Contents

[New Variables — Creation, Updating and Scoping 5](#_Toc40712016)

[let & const in real world 5](#_Toc40712017)

[var Scoping Refresher 6](#_Toc40712018)

[let versus const 6](#_Toc40712019)

[let and const in the Real World 6](#_Toc40712020)

[Temporal Dead Zone 8](#_Toc40712021)

[Is var Dead? What should I use? 8](#_Toc40712022)

[Function Improvements: Arrows and Default Arguments 8](#_Toc40712023)

[Arrow Functions Introduction 9](#_Toc40712024)

[More Arrow Function Examples 10](#_Toc40712025)

[Arrow Functions and ‘this’ 10](#_Toc40712026)

[Default Function Arguments 11](#_Toc40712027)

[when not use arrow functions 11](#_Toc40712028)

[Arrow Function Exercises 12](#_Toc40712029)

[arrow functions and 'this' 12](#_Toc40712030)

[**Template Strings** 13](#_Toc40712031)

[Strings Introduction 13](#_Toc40712032)

[Tagged Template Literals 13](#_Toc40712033)

[Tagged Templates Exercise 13](#_Toc40712034)

[Sanitizing User Data with Tagged Templates 13](#_Toc40712035)

[New String Methods 13](#_Toc40712036)

[new Strings methods example 13](#_Toc40712037)

[Abreviations 17](#_Toc40712038)

[Sanitizing data 19](#_Toc40712039)

[Using the unminified development version 19](#_Toc40712040)

[Using the minified and tested production version (source-map available) 20](#_Toc40712041)

[Class 04-30-2020 20](#_Toc40712042)

[Proxies 20](#_Toc40712043)

[Destructuring 21](#_Toc40712044)

[Destructuring Objects 21](#_Toc40712045)

[destructuring objects example 22](#_Toc40712046)

[Destructuring Arrays 22](#_Toc40712047)

[destructuring arrays example 22](#_Toc40712048)

[Swapping Variables with Destructuring 22](#_Toc40712049)

[swapping variables with destructuring example 23](#_Toc40712050)

[Destructuring Functions — Multiple returns and named defaults 23](#_Toc40712051)

[destructuring functions examples 23](#_Toc40712052)

[Iterables & Looping 24](#_Toc40712053)

[The for of loop 24](#_Toc40712054)

[The for of Loop in Action 24](#_Toc40712055)

[for of loop examples 25](#_Toc40712056)

[Using for of with Objects 25](#_Toc40712057)

[using for in in objects example 26](#_Toc40712058)

[An Array of Array Improvements 26](#_Toc40712059)

[Array.from and Array.of 26](#_Toc40712060)

[Array.from() and Array.of example 27](#_Toc40712061)

[Array.find and Array.findIndex 27](#_Toc40712062)

[Array.find() and .findIndex() example 27](#_Toc40712063)

[Array some and every 28](#_Toc40712064)

[…Spread and …Rest 28](#_Toc40712065)

[Spread Operator Introduction 28](#_Toc40712066)

[Spread operator example 28](#_Toc40712067)

[Spread Exercise 29](#_Toc40712068)

[More Spread Examples 29](#_Toc40712069)

[Spreading into a Function 29](#_Toc40712070)

[*Spreading into a function example* 29](#_Toc40712071)

[The …rest param in Functions and destructuring 30](#_Toc40712072)

[**Object Literal Upgrades** 31](#_Toc40712073)

[**Promises** 33](#_Toc40712074)

[Basic example 35](#_Toc40712075)

[Building Promises 35](#_Toc40712076)

[Chaining Promises 35](#_Toc40712077)

[Multiple Promises 36](#_Toc40712078)

[Symbols 37](#_Toc40712079)

[Code Quality with ESLint 37](#_Toc40712080)

[JavaScript Modules 37](#_Toc40712081)

[ES6 Tooling 38](#_Toc40712082)

[Classes 38](#_Toc40712083)

[Prototypal Inheritance Review 38](#_Toc40712084)

[Classes 38](#_Toc40712085)

[Classes example 38](#_Toc40712086)

[Extending Classes and using super() 39](#_Toc40712087)

[Extending Classes and using super() example 39](#_Toc40712088)

[Extending Arrays with Classes for Custom Collections 40](#_Toc40712089)

[Extending Arrays with Classes for Custom Collections example 40](#_Toc40712090)

[Generators 41](#_Toc40712091)

[Generators example 43](#_Toc40712092)

[Using Generators for Ajax Flow Control Example 43](#_Toc40712093)

[Looping Generators with for of example 43](#_Toc40712094)

[Sets and Weak Sets 43](#_Toc40712095)

[Map and Weak Map 44](#_Toc40712096)

[**Async + Await Flow Control** 45](#_Toc40712097)

[ES7, ES8 + Beyond 46](#_Toc40712098)

What is ECMA and ES6?

JavaScript was originally named JavaScript in hopes of capitalizing on the success of Java.

Netscape then submitted JavaScript to ECMA International for Standardization. (ECMA is an organization that standardizes information)

This results in a new language standard, known as ECMAScript.

Put simply, ECMAScript is a standard. While JavaScript is the most popular implementation of that standard. JavaScript implements ECMAScript and builds on top of it.

If you try to use modern syntax to recreate a familiar old technique, or stick it in without understanding how it actually behaves, you run the risk of:

* having to debug code that worked perfectly before
* introducing subtle mistakes that may catch you at runtime only and potentially not even in QA
* creating code that fails silently when you least expect it.

In fact, several of the changes that appear to be drop-in replacements for existing techniques actually behave differently from the code that they supposedly replace. In many cases, it can make more sense to use the original, older style to accomplish what you’re trying to do. Recognizing when that’s happening, and knowing how to make the choice, is critical to writing effective modern JavaScript.

When you declare a variable using let or const, the scope for that variable is limited to the local block where it’s declared. A block in JavaScript is distinguished by a set of curly braces {}, such as the body of a function or the executable code within a loop.

This is a great convenience for block-scoped uses of variables such as iterators and loops. Previously, variables declared within loops would be available to the containing scope, leading to potential confusion when multiple counters might use the same variable name. However let can catch you by surprise if you expect your variable declared somewhere inside of one block of your script to be available elsewhere.

Object and ARRAY declared as CONST in some cases can be modified so be careful.

