Var, let and const.

Introduction.

With ES6, JavaScript came out with a lot of great features. One of those awesome features was how we declare variables and the scoping ability of each of those variable declarations. Previously, many JS developers were used to using var for assigning variables, but now we have let and const.

– Var

When I was learning JavaScript, var was how I learned to declare variables. How var is scoped pertains to where var is accessible in your code. If var is declared within a function, then it is functionally scoped. This means that it can only be accessed within that function.
Example:

```
function exampleOne(){
  var scopingWithVarFunc = "using inside function"
  console.log(scopingWithVarFunc)
}

console.log(scopingWithVarFunc)
 //Uncaught ReferenceError: scopingWithVar is not defined

console.log(exampleOne())
// using inside function
```

As you can see, we cannot access scopingWithVarFunc outside of the function as we tried on line 6. Yet, when we console.log() the function, we are able to access the variable. This is because when the function is invoked, we get access to the variable that was declared inside of it.

Let see what happens when we declare the variable outside of the function?

```
//Global Scoped Variable
var scopingWithVarGlobal = "using outside function"

function exampleOne(){
  var scopingWithVarFunc = "using inside function"
  console.log(scopingWithVarFunc)
  console.log(scopingWithVarGlobal)
}
console.log(scopingWithVarFunc)
//Uncaught ReferenceError: scopingWithVar is not defined

console.log(scopingWithVarGlobal)
// "using outside function"
console.log(exampleOne())
// using inside function
// using outside function
```

With the var declared outside of the function, it becomes known as a global variable. This is an important distinction because when the variable is globally scoped, then everything has access to it including our function exampleOne().

When I console.log(exampleOne()) on line 19, the console will log out both the globally scoped variable and our functionally scoped variable. When we console.log(scopingWithVarGlobal) outside of the function, we are still able to get a logged value.

Var can also be redeclared and updated.

This means that when I write var declaredVar the first time, I can then write another variable declaration with the same name, but with another value, and not receive an error.

Example:

```
var declaredVar = "First time"
var declaredVar = "Second time"
console.log(declaredVar)
// Second Time
```

Also when you write an assignment without using the var keyword, it automatically becomes globally scoped, so be careful if you do so.

```
newDeclaredVar = "This is global"
```

You might be wondering why most developers have switched to let and const? The reasoning is because you have less control, especially when dealing with code that is block scoped.

Example:

```
var greetingUsingVar = "hello"
if(true){
  var greetingUsingVar = 'time'
}
console.log(greetingUsingVar) // time
```

What I mean by this is that if you declare a variable using var that is globally scoped and then declare another variable below it with the same name that is in blocks such as a loop or conditional, you may find that when you console.log that variable, you may get the value you do not want. In the code above, the console logs the one that is inside of the block scope.

What if I did not want that? What if I wanted the value “hello” instead? What if I want to know that when I console.log any variable that its the one that I want.

This is what the benefits of using both let and const are. I will begin by explaining let.

Let

The best thing about let is that it improves upon var by being what is called block-scoped {}. I mentioned earlier how this might be a problem with var. Let me show you an example:

```
let greetingUsingLet = ’first time’
if(true){
      let greetingUsingLet = ’second time’
}
console.log(greetingUsingLet) //first time
```

As you can see in this code example when you console.log (greetingUsingLet), you only have access to the variable that is outside of the block scope. This solves the issue we had earlier when using var.

What if I want to only access the variable inside the blocks? Then you will have to put your console.log inside of the block code and then you will see the value. This is useful because I can now have two variables that are named the same, but know exactly what variable I want to access because letfixes the scoping issue that var has.

You can update the value with let, but unlike var, you cannot redeclare them in the same scope. I am going to modify the code example above just a bit.

As you can see I have tried to redeclare greetingUsingLet underneath the first one which is within the same scope and I get a syntax error, “Identifier greetingUsingLet has already been declared”.

Const

Just like in its name, const variables are constant. In other words, they cannot be redeclared or updated and if you try to do so, you will get an error.

When using var, the way one would signify a constant variable was to write the variable name in all caps.

```
var GREETINGUSINGVAR = "greeting using var"
GREETINGUSINGVAR = "new greeting"

console.log(GREETINGUSINGVAR)
//new greetingvar
GREETINGUSINGVAR = "greeting using var"
GREETINGUSINGVAR = "new greeting"

console.log(GREETINGUSINGVAR)
//new greeting
```

The only issue with this is that you could still update the value and not get an error. This is why using const to save you from having your variable be updated.
Const is also block–scoped and I showed you what this means when showing how let works.

Hoisting
One thing I haven’t spoken about is hoisting with regard to var, let, and const. I will briefly talk about hoisting. To start, hoisting is when variable and function declarations are moved to the top of their scope before the code is executed. With var, variables are hoisted to the top, but variables written with let and const, are not.
Var
Since variable declarations are hoisted, we get back undefined instead of not defined. In JavaScript, it knows that the variable has been declared, but it returns undefined because it does not yet know the value. When we console.log the greeting variable, the code has not gotten to the initialization.
Let & **Const**
Unlike var, let and const declarations are not hoisted. This is why you will receive an error as opposed to undefined.
You will receive a reference error that you cannot access the variable. This is actually quite helpful in that you make sure to declare your variables at the top or at least before you decide to do any actions with the variable you are declaring.
With const, you must always declare and initialized the variable before using it. If you don’t, you will get an error telling you that there is a missing initializer.