Lexical scoping

(https://github.com/learn-co-curriculum/phase-1-lexical-scoping-readme) (https://github.com/learn-co-curriculum/phase-1-lexical-scoping-readme/issues/new)

Learning Goals

- Explain the concept of lexical scoping.
- Describe how lexical scoping informs the scope chain of a function.

Introduction

In this lesson, we'll learn how JavaScript decides which outer scope to place into the scope chain for a new function.

First, a brief note. Some of the material introduced in this section might feel a bit confusing or esoteric, but, fear not, you're going to get more and more comfortable with these concepts throughout this course. If you're finding it difficult to wrap your brain around some of these more abstract concepts at this point, **don't freak out**. This is really difficult material that even seasoned programmers occasionally struggle with. After you've read the material and given it a college try, feel free to move on to the next lesson — with a mental note to return once you have more JavaScript experience under your belt. Okay, onwards and upwards!

Take a look at the following code snippet:

```
const myVar = "Foo";

function first() {
  console.log("Inside first()");

  console.log("myVar is currently equal to:", myVar);
}

function second() {
```

```
  const myVar = "Bar";

  first();
}
```

Think about what we've learned in previous lessons about how JavaScript looks up the scope chain to perform identifier resolution. Given that information, what do you think will get logged out to the console when we invoke `second()`? Let's try it out:

```
second();
// LOG: Inside first()
// LOG: myVar is currently equal to: Foo
// => undefined
```

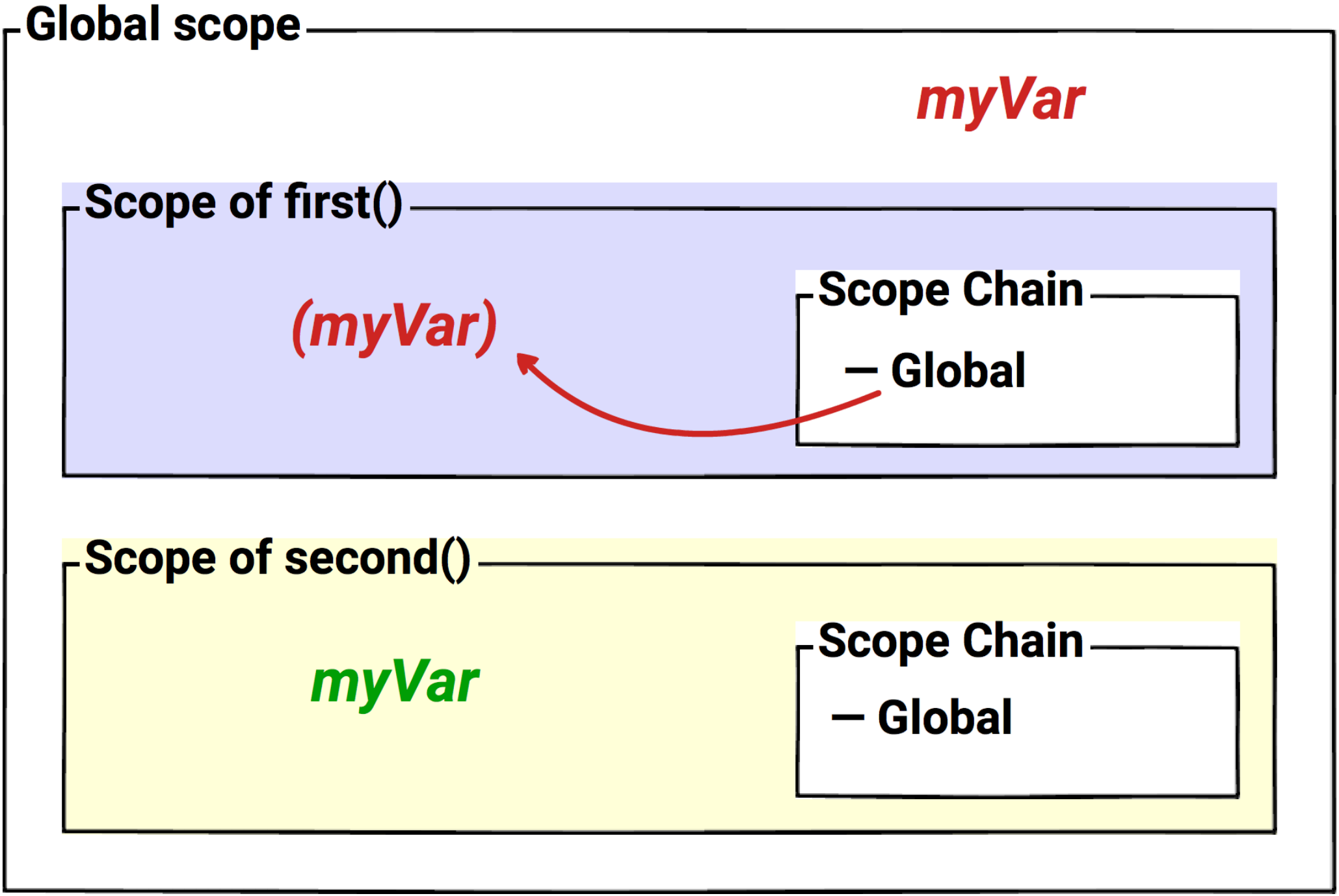
Did that catch you by surprise? At first glance, it might seem like `Bar` should get printed out. Inside `second()`, that string is assigned to the `myVar` variable right before `first()` is invoked:

```
function second() {
  const myVar = "Bar";

  first();
}
```