One part of the traditional function has been taken over by the class keyword. This allows programmers to choose whether they would prefer to follow a more functional programming paradigm with callable arrow functions, or use a more object-oriented approach with classes to substitute for the prototypal inheritance of traditional functions.

Classes in JavaScript look and act a lot like simple classes in other object-oriented languages, and may be an easy stepping stone for Java and C++ developers looking to expand into JavaScript as JavaScript expands out to the server.

One difference between functions and classes when doing object-oriented programming in JavaScript is that classes in JavaScript require forward declaration, the way they do in C++ (although not in Java). That is, a class needs to be declared in the script before it is instantiated with a new keyword. Prototypal inheritance using the function keyword works in JavaScript even if it’s defined later in the script, since a function declaration is automatically hoisted to the top, unlike a class.

# New Variables — Creation, Updating and Scoping

* var
  + are **function scoped**
  + can be reassigned
* let and const
  + are **block {} scoped**
* let
  + cannot be re-declared in the same scope
* const
  + cannot be re-declared nor updated
  + doesn't mean they're immutable (eg. object content)

## let & const in real world

* they allow to **replace the IFEE trick** by simply inserting code inside a block

{  
const days = 365;  
}

* they allow to **fix for loops problem** (variable leaks outside)

for (let i=0; i<10; i++) {  
console.log(i);  
setTimeout(function(){  
console.log('The number is ' + i);  
// works fine because the value of let variable doesn't get overridden everytime  
}, 1000);  
}

## var Scoping Refresher

* var can be updated, function scope — only available inside the function where they’re created, or globally if not in a function
* can update a globally scoped var inside a function
* defining a var inside an if statement (which isn’t a function) means it will be available outside of the if
* let defines a variable inside a block scope; a block is anything between {}, so a variable defined inside an if block is not available outside of that block; same with const
* let and const are both block-scoped

## let versus const

* can only let once inside a block, whereas var can be defined more than once, even though this isn’t good practice
* can define variables with the same name in different blocks, which can cause confusion
* const variables cannot be updated, whereas let variables can be
* properties of a const can change, whereas the variable itself cannot be reassigned
* can prevent this if necessary with Object.freeze(person), though console will not throw an error when attempting to change a property on a frozen object/variable

## let and const in the Real World

* IIFE - immediately-invoked function expression
* if you define a variable outside of a function (in the window scope for example), it will be set in the global scope, which isn’t always what you want; IIFE is a function that is immediately executed, and keeps the variable within the function scope

(function() {

var name = 'wes';

console.log(name);

})();

* instead of using IIFE, use const or let inside a block {}
* with a for loop for(var i = 0; i < 10; i++), i is overwritten to 10 right away, so can’t do something like console.log(i) and expect 1, 2, 3..10 - instead you’ll get 10, 10, 10..; solution is to use for(let i = 0;...);

There are different opinions on CONST and LET and VAR. The recommendation by the teacher and also me is this.

Assume you will always use CONST. Use LET when you know rebinding or meaning you will update the value. Avoid VAR at All times.

You will generally be able to spot people who have a background in languages like CLIPPER, VB 6, etc because they gravitate to VAR. This is what they know. But it is not a good idea at all. Also many examples on the web will be using VAR for the same reason. You should clean those up before you use them.

A CONST var is immutable. A LET var is changeable. However, if the CONST defines an object the properties are not immutable. Therefore, it is not necessary to avoid CONST for object declaration.

While ES6 does not include a way to make the entire object immutable there is a function from MDN called Freeze that will freeze all properties essentially making them CONST at a point in time. MDN grew out of what was originally the Mozilla Developer Network. Hence the name MDN.

IIFE is an Immediately-Invoked Function Expression - A JavaScript function that runs as soon as it is defined.

It is a design pattern which is also known as a Self-Executing Anonymous Function and contains two major parts:

* The first is the anonymous function with lexical scope enclosed within the Grouping Operator (). This prevents accessing variables within the IIFE idiom as well as polluting the global scope.
* The second part creates the immediately invoked function expression () through which the JavaScript engine will directly interpret the function.

Using LET and CONST are created at the BLOCK level so they are private to the code block they are part of by default scope.

You can avoid your VAR from leaking into the main stack by using let instead of var to define the variable.

# Temporal Dead Zone

* var variables can be accessed before they’ve been assigned; can’t access value, but can see that it’s been defined
* const and let vars are not defined before they’re assigned

Temporal Dead Zone – This exist because of scoping related to var variables. This goes back to rules all languages comply with and some languages like Pascal actually are built around. It also has to do with reference and value pointers. Which we won’t get into here but it is the reason this exist.

## Is var Dead? What should I use?

According to Mathias Bynens (and the pattern Wes follows):

* use const by default
* use let if updating variable (rebinding) is needed
* var shouldn’t be used in ES6

According to Kyle Simpson:

* use var for top-level variables that are shared across many scopes
* use let for localized variables in smaller scopes
* refactor let to const only after some code has been written and you’re reasonably sure you’ve got a case where there shouldn’t be variable reassignment

# Function Improvements: Arrows and Default Arguments

***=>*** Arrow functions Also called the FAT ARROW FUNCTION

Instead of a require you can reference modules directly that are imported by other modules you have already imported. This is like a static class or method in object-oriented code. Wes calls it a static member reference.

You use a strict schema to control data passed to your model.

JavaScript is asynchronous by default. So you need to use calls such as ***await*** to force the code to execute synchronously when desired.

You need to mark your function as ***async*** to use the ***await*** callback

Arrow functions inside other arrow functions do not rebind the value

It greatly simplifies code because the return value is implicit and there is no need for curly braces

Things to know

1. Harder to debug
2. They can’t self-reference, also known as recursion
3. This become lexically bound in an Arrow function
4. With Arrow functions since you can’t bind this it will lexically go up the chain to first defined case which may be incorrect

Best use

* When this is required to bound to the context and not the function itself
* Map and reduce as the code can be more readable

No parameter

() => 100

1 parameter

x => 50 || (x) => 50

Multi parameter

(x, y) => 42

Putting parentheses around the return block means you will be returning an object

When the property and variable name are the same you can put the name only once for readability in ES6.

