

# Modern JavaScript - pt1

Andrew.Kaluba

Jul '20



## Modern JavaScript (Tyler Mcginnis) - Video Tutorials

This is a “live” course that is updated as JavaScript updates. From destructuring objects and template strings to two new ways to declare variables, you'll never look at JavaScript the same way. In this course, you will learn all the latest ...

Jul 2020

1 / 1  
Jul 2020

## Introduction, Philosophy, and Tips

### Welcome!

Before we dive into the material, there are some important housekeeping items to mention first.

1. Unlike our other courses, the current version of this course is pretty hands off. That'll change in future versions. For now, enjoy the video, text, and quiz - then you'll be able to start applying the topic in the code you work on.
2. Unlike our other courses, the current version of this course doesn't have any curriculum. That will also change in future versions.
3. You can take this course regardless of if you're on Mac, Linux, or Windows. If you're wondering, I'm running Node v13.12.0 and NPM v6.14.4.
4. If you haven't already, request to join our [official private Facebook Group](#) for subscribers. It's where other members of the team and I will be hanging out answering support questions. Again, if your code isn't working after following a video, very carefully compare it to the specific commit for that video before asking a question in the facebook group.

Jul 2020



Good luck!

Tyler

## ECMAScript, TC39, and the Standardization Process

JavaScript is a living language that is constantly adding new features. As a JavaScript developer, it's important to understand the underlying process that's needed to take a new feature and transform it from a simple idea, to part of official language specification. To do that, we'll cover three topics - Ecma, EcmaScript, and the TC39.

First, let's take ourselves back to 1995. The cult classic [Heavy weights](#) was in theaters, Nicolas Cage won an Oscar, and websites typically looked [something like this](#). Now, odds are the way you would view that website would be with [Netscape Navigator](#). At the time, Netscape Navigator was the most popular web browser with almost 80% market share. The founder of Netscape, the company behind Netscape Navigator, was [Marc Andreessen](#). He has a vision for the future of the web and it was more than just a way to share and distribute documents. He envisioned a more dynamic platform with client side interactivity - a sort of “glue language” that was easy to use by both designers and developers. This is where [Brendan Eich](#) comes into the picture.

Brendan was recruited by Netscape with the goal of embedding the Scheme programming language into Netscape Navigator. However, before he could get started, Netscape collaborated with [Sun Microsystems](#) to make their up and coming programming language Java available in the browser. Now, if Java was already a suitable language, why bring on Brendan to create another one?. If you remember back to Netscape's goal, they wanted “a scripting language that was simple enough for designers and amateurs to use” - sadly, Java wasn't that. From there, the idea became that Java could be used by “professionals” and this new language “Mocha” (which was the initial name of JavaScript) would be used by everyone else. Because of this collaboration between languages, Netscape decided that Mocha needed to compliment Java and should have a relatively similar syntax.

From there, as the legend states, in just 10 days Brendan created the first version of Mocha. This version had some functionality from Scheme, the object orientation of SmallTalk, and because of the collaboration, the syntax of Java. Eventually the name Mocha changed to LiveScript then LiveScript changed to JavaScript as a marketing ploy to ride the

hype of Java. So at this point, JavaScript was marketed as a scripting language for the browser - accessible to both amateurs and designers while Java was the professional tool for building rich web components.

Now, it's important to understand the context of when these events were happening. Besides Nicolas Cage winning an Oscar, Microsoft was also working on Internet Explorer. Because JavaScript fundamentally changed the user experience of the web, if you were a competing browser, since there was no JavaScript specification, you had no choice but to come up with your own implementation. As history shows, that's exactly what Microsoft did and they called it JScript.

This then led to a pretty famous problem in the history of the internet. JScript filled the same use case as JavaScript, but its implementation was different. This meant that you couldn't build one website and expect it to work on both Internet Explorer and Netscape Navigator. In fact, the two implementations were so different that "Best viewed in Netscape" and "Best viewed in Internet Explorer" badges became common for most companies who couldn't afford to build for both implementations. Finally, this is where Ecma comes into the picture.

Ecma International is "an industry association founded in 1961, dedicated to the standardization of information and communication systems". In November of 1996, Netscape submitted JavaScript to Ecma to build out a standard specification. By doing this, it gave other implementors a voice in the evolution of the language and, ideally, it would help keep other implementations consistent across browsers.

Under Ecma, each new specification comes with a standard and a committee. In JavaScript's case, the standard is ECMA-262 and the committee who works on the ECMA-262 standard is the TC39. If you look up the ECMA262 standard, you'll notice that the term "JavaScript" is never used. Instead, they use the term "EcmaScript" to talk about the official language. The reason for this is because Oracle owns the trademark for the term "JavaScript". To avoid legal issues, Ecma decided to use the term EcmaScript instead. In the real world, ECMAScript is usually used to refer to the official standard, ECMA-262, while JavaScript is used when talking about the language in practice. As mentioned earlier, the committee which oversees the evolution of the Ecma262 standard is the TC39, which stands for Technical Committee 39. The TC39 is made up of "members" who are typically browser vendors and large companies who've invested heavily in the web like Facebook and PayPal. To attend the meetings, "members" (again, large companies and browser vendors) will send "delegates" to represent said company or browser. It's these delegates who are responsible for creating, approving, or denying language proposals.

When a new proposal is created, that proposal has to go through certain stages before it becomes part of the official specification. It's important to keep in mind that in order for any proposal to move from one stage to another, a consensus among the TC39 must be met. This means that a large majority must agree while nobody strongly disagrees enough to veto a specific proposal.

Each new proposal starts off at Stage 0. This stage is called the "Straw man" stage. Stage 0 proposals are "proposals which are planned to be presented to the committee by a TC39 champion or, have been presented to the committee and not rejected definitively, but have not yet achieved any of the criteria to get into stage 1." So the only requirement for becoming a Stage 0 proposal is that the document must be reviewed at a TC39 meeting. It's important to note that using a Stage 0 feature in your codebase is fine, but even if it does continue on to become part of the official spec, it'll almost certainly go through a few iterations before then.

The next stage in the maturity of a new proposal is Stage 1. In order to progress to Stage 1, an official "champion" who is part of TC39 must be identified and is responsible for the proposal. In addition, the proposal needs to describe the problem it solves, have illustrative examples of usage, a high level API, and identify any potential concerns and implementation challenges. By accepting a proposal for stage 1, the committee signals they're willing to spend resources to look into the proposal in more depth.

The next stage is Stage 2. At this point, it's more than likely that this feature will eventually become part of the official specification. In order to make it to stage 2, the proposal must, in formal language, have a description of the syntax and semantics of the new feature. In other words, a draft, or a first version of what will be in the official specification is written. This is the stage to really lock down all aspects of the feature. Future changes may still likely occur, but they should only be minor, incremental changes.

Next up is Stage 3. At this point the proposal is mostly finished and now it just needs feedback from implementors and users to progress further. In order to progress to Stage 3, the spec text should be finished and at least two spec compliant implementations must be created.

The last stage is Stage 4. At this point, the proposal is ready to be included in the official specification. To get to Stage 4, tests have to be written, two spec compliant implementations should pass those tests, members should have significant practical experience with the new feature, and the EcmaScript spec editor must sign off on the spec text. Basically once a proposal makes it to stage 4, it's ready to stop being a proposal and make its way into the official specification. This brings up the last thing you need to know about this whole process and that is TC39's release schedule.

As of 2016, a new version of ECMAScript is released every year with whatever features are ready at that time. What that means is that any Stage 4 proposals that exist when a new release happens, will be included in the release for that year. Because of this yearly release cycle, new features should be much more incremental and easier to adopt.

## Variable Declarations (var vs let vs const)

ES2015 (or ES6) introduced two new ways to create variables, `let` and `const`. But before we actually dive into the differences between `var`, `let`, and `const`, there are some prerequisites you need to know first. They are variable declarations vs initialization, scope (specifically function scope), and hoisting.

### Variable Declaration vs Initialization

A variable declaration introduces a new identifier.

```
var declaration
```

Above we create a new identifier called `declaration`. In JavaScript, variables are initialized with the value of `undefined` when they are created. What that means is if we try to log the `declaration` variable, we'll get `undefined`.

```
var declaration
```

```
console.log(declaration) // undefined
```

So if we log the `declaration` variable, we get `undefined`.

In contrast to variable declaration, variable initialization is when you first assign a value to a variable.

```
var declaration
```

```
console.log(declaration) // undefined
```

```
declaration = 'This is an initialization'
```

So here we're initializing the `declaration` variable by assigning it to a string.

This leads us to our second concept, Scope.

### Scope

Scope defines where variables and functions are accessible inside of your program. In JavaScript, there are two kinds of scope - **global scope**, and **function scope**. According to the official spec,

"If the variable statement occurs inside a `FunctionDeclaration`, the variables are defined with function-local scope in that function."

What that means is if you create a variable with `var`, that variable is "scoped" to the function it was created in and is only accessible inside of that function or, any nested functions.

```
function getDate () {  
  var date = new Date()  
  
  return date  
}
```

```
getDate()  
console.log(date) // ❌ Reference Error
```

Above we try to access a variable outside of the function it was declared. Because `date` is "scoped" to the `getDate` function, it's only accessible inside of `getDate` itself or any nested functions inside of `getDate` (as seen below).

```
function getDate () {
  var date = new Date()

  function formatDate () {
    return date.toString().slice(4) // ✓
  }

  return formatDate()
}

getDate()
console.log(date) // ✗ Reference Error
```

Now let's look at a more advanced example. Say we had an array of prices and we needed a function that took in that array as well as a discount and returned us a new array of discounted prices. The end goal might look something like this.

```
discountPrices([100, 200, 300], .5) // [50, 100, 150]
```

And the implementation might look something like this

```
function discountPrices (prices, discount) {
  var discounted = []

  for (var i = 0; i < prices.length; i++) {
    var discountedPrice = prices[i] * (1 - discount)
    var finalPrice = Math.round(discountedPrice * 100) / 100
    discounted.push(finalPrice)
  }

  return discounted
}
```

Seems simple enough but what does this have to do with block scope? Take a look at that for loop. Are the variables declared inside of it accessible outside of it? Turns out, they are.

```
function discountPrices (prices, discount) {
  var discounted = []

  for (var i = 0; i < prices.length; i++) {
    var discountedPrice = prices[i] * (1 - discount)
    var finalPrice = Math.round(discountedPrice * 100) / 100
    discounted.push(finalPrice)
  }

  console.log(i) // 3
  console.log(discountedPrice) // 150
  console.log(finalPrice) // 150

  return discounted
}
```

If JavaScript is the only programming language you know, you may not think anything of this. However, if you're coming to JavaScript from another programming language, specifically a programming language that is blocked scope, you're probably a little bit concerned about what's going on here. It's not really broken, it's just kind of weird. There's not really a reason to still have access to `i`, `discountedPrice`, and `finalPrice` outside of the for loop. It doesn't really do us any good and it may even cause us harm in some cases. However, since variables declared with `var` are function scoped, you do.

Now that we've discussed variable declarations, initializations, and scope, the last thing we need to flush out before we dive into `let` and `const` is hoisting.

## Hoisting

Remember earlier we said that "In JavaScript, variables are initialized with the value of `undefined` when they are created.". Turns out, that's all that "Hoisting" is. The JavaScript interpreter will assign variable declarations a default value of `undefined` during what's called the "Creation" phase.

For a much more in depth guide on the Creation Phase, Hoisting, and Scopes see ["The Ultimate Guide to Hoisting, Scopes, and Closures in JavaScript"](#)

Let's take a look at the previous example and see how hoisting affects it.

```
function discountPrices (prices, discount) {
  var discounted = undefined
  var i = undefined
  var discountedPrice = undefined
  var finalPrice = undefined

  discounted = []
  for (i = 0; i < prices.length; i++) {
    discountedPrice = prices[i] * (1 - discount)
    finalPrice = Math.round(discountedPrice * 100) / 100
    discounted.push(finalPrice)
  }

  console.log(i) // 3
  console.log(discountedPrice) // 150
  console.log(finalPrice) // 150

  return discounted
}
```

Notice all the variable declarations were assigned a default value of `undefined`. That's why if you try access one of those variables **before** it was actually declared, you'll just get `undefined`.

```
function discountPrices (prices, discount) {
  console.log(discounted) // undefined
  var discounted = []

  for (var i = 0; i < prices.length; i++) {
    var discountedPrice = prices[i] * (1 - discount)
    var finalPrice = Math.round(discountedPrice * 100) / 100
    discounted.push(finalPrice)
  }

  console.log(i) // 3
  console.log(discountedPrice) // 150
  console.log(finalPrice) // 150

  return discounted
}
```

Now that you know everything there is to know about `var`, let's finally talk about the whole point of why you're here, what's the difference between `var`, `let`, and `const`?

## var VS let VS const

First, let's compare `var` and `let`. The main difference between `var` and `let` is that instead of being function scoped, `let` is block scoped. What that means is that a variable created with the `let` keyword is available inside the "block" that it was created in as well as any nested blocks. When I say "block", I mean anything surrounded by a curly brace `{}` like in a `for` loop or an `if` statement.

So let's look back to our `discountPrices` function one last time.

```
function discountPrices (prices, discount) {
  var discounted = []

  for (var i = 0; i < prices.length; i++) {
    var discountedPrice = prices[i] * (1 - discount)
    var finalPrice = Math.round(discountedPrice * 100) / 100
    discounted.push(finalPrice)
  }

  console.log(i) // 3  console.log(discountedPrice) // 150  console.log(finalPrice) // 150
  return discounted
}
```

Remember that we were able to log `i`, `discountedPrice`, and `finalPrice` outside of the `for` loop since they were declared with `var` and `var` is function scoped. But now, what happens if we change those `var` declarations to use `let` and try to run it?

```
function discountPrices (prices, discount) {
  let discounted = []

  for (let i = 0; i < prices.length; i++) {
    let discountedPrice = prices[i] * (1 - discount)
    let finalPrice = Math.round(discountedPrice * 100) / 100
    discounted.push(finalPrice)
  }

  console.log(i)  console.log(discountedPrice)  console.log(finalPrice)
  return discounted
}
```

```
discountPrices([100, 200, 300], .5) // ❌ ReferenceError: i is not defined
```

🙄 We get `ReferenceError: i is not defined`. What this tells us is that variables declared with `let` are block scoped, not function scoped. So trying to access `i` (or `discountedPrice` or `finalPrice`) outside of the "block" they were declared in is going to give us a reference error as we just barely saw.

## `var` VS `let`

**`var`:** function scoped

**`let`:** block scoped

The next difference has to do with Hoisting. Earlier we said that the definition of hoisting was "The JavaScript interpreter will assign variable declarations a default value of `undefined` during what's called the 'Creation' phase." We even saw this in action by logging a variable before it was declared (you get `undefined`)

```
function discountPrices (prices, discount) {
  console.log(discounted) // undefined
  var discounted = []

  for (var i = 0; i < prices.length; i++) {
    var discountedPrice = prices[i] * (1 - discount)
    var finalPrice = Math.round(discountedPrice * 100) / 100
  }
```

```

    discounted.push(finalPrice)
  }

  console.log(i) // 3
  console.log(discountedPrice) // 150
  console.log(finalPrice) // 150

  return discounted
}

```

I can't think of any use case where you'd actually want to access a variable before it was declared. It seems like throwing a `ReferenceError` would be a better default than returning `undefined`. In fact, this is exactly what `let` does. If you try to access a variable declared with `let` before it's declared, instead of getting `undefined` (like with those variables declared with `var`), you'll get a `ReferenceError`.

```

function discountPrices (prices, discount) {
  console.log(discounted) // ❌ ReferenceError
  let discounted = []

  for (let i = 0; i < prices.length; i++) {
    let discountedPrice = prices[i] * (1 - discount)
    let finalPrice = Math.round(discountedPrice * 100) / 100
    discounted.push(finalPrice)
  }

  console.log(i) // 3
  console.log(discountedPrice) // 150
  console.log(finalPrice) // 150

  return discounted
}

```

var VS `let`

var:

function scoped  
undefined when accessing a variable before it's declared

let:

block scoped  
ReferenceError when accessing a variable before it's declared

## let VS const

Now that you understand the difference between `var` and `let`, what about `const`? Turns out, `const` is almost exactly the same as `let`. However, the only difference is that once you've assigned a value to a variable using `const`, you can't reassign it to a new value.

```


let name = 'Tyler'
const handle = 'tylermcginnis'

name = 'Tyler McGinnis' // ✅
handle = '@tylermcginnis' // ❌ TypeError: Assignment to constant variable.

```

The take away above is that variables declared with `let` can be re-assigned, but variables declared with `const` can't be.

Cool, so anytime you want a variable to be immutable, you can declare it with `const`. Well, not quite. Just because a variable is declared with `const` doesn't mean it's immutable, all it means is the value can't be re-assigned. Here's a good example.

```
const person = {  
  name: 'Kim Kardashian'  
}  
  
person.name = 'Kim Kardashian West' //   
  
person = {} //  Assignment to constant variable.
```

Notice that changing a property on an object isn't reassigning it, so even though an object is declared with `const`, that doesn't mean you can't mutate any of its properties. It only means you can't reassign it to a new value.

Now the most important question we haven't answered yet, should you use `var`, `let`, or `const`? The most popular opinion, and the opinion that I subscribe to, is that you should always use `const` unless you know the variable is going to change. The reason for this is by using `const`, you're signalling to your future self as well as any other future developers that have to read your code that this variable shouldn't change. If it will need to change (like in a `for` loop), you should use `let`.

So between variables that change and variables that don't change, there's not much left. That means you shouldn't ever have to use `var` again.

Now the unpopular opinion, though it still has some validity to it, is that you should never use `const` because even though you're trying to signal that the variable is immutable, as we saw above, that's not entirely the case. Developers who subscribe to this opinion always use `let` unless they have variables that are actually constants like `_LOCATION_ = ...`.

So to recap, `var` is function scoped and if you try to use a variable declared with `var` before the actual declaration, you'll just get `undefined`. `const` and `let` are block scoped and if you try to use variable declared with `let` or `const` before the declaration you'll get a `ReferenceError`. Finally the difference between `let` and `const` is that once you've assigned a value to `const`, you can't reassign it, but with `let`, you can.

