patterns

- Deign Patterns Game
- 4 JavaScript Design Patterns You Should Know
- Essential JS Design Patterns Book
- GoF Design Patterns Implemented in Javascript
- Javascript Design Patterns
- JavaScript Design Patterns
- Understanding Design Patterns in JavaScript
- Design Patterns For Humans
- JS Design Patterns

Books

- You Don't Know JS

Online Tools

- CanIUse

- Canva
- Codepen
- Stat Counter

Fonts & Icons

- Google Fonts
- IcoMoon
- Font Awesome

Images and Videos

- Coverr.co - Free videos you can use on your website, useful for background videos
- CutMyPic - Online photo and effects editor. Previously, I use this for the creating round pictures with shadow, although you can get a similar effect now just using CSS
- FreeImages.co.uk - Free stock images
- Graffiti Creator
- Tuxpi - Online photo and effects editor. I use this for the Polaroid pictures for the banners on this site
- Unsplash - Hundreds of free stock imagery you can use

Content

- Copyscape - Check for plagiarized text
- Hilite - Code Snipper Pretty Printer
- Pingdom Website Speed Tester - Check your webpage to ensure they load quickly

Javascript Dictionary

Dyadic function A function with two arguments

```
function(one, two) {};
```

Idempotent You can call a function repeatedly the same way and it will always produce the same result.

```
const myFunction => 1 + 1;
```

Immediately Invoked Function Expression Usually simplified to IIFE, pronounced iffy.

```
(function() {
    alert("IIFE");
}());
```

Isomorphic Application Building an application that looks the same on the server and the client

Monadic function A function with one argument

```
function(one) {};
```

Pointfree style/Tacit Programming Point refers to a function parameter. Pointfree refers to not naming those parameters.

```
// not pointfree because we mention myParam in the code
const example = myParam => myParam.toLowerCase();

// pointfree
const example = compose(myParam, toLowerCase);
```

Pure Functions A function where the return value is only determined by its input values. Having a function that does not add any side effects is desirable, as it reduces the chance of bugs being introduced. Pure functions are also a lot easier to test.

```
const result = x => x * 2;
```

Stuff I Use

Software

- Visual studio code My preferred text editor

My Top 10 List Of Books Every Developer Should Read

- Code Complete: A Practical Handbook of Software Construction, Second Edition
  This is my bible of software development. It's a monster to read but it contains pretty much everything you need to know about crafting good code. After reading this book people commented about how much better my coding became.
- Clean Code: A Handbook of Agile Software Craftsmanship
  Everyone of Uncle Bobs books are essential reading this one in my opinion is the best. Theres a big difference between writing code and writing good code. This book will help you to think about what good code is and how you can write it
- Refactoring: Improving the Design of Existing Code
  I'm a strong believer in iterative design. Build something basic that works and fits the bill and then improve it constantly. This constant improvement is called refactoring. This book will teach you all the patterns and techniques you will need to accomplish that
- Dependency Injection in .NET
  To check that the code you write works, you will need to test it. To write testable code, you need to understand dependency injection. This book will teach you how you should use dependency injectionS
- Head First Design Patterns: A Brain-Friendly Guide
  Every software developer needs to know about design patterns. This book makes it super simple to learn the main patterns. If design patterns scare you, then this should be your next read.
- Code: The Hidden Language of Computer Hardware and Software
  This book will help you understand how your computer works. It's easy to read, and it will help you understand concepts like binary addition, gates and circuit boards.

- The Pragmatic Programmer
  This book is one of the first software development books I read. This book will provide you with a plan on how to improve your coding skills.
- The DevOps Handbook
  Writing code isn't the only part of software development. You will also have to release it. This book will teach you everything you will need in order to be able to release your code correctly.
- The Clean Coder
  This book will teach you how to be a professional. This book isn't about coding. It's how to behave like a great coder.
- Design Patterns: Elements of Reusable Object-Oriented Software
  Every software developer should read this book. This isn't the easiest book to read, which is why I recommend reading head first patterns first. You should read this book for bragging rights!

List Of Great Books That Will Help You Get Promoted

- Outliers: The Story of Success
  This book will teach you about practice. To be a good coder you will need to spend at least 10,000 hours coding. Being great isn't about talent, it's about practice. People who practice will become better. You will also learn about the right type of practice
- Deep Work: Rules for Focused Success in a Distracted World
  This book changed my life. Just because your busy doesn't mean you're succeeding. Deep work will give you a system to prioritise your life so you can achieve more
- Bounce: Mozart, Federer, Picasso, Beckham, and the Science of Success
  When I started programming it seemed like some people were naturally just gifted programmers. I was not one of them. I thought being good was down to your genes. This book proves differently. This book will teach you about practice and more importantly deliberate practice. Understanding this is essential in becoming a great programmer.
- The Adweek Copywriting Handbook
  We all need to write emails, blog posts, memo's. Learning how to improve how you write is an important skill that you will use throughout your life. This book will help you do that.
- The Power of Habit: Why We Do What We Do in Life and Business
  We all need to write emails, blog posts, memo's. Learning how to improve how you write is an important skill that you will use throughout your life. This book will help you do that.
- The Intelligent Investor: The Definitive Book on Value Investing
  Software development is a well paying job, however, if you don't know how to manage your money you will be working all your life. This book was originally written in the 1930's. The advice in this book is still true.
- Henry Ford - My Life and Work
  This book was written over 100 years ago. How Ford managed his planet..

was the original agile manifesto. The principles For applied to creating motorcars can be applied to modern software design.