## Arrow Functions Introduction

* more concise
* implicit returns
* doesn’t rebind value of this when using an arrow function inside another function
* arrow functions are always anonymous, cannot be named; can create a const that is an arrow function though, like const sayMyName = (name) => { alert(`Hello ${name}!`) }

const names = ['wes', 'kait', 'lux'];

const fullNames = names.map(function(name) {

return `${name} bos`;

});

const fullNames2 = names.map((name) => {

return `${name} bos`;

});

// if only one param

const fullNames3 = names.map(name => {

return `${name} bos`;

});

// implicit return

const fullNames4 = names.map(name => `${name} bos`);

// if no arguments, need to pass empty parens

const fullNames5 = names.map(() => `foo bar`);

## More Arrow Function Examples

* When attempting to return an object literal in an arrow function, have to wrap braces in () to indicate returning an object
* use console.table to show table-formatted object
* When constructing an object, in ES6 if the key has the same name as the value variable, you can just use the var name. For example: {name: name, race: race} is equivalent to {name, race}
* Filter with arrayName.filter(age => age >= 60)

## Arrow Functions and ‘this’

* When you use an arrow function, this is not re-bound inside the function, but rather this is inherited by the parent (parent if one level deep); use a regular function if you need to re-bind this
* If you enter a new function, this will not be bound to anything

function() {

this

function() {

this // not the same as parent this! here, this is the window scope

}

}

Solved by using var self = this or var that = this. Not necessary though, can use an arrow function instead.

function() {

this

() => {

this // same as parent this

}

}

* If you want to switch two variables, you can use [first, second] = [second, first];

## Default Function Arguments

* Set in function declaration
* can pass undefined as value if default is set, for example calculateBill(100, undefined, 0.2)

## when not use arrow functions

* when adding event listeners (this doesn't refer to the element)
* when you need to bing a method to an object
* when adding prototype method
* when you need to access the arguments object

## Arrow Function Exercises

* item.textContent.includes checks text content; use instead of indexOf
* chain on multiple lines

## arrow functions and 'this'

The **this** keyword does not get rebound but is inherited form the parent

const btn = document.querySelector('button');  
  
btn.addEventListener('click', () => {   
console.log(this);   
// this == window (button's parent)  
});  
  
btn.addEventListener('click', function () {   
console.log(this);   
// this == button   
});  
  
btn.addEventListener('click', function () {   
console.log(this); // this == button   
setTimeout(() => {  
this.classList.toggle('active');  
// Using an arrow function 'this' on the setTimeout still point at the outer context, the button  
});  
});

**Template Strings**

# Strings Introduction

* aka template literals
* use ${} and backticks; can run any js inside braces

## Tagged Template Literals

* takes in strings and values
* ...values is the rest operator, returns all values without having to know how many arguments there are
* you get a strings array

## Tagged Templates Exercise

* can tack on a function to a string like

addAbbreviations`Hello my name is ${first}`

* is an abbreviation tag; hovering over word shows title value

## Sanitizing User Data with Tagged Templates

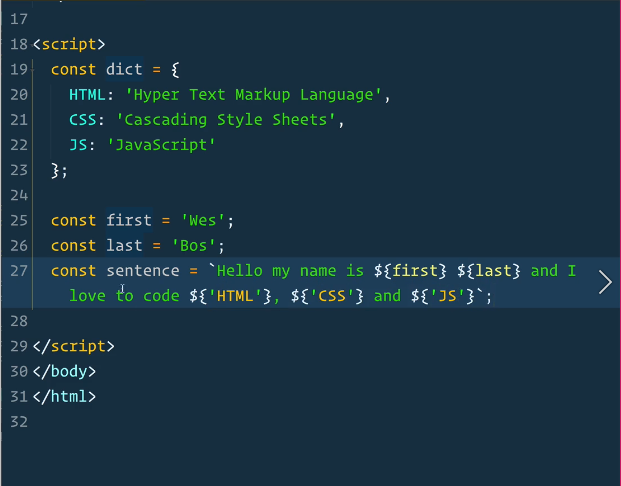
* Cant let users run JS on your page, so need to sanitize
* Run through a sanitize tag template

## New String Methods

* startsWith(), endsWith(), includes(), repeat()
* no way to do case sensitivity; still need regex
* can pass a number to startsWith('AC', 3) which will skip fist 3 chars
* can pass a number to endsWith('RT', 11) to only consider the first 11 chars
* includes used to be contains, but renamed because of conflict with moo tools
* repeat(10) repeats string 10 times; can be useful to create padding in a pad function, for example

### new Strings methods example

const string = "Lorem ipsum";  
  
string.startsWith('Lorem'); // true, it's case sensitive  
string.startsWith('rem', 3); // true, you can specify the starting character  
  
string.endsWith('sum'); // true  
string.endsWith('Lo'); // true, only take the specified number of characters  
  
string.includes('red'); // true  
  
string.repeat(3); // Lorem ipsumLorem ipsumLorem ipsum



Above in our sentence you see an example of interpolation.

So what is interpolation? Well outside of code interpolation is: the insertion of something of a different nature into something else. But in our world of code it is more specific.

String Interpolation using Template Literals

The syntax for template literals will be a bit strange for anyone who is a JavaScript purist. First, you need to surround the string you want to interpolate with backticks, which you can find at the top left of your keyboard under the esc key (American keyboards).

"wrong"

'wrong'

`correct`

Second, you need to surround the JavaScript code that you want interpolated with these brackets: ${}. And yes, the $ is required.

`{wrong}`

`$wrong`

`${correct}`

Here’s what an example code snippet using ES6 template literals would look like:

const names = ['Curly', 'Moe', 'Larry'];

`The Three Stooges were ${names.slice(0, 2).join(', ')} and ${names[2]}.`

// "The Three Stooges were Curly, Moe and Larry."

Escaping Nested Quotes

One of the things I love about template strings is that they are declarative. You just need to write the string how you want it to look and the JavaScript interpreter will take care of the rest.

Take nested quotation marks for instance. Typically, if you wanted to create nested quotes, you would have to escape the quotation characters so the interpreter wouldn’t accidentally end the string early or switch between double and single quotes.

// ES5

"This is a \"nested string\"."

// or

'This is also a \'nested string\'.'

// or

"This is another 'nested string'."

Template literals remove that headache.

// ES6

`This is a "nested string".`

// "This is a \"nested string\"."

Multi-line Strings

Multi-line strings in ES5 are a pain. Not only do you need to include a new line character, \n, you need concatenate the strings across the lines as well.

// ES6

console.log(`string text line 1

string text line 2`);

// "string text line 1

// string text line 2"

Tagging

Tag functions are a great way to keep your code DRY. If you find yourself needing to interpolate strings over and over in the same way, you can create a tag function to do it for you.

