Lesson Tuesday

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

Text

In this lesson, we'll discuss **scope**, one of the most important concepts in computer programming. The scope of our code determines where in our computer program code is available, and whether we can access or modify it. All of our variables and functions have scope in our code. As we'll soon learn, scope is divided into two main categories: **global** and **local**. There are other types of scope that we'll learn about in future sections.

Let's start by learning about local scope.

Local scope

Variables declared inside of functions have **local scope**. This means they are only locally available during the execution of the function. When we look at a variable that is defined within a function, we find that its scope ends when the function is finished processing. In fact, the variable is created and destroyed each time the function runs.

Let's look at an example.

```
function sampleFunction() {
  let localString = "This is a local variable";
  window.alert(localString);
  localString = "This is a local variable update!!";
  window.alert(localString);
}

sampleFunction();
window.alert(localString);
```

If we run this code in the DevTools console (you can copy and paste the above code), the first two alerts inside the function will run as expected when the function is called.

However, the third alert, which is outside of the function, will not run. Instead, we'll get the following error:

```
Uncaught ReferenceError: localString is not defined
```

This is because the `localString` variable is defined in the `sampleFunction()` function, and is locally scoped to that function. If we wanted to access the value of `localString` outside of the function, we'd need to add a `return` statement:

```
function sampleFunction() {
  let localString = "This is a local variable";
  window.alert(localString);
  localString = "This is a local variable update!!";
  window.alert(localString);
  return localString; // new line of code!
}

const globalString = sampleFunction();  // updated code!
window.alert("The value of 'localString' at the global scop
e: " + globalString); // updated code!
```

Notice that we've updated the name of the `localString` variable to `globalString` when we save the value returned from the `sampleFunction()` call. This is because we're at the global scope: all variables declared at the top level of a `.js` file, outside of functions, have **global scope**. Let's discuss global scope now.

Global scope

As noted above, variables declared outside of functions have **global scope**. These variables are at the "top level" of a `.js` file, which means that all code can access and modify them.

Let's look at an example:

```
let globalString = "This is a global variable";

function sampleFunction() {
  window.alert(globalString);
  globalString = "This is a global variable update!!";
  window.alert(globalString);
}

window.alert(globalString);
sampleFunction();
window.alert(globalString);
```

In the code snippet, we're defining the `globalString` variable at the global scope. We then alert the value of the `globalString` variable outside of the `sampleFunction()` function and inside of that function. When we run this code in the DevTools console, we'll get "This is a global variable" twice followed by "This is a global variable update!!" twice. Notably, we can modify the value of the `globalString` variable from inside of the function, because it is initially defined at the global scope, which makes it available everywhere in our code.

Why Scoping Variables is Important

In computer programming, it's always the developer's goal to create code that is organized, easy to test, and bug-free. We haven't got to testing, but we can still understand what buggy or organized code looks like. Along with separating code that handles user interface logic and business logic, scope is another tool that lets us separate code into organized and distinct blocks. For example, when we create a variable that is local to a function, we're saying that we don't want any code to be able to access or modify that variable except for code in that function.

Let's take one more look at our local variable example from earlier in this lesson. We will make one small change:

```
function sampleFunction() {
  localString = "This is a local variable";
  window.alert(localString);
  localString = "This is a local variable update!!";
  window.alert(localString);
}

sampleFunction();
window.alert(localString);
```

In this example, we assign a new variable `localString` inside `sampleFunction()` *without* adding `let`.

What happens when we run the code? Do you think it will work correctly? Try it out in the console.

Everything runs correctly, including the alert outside of the function at the end of the code snippet. However, this is *not* what we want to happen. The variable has been declared inside the function, so shouldn't it be scoped there, too?

This is a bad thing that JavaScript does that's a holdover from the early days of JavaScript. If a variable is declared without `let`, `const`, or `var`, it's automatically global *no matter where it is defined*.

You will run into errors and bad bugs if you let variables run amok like this, so it's important to always declare a variable with `let` or `const`.

This code is very small, so you may not see the problem with using a global variable. But imagine a code base that's thousands or tens of thousands of lines long. For instance, let's say you were working with a huge codebase that had a function like this:

```
function(event) {
  window.alert(whatToSay);
}
```

It might be incredibly difficult to figure out where `whatToSay` was defined. And if `whatToSay` was used and changed in multiple places, it would be next to impossible to figure out where its value was last set.

Let's take a look at an analogy to solidify the concept. Let's say we have a fun but overly feisty golden retriever named Max. Max isn't allowed into most rooms of the house because he will chew everything up and make a mess. However, it's okay if he hangs out in the den because the den has been Max-proofed. If we wanted to ensure this in our code, we need to scope Max to just the den like this:

```
function den() {
  let dog = "Max";
}

function livingRoom() {

}

function kitchen() {

}
```

In the code above, Max is locally scoped to the den. He can't be called or redefined in the `livingRoom()` or `kitchen()` function.

However, he will certainly run amok if we do this:

```
let dog = "Max";

function den() {

}

function livingRoom() {

}

function kitchen() {

}
```

Now he is global in scope and he has access to *all* the rooms. He could chew up a couch in the `livingRoom()` function and eat all the food lying out in the `kitchen()` function. As you can imagine, this is not at all good! Even if you don't yet see how global variables can cause harm to our code (because our code samples are small), just trust the analogy above. Global variables can truly wreak havoc.

For that reason, **avoid using global variables, including in your independent projects**. There will be rare occasions when we will use global variables in lessons (or when you might see a reason to use them in your own code). Global variables exist for a reason — and they can be needed sometimes. However, they are too often used as a crutch for people new to coding. In the coming weeks, **if the lessons don't specify using a global variable, you almost certainly don't need one.** So stick with using locally scoped variables!

Test Your Understanding

Below are two code snippets that each have two alerts that display the variable `bunnyName` , one when the function runs and one at the end of the code. What value do you expect for each alert? Use the

DevTools console to test your theories.

Code Snippet #1

```
let bunnyName = "Flopsy";

function hippityHoppity() {
  window.alert(bunnyName);
  bunnyName = "Cottontail";
}

hippityHoppity();
window.alert(bunnyName);
```

Code Snippet #2

What values do you expect for these alerts?

```
let bunnyName = "Flopsy";

function hippityHoppity() {
  let bunnyName = "Mopsy";
  window.alert(bunnyName);
  bunnyName = "Cottontail";
}

hippityHoppity();
window.alert(bunnyName);
```

Previous (/introduction-to-programming/javascript-and-web-browsers/homework-identifying-and-preventing-microaggressions)
Next (/introduction-to-programming/javascript-and-web-browsers/practice-more-function-writing)

(http://www.epicodus.com)