Lesson Tuesday

Introduction to Programming (/introduction-to-programming) / JavaScript and Web Browsers (/introduction-to-programming/javascript-and-web-browsers) / JavaScript's Global Object

Text Cheat sheet

In this lesson, we're going to revisit the topic of scope and learn something new about global scope in JavaScript: it is represented by a global object. For JavaScript that's run in the browser, its global object is set to the `window` object. We'll start with a review of global and local scope, and then move onto new information.

Review: Global and Local Scope

In the "Variable Scope" lesson, we learned that where a variable is declared in our code determines the access we have to it. This is called **scope**. If a variable is declared inside of a function, that variable is **locally scoped** to the function and we only have access to it within the function. Because of this, if we want to access any variable declared in a function, *outside* of that function, we need to use a `return` statement.

Check out the following code snippet for an example of local scope. If we run this in our DevTools console, our second alert will cause this error: `Uncaught ReferenceError: result is not defined`. This is because the `result` variable was declared within the `add()` function and not returned from the function. In other words, `result` is locally scoped within the `add()` function and we can't access it outside of it.

```
function add() {          // function declaration
  const result = 1 + 3;  // locally scoped variable declara
tion (result)
  window.alert(result);  // the value of the 'result' varia
ble is successfully alerted as 4
}

add();                    // function call
window.alert(result);     // alerting the value of the 'resu
lt' variable causes an error
```

Just as noted above, because the `result` variable is not returned from the `add()` function, we can't access it at the global scope.

On the other hand, a **globally scoped** variable is one that is declared at the top level of our script file or our DevTools console. When we give a variable a global scope, it becomes available everywhere in our code. Check out the following code snippet that demonstrates this. In this example, the `result` variable is declared outside of the `add()` function and at the top level of our scripts at the global scope.

```
const result = 1 + 3;    // globally scoped variable declar
ation (result)

function add() {         // function declaration
  window.alert(result);  // the value of the 'result' varia
ble is successfully alerted as 4
}

add();                   // function call
window.alert(result);    // the value of the 'result' varia
ble is successfully alerted as 4
```

As we can see in the above example, because the `result` variable is declared at the global scope, we can access it anywhere in the global scope, or in the local scope of functions declarations.

Scope can be a tricky concept to wrap your head around as a newcomer to computer programming. It's important to understand that scope gives developers control over which code our variables and functions can access or "talk to". Just like with the example of the feisty golden retriever named Max, we only wanted Max to be able to access the den that we've "Max-proofed" and not the rest of the house. Scope allows us to do exactly that in our code — control the access our code has to other code in our program.

Ultimately, scope helps our code be less buggy and more organized. Global and local scope are two foundational categories within scope, and over the course of the program we'll learn more about scope. Next up, we'll learn that global scope is represented by a global object in JavaScript.

JavaScript Global Scope Is Represented by a Global Object

In JavaScript, the global scope is by definition always available. We know this because when we declare a variable or function in the global scope, we can access it from anywhere in our code. However, "scope" so far has been more of a concept. How is global scope represented in actual code? Let's first look at local scope.

With local scope, we can see that it's created every time we declare a function. In other words, local scope is inside of any function we declare:

```
// global scope

function add() {
  // local scope
}

// global scope

function subtract() {
  // local scope that is completely separate
  // from the add() function's local scope
}

// global scope
```

With global scope, it's a bit trickier to see represented in code. So far, we know that declaring a variable or function at the top level of our script file or DevTools console makes that variable or function globally scoped. Well, in JavaScript the global scope is represented in code by a global object. The **global object** is just an object that is always available and accessible in our script file or DevTools console. This means that when we create a function or variable at the global scope, it gets added to the global object as a property! Let's see exactly how this works.

The Global Object for JavaScript Changes Based on Its Execution Context

For JavaScript code that's run in a web browser, the global object (the global scope) is set to the `window` object. Yes — the very same `window` object that we've been working with to run our alerts and prompts. Let's work through an example. When you input this code in your DevTools console:

```
> function add() {
  return 1 + 3;
}
```

We can call it like this:

```
> add();
4
```

Or, we can call it like this:

```
> window.add();
4
```

`window.add()` is the exact same function as `add()`. This demonstrates how the functions we declare at the global scope in our code get added to the JavaScript's global object.

Keep in mind that it's only when we run JavaScript in our web browsers that JavaScript's global object is set to the browser's `window` object. If we run JavaScript on a server, the global object is set to an object called `global`, but we won't spend time learning about this object. The technical way to describe this behavior is that JavaScript's global object depends on its execution context. An

execution context is the location where our code is being run. For example, are we running our JavaScript in the browser? If so, then the browser is our execution context, and as we learned recently, JavaScript's global object is set to the `window` object. Or, are we running JavaScript in a server, and not in a browser? We won't worry about this second case now. The main take-away is this:

- JavaScript's global scope is represented by a global object that changes based on its execution context.
- A global object is just an object that is always available and accessible.
- The execution context of our JS code corresponds to where we're running the code (in browser or in a server).

You may be asking yourself why is JavaScript's global scope represented by an object? Well, just like browsers create the DOM to represent our HTML, and the `window` object to represent our current browsing context, JavaScript needs a way to represent global scope in code so that we can write code to interact with it.

In fact, all of JavaScript's global built-in functions are properties of JavaScript's global object. Let's look at an example. Do you remember JavaScript's global built-in functions for parsing numbers? Theses ones:

```
> const myNumber = parseInt("3");
> myNumber;
3
> const myPi = parseFloat("3.14");
> myPi;
3.14
```

When we're running our JS in the browser, these global built-in JavaScript functions can be re-written and called as `window` object methods, and the functionality is the exact same:

```
> const myNumber = window.parseInt("3");
> myNumber;
3
> const myPi = window.parseFloat("3.14");
> myPi;
3.14
```

Note — there's no reason to use `window.parseInt()` instead of `parseInt()` in your projects. The point of these examples is to show you how JavaScript's global built-in functions (like `parseInt()`) and the functions we create at the global scope (like `add()`) are added to JavaScript's global object, which in the case of JavaScript run in the browser is set to the `window` object.

At this point, it should be relatively clear that JavaScript has a global object that represents the global scope of our script files or the DevTools JavaScript console. Next, we'll cover more details about the global object and we'll explain a change in the way we'll write certain code:

- Only variables declared at the global scope with `var` are added to JavaScript's global object.
- It's possible to accidentally (or purposefully) override `window` properties.
- We'll use a shortcut for JavaScript run in the browser, which is to NOT call on the `window` object to access its properties.

You Don't Need to Include `window` to Access Its Properties

Let's look at another example. Try putting this into your DevTools console:

```
> window.alert("I'm using the window object's alert metho
d!");
```

And then input this:

```
> alert("I'm STILL using the window object's alert metho
d!");
```

Writing `window.alert()` is the same as writing `alert()`! This is because the `window` object is the global object — it is the execution context of our code in the browser, and by default it's always available and accessible. To use an analogy, the global object for JavaScript is like an aquarium fish tank. The JavaScript functions we create are the aquarium fish, rocks, plants, and sand, and the fish tank is the default context that supports the existence of the fish, plants, etc. — it's always available and accessible.

Using another analogy, the global object in JavaScript is pretty similar to an omniscient narrator that knows everything about the characters, their motives, and the events in a book. As the global object, the `window` object knows about all of the code — the HTML, the JS, the CSS, and other browser Web APIs. Because the global object is the default context, we don't have to directly call on the `window` object to access it's properties.

If this is confusing to you, that's to be expected. A good starting point is to just accept that this is how JavaScript and web browsers are structured and it helps both technologies interact with each other.

The big takeaway here is that we don't ever have to explicitly access the `window` object to first access its properties — we can just call directly on the property names. This means that all of these `window` properties that we've learned about:

```
window.innerHeight;
window.innerWidth;
window.open();
window.location.reload();
window.location.host;
window.location.href;
window.alert();
window.prompt();
window.confirm();
window.document;
window.document.body;
window.document.head;
window.document.getElementById();
window.document.querySelectory();
```

Can all be written without first accessing the `window` object, like this:

```
innerHeight;
innerWidth;
open();
location.reload();
location.host;
location.href;
alert();
prompt();
confirm();
document;
document.body;
document.head;
document.getElementById();
document.querySelectory();
```

In fact, **it's convention to not reference the `window` object explicitly**. Developers just don't want to type out the extra `window.` when they don't have to. However, when you are just learning to code, this can cause a lot of confusion — you may think that

`alert()` is a function when in fact it is a `window` object method. Now that we know the `window` object is the global object for JavaScript that's run in the browser, we'll be able to see `alert()` and know that it's a `window` object method.