The first argument passed into a tag function is the string you want to convert. The rest of the arguments correlate with any ${} expressions you have in your string. The function will parse those expressions out and use them.

// Tag Function

var person = 'Mike';

var age = 28;

function myTag(strings, personExp, ageExp) {

// The string is split using the expressions as delimiters.

// the 'strings' argument is returned as an array

// of the text snippets which you can access.

var str0 = strings[0]; // "that "

var str1 = strings[1]; // " is a "

var ageStr;

if (ageExp > 99){

ageStr = 'centenarian';

} else {

ageStr = 'youngster';

}

return str0 + personExp + str1 + ageStr;

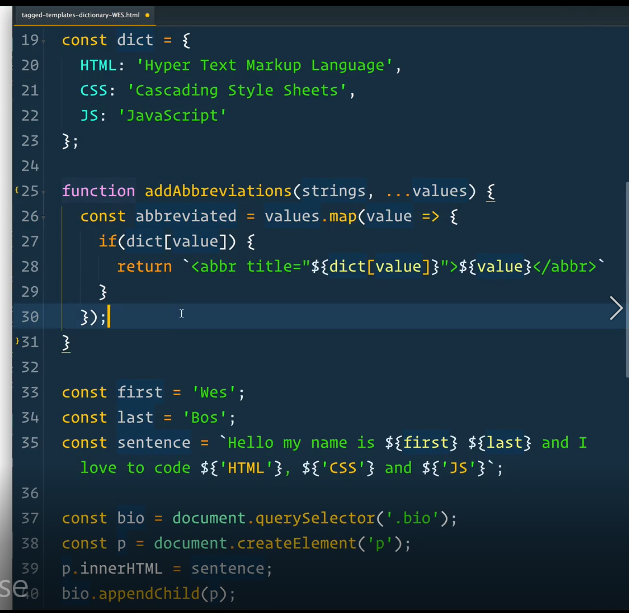
}

var output = myTag`that ${ person } is a ${ age }`;

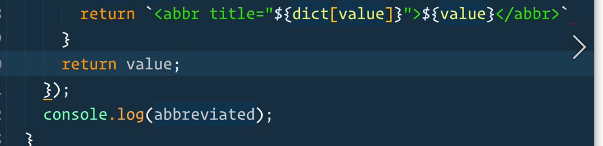
console.log(output);

// that Mike is a youngster

## Abreviations



Don’t forget to return the value if it is not found in the array.



A couple of ways to get abbreviations or acronyms automatically from words

let abbrev = 'INTERNATIONAL Monetary Fund'.match(/\b([A-Z])/g).join('');

alert(abbrev);

let input = "Content Management System";

let abbr = input.match(/[A-Z]/g).join('');

let toMatch = "hyper text markup language";

let result = toMatch.replace(/(\w)\w\*\W\*/g, function (\_, i) {

return i.toUpperCase();

}

)

alert(result);

## Sanitizing data

You have to stop users from running Java Script on your page with things they input or upload.

For our class we are using DOMPURIFY. <https://github.com/cure53/DOMPurify>

DOMPurify is a DOM-only, super-fast, uber-tolerant XSS sanitizer for HTML, MathML and SVG

DOMPurify sanitizes HTML and prevents XSS attacks. You can feed DOMPurify with string full of dirty HTML and it will return a string (unless configured otherwise) with clean HTML. DOMPurify will strip out everything that contains dangerous HTML and thereby prevent XSS attacks and other nastiness. It's also damn bloody fast. We use the technologies the browser provides and turn them into an XSS filter. The faster your browser, the faster DOMPurify will be.

It's easy. Just include DOMPurify on your website.

### Using the unminified development version

<script type="text/javascript" src="src/purify.js"></script>

### Using the minified and tested production version (source-map available)

<script type="text/javascript" src="dist/purify.min.js"></script>

Afterwards you can sanitize strings by executing the following code:

var clean = DOMPurify.sanitize(dirty);

The read me an GITHUB have examples of just about everything you may want to do with it.

## Class 04-30-2020

Recommended Book:

<https://eloquentjavascript.net/>

Home work :

Take the completed class work (in GITHUB REPO JAVA30 FlexPanel branch file name: FlexPanel-Finished.HTML

Your assignment:

1. Make side panel that does not react or change size.   Show small icon of photos and let users pick the photo for the panels from your list and either upload new or change the URL reference.
2. Allow the words to be photos that slide over the background photo
3. Allow the user to pick the number of panels which would mean number of photos.  Kind of like a collage maker that shows a larger portion of the individual photos when you click them

Extra credit

Let the user choose how the photo is centered so when it expands they are controlling which part of the larger photo you are seeing.

# Proxies

* Allow you to overwrite default behavior for many of an object’s default operations
* Proxy takes a target (what object you want to proxy) and a handler, which is where you specify the operations you wish to overwrite; when you want to overwrite a default an operation, a “trap” sits between you and the operation

const person = { name: 'Wes', age: 100 };

const personProxy = new Proxy(person, {

get(target, name) {

console.log('someone is asking for ', target, name);

return 'Nahhh ';

}

});

personProxy.name = 'Wesley'; // = is where the trap is

// returns "Nahhh" when personProxy.name is called

* Examples:
  + uppercase a string
  + removing whitespace when setting a value
  + stripping special characters from a phone number
  + correcting common mistakes, such as using the wrong case when using a key, or when trying to set a key that’s the same as another key but different case (ID vs id)

# Destructuring

## Destructuring Objects

* Allows properties to be extracted into their own objects: const { first, last } = person; creates variables first and last from person
* Can extract nested properties too, like const { twitter, facebook } = wes.links.social; where twitter and facebook are nested 3 levels down
* Can rename as you destructure with const { twitter:tweet, facebook:fb } = wes.links.social;
* Can set defaults too; prevoiusly you’d have a default settings object that you merge with your new settings, but in es6: const { width = 100, height = 100, color = 'blue', fontSize = 25} = settings; where the values that are set are fallbacks in case not set in settings

### destructuring objects example

const person = {  
name: 'Wes',  
surname: 'Bos',  
socials: {  
links: {  
twitter: '@wesbos',  
facebook: '/wesbos.developer'  
}  
}  
};  
  
const { name, surname } = person; // Extracts from the given object the values with the names of the new constants declared  
const { twitter, facebook } = person.socials.links; // Can be used on nested nodes of an object  
const { tw:twitter, fb:facebook } = person.socials.links; // You can rename the key whatever you like  
const { tw:twitter = 'some twitter account', fb:facebook = 'some facebook account' } = person.socials.links; // You can also set default values in case they are not included in the source object

## Destructuring Arrays

* Same as destructuring objects, but use [ ]
* Useful if data is passed as a string that can be split into an array
* When destructuring an array that contains more elements than the destructuring array, the rest of the elements will be discarded
* Use rest operator to grab remainder ...

