<https://medium.com/dailyjs/javascript-basics-the-execution-context-and-the-lexical-environment-3505d4fe1be2>

**JavaScript basics: The Execution Context and the Lexical Environment**

What is actually happening when code is executed/functions are invoked?

https://cdn-images-1.medium.com/max/640/1*4877k4Hq9dPdtmvg9hnGFA.jpeg

A lot of the magic of code happens behind-the-scenes when the code is compiled and interpreted.

JavaScript is described as an interpreted language rather than a compiled language like C or Java. *The distinction between an interpreted language and a compiled language might be a future blog post*. However, much of the discussion for new JavaScript developers on this subject tend to be confused over the *compilation* step.

Semantically, you have two ways of understanding compilation:

1. Compilers take the **entire** program code and “compiles” it into machine code *before* the code is executed.
2. Compilation can simply mean making program code digestible for the machine to run it.

The first definition is the traditional definition of what being a compiled language means. After all, if you think about it, if compilation simply means taking program code and making it machine-readable, all languages would be “compiled”. It is under the traditional definition, JavaScript is understood as an interpreted language, where Google’s V8 engine compiles the program code line-by-line just as it is being run — called the Just-In-Time (JIT) Compiler. *But more on this in a later post*.

https://cdn-images-1.medium.com/max/640/1*ZrJKJqBsksWd-8uKM9OvgA.png

**Execution Context**

The **Execution Context** is the *abstract concept* of the environment in which the current code is being evaluated in.

In the ