Variable Declaration

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 we 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 newvariabledeclaration;

console.log(newvariabledeclaration) // 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 newvariabledeclaration

console.log(newvariabledeclaration) // undefined

newvariabledeclaration = '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 function*s.

```
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 getData 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.toDateString().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], 0.5); // [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.

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 we 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.

So anytime you want a variable to be immutable, you can declare it with const. Well, not quite, it's little more tricky in Javascript. 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 signaling 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 needs 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 blocked 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.