Contents

[New Variables — Creation, Updating and Scoping 3](#_Toc36405191)

[let & const in real world 3](#_Toc36405192)

[var Scoping Refresher 4](#_Toc36405193)

[let versus const 4](#_Toc36405194)

[let and const in the Real World 4](#_Toc36405195)

[Temporal Dead Zone 6](#_Toc36405196)

[Is var Dead? What should I use? 6](#_Toc36405197)

[Function Improvements: Arrows and Default Arguments 6](#_Toc36405198)

[Arrow Functions Introduction 7](#_Toc36405199)

[More Arrow Function Examples 8](#_Toc36405200)

[Arrow Functions and ‘this’ 8](#_Toc36405201)

[Default Function Arguments 9](#_Toc36405202)

[when not use arrow functions 9](#_Toc36405203)

[Arrow Function Exercises 10](#_Toc36405204)

[arrow functions and 'this' 10](#_Toc36405205)

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  
});  
});