- Henry Ford - My Life and Work
  To be a good software craftsman you need to understand the beauty of design. Steve jobs was known for his eye for detail and his never ending pursuit for making things better. Reading this book will help you appreciate design more
- Eric Schmidt - How Google Works
  Every developer has heard what a great place Google is to work. This book gives you an insight into how Google was started and the ethos that made them so great.

Equipment I Use

- Dell UltraSharp U3415W 34-Inch Curved LED-Lit Monitor
  Dell is known for making great products. This monitor is no exception. It doesn't give me eye-strain. I can have a split screen on two HDMIs, so I view a MAC and a PC on the screen at the same time. Plugging the HDMI cables could be easier, aides from that its worth buying.
- ASUS ZenBook As a contractor I've had a lot of laptops (currently over 10 dells)> I prefer the Zenbook over all of them. It's quick, light, keyboards good. The only downside are the speakers are too quiet.
- Apple Mac Mini
  When I've doing Javascript coding or video editing I use a mac-mini. Its handy to have a desktop and the mac-mini is the best small looking computer.
- Bose QuietComfort 35 Wireless Noise Cancalling Headphones
  I can't recommend these enough. When you put them on they block everything out. Sounds amazing. The charge lasts for ages. It recharges really quickly. I was skeptical about buying battery powered headphones, but these are much better than wired
- FSDUALWIN Aluminum Laptop Monitor Stand Space Bar, Monitor Riser MacBook Monitor Dock Desk Organizer with 4 USB Ports, Keyboard Storage
  I spent ages looking for a good looking monitor stand and this one fits the bill. Looks good with brushed aluminium. HAs 4 USB ports so I can plug the Echo dot in. Makes my desk look cleaner as I can hide my mouse and keyboard under it when not in use
- WD 4TB My Cloud Personal Network Attached Storage
  If you need an external hard drive this is a great choice. Connect it to your broadband and you can access your files anywhere in the world. I can view my media on my Xbox. I can access it from any device in my house. Looks good
- Canon EOS REBEL T7i EF-S 18-55
  This is the camera loads of Vloggers recommended buy. The quality of the

picture and video is good. Worth buying

- Rode VMGO Video Mic GO
  This is the best camera MIC you can buy. Good sound quality. Not bad price
- Rode Podcaster USB Dynamic Microphone
  This USB MIC plugs into my PC/Mac. Great sound quality. I recommend this MIC for anyone wanting to do voice and screen-sharing
- Xbox One X 1TB Console
  I'm an Xbox guy. Xbox One is the best games and media player. FACT!
- Echo Dot
  Its fun :)
- Samsung Galaxy S7 Edge
  My first smart phone was an iPhone. After trying a Samsung I switched an d never looked back. Camera on this phone is great. Battery life is good.
- Fitbit Charge 2
  I never thought I'd buy a smart-watch but I love this. If you want to become more active, a great challenge is to try and walk 10,000 steps a day. This watch helps you to do that. Since I've brought it I'm on a 50 day streak.
- Fitbit Aria 2 Wifi + Bluetooth Smart Scales
  I never thought I'd buy a smart-watch but I love this. If you want to become more active, a great challenge is to try and walk 10,000 steps a day. This watch helps you to do that. Since I've brought it I'm on a 50 day streak
- TomTom Go 510 5 Inch Sat Nav With World Maps
  Need a Satnav? This is worth considering. Comes with speed camera detection, which 'may' have saved me on a few occasions!
- Dyson AM04 Hot + Cool Heater/Table Fan
  This is expensive, but it is the coolest looking fan/heater on the market. In the summer acts as a fan. In the winter a heater

My Code

Below gives an overview of all the repos that you can find on my Github account

Javascript Projects

- Todo App -
- Hangman App -

React Projects

- Todo App -
- Portfolio
- Expense Tracker Website Allow users to add/edit/remove expenses. Integrates with firebase
- React Sample Website

Node Projects

- Node Examples

My NPM Packages

- https-status-lookup
- js-number-formatter

Misc Projects

- CSS Playground
- HackDays
- Packages Playground

Episerver Projects

- Episerver BaseBuild
- Episerver Goodies
- EpiServerBlogSampleSite
- DeploymentTool
- EpiServerDonutCaching
- Episerver10DisplayOptions
-
-

Umbraco Projects

- UmbracoSampleSite 76
- Surface Controller Example