Lesson Weekend

Introduction to Programming (/introduction-to-programming) / JavaScript and Web Browsers (/introduction-to-programming/javascript-and-web-browsers) / JavaScript Variables

Text Cheat sheet

Variables are one of the major features of computer programming and we will use them all the time. A variable is just a container that stores some information such as a number, an object, or a string.

A variable can be declared with one of the following JavaScript reserved words:

- `var` is short for variable. This is the traditional way of declaring a variable but it has some longstanding issues so we won't use this option.
- `let` is a more modern way to declare a variable whose value will change.
- `const` is a more modern way to declare a variable whose value will never change.

These reserved words are called **statements**. There are different types of statements in JavaScript that all serve a specific purpose. For `var`, `let`, and `const`, these statements are declaration statements that create variables.

We will learn more about the advantages of `let` and `const` in this lesson. In a future lesson, we will learn about the major disadvantage of `var`.

var

Up until ES6, which was released in 2015, all variables in JavaScript were declared using `var`. Remember, ES is short for ECMAScript, the scripting-language specification for JavaScript. Here's how we can declare a variable using `var`.

```
> var myNumber;
```

In the example above, we declare a variable called `myNumber`. It doesn't have a value yet. In fact, if we check its value in the console, it will be `undefined`. Also, note the semicolon at the end of the line. For now, we will always be including semicolons at the end of each line. We will let you know when that changes.

Note that the first word `my` is lowercased while the second word `Number` is capitalized. This naming convention is called **lower camel case** because the lower and upper case letters look a little like the humps on a camel. (There are other camel case naming conventions such as upper camel case, too, which will learn about when they are needed.)

With lower camel case, we always lowercase the first letter of the first word but then capitalize the first letter of any other word in the variable.

Here's another example of lower camel case:

```
> var thisIsALongNumber;
```

In general, we want to keep variable names short and concise, which the example above certainly doesn't do. However, it does illustrate how lower camel case looks.

JavaScript doesn't care whether you capitalize the letters or not but other developers do. Folks will get upset if you don't follow proper naming conventions! **You'll be expected to always use lower camel case and be consistent when declaring JavaScript variables.**

Using the Assignment Operator `=`

In the example above, we declared a variable without assigning it a value. We could've declared the variable *and* assigned it a value at the same time like this:

```
> var myNumber = 45;
undefined
```

In the example above, `myNumber` is a number, but as we mentioned before, a variable can hold many different kinds of data types. Of course we'd want to name it differently if it's something other than a number — otherwise our variable names would confuse other developers.

JavaScript Statements

In the example above, take note that JavaScript returns `undefined` because `var myNumber = 45;` is a **statement**. A statement doesn't return a value, which means its return value is just `undefined`.

That doesn't mean `myNumber` is `undefined`, though. Type the variable name in the console:

```
> myNumber;
45
```

The console returns the value of `myNumber`, which is `45`.

Reassigning a Variable's Value

We can modify `myNumber` if we want:

```
> myNumber = myNumber + 5;
50
```

Note that we don't use `var` again when we modify `myNumber`. We only use `var` when we are **declaring** a variable — that is, the first time it shows up in our code.

Think of a variable as being like an actor that comes on stage and introduces their character. Over the course of the play, their character may change but the actor doesn't need to introduce themselves again — like a variable, they've already been declared. We'd be pretty annoyed if they introduced themselves again every time they came out.

Let's take another look at the example above:

```
> myNumber = myNumber + 5;
50
```

If we look at this from the perspective of using variables in math class in school, this would be mathematically impossible. But things are a little different in programming and this is one of those things

that takes a little getting used to. When we modify a variable, we usually want to take the *old* value of the variable, make a change to it, and then have the variable be equal to the *new* value.

This is called **reassigning a variable**. It just means we are changing the value of the variable to something else. In the example above, we are doing the following:

```
New Value = Original Value + Change We Want to Make to Orig
inal Value
```

Note, the variable that is holding the value is always on the left side of the assignment operator `=`. In our example, the variable holding the value is `myNumber`. The expression that's going to be evaluated is always on the right side of the assignment operator. In our example, the expression is `myNumber + 5`, which is accessing the original value of the `myNumber` variable and adding 5 to it.

We will get plenty of practice with reassigning variables — though many developers don't think reassigning variables is a great idea. We will cover this more when we get to declaring variables with `const`.

Here's a little shortcut we can do that's exactly the same thing as `myNumber = myNumber + 5`:

```
> myNumber += 5;
50
```

You'll see (and use) this shorthand a lot in JavaScript.

Before we move on, what do you think happens to the value of `myNumber` if we do this?

```
> myNumber + 5;
50
```

The console will correctly return the new value. However, if we check the value of `myNumber`, we will see that it hasn't changed. That's because we didn't actually *reassign* the value of `myNumber`. To actually change the value, you can probably guess by this point what we need to do:

```
> myNumber = myNumber + 5;
50
```

Alternatively, we can use the shorthand `myNumber +=5`.