`var` VS `let` VS `const`

`var`:  
function scoped  
undefined **when** accessing a variable before it's declared

`let`:  
block scoped  
**ReferenceError** when accessing a variable before it's declared

`const`:  
block scoped  
**ReferenceError** **when** accessing a variable before it's declared  
**can't** be reassigned

## Object and Array Destructuring

In this post, we'll cover an ES2015 feature called `destructuring`. To better understand it, let's take a look at some of the basics of Javascript objects. To add a single property to an object, you use dot notation. With dot notation, you can only add properties to an object one at a time. The same syntax can be used to extract data, again, one property at a time.

```
const user = {};  
user.name = 'Tyler McGinnis';  
user.handle = '@tylermcginnis';  
user.location = 'Eden, Utah';  
  
const name = user.name;  
const handle = user.handle;
```

If you wanted to add multiple properties to an object at the same time, you would need to use JavaScript's "object literal notation" when you initialize the object.



```
const user = {
  name: 'Tyler McGinnis';
  handle: '@tylermcginnis';
  location: 'Eden, Utah';
};

const name = user.name;
const handle = user.handle;
```

There's a way to add properties one at a time, extract properties one at a time, add multiple properties at the same time, but unfortunately, there's no comparable way to extract multiple properties from an object at the same time. That is, until "destructuring" was introduced in ES2015. **Destructuring allows us to extract multiple properties from an object**. This can drastically decrease the amount of code we need to write when we want to extract data from an object, because what used to look like this,

```
const name = user.name;
const handle = user.handle;
const location = user.location;
```

can now look like this,

```
const { name, handle, location } = user;
```

The syntax can be a little bit weird but know that these two blocks of code are identical in that they both create and initialize three new variables. You can think of it like this, if you want to add properties to an object, do it as you are used to, on the right-hand side of the equal sign. If you want to extract properties from an object, do it on the left-hand side of the equal sign.

Destructuring also allows you to destructure the results of function invocations. For example, below we have a function called `getUser()` which returns the user object. Rather than invoking `getUser()` and grabbing all of the properties off of it one by one, we could get the same result by destructuring the result of that invocation.

```
function getUser () {
  return {
    name: 'Tyler McGinnis',
    handle: '@tylermcginnis',
    location: 'Eden, Utah'
  };
}

const { name, handle, location } = getUser();
```

Up until this point we've talked about how destructuring helps us extract data from objects, but what about arrays? Though not as common as object destructuring, array destructuring is a thing and it is still pretty useful in certain circumstances, specifically when the location of an item in the array is the main differentiator for that item. So here we have a user array with each item being a unique piece of information about the user,

```
const user = ['Tyler McGinnis', '@tylermcginnis', 'Eden, Utah'];
```

You'll notice that this array probably should just be an object. But sometimes you have to take what you can get from weird external API's. Typically if we want to better identify each item in the array we need to create a variable for each item.

```
const name = user[0];
const handle = user[1];
const location = user[2];
```

However just like with objects, array destructuring allows us to more effectively extract items from an array so the above code, can now look like the code below.

```
const [ name, handle, location ] = user;
```

Just as we saw from objects you can use array destructuring with function invocations. For example, below “split” is going to return an array with each item in the array being a specific property of the car.

```
const cvs = '1997,Ford,F350,MustSell!'
const [ year, make, model, description ] = cvs.split(',');
```

By using array destructuring, we are able to extract each property into their own, user readable variable.

So that's it in regards to the basics of destructuring, again destructuring allows us to easily extract data from an object or an array. There are, however, what I'd consider to be more advanced features of destructuring that are worth taking a look at.

For example, what if when we do destructure an object, we wanted the variable name to be different than the property name on that object. So say we had an object that looked like this,

```
const user = {
  n: 'Tyler McGinnis',
  h: '@tylermcginnis',
  l: 'Eden, Utah'
};
```

Since we are not masochists and we actually like the other developers on our team, we don't want to make three one letter variable names. Instead, we can have the property names on the left of the colon and the new variable names on the right. Now, we are not only destructuring the user object, but we are also renaming the poorly named properties into more easily understood variable names.

```
const { n: name, h: handle, l: location } = user;
console.log(name) // Tyler McGinnis
console.log(handle) // @tylermcginnis
console.log(location) // Eden, Utah
```

This may seem like a rarely used feature, but it is actually pretty common. To find a real world example we don't have to look very far. This is the implementation of the render method in React Router Native's Link component. Note how we're renaming component with a lowercase “c” to Component with a capitalized “C”.

```
render () {
  const { component: Component, to, replace, ...rest } = this.props
  return <Component {...rest} onPress={this.handlePress}/>
}
```

Next, let's talk about function arguments and parameters. Below we have a fetchRepos() function which is going to be in charge of fetching a group of repositories from the Github API.

```
function fetchRepos (language, minStars, maxStars, createdBefore, createAfter) {
}
```

The first thing you'll notice is that we have a lot of control over the type of repositories that we will be fetching. Fortunately, this leads to a stupid amount of arguments that can be passed into the function. Currently when we invoke our fetchRepos() function, we have two issues. First, we need to remember or look up which arguments go in which order. Second, we need to read and hope that the documentation has instructions for what to do with our arguments that we do not care about. In this case, we will just use null and hope for the best.

```
function fetchRepos (language, minStars, maxStars, createdBefore, createAfter) {
}

fetchRepos('JavaScript', 100, null, new Date('01.01.2017').getTime(), null);
```

The good news is that destructuring helps us with both of these problems. First, let's solve the positional parameters problem. What if instead of passing in each argument one by one, we pass in an object instead? Now, before we ever need

to look at the function definition of `fetchRepos`, we know exactly what information it needs. Even more important, order no longer matters.

```
function fetchRepos (language, minStars, maxStars, createdBefore, createAfter) {  
  
}  
  
fetchRepos({  
  language: 'JavaScript',  
  maxStars: null,  
  createdAfter: null,  
  createdBefore: new Date('01/01/2017').getTime(),  
  minStars: 100,  
});
```

Now we need to modify the `fetchRepos` function definition. This is where destructuring comes into play. Because we are receiving an object as the argument to the function, we can destructure it. So now the code above, can be changed to this.

```
function fetchRepos ({ language, minStars, maxStars, createdBefore, createAfter }) {  
  
}  
  
fetchRepos({  
  language: 'JavaScript',  
  maxStars: null,  
  createdAfter: null,  
  createdBefore: new Date('01/01/2017').getTime(),  
  minStars: 100,  
});
```

Again, the biggest benefit here is that we have removed the order out of the equation entirely, so that's one less thing we have to worry about.