### destructuring arrays example

const myData = ['Alessandro', '35', 'Italy', 'spaghetti', 'afalchi82'];  
  
const [ name, age, location ] = myData; // Same as for object, but uses square bracket for output variables  
const [ name, age, ...misc ] = myData; // Can be use in conjunction with the rest operator ... to put the rest of the values into a new array

## Swapping Variables with Destructuring

* Previously you’d have to use an intermediate/temp var to swap values; now, can use [a, b] = [b, a];

### swapping variables with destructuring example

let inRing = 'Hulk Hogan'; // Using let because we want to update them  
let onSide = 'The Rock'; // Using let because we want to update them  
  
[ inRing, onSide ] = [ onSide, inRing]; // Sweet.

## Destructuring Functions — Multiple returns and named defaults

* Can get multiple values back from a function by passing in multiple values and returning an object that you can then destructure
* Order passed in doesn’t matter, and can pass in just the variables you want
* Can make function order-independent by making an object to be destructured; other benefit here is don’t need to pass each variable in
* Do have to pass at least something in to a function, OR can set a default value for the entire function

Instead of function tipCalc(total, tip, tax), use function({ total, tip, tax }) and pass in an object

### destructuring functions examples

You can destructure the result of a function by returning an object

function convertCurrency (amount) {  
const converted = {  
USD: amount \* .76,  
GPB: amount \* .53,  
AUD: amount \* 1.01  
}:  
  
return converted;  
}  
  
const { USD } = convertCurrency(100);

Another example of destructuration of the arguments of a fundction:

function tipCalc ({total = 100, tip = .15, tax = .13} = {}) {   
  
// An empty object has to be passed as default   
// in case the function is called without params  
// (it would give an error, beacause it tries to destructure againt an undefined object)  
  
// We are destructuring the arguments  
// it's like saying  
// {total, tip = .15, tax = .13} = { total: 100 };  
// Where only the values passed override the defaults  
  
return total + (tip \* total) + (tax \* total);  
}  
  
const bill = tipCalc({ total: 100 });  
// We can pass whatever argument we need  
// and they're position indipendent

# Iterables & Looping

## The for of loop

* Iterable is anything that can be looped over
* Loop types are:
  + for loop (confusing, overly verbose)
  + forEach (can’t break or skip items)
  + for in for (const index in cuts) { console.log(cuts[index]); (if methods have been added to the prototype like monkey-patching, added methods will be displayed; unexpected behavior)
  + for of gets around the prototype issue, clean syntax, allows break and skip

const cuts = ['Chuck', 'Shank', 'Brisket'];

for (const cut of cuts) {

if (cut === 'Brisket') {

break; // or continue if you want to keep going through elements

}

}

## The for of Loop in Action

* Each iterable has an iterator, can be accessed with .entries(); each element in the array is in its own array with an index, can use destructuring to grab value and index

### for of loop examples

const cuts = ['Chuck', 'Brisket', 'Shank', 'Short Rib'];  
  
// Classic for syntax  
// A little confused  
for (let i=0; i<cuts.length; i++) {  
console.log(cuts[i]);  
}  
  
// forEach  
cuts.forEach( cut => {  
console.log(cut);  
if (cut === 'Brisket') {  
break; // This DOESN'T WORK!  
continue; // This DOESN'T WORK!  
}  
});  
  
// for ... in   
// Only return the index of the element  
// has unexpected results when the prototype of the array has been modified  
for (const cut in cuts) {  
console.log(cut); // 0,1,2,3 -> returns the index of the current element  
}  
  
  
// for ... of   
for (const cut of cuts) {  
console.log(cut);  
if (cut === 'Brisket') {  
break; // This WORKS!  
continue; // This WORKS!  
}  
}

for (const [i, cut] of cuts.entries()) {

console.log(`${cut} is the ${i + 1} item`);

}

* Useful when iterating over arguments object. arguments is a reserved word, returns arguments of a function in an object; not a regular array but does have an iterator. Typically convert arguments to an array.
* Use arguments when you don’t know how many arguments are going to be passed in, and loop over arguments.
* Node lists have an iterator, too

## Using for of with Objects

* Objects are not iterables, no iterator
* Object does have entries, but that doesn’t have an iterator either; can use polyfill and will be coming soon to ES
* Can use Object.keys and get prop and value
* Can also use for in but will have same limitations as when used elsewhere (can’t break or skip entries)

### using for in in objects example

const apple = {  
color: 'red',  
size: 'medium',  
weight: 50,  
sugar: 10  
}  
  
// Using for ... of  
for (const prop of Object.keys(apple) ) {  
  
console.log.( Object.keys(apple) ); // returns an array of the keys  
  
const value = apple[prop];  
console.log(prop, value);  
}  
  
  
// Using for ... in  
for (const prop in apple) {  
const value = apple[prop];  
console.log(prop, value);  
}

# An Array of Array Improvements

## Array.from and Array.of

* Neither are on prototype but are on array itself (can do Array.of but not [0,1,2].of
* Useful when working with DOM elements, which are returned in a node list
* Array.from can take a second param which is a map function; the following will convert the peopleArray to an actual array (from a node list), map over the peopleArray and create a new array with the text content of each element

const people = document.querySelectorAll('.people p');

const peopleArray = Array.from(people, person => {

console.log(person);

return person.textContent;

});

* useful when working with the arguments array too, because it’s not an actual array
* Array.of(x, y, z) takes in arguments and creates an array

### Array.from() and Array.of example

**Array.from** is useful to create a true array instance from an array-ish one (like a NodeList)

// Returns an array-ish object  
const people = document.querySelectorAll('p');  
  
// Convert into a real array  
const peopleArray = Array.from(people);  
  
// It accepts a map function as a second argument  
const peopleMap = Array.from(people, person => person.textContent);