Variables Can Be Treated as the Data Types They Represent

Let's look at another thing we can do with variables. We can combine several together to make something new. Here's an example:

```
> var num1 = 1;
> var num2 = 2;
> var num3 = num1 + num2;
3
```

In the example above, we assign values to two different variables. Finally, we add those two variables together and assign that value to a new variable. Because `num1` and `num2` represent numbers, we can treat them as numbers and do math with them.

In short, we can do anything with variables that we'd do with the things contained inside those variables. If the variables contain numbers, we can do anything that we can do with numbers. If they contain strings, we can do anything we'd do with strings. And so on and on.

Naming Variables

You can give variables any name. Using our last example, we could rewrite it to look like this:

```
> var pig = 1;
> var horse = 2;
> var cow = pig + horse;
3
```

And we will still get the number `3` returned.

Using the same example, we could also name our variables like this:

```
> var a = 1;
> var b = 2;
> var ab = a + b;
3
```

And just as before, we will still get the number `3` returned.

So what's best practice? Generally it's best to give your variables a name that is descriptive of what it represents. Using our example, `num1` and `num2` are the most descriptive, because we're using "num", short for "number" to indicate its data type, and we're distinguishing each variable by numbering them with "1" or "2".

Variable naming can be a subjective process. When deciding what to name your variables, think about someone else reading your code and trying to understand what the variable represents and does. This means naming your variable based off of the answers to these questions:

- What is the variable's data type?
- How will the variable be used in the code?

Using `Var` Is Outmoded

It's important to know about `var` because you will see it frequently in code examples and legacy code. It was the only option for declaring a variable up until 2015, and even after that, it's taken additional time for some developers to change their practices and update old code.

However, we won't be using `var` at Epicodus — as we mentioned before, it has some problems that `let` and `const` address. Modern JavaScript developers don't use `var`, and you shouldn't, either!

`let`

`let` is an ES6 (ECMAscript 6) feature that came out in 2015. Like `const`, it has a couple of major advantages over `var`. First, it's more descriptive. We will discuss that further in this lesson. Second, it doesn't have the problems with scope that `var` has. We will cover these specific scoping issues later in this section when we get to branching.

In most ways, both `let` and `const` work exactly the same as `var`. Let's look at the same example we used with `var`, but this time we'll use `let` instead. (You'll need to refresh the browser to do this in the console or you'll get an error. We'll discuss this error towards the end of this lesson.)

```
> let myNumber = 45;
undefined
> myNumber = myNumber + 5;
50
```

As we mentioned at the beginning of this lesson, we can use `let` to signify a variable that will change over time. As you can see, `let` already has a huge advantage over `var`. Any developer can tell just by looking at it that the value of the variable will change at some point.

A big part of being a good developer is good communication. Part of good communication is code that clearly shows what it's meant to do. If another developer sees `let` in front of a variable, they've already learned something important about that variable that they wouldn't have known with just `var`. It's going to change over time. Now they might want to see exactly where it's going to change and how — especially if they are hunting down a bug or refactoring code.

`const`

Finally, we have `const`, which is short for constant. As the name implies, a `const` is a variable that doesn't change over time — it stays constant. For this reason, there are a few ways in which `const` works differently than `let` and `var`.

Let's take a look at `const` in action (refresh your browser first):

```
> const myNumber = 45;
> myNumber = myNumber + 5;
```

If we try to do this, we get an error:

```
Uncaught TypeError: Assignment to constant variable.
```

We can't reassign `myNumber` because it is a constant. This is a very useful advantage that a `const` provides that we can't get with `var`. It will "freeze" the value of our variable, which is very nice if we want to make sure it doesn't change.

As you might expect, you can't just declare a constant without assigning a value to it. Try this out in the console:

```
const thisIsAConstant;
```

We'll get the following error:

```
Uncaught SyntaxError: Missing initializer in const declarat
ion
```

This just means that when we create a constant, we have to assign a value to it. That's because we aren't ever supposed to reassign new values to constants. Even if a variable is declared without a value, reassigning it to have a value is still changing it.

Note that JavaScript actually isn't always very good about freezing constants. JavaScript will make sure that primitives like numbers can't be reassigned. However, we can change more complex objects even if they are constants and JavaScript won't complain. We should never do this — and we will cover this problem further once we start working with those more complex types of objects. It's not something you need to worry about right now, but it's important to know that constants in JavaScripts have their issues, too.

Now let's get back to our example above. What if we want to compute a new value based on `myNumber`? We can do this:

```
> const myNumber = 45;
> const myNewNumber = myNumber + 5;
> myNewNumber;
50
```

We just need to create a *new* variable to hold our new value. Meanwhile, `myNumber` doesn't change — it is still `45`. This is because we didn't reassign `myNumber`. Instead, we declared a new `const` called `myNewNumber` and assigned the value to our new variable.