The second problem we had earlier with our code was that we needed to figure out what to do with the arguments we did not care about. Before we just passed in `null`, but now that we are passing in an object rather than arguments one by one, we can actually just remove the `null` values altogether and that will give us a function invocation that looks like this.

```
function fetchRepos ({ language, minStars, maxStars, createdBefore, createAfter }) {  
  
}  
  
fetchRepos({  
  language: 'JavaScript',  
  createdBefore: new Date('01/01/2017').getTime(),  
  minStars: 100,  
});
```

This now leads us back to our function definition of `fetchRepos`. We need a way to establish default values for any properties that aren't on the arguments object when the function is invoked. Typically that would look like this.

```
function fetchRepos ({ language, minStars, maxStars, createdBefore, createAfter }) {  
  language = language || 'All';  
  minStars = minStars || 0;  
  maxStars = maxStars || '';  
  createdBefore = createdBefore || '';  
  createdAfter = createdAfter || '';  
}  
  
fetchRepos({  
  language: 'JavaScript',  
  createdBefore: new Date('01/01/2017').getTime(),
```

```

    minStars: 100,
  });

```

For each different possible property, we'd set the value of that property to itself or a default value if the original value was undefined. Luckily for us, another feature of destructuring is it allows you to set default values for any properties. If a partially destructured value is undefined, it will default to whatever you specify. What that means is that the ugly code above can be transformed into this,

```

function fetchRepos({ language='All', minStars=0, maxStars='', createdBefore='', createdAf
}

```

We set the default value of each property in the same place where we just destructured the parameters. Now that we've seen the power of using object destructuring to destructure an object's parameters, can the same thing be done with array destructuring? Turns out, it can.

My favorite example of this is with `Promise.all`. Below we have a `getUserData` function.

```

function getUserData (player) {
  return Promise.all([
    getProfile(player),
    getRepos(player)
  ]).then(function (data) {
    const profile = data[0];
    const repos = data[1];

    return {
      profile: profile,
      repos: repos
    }
  })
}

```

Notice it's taking in a `player` and returning us the invocation of calling `Promise.all`. Both `getProfile` and `getRepos` return a promise. The whole point of this `getUserData` function is that it's going to take in a `player` and return an object with that `player`'s profile as well as that `player`'s repositories. If you're not familiar with the `Promise.all` API, what's going to happen here is `getProfile` and `getRepos` are both asynchronous functions. When those promises resolve (or when we get that information back from the Github API), the function that we passed to `then` is going to be invoked receiving an array (in this case we are calling it `data`). The first element in that array is going to be the user's profile and the second item in the array is going to be the user's repositories. You'll notice that order matters here. For example, if we were to pass another invocation to `Promise.all`, say `getUsersFollowers`, then the third item in our data array would be their followers.

The first update we can make to this code is we can destructure our data array. Now we still have our `profile` and `repos` variables, but instead of plucking out the items one by one, we destructure them.

```

function getUserData (player) {
  return Promise.all([
    getProfile(player),
    getRepos(player)
  ]).then(function (data) {
    const [ profile, repos ] = data
    return {
      profile: profile,
      repos: repos
    }
  })
}

```

Now just as we saw with objects, we can move that destructuring into the parameter itself.

```
function getUserData (player) {
  return Promise.all([
    getProfile(player),
    getRepos(player)
  ]).then(([ profile, repos ]) => {
    return {
      profile: profile,
      repos: repos
    }
  })
}
```

Now we still have `profile` and `repos` , but those are being created with array destructuring inside of the function's parameters.

## Shorthand Properties and Method Names

ES6 introduced two new features to make objects more concise - Shorthand Properties and Shorthand Method Names.

### Shorthand Properties

With Shorthand Properties, whenever you have a variable which is the same name as a property on an object, when constructing the object, you can omit the property name.

What that means is that code that used to look like this,

```
function formatMessage (name, id, avatar) {
  return {
    name: name,    id: id,    avatar: avatar,    timestamp: Date.now()
  }
}
```

can now look like this.

```
function formatMessage (name, id, avatar) {
  return {
    name,    id,    avatar,    timestamp: Date.now()
  }
}
```

### Shorthand Method Names

Now, what if one of those properties was a function?

A function that is a property on an object is called a method. With ES6's Shorthand Method Names, you can omit the `function` keyword completely. What that means is that code that used to look like this,

```
function formatMessage (name, id, avatar) {
  return {
    name,
    id,
    avatar,
    timestamp: Date.now(),
    save: function () {      // save message    }  }
}
```

can now look like this

```
function formatMessage (name, id, avatar) {
  return {
    name,
```

```

    id,
    avatar,
    timestamp: Date.now(),
    save () {          //save message    }  }
}

```

Both Shorthand Properties and Shorthand Method Names are just syntactic sugar over the previous ways we used to add properties to an object. However, because they're such common tasks, even the smallest improvements eventually add up.

## Computed Property Names

ES6's "Computed Property Names" feature allows you to have an expression (a piece of code that results in a single value like a variable or function invocation) be computed as a property name on an object.

For example, say you wanted to create a function that took in two arguments ( `key` , `value` ) and returned an object using those arguments. Before Computed Property Names, because the property name on the object was a variable ( `key` ), you'd have to create the object first, then use bracket notation to assign that property to the value.

```

function objectify (key, value) {
  let obj = {}
  obj[key] = value
  return obj
}

objectify('name', 'Tyler') // { name: 'Tyler' }

```

However, now with Computed Property Names, you can use object literal notation to assign the expression as a property on the object without having to create it first. So the code above can now be rewritten like this.

```

function objectify (key, value) {
  return {
    [key]: value
  }
}

objectify('name', 'Tyler') // { name: 'Tyler' }

```

Where `key` can be any expression as long as it's wrapped in brackets, `[]` .