**Array.of** simply creates an array from the provided arguments Javascript const people = Array.of('Gianni', 'Luigi', 'Sergio');

## Array.find and Array.findIndex

* If you have an array, can use objectName.find which checks each item in the array until the condition is true: posts.find(post => post.code === 'abc'); but only returns first match
* Can use filter to find multiple entries that match
* Can use findIndex to get index

### Array.find() and .findIndex() example

const people = [  
{ name: 'Mario', age: 32 },  
{ name: 'Angelo', age: 45 },  
{ name: 'Franco', age: 78 }  
];  
  
// .find() Long version  
const friendLong = people.find( person => {  
if( person.name === 'Franco') {  
return true;  
} else {  
return false;  
}  
});  
  
// .find() Short version  
const friendShort = people.find( person => person.name === 'Franco');  
  
  
// .findIndex Long version  
const indexLong = people.findIndex( person => {  
if ( person.name === 'Franco') {  
return true;   
} else {  
return false;   
}  
});  
  
// .findIndex Short version  
const indexShort = people.findIndex( person => person.name === 'Franco');

## Array some and every

* some checks if at least one element in the array matches the condition
* every checks if all elements in the array match the condition

# …Spread and …Rest

## Spread Operator Introduction

* Spread good for copying an array (setting array1 = array2 would create a reference instead of a copy)

### Spread operator example

The spread operator takes every item in an **iterable** and spreads it into a new array.

// It's useful for combining existing arrays into a new one  
const array1 = [1,2,3,4,5];  
const array2 = ['a', 'b', 'c', 'd'];  
const combinedArrays = [...arra1, ...array2];  
  
// or creating non-destructive copies  
const array1Copy = [...array1];

## Spread Exercise

* Spread and wrap each item of a string array with map

## More Spread Examples

* Good for copying an array instead of Array.from
* Good for adding items from one array into another like ['Milk', 'Flour', ...otherArray.]
* When you need to remove an item from an array, need to slice and spread

Using spread in place of Array.from()

const people = document.querySelectorAll('p.person'); // Returns a nodelist  
const peopleArray = Array.from(people); // Converts to array  
const peopleSpreadArray = [...people]; // Also converts to array

Using spread to pop an element from an array

## Spreading into a Function

* To add elements of one array to another you can use apply
* Or, you can use push with spread

const inventors = ['Einstein', 'Newton'];

const newInventors = ['Musk', 'Jobs'];

inventors.push(...newInventors);

* Also useful for passing an array into a function. Given const array = ['Rob', 'Dyson'], instead of sayHi(array[0], array[1]) can use sayHi(...array)

### Spreading into a function example

const inventors = ['einstein', 'newton', 'galileo'];  
const newInventors = ['musk', 'wozniak']; // Woz, Wes, not Jobs  
  
// Old method  
// inventors.push.apply(inventors, newInventors);  
  
// Using ES6 spread operator  
inventors.push(...newInventors);  
  
// Spreading array into function arguments  
const name = ['Gianni', 'Morandi'];  
  
function sayHi(first, last) {  
console.log(`Hi ${first} ${last}`);  
}  
  
sayHi(...name);

#### [**#**](https://codepen.io/afalchi82/post/es6#32-the-rest-param-40)32 The ...Rest Param

# The …rest param in Functions and destructuring

* Will use the rest of the params given

function convertCurrency(rate, ...amounts) {

}

convertCurrency(1.54, 10, 20, 30)

Uses 1.54 as rate, the rest as amounts.

const runner = ['Wes Bos', 123, 5.5, 5, 3, 6, 35];

const [name, id, ...runs] = runner;

Runs is the 3rd element on

#### Object literals upgrades

// No need to repeat value if the name is equal to variable to assign  
const name, age, country;  
const person = { name, age, person };  
  
// No need to write anonymous function for methods  
const modal = {  
create() {},  
open() {},  
close() {}  
};  
  
// Dynamic key names  
const key = 'pocketColor';  
const value = 'red';  
const tShirt = {  
[key]: value; // old way  
[`${key}Alternate`]: value // key name can be evaluated here  
};

The same symbol as the spread operator acts as its opposite

function convertCurrency(value, ...amounts) {  
return amounts.map( item => item \* rate);  
}  
  
convertCurrency(1.968, 10, 15, 24);

**Object Literal Upgrades**

const first = 'snickers';

const last = 'bos';

const age = 2;

const breed = 'King Charles Cav';

const dog = {

first: first,

last: last,

age: age,

breed: breed,

};

can be rewritten as follows if property name and variable are the same

const dog = {

first,

last,

age,

breed,

};

Also useful to define an object of functions.

const modal = {

create: function() {

},

open: function() {

},

close: function() {

}

}

can be rewritten as (same scoping of this)

const modal = {

create() {

},

open() {

},

close() {

}

}

* computed property names

const key = 'pocketColor';

const value = '#ffc600';

const tShirt = {

};

// can compute a value given a function, like invertColor

const tShirt = {

};

// previously would have had to define the object and then update the 'opposite' key value

* different value types

given

const keys = ['size', 'color', 'weight'];

const values = ['medium', 'red', 100];

const shirt = {

}

// creates an object with [0] [1] [2] set as key value pairs

**Promises**

* Javascript is async; fetch does not return actual data, but returns a promise that you can listen to with then (like a callback)
* Fetch can return json or any other type of data, so need to specify, for example data.json()
* In a promise, then only fires on success, catch on error
* Promise created as follows:

const p = new Promise((resolve, reject) => {

resolve('Wes is cool');

// reject(Error('Error')); wrapped in an Error object to give you information about where the error actually occurred as opposed to just where the error was caught

});

p

.then(data => {

console.log(data);

})

.catch(err => {

console.error(err);

});

* Can chain promises by returning a promise in a then of another promise

TODO: How do promises compare to observables?

getPostById(2)

.then(post => { // promise 1

return hydrateAuthor(post);

})

.then(post => { // promise 2

console.log(post);

})

.catch(err => {

console.error(err);

});

* When needing to fire multiple promises at once, instead of using .then use .all; waits for all promises to return, so will be as slow as the slowest response. Returns an array of responses.
* Can’t use fetch from local filesystem, use a server

Promise

.all([weather, tweets])

.then(responses => {

console.log(responses)

})

* When using .all, can call .all on the response to convert, for example all to json
* TODO: What if you need to do different things with each response? As opposed to all being converted to json, for example