Going forward in LHTP lessons, we won't always explicitly include `window` when we want to access `window` properties, however, anytime it's important to know how a new object (like `document` or a global built-in JavaScript function) is connected to the `window` object we'll make that clear. Shifting to this new convention will take a bit of time to get used to, but we'll have lots of practice and you'll be given reminders.

Also, when we see code like this:

```
> alert("I'm STILL using the window object's alert metho
d!");
```

It's completely fine to say "this is the `alert()` function". Many places online do this. Why? Well, it's because in the code `alert()` is being called as a function.

If you prefer to type out `window.alert()` instead of `alert()` to call on this method, that is completely acceptable. The same goes for every other example we've covered in this lesson — do what's most comfortable for you and best for your learning. **In any independent project, it is your choice whether or not to omit `window` when you are accessing a property of it.**

Only Variables Declared with `var` Are Added to the Global Object

Any variable declared with `let` or `const` at the global scope of your scripts (or DevTools console) are not added to the global object. They still have global scope in your scripts/console and can be

accessed anywhere, even within the local scope of a function. However, only variables declared with `var` are actually added to the global object.

It's Possible to Override `window` Properties

If we write a function with the same name as a `window` method, it will overwrite the `window` method. Check out this example, and try it out in your DevTools console if you like.

```
> window.alert
ƒ alert() { [native code] }
> function alert() {
  return "Howdy! I've overwritten window.alert().";
}
undefined
> window.alert
ƒ alert() {
  return "Howdy! I've overwritten window.alert().";
}
```

We've entered three commands sequentially in the above code snippet:

- First, we accessed the value of the `window.alert` property. We receive `ƒ alert() { [native code] }` as the return value of the `window.alert` property. Here we can see that alert is a built-in (`[native code]`) function (`ƒ`). Since it belongs to the `window` object, we know this function is actually a method.
    - If you are wondering why we didn't enter in `window.alert()` with parens, this is because we did not want to call the method to execute its code. Instead, we want to know the definition of the method, so instead we input `window.alert` to access the value of the window's `alert` property.
- Second, we declared a new function called `alert()`. In this function, we simply return the string `"Howdy! I've overwritten`

`window.alert()."`.

- Third, we access the value of the `window.alert` property again. Here, we're confirming that the definition of the window's alert method has actually changed, and indeed it has! We receive the code of our new `alert()` function instead of the original definition.

The same goes for other `window` properties. We can easily overwrite those if we are declaring variables of the same name at the global scope with `var`. This issue can easily be avoided if we stick to using `let` and `const`, which is expected in all of your projects.

The takeaway here is just to be aware that you can accidentally overwrite `window` properties.

Summary

In this lesson, we learned more about JavaScript's global scope:

- JavaScript's global scope is represented by a global object that changes based on its execution context.
- A global object is just an object that is always available and accessible.
- The execution context of our JS code is where we're running the code (for example, in a web browser or in a server).
- For JavaScript run in the browser, the global object is set to the `window` object.
- All globally scoped built-in or custom JS functions are added to the global object. In our projects, that means these functions are added to the `window` object.
    - However, only globally scoped variables declared with `var` are added to the global object.
    - We also need to be careful not to accidentally override `window` properties.
- Because the `window` object is the global object for our JS, we don't have to explicitly reference the `window` object to access its properties. For example, `window.document` can be written

just as `document`, and this is the conventional way of doing things.

Previous (/introduction-to-programming/javascript-and-web-browsers/practice-accessing-the-dom)
Next (/introduction-to-programming/javascript-and-web-browsers/accessing-html-element-attributes-and-properties-in-the-dom)

Lesson 46 of 75
Last updated more than 3 months ago.

disable dark mode

(http://www.epicodus.com)