## Template Literals

String concatenation is hard. Take this code for example.

```

function makeGreeting (name, email, id) {
  return 'Hello, ' + name +
    '. We\'ve emailed you at ' + email +
    '. Your user id is ' + id + '."'
}

```

All we're trying to do is take three variables ( `name` , `email` , and `id` ) and create a sentence using them. Sadly, in order to do that, it's a balancing act between using the right quotations, + signs, and escaping ( `\` ) the right characters. This is the exact problem that Template Literals (also called Template Strings) was created to solve.

With Template Literals, instead of using single ( `' '` ) or double quotes ( `" "` ), you use backticks ( ``` ) (located to the left of the `1` key if you're using a QWERTY keyboard 😊). Anywhere inside of your backticks where you have an expression (a piece of code that results in a single value like a variable or function invocation), you can wrap that expression in `${expression goes here}` .

So using Template Literals, we can take the confusing `makeGreeting` function above and simplify it to look like this.

```
function makeGreeting (name, email, id) {
  return `Hello, ${name}. We've emailed you at ${email}. Your user id is "${id}".`
}
```

Much better. No more worrying about using the right quotations, + signs, and escaping the right characters. Not only is it easier to write, but it's also much easier to read.

Now instead of having a makeGreeting function, say we wanted a makeGreetingTemplate function that returned us an HTML string that we could throw into the DOM. Without template strings, we'd have something like this.

```
function makeGreetingTemplate (name, email, id) {
  return '<div>' +
    '<h1>Hello, ' + name + '.</h1>' +
    '<p>We\'ve emailed you at ' + email + ' . ' +
    'Your user id is "' + id + '".</p>' +
    '</div>'
}
```

Perfect, except for the fact that not only is it terribly hard to write, it's even harder to read. What's nice about ES6's Template Strings is they also support multi-line strings. That means, using Template Strings, we can rewrite makeGreetingTemplate to look like this.

```
function makeGreetingTemplate (name, email, id) {
  return `
    <div>
      <h1>Hello, ${name}</h1>
      <p>
        We've email you at ${email}.
        Your user id is "${id}".
      </p>
    </div>
  `
}
```

I consider that an absolute win.

## Arrow Functions

Arrow functions provide two main benefits over regular functions. First, they're more terse. Second, they make managing the `this` keyword a little easier.

What I've seen with new developers learning about Arrow Functions is that it's not really the concept itself that's difficult to grasp. Odds are you're already familiar with functions, their benefits, use cases, etc. However, for some reason, it's the actual syntax that throws your brain for a loop when you're first exposed to them. Because of that, we're going to take things slow and first just introduce how the syntax compares with typical functions you're used to.

Here we have a very basic function declaration and a function expression.

```
// fn declaration
function add (x,y) {
  return x + y;
}

// fn expression
var add = function (x,y) {
  return x + y;
}
```

Now, if we wanted to change that function expression to an arrow function, we'd do it like this.

```
var add = function (x,y) {
  return x + y;
}
```

```
var add = (x,y) => {
  return x + y;
}
```

Again, the most difficult part about getting started with arrow functions is just getting used to the syntax. Once you're cool with it, move on and we'll dive deeper.

At this point you may be wondering what all the hype is about with arrow functions. Truthfully, the example above doesn't really lend well to their strengths. What I've found is that arrow functions really thrive when you're using anonymous functions. We can warm our brain up a little more to the syntax by looking at another basic example of this is using `.map`.

```
users.map(function () {

})

users.map(() => {

})
```

Alright enough with the warm up. Let's dive into it.

Let's say we had a `getTweets` function that took in a user id and, after hitting a poorly designed API, returned us all of the user's Tweets with over 50 stars and retweets. Using promise chaining, that function may look something like this,

```
function getTweets (uid) {
  return fetch('https://api.users.com/' + uid)
    .then(function (response) {
      return response.json()
    })
    .then(function (response) {
      return response.data
    }).then(function (tweets) {
      return tweets.filter(function (tweet) {
        return tweet.stars > 50
      })
    }).then(function (tweets) {
      return tweets.filter(function (tweet) {
        return tweet.rts > 50
      })
    })
}
```

Well, it works. But it's not the prettiest function in the world 🙄. Even though this specific implementation is kind of dense, the idea is all too common. Let's take a look at how what we know about arrow functions thus far, can improve our `getTweets` function.

```
function getTweets (uid) {
  return fetch('https://api.users.com/' + uid)
    .then((response) => {
      return response.json()
    })
    .then((response) => {
      return response.data
    }).then((tweets) => {
      return tweets.filter((tweet) => {
        return tweet.stars > 50
      })
    })
}
```



```

    }).then((tweets) => {
      return tweets.filter((tweet) => {
        return tweet.rts > 50
      })
    })
  })
}

```

OK, cool. It looks basically the same we just didn't have to type `function`. Beneficial, but nothing worth Tweeting about. Let's look at the next benefit of arrow functions, "implicit returns".

With arrow functions, if your function has a "concise body" (a fancy way for saying one line function), then you can omit the "return" keyword and the value will be returned automatically (or implicitly).

So the add example from earlier can be updated to look like this,

```

var add = function (x,y) {
  return x + y;
}

var add = (x,y) => x + y;

```

and more importantly, the `getTweets` example can be update to look like this,

```

function getTweets (uid) {
  return fetch('https://api.users.com/' + uid)
    .then((response) => response.json())
    .then((response) => response.data)
    .then((tweets) => tweets.filter((tweet) => tweet.stars > 50))
    .then((tweets) => tweets.filter((tweet) => tweet.rts > 50))
}

```

Now we're talking . That code is not only much easier to write, but more importantly, it's much easier to read.

Now, one further change we can make is that if the arrow function only has one parameter, you can omit the `()` around it. With that in mind, `getTweets` now looks like this,

```

function getTweets (uid) {
  return fetch('https://api.users.com/' + uid)
    .then(response => response.json())
    .then(response => response.data)
    .then(tweets => tweets.filter(tweet => tweet.stars > 50))
    .then(tweets => tweets.filter(tweet => tweet.rts > 50))
}

```

Overall, I'd say that's a huge improvement in just about every category.

The next benefit of arrow functions is how they manage the `this` keyword. If you're not familiar with the `this` keyword, check out this post - [Understanding the this keyword, call, apply, and bind in JavaScript](#).