### Basic example

const peoplePromise = fetch('https://randomuser.me/api/?inc=name,picture&results=10&inc=id,location,nat,name,picture,age');  
peoplePromise  
.then( data => {  
// console.log(data.json());  
return data.json();  
})   
.then( data => {  
//console.log(data.results);  
people = data.results;  
console.log(people)  
})  
.catch();

### Building Promises

const p = new Promise(resolve, reject) {  
  
// On success   
resolve('Some data to be returned');  
  
// On error  
reject(Error('Some error occured'));  
};  
  
p  
.then( data => { console.log(data); })  
.catch( err => { console.error(err); });

### Chaining Promises

const posts = [  
{t: 't1', a: 'a1', id: 1},  
{t: 't2', a: 'a2', id: 2},  
{t: 't3', a: 'a3', id: 3}  
];  
  
const authors = [  
{n: 'a1', tw: 'twitter 1', bio: 'bio 1'},  
{n: 'a2', tw: 'twitter 2', bio: 'bio 2'},  
{n: 'a3', tw: 'twitter 3', bio: 'bio 3'}  
];  
  
function getPostById(id) {  
return new Promise( (resolve, reject) => {   
setTimeout( () => {  
const item = posts.find( item => item.id === id);   
resolve(item);   
reject(Error('No Item Found!'));   
}, 1000);  
});  
}  
  
function hydrateAuthor(post) {  
return new Promise( (resolve, reject) => {  
// debugger;  
if (post) {  
post.a = authors.find( item => item.n === post.a);  
resolve(post);  
} else {  
reject(Error('Author wasn\'t hydrated.'));  
}  
});  
}   
  
getPostById(3)  
.then( item => {  
// console.log(item);  
return hydrateAuthor(item);  
})  
.then( post => {  
console.log(post);  
})  
.catch( err => { console.log(err); })  
;

### Multiple Promises

const dogs = fetch('https://dog.ceo/api/breeds/list');  
const ghibliMovies = fetch('https://ghibliapi.herokuapp.com/films');  
  
Promise  
.all([dogs, ghibliMovies])  
.then( responses => {  
return Promise.all( responses.map( res => res.json() ) );   
})  
.then( response => {  
console.log(response)  
})  
;

# Symbols

* Primitives are: number, string, object, boolean, null, undefined
* New primitive type is symbol, which is a unique identifier. const wes = Symbol('wes') where wes is a descriptor. Symbols are always unique. Can have same descriptor for multiple symbols.
* By default cannot loop over symbols, data is private. Can get data by calling Object.getOwnPropertySymbols(foo); which you can then loop/map over

const classRoom = {

}

const syms = Object.getOwnPropertySymbols(classRoom);

const data = syms.map(sym => classRoom[sym]);

console.log(data)

# Code Quality with ESLint

* rules can be set as off, warn, error or 0, 1, 2
* can set file or line specific rules
* can eslint markdown and html

# JavaScript Modules

* need webpack bc importing modules not supported in browser
* –save-dev for dev tools not part of app
* variables are not global with modules—scoped to module
* import your own modules with import apiKey from './src/config'; for example
* default export means you can import as anything you want, named means you need to import and use name defined in module
* export default apiKey means apiKey is available wherever it is imported; could import as whatever name you want. Only 1 default export, but can have as many named exports as needed. Use { } for importing named exports. Leave off for default.
* can export functions, too
* can import named module as to rename; can export as, too

# ES6 Tooling

* SystemJS/JSPM works in browser, vs webpack
* @next in an npm install command grabs stable+1 version
* env preset in babel lets you specify which browsers you want to support
* Babel only works on syntax, not on methods like Array.from; need polyfills using babel-polyfill module, or if not using modules can include polyfill.io script, which detects what browser the user is using and polyfills as necessary

# Classes

## Prototypal Inheritance Review

* When you put a method on the original constructor, will be inherited by all instances
* use capital name for prototype function, like function Dog
* even after creating an instance, can add functions to prototype and have them accessible by instances

## Classes

* class declaration: class Dog { }
* class expression: const Dog = class { }
* only necessary method is a constructor method
* adding methods in classes, don’t use comma between functions
* static methods are only available on class directly, for example if you have a Dog class can call a static method bark like Dog.bark but not on a dog instance
* can add getters and setters to classes, too

TODO: Why use classes instead of prototype functions?

### Classes example

class Dog {  
constructor (name, breed) {  
this.name = name;  
this.breed = breed;  
}  
  
// Inheritable method  
bark() {  
console.log(`My name is ${this.name}`);  
}   
  
// Private method (not inheritable)  
static info() {  
console.log(`My breed is ${this.breed}`);  
}  
  
// Define a getter  
get description() {  
console.log(`I'm ${this.name} and I'm a ${this.breed}`);  
}  
  
  
// Define a setter  
set nickname(value) {  
this.nick = value;  
}  
  
// Define a setter  
get nickname() {  
return this.nick;  
}  
}  
  
  
const fido = new Dog('fido', 'cane brutto');

## Extending Classes and using super()

* can extend classes with class Dog extends Animal { constructor() { } }; needs a constructor
* this is not defined inside a class that extends a class without first calling super()
* can add methods to a class that extends a class

### Extending Classes and using super() example

class Animal {  
constructor (name) {  
this.name = name;  
this.thirst = 100;  
this.belly = [];  
}   
drink() {  
this.thirst -= 10;  
return this.thirst;  
}   
eat(food) {  
this.belly.push(food);  
return this.belly;  
}  
}  
  
class Dog extends Animal {  
constructor (name, breed) {  
  
// You need to call super()   
// -a reference to the extended class-  
// to use the class you are creating  
super(name);   
  
this.breed = breed;  
}  
bark(message) {  
console.log(`Bark bark ${message}`);  
}  
}  
  
const rinho = new Animal('Rhiney');  
const doggo = new Dog('fido', 'bastard');

## Extending Arrays with Classes for Custom Collections

* can add methods to arrays and other collections with class MovieCollection extends Array { constructor() } }
* have to call super before using
* remember that adding methods to an array means the array will have a new element that is the method; use for of instead of for in

### Extending Arrays with Classes for Custom Collections example

class MovieCollection extends Array {  
constructor (name, ...items) {  
super(...items);   
this.name = name;  
}  
add(movie) {  
this.push(movie);  
}  
topRated(limit = 10) {  
return this.sort( (a,b) => (a.stars > b.stars ? -1 : 1)).slice(0, limit);  
}  
}  
  
  
const myCollection = new MovieCollection('Ale\'s collection',   
{title: 'Titanic', stars: 5 },  
{title: 'Avatar', stars: 3 },  
{title: 'Baraka', stars: 5 },  
{title: 'The Simpsons', stars: 0 }  
);  
  
  
myCollection.add({title: 'Dillinger è morto', stars: 5});  
  
console.table(myCollection.topRated(2) ) // Why do I get also the name property?  
  