However, the assignment of `myVar` as `'Bar'` is **not visible to** `first()`. This is because `second()` is **not** the parent scope of `first()`.

In the following diagram, the red `myVar` is declared in the global scope, and the green `myVar` is declared inside `second()`:


Global scope
myVar
Scope of first()
(myVar)
Scope Chain
— Global
Scope of second()
myVar
Scope Chain
— Global

No variable named `myVar` exists inside `first()`. When the JavaScript engine reaches the second line of code inside the function, it has to consult the scope chain to figure out what the heck this `myVar` thing is:

```
console.log("myVar is currently equal to:", myVar);
```

The engine's first (and only) stop in the scope chain is the global scope, where it finds a variable named `myVar`. The reference to `myVar` inside `first()` is pointed at that external variable, so `console.log()` prints out `myVar is currently equal to: Foo`.

`first()` is declared in the global scope, and, when it comes to the scope chain, JavaScript functions don't care where they are invoked. **The only thing that matters is where they are declared**. When we declare a new function, the function asks, "Where was I created?" The answer to that question is the outer environment (the outer scope) that gets stored in the new function's scope chain.

This is called *lexical scoping*, and *lexical environment* is a synonym for *scope* that you might encounter in advanced JavaScript materials. *Lexical* means "having to do with words," and for lexical scoping what counts is where we, the programmer, typed out the function declaration within our code.

In the example above, we typed out our declaration for `first()` in the global scope. If we instead declare `first()` inside `second()`, then `first()`'s reference to its outer scope points at `second()` instead of at the global scope:

```
const myVar = "Foo";

function second() {
  function first() {
    console.log("Inside first()");

    console.log("myVar is currently equal to:", myVar);
  }

  const myVar = "Bar";

  first();
}
```

When we invoke `second()` this time, it creates a local `myVar` variable set to `'Bar'`. Then, it invokes `first()`:

```
second();
// LOG: Inside first()
// LOG: myVar is currently equal to: Bar
// => undefined
```

While `first()` is executing, it again encounters the reference to `myVar` and realizes it doesn't have a local variable or function with that name. `first()` looks up the scope chain again, but this time `first()`'s outer scope isn't the global scope. It's the scope of `second()` **because `first()` was declared inside `second()`**. So `first()` uses the version of the `myVar` variable from the `second()` scope, which contains the string `'Bar'`.

Wrapping up

If this isn't making a ton of sense, don't sweat it too much! We're spending time on things like the *scope chain* and the *lexical environment* now because they're fundamental to the language, but they are not easy concepts to grasp! Keep these concepts in mind as you move through the rest of the course. As you write more and more JavaScript code, you'll notice some of the language's eccentricities cropping up. But then you'll remember things like lexical scoping and the scope chain, and you'll be in a much better position to explain what's going on — **why** your code is being interpreted a certain way.

Investing the time and effort now will pay huge dividends throughout your JavaScript programming career. Knowing how to declare and invoke a function is great and necessary, but knowing what's actually going on under the hood during the declaration and invocation is exponentially more powerful.

When a variable contains an unexpected value, understanding the scope chain will save you countless hours of painful debugging. When you're wondering where to declare a function so that it can access the proper variables, your familiarity with JavaScript's lexical scoping will save the day. When you want to impress some new friends at a party, hit 'em with a quick lesson on how running JavaScript code consists of distinct compilation and execution phases.

Resources

-

JavaScript: Understanding the Weird Parts - The First 3.5 Hours (https://www.youtube.com/watch?v=Bv_5Zv5c-Ts)

(https://www.youtube.com/watch?v=Bv_5Zv5c-Ts)

(Video)