Many developers believe that `const` is the only way to go and that we shouldn't use `let` or `var` at all. This is because when we reassign variables, we might introduce a problem that's hard to track down. With `let` and `var`, we can never be sure of the value of a variable unless we check it, since it's always possible it will change. With `const`, we assign a value to a variable and then we know what it is. We don't have to worry about it changing!

In the example above, we don't have to worry whether `myNumber` is 45, 50, or something else. `myNumber` will always be 45, `myNewNumber` will always be 50, and we can always make good choices about our variable naming to ensure that other developers understand how the values are changing. It's another opportunity to communicate clearly, which will make everyone happy.

We recommend using `const` wherever possible, but you may use both `let` and `const` to declare variables at this point. For now, there is one hard and fast rule — don't use `let` if the variable's value won't change in your application. Even though things will work fine, that's not clear communication. That's like saying, "yeah, this is gonna change... no, wait, never mind."

One More Advantage of `let` and `const`

Before we move on, let's take a look at another advantage of `let` and `const` over `var`. As we discussed earlier in this lesson, we should only declare a variable once.

Try doing the following in the console (one at a time, pressing `Enter` after each entry):

```
> var myNumber = 1;
> var myNumber = 2;
> let myNumber = 3;
> const myNumber = 4;
```

When we declare `myNumber` a second time with `var`, JavaScript doesn't complain even though it would be a very bad practice to declare two variables with the exact same name in an application. Why is it a bad practice? Well, since we are just beginning to learn about variables, it's hard to show an example just yet — however, it can make our code much buggier and more unreliable because it indicates that one or more developers are trying to do different things with two different variables — but when our code executes, our machine will think they are the same variable because they have the same name, leading to problems.

Here's another way of putting it. If two Janes work at your company, there's a good chance you'll cause confusion if you always refer to both of them as Jane without any clarification about which Jane you're referring to. Did you mean Jane the senior developer or Jane the data analyst? You might need to refer to them as Jane M. and Jane S. instead.

Returning to our example above, when we use `let` or `const` to try to declare a variable that's already been declared, we get the following error:

```
Uncaught SyntaxError: Identifier 'myNumber' has already bee
n declared
```

This is a good error! `let` and `const` won't let us make this mistake.

Going back to our example about two Janes, this would be like someone stopping us and asking, "Wait, we are still talking about Jane S., right?" And then, to clarify, we'd say, "No, actually, I'm talking about Jane M. now. She's the one that will be working on this new project."

This is a major difference between the JavaScript of the past and newer specifications that are being added to JavaScript. The JavaScript of the past is very flexible — in fact, too much so. The JavaScript of the past doesn't care which Jane we are referring to, even if it means assigning a new project to the wrong Jane.

There'd be nary a complaint from the JavaScript interpreter when we did something wrong — the code would just execute until something silently broke. But that's not helpful for developers. We want things to break loudly and clearly so we can see exactly where things went wrong. Otherwise, tenacious bugs will be hard to find and fix — and even worse, they could be so silent at first that they sneak into production code, making the users of our applications angry. Not good if you are working on an application for a bank. When a user's trust goes away, they probably won't use your application anymore. Likewise, when the wrong Jane goes to a meeting with the wrong client, neither Jane nor the client are going to be very happy. For that reason, we need to combine good communication in our code with modern JavaScript specifications.

Declaring a variable without `var`, `let` or `const`

JavaScript is so flexible that it won't complain if we declare a variable like this:

```
> myNumber = 10;
```

In the example above, we don't use `var`, `let`, or `const`.

However, this is very bad for two reasons.

Reason #1 has to do with a slightly more advanced concept called scope that we will learn about later in this section. We won't cover that in depth now, but it's enough to say that if we declare a variable in this way, it won't just be where you think it is, it'll be everywhere in your code, potentially causing trouble. Going back to our coworker Jane, it would be a little like the team lead accidentally assigning her to *all* of the projects at our company instead of just the one she's focusing on.

Reason #2 goes back to clear communication. `var`, `let` and `const` communicate very clearly to other developers that a variable is being declared. Writing `const myNumber` in our code is like saying "Hey everyone, I'm introducing a new variable that should never change called myNumber."

However, if we declared `myNumber` on its own, another developer might think it's been declared elsewhere. We won't even be communicating that we declared a new variable. That is poor communication and it will make other developers upset.

So going forward, make sure to use `let` or `const` (and preferably `const`). JavaScript won't care but other developers will thank you.

Previous (/introduction-to-programming/javascript-and-web-browsers/practice-arithmetic)

Next (/introduction-to-programming/javascript-and-web-browsers/strings)

Lesson 9 of 75
Last updated March 24, 2023

disable dark mode

(http://www.epicodus.com)