// Only get the index  
for (const movie in myCollection) {  
// console.log(movie);   
}  
  
// Get the actual objects (works with iterables)  
for (const movie of myCollection) {  
// console.log(movie);

## Generators

* Generators can be paused, passed information at a later time

function\* listPeople() {

yield 'Wes';

yield 'Kait';

yield 'Snickers';

}

const people = listPeople();

* When being called, returns first item in an object with the value, and done which is a boolean. Call .next to get next values.
* Values declared in generator function retain their value until the generator is finished.
* TODO: review for in and for of
* can call .next().value if you don’t want the done value
* can put an asterisk in front of the function name if you prefer
* can use generator for multiple API calls

function ajax(url) {

fetch(url).then(data => data.json()).then(data => dataGen.next(data))

}

function\* steps() {

const beers = yield ajax('http://beer...');

const wes = yield ajax('http://wes...');

const fatJoe = yield ajax('http://joe...');

}

const dataGen = steps();

dataGen.next(); // kick it off

* Whatever gets passed from next will be returned to the variable in the generator function. The above example gets the three API requests in sequence.
* Instead of calling next repeatedly on a generator, can use for of—don’t need to call next.

function\* lyrics() {

yield "line 1";

yield "line 2";

yield "line 3";

}

const achy = lyrics()

// instead of calling .next .next .next can do the following

for (const line of achy) {

console.log(line);

}

// don't need to call next, automatically happens with a for of loop

### Generators example

function\* listPeople() {   
yield 'Wes';  
yield 'Kait';  
yield 'Snickers';   
}  
  
const people = listPeople(); // = Generator  
console.log(people.next()); // = {value, done}

### Using Generators for Ajax Flow Control Example

function ajax (url) {  
return fetch(url).then( data => data.json()).then( data => {  
apiGen.next(data);   
});   
}  
  
  
function\* steps() {   
const ale = yield(ajax('https://api.github.com/users/afalchi82'));   
console.log(ale);  
  
const prigara = yield(ajax('https://api.github.com/users/prigara'));   
console.log(prigara);  
  
const eenagy = yield(ajax('https://api.github.com/users/eenagy'));   
console.log(eenagy);  
}  
  
  
const apiGen = steps();  
  
apiGen.next();

### Looping Generators with for of example

function\* lyrics() {  
yield `Near, far, wherever you are`;  
yield `I believe that the heart does go on `;  
yield `Once more you open the door`;   
yield `And you're here in my heart and my heart will go on and on`;  
}  
  
const breakHearts = lyrics();  
  
for (const line of breakHearts) {  
console.log(line);  
}

# Sets and Weak Sets

* A set is a unique array, can’t add same value more than once; can’t access items individually, and is not index based

const people = new Set();

people.add('Wes');

pepole.add('Snickers');

people.add('Kait');

* Add with .add, remove with .remove
* Don’t have to know where index is to delete
* Can clear with .clear
* Use .size instead of length (because not index-based)
* .values returns a set iterator, which is a generator, can call .next and use for of to loop over it
* keys and values are the same thing in a set, different in a map
* can pass an array into a new set, or add values when creating a new set
* .has returns a boolean, whether value is present in set
* when calling .next on the iterator, values are removed, whereas the values remain in the set
* a weak set is the same as a set but it can only contain objects (not strings, arrays, etc.) and cannot enumerate over it (for example with for of); no .clear—elements gets GC’d when there’s no reference to an item in the set, can’t get size

# Map and Weak Map

* set is to array as map is to object
* .set to set a key/value, .get, .delete, .clear
* can loop over with .forEach and for of loop (returns an array for each element, where first value is key, second is value, can use destructuring)
* why is a map better than an ojbect? can use for of (probably coming soon in js)
* can be used to hold metadata about an object that you don’t want to store on the object
* weak maps get GC’d, not enumerable, can’t get size

const clickCounts = new Map();

const buttons = document.querySelectorAll('button');

buttons.forEach(button => {

clickCounts.set(button, 0);

});

**Async + Await Flow Control**

* sync means you wait for something to finish before moving on, async you don’t wait
* can be used instead of chaining then, catch
* async await requires a promise function; await must be called inside an async function
* when setting a variable or a constant const res = await breathe(1000) the constant will be set to the actual value rather than a promise
* cannot use await in the open, needs to be used in an async function, but no problem setting a regular function to async
* to catch error, use try { ...await } catch (err) { error... }
* can use higher order function so you don’t need to try catch each async await
* higher order function takes in a function as an argument returns a new function, in effect wrapping the function
* can use … in the higher order function so the function is flexible in that it can take any number of arguments
* great for when you’re handling a bunch of errors in the same way
* when dealing with multiple promises, can await Promise.all([promise1, promise2]);

async function go() {

const promise1 = fetch('http://p1').then(r => r.json());

const promise2 = fetch('http://p2').then(r => r.json());

const res = await Promise.all([promise1, promise2]);

}

go();

or could map over promises ``` const dataPromises = res.map(r => r.json()); const wesAndScott = await Promise.all(data.Promises);

// then syntax Promise .all([weather, tweets]) .then(responses => { console.log(responses) }) ```

* Promise.race resolves as soon as the first promise is resolved, stops
* TODO: Review then vs async await
* can promise-ify callback-based functions

# ES7, ES8 + Beyond

* If you want a property on the instance, have to put it in the constructor

TODO: What’s great about class properties?

* padStart and padEnd pad start or end of a string by adding spaces or whatever character is passed
* exponential operator 3 \*\* 3 is 33, previously had to use Math.pow(3, 3)—can’t do something like 3 \*\* 3 \*\* 3
* Trailing commas or comma dangle have been allowed in arrays and objects (benefit was changing 1 line instead of 2 in git); now allowed in function arguments function fam (mom, dad, children,)
* Object.keys has been around for a while, now can use Object.entries (keys and values) and Object.values; all iterables