Lesson Thursday

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

Text

When we first introduced variables, we mentioned that there are differences in how `var`, `let`, and `const` are scoped. At the time, we weren't ready to discuss those differences. Now that we are actively branching, though, we can explore this difference in a little more detail. It's important because there is another layer of scope that goes beyond local and global scope: function scope and block scope. As you read along with the lesson, you are welcome to try out the different code snippets in the DevTools console.

Function Scoping

Let's take a look at a function that includes branching and then illustrate the differences between `var` and `let` / `const`.

```
function doYouLikeApples(bool) {
  if (bool) {
    var str = "Apples are delicious!";
  } else {
    str = "Maybe oranges would be better.";
  }
  return str;
}
```

In the function above, the value of `str` is available everywhere in the `doYouLikeApples()` function because it is initialized with `var`. As a result, the code above is poorly written and should be refactored to look like this:

```
function doYouLikeApples(bool) {
  var str;
  if (bool) {
    str = "Apples are delicious!";
  } else {
    str = "Maybe oranges would be better.";
  }
  return str;
}
```

The functionality in both of the above examples is exactly the same. With `var`, all variables are "hoisted" to the outermost level of the function. This is called **function scoping**.

However, we should always try to scope all variables as tightly as possible. What if we have a variable that we only need to use inside one conditional block within the `if...else` statement? Take note, a **conditional block** is one condition of an `if...else` statement, including any condition that needs evaluating and any code that should be run if the condition is true. In the above

`doYouLikeApples()` function, we have two conditional blocks in the `if...else` statement. Here we've separated each conditional block into two code blocks:

```
if (bool) {
  str = "Apples are delicious!";
}
```

```
else {
  str = "Maybe oranges would be better.";
}
```

So, if we have a variable that we only need to use inside one conditional block and we want to scope all variables as tightly as possible, it would be better if we could only scope the variable to the conditional block instead of to the whole function. `var` doesn't let us do that.

This is one of the problems that `let` and `const` fix.

Block Scoping

What happens if we rewrite our first example above to use `let` or `const` instead of `var`? Well, let's take a look. Be warned — we are going to see some tricky behavior.

```
function doYouLikeApples(bool) {
  if (bool) {
    let str = "Apples are delicious!";
  } else {
    str = "Maybe oranges would be better.";
  }
  return str;
}
```

All we are doing is changing `var` to `let` from the first example in this lesson.

So what happens if we call `doYouLikeApples(true)`?

We'll get the following error:

```
Uncaught ReferenceError: str is not defined
```

This is because `let` and `const` use **block scoping**.

When we scope to a block, the scope remains inside the curly braces of each conditional statement within the `if...else` statement.

In the example above, that means the `str` variable is scoped inside the curly braces:

```
if (bool) {
  let str = "Apples are delicious!";
}
```

When we try to `return str`, we are at a higher level of scope than the block where `str` was defined. For that reason, the variable has fallen out of scope and the function doesn't have access to it.

Now let's do something tricky. What do you think happens when we try calling `doYouLikeApples(false)`?

It will return `"Maybe oranges would be better."`. So what happened?

Well, with `str = "Maybe oranges would be better.";` in the second conditional block:

```
else {
  str = "Maybe oranges would be better.";
}
```

We've created a global variable! Because `let` and `const` can be scoped to blocks, when we use `str` a second time, it's not referring to the variable we declared in the first conditional. Even though it has the same name, it's not in the same scope. Instead, a *new* variable is created. Because we don't use `let`, `const`, or `var` to declare it, it defaults to the global scope. We can confirm this in the console by checking the value of `str` after the function has been called. We'll see that `str` retains its value even though the variables inside the function should no longer be in scope.

We can still use `let` and `const` to have local scope at the outermost level of a function. We just need to do something like this:

```
function doYouLikeApples(bool) {
  let str;
  if (bool) {
    str = "Apples are delicious!";
  } else {
    str = "Maybe oranges would be better.";
  }
  return str;
}
```

Because `str` is being declared at the uppermost level of the function, it can be accessed anywhere in the function.

So as we can see, block scoping gives us more granular control over scope, which is a good thing. Since we want to always scope variables as tightly as possible, we should try to scope to the level of the block. Old school JavaScript with `var` doesn't allow that. `let` and `const` do.

Here's one other little behavior that's different between `var` and `let` / `const` . It's a little thing, but it's one more way `let` and `const` make JavaScript more consistent and developer-friendly.

```
function doYouLikeApples(bool) {
  if (bool) {
    str = "Apples are delicious!";
  } else {
    str = "Maybe oranges would be better.";
  }
  let str;
  return str;
}
```

If we call this function, we'll get the following error:

```
Uncaught ReferenceError: Cannot access 'str' before initial
ization
```

This makes sense. We aren't defining `str` until *after* we use it. Even if that worked, it would be sloppy code and hard to read.

However, replace the `let` with `var` in the function above and everything works just as if `str` were defined at the beginning of the function. This is because `var` automatically scopes all variables to the level of the function no matter where they are declared. It's not really a convenience at all — in fact, it's JavaScript being a bit too loosey-goosey. The problem with loosey-goosey code is that it results in annoying bugs.

Summary

In this lesson, we covered the difference between block and function scope and a few more reasons why `let` and `const` are better than `var`. Block scoping doesn't just apply to conditionals (branching with `if` statements) — it also applies to switch statements (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/switch) (another form of branching that we'll cover later in the program) and loops. We won't cover looping until the next section, so don't worry about that yet.

When you write functions that include blocks, always consider whether any variables you declare can be scoped more tightly. There is no need to scope a variable to the top level of the function if it's only needed in a block. Paying close attention to this granularity of scope is a key step you can take towards becoming a better programmer.

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

Next (/introduction-to-programming/javascript-and-web-browsers/debugging-in-javascript-pausing-on-exceptions)

Lesson 71 of 75

Last updated more than 3 months ago.

disable dark mode

(http://www.epicodus.com)