Let's take a look at some typical React code.

```

class Popular extends React.Component {
  constructor(props) {
    super();
    this.state = {
      repos: null,
    };

    this.updateLanguage = this.updateLanguage.bind(this);
  }
  componentDidMount () {

```

```

    this.updateLanguage('javascript')
  }
  updateLanguage(lang) {
    api.fetchPopularRepos(lang)
      .then(function (repos) {
        this.setState(function () {
          return {
            repos: repos
          }
        });
      });
  }
  render() {
    // Stuff
  }
}

```

When the component mounts, it's making an API request (to the Github API) to fetch JavaScript's most popular repositories. When it gets the repositories, it takes them and updates the component's local state, or at least that's what we want it to do. Unfortunately, it doesn't do that. Instead, we get an error. Can you spot the bug?

The error the code above is going to throw is "cannot read setState of undefined". Now, **why** that's happening is outside the scope of this post (again, read or watch [Understanding the this keyword, call, apply, and bind in JavaScript](#) if you need it) but a typical ES5 solution uses `.bind` and looks something like this

```

class Popular extends React.Component {
  constructor(props) {
    super();
    this.state = {
      repos: null,
    };

    this.updateLanguage = this.updateLanguage.bind(this);
  }
  componentDidMount () {
    this.updateLanguage('javascript')
  }
  updateLanguage(lang) {
    api.fetchPopularRepos(lang)
      .then(function (repos) {
        this.setState(function () {
          return {
            repos: repos
          }
        });
      }).bind(this));
  }
  render() {
    // Stuff
  }
}

```

This is the second major benefit as to why arrow functions are great, they don't create **their own context**. What that means is that typically the `this` keyword Just Works™ without you having to worry about what context a specific function is going to be invoked in. So by using arrow functions in the `updateLanguage` method, we don't have to worry about `this` which means we don't have to call `.bind` anymore.

```

updateLanguage(lang) {
  api.fetchPopularRepos(lang)
    .then((repos) => {
      this.setState(() => {

```

```

        return {
          repos: repos
        }
      });
    });
  }
}

```



### Nice to knows

At this point, we've covered all of the "need to knows" about arrow functions. There are, however, two different "nice to knows" that I think are worth mentioning.

Looking at the `updateLanguage` method again, if we wanted to implicitly return the object inside of the `setState` callback, how would we do that? Your first intuition would be to remove the `return` statement and just return an object.

```

api.fetchPopularRepos(lang)
  .then((repos) => {
    this.setState(() => {
      repos: repos
    });
  });
}

```

The problem with this, as you probably guessed, is that that syntax is the exact same as creating a function body. JavaScript can't magically tell the difference between when you want to create a function body and when you want to return an object so it'll throw an error. To fix this, we can wrap the object inside of `()`.

```

api.fetchPopularRepos(lang)
  .then((repos) => {
    this.setState(() => ({
      repos: repos
    }));
  });
}

```

Now, with that syntax, we can use an arrow function to implicitly return an object.

Now I know if I don't put this, someone will mention it. As a bonus since we're using ES6, we can go ahead use ES6's [shorthand property and method names](#) feature to get rid of the `repos: repos` and use Arrow Function's implicit return to shorten it up a bit.

```

api.fetchPopularRepos(lang)
  .then((repos) =>
    this.setState(() => repos)
  );
}

```

Next tip. Say we wanted to examine the previous state of the component inside of `setState` by logging it. If this was your `setState` function, how would you approach logging `nextState`?

```

this.setState((nextState) => ({
  repos: repos
}));

```

The obvious move would be to change your implicit return to an explicit return, create a function body, then log inside of that body.

```

this.setState((nextState) => {
  console.log(nextState)
  return {
    repos: repos
  }
}

```

```
}  
});
```

Well, that's pretty annoying. There is a better way though and it's done using the `||` operator. Instead of messing with all of your code, you can do something like this

```
this.setState((nextState) => console.log(nextState) || ({  
  repos: repos  
}));
```

So clever.

## Default Parameters

Often times when writing a function, you need to assign default values for arguments that weren't passed to the function when it was invoked.

For example, let's say we were creating a `calculatePayment` function. This function has three parameters, `price`, `salesTax`, and `discount`. The purpose of this function, as the name implies, is to calculate the final price of a bill taking into account the initial price as well as any sales tax or discounts that should be applied.

With a function like this, the only parameter that we want to make required is the `price`. We'll set the default value of `salesTax` to `0.05` (5%) and the default value of `discount` to `0` so our function will still work if those values aren't passed in when the function is invoked. This way, the consumer of this function can supply a sales tax as well as a discount if they want, but if they don't, the default values will kick in.

```
calculatePayment(10) // 9.50  
calculatePayment(10, 0, 10) // 9.00
```

Historically, one way you could accomplish this is by using the Logical `||` operator.

```
function calculatePayment (price, salesTax, discount) {  
  salesTax = salesTax || 0.05  
  discount = discount || 0  
  
  // math  
}
```

If you're not familiar with `||`, you can think of it like you would an `if` statement checking for falsy values.

```
function calculatePayment (price, salesTax, discount) {  
  if (!salesTax) {  
    salesTax = 0.05  
  }  
  
  if (!discount) {  
    discount = 0  
  }  
  
  // math  
}
```

However, this approach has some downsides. Can you spot the issue? What if we wanted to set the `salesTax` to `0`? With our current implementation that would be impossible since `0` is classified as a falsy value so our `if (!salesTax)` would always evaluate to true setting the `salesTax` to our default value of `0.05`. To fix this, let's check for `undefined` rather than falsy.

```
function calculatePayment (price, salesTax, discount) {  
  salesTax = typeof salesTax === 'undefined' ? 0.05 : salesTax  
  discount = typeof discount === 'undefined' ? 0 : discount
```

```
// math
}
```

Now, both `salesTax` and `discount` will only take on their default values if their arguments are undefined .

At this point our code works well, but as you'll see, there's now a better way to do this with ES6's "Default Parameters".

## Default Parameters

Default Parameters allow you to set the default values for any parameters that are undefined when a function is invoked. Using Default Parameters, we can now update our `calculatePayment` function to look like this,

```
function calculatePayment (price, salesTax = 0.05, discount = 0) {

  // math
}
```

Now, just as we had before, if `salesTax` or `discount` are undefined when `calculatePayment` is invoked, they'll be set to their default values of `0.05` and `0` .

## Required Arguments

One neat trick you can do using Default Parameters is to throw an error if a function is invoked without a required argument. For example, what if we wanted `calculatePayment` to throw an error if the price wasn't specified when it was invoked?

To do this, first create the function that will throw the error.

```
function isRequired (name) {
  throw new Error(`${name}` is required)
}
```

Next, using Default Parameters, assign the required parameter to the invocation of `isRequired`

```
function calculatePayment (
  price = isRequired('price'), salesTax = 0.05,
  discount = 0
) {

  // math
}
```

Now if `calculatePayment` is invoked without a `price` , JavaScript will invoke the `isRequired` function, throwing the error. Clever? Totally. A good idea? I'll leave that up to you.

## Compiling vs Polyfills with Babel

JavaScript is a living language that's constantly progressing. As a developer, this is great because we're constantly learning and our tools are constantly improving. The downside of this is that it typically takes browsers a few years to catch up. Whenever a new language proposal is created, it needs to go through five different stages before it's added to the official language specification. Once it's part of the official spec, it still actually needs to be implemented into every browser you think your users will use. Because of this delay, if you ever want to use the newest JavaScript features, you need to either wait for the latest browsers to implement them (and then hope your users don't use older browsers) or you need to use a tool like Babel to compile your new, modern code back to code that older browsers can understand. When it's framed like that, it's almost certain that a compiler like Babel will always be a fundamental part of building JavaScript applications, assuming the language continues to progress. Again, the purpose of Babel is to take your code which uses new features that browsers may not support yet, and transform it into code that any browser you care about can understand.

So for example, code that looks like this,

```
const getProfile = username => {
  return fetch(`https://api.github.com/users/${username}`)
    .then((response) => response.json())
    .then(({ data }) => ({
      name: data.name,
      location: data.location,
      company: data.company,
      blog: data.blog.includes('https') ? data.blog : null
    }))
    .catch((e) => console.warn(e))
}
```

would get compiled into code that looks like this,

```
var getProfile = function getProfile(username) {
  return fetch('https://api.github.com/users/' + username).then(function (response) {
    return response.json();
  }).then(function (_ref) {
    var data = _ref.data;
    return {
      name: data.name,
      location: data.location,
      company: data.company,
      blog: data.blog.includes('https') ? data.blog : null
    };
  }).catch(function (e) {
    return console.warn(e);
  });
};
```

You'll notice that most of the ES6 code like the Arrow Functions and Template Strings have been compiled into regular old ES5 JavaScript. There are, however, two features that didn't get compiled: `fetch` and `includes`. The whole goal of Babel is to take our "next generation" code (as the Babel website says) and make it work in all the browsers we care about. You would think that `includes` and `fetch` would get compiled into something native like `indexOf` and `XMLHttpRequest`, but that's not the case. So now the question becomes, why didn't `fetch` or `includes` get compiled? `fetch` isn't actually part of ES6 so that one at least makes a little bit of sense assuming we're only having Babel compile our ES6 code. `includes`, however, is part of ES6 but still didn't get compiled. What this tells us is that compiling only gets our code part of the way there. There's still another step, that if we're using certain new features, we need to take - polyfilling.

What's the difference between compiling and polyfilling? When Babel compiles your code, what it's doing is taking your syntax and running it through various syntax transforms in order to get browser compatible syntax. What it's not doing is adding any new JavaScript primitives or any properties you may need to the browser's global namespace. **One way you can think about it is that when you compile your code, you're transforming it. When you add a polyfill, you're adding new functionality to the browser.**

If this is still fuzzy, here are a list of new language features. Try to figure out if they are compiled or if they need to be polyfilled.

- Arrow Functions
- Classes
- Promises
- Destructuring
- Fetch
- String.includes

**Arrow functions** : Babel can transform arrow functions into regular functions, so, they can be compiled.

**Classes** : Like Arrow functions, `Class` can be transformed into **functions with prototypes**, so they can be compiled as well.

**Promises** : There's nothing Babel can do to transform promises into native syntax that browsers understand. More important, compiling won't add new properties, like `Promise` , to the global namespace so Promises need to be polyfilled.

**Destructuring** : Babel can transform every destructured object into normal variables using dot notation. So, compiled.

**Fetch** : `fetch` needs to be polyfilled because, by the definition mentioned earlier, when you compile code you're not adding any new global or primitive properties that you may need. `fetch` would be a new property on the global namespace, therefore, it needs to be polyfilled.

**String.includes** : This one is tricky because it doesn't follow our typical routine. One could argue that `includes` should be transformed to use `indexOf` , however, again, compiling doesn't add new properties to any primitives, so it needs to be polyfilled.

Here's a pretty extensive list from the Babel website as to what features are compiled and what features need to be polyfilled.

### Features that need to be compiled

- Arrow functions
- Async functions
- Async generator functions
- Block scoping
- Block scoped functions
- Classes
- Class properties
- Computed property names
- Constants
- Decorators
- Default parameters
- Destructuring
- Do** expressions
- Exponentiation **operator**
- For-of**
- Function** bind
- Generators
- Modules
- Module export** extensions
- New** literals
- Object** rest/spread
- Property method assignment
- Property **name** shorthand
- Rest **parameters**
- Spread
- Sticky regex
- Template** literals
- Trailing **function** commas
- Type** annotations
- Unicode** regex

### Features that need to be polyfilled

- ArrayBuffer
- Array.from
- Array.of
- Array#copyWithin
- Array#fill
- Array#find
- Array#findIndex
- Function#name
- Map
- Math.acosh

Math.hypot  
Math.imul  
Number.isNaN  
Number.isInteger  
Object.assign  
Object.getPrototypeOf  
Object.is  
Object.entries  
Object.values  
Object.setPrototypeOf  
Promise  
Reflect  
RegExp#flags  
Set  
String#codePointAt  
String#endsWith  
String.fromCodePoint  
String#includes  
String.raw  
String#repeat  
String#startsWith  
String#padStart  
String#padEnd  
Symbol  
WeakMap  
WeakSet

Because adding new features to your project isn't done very often, instead of trying to memorize what's compiled and what's polyfilled, I think it's important to understand the general rules behind the two concepts, then if you need to know if you should include a polyfill for a specific feature, check [Babel's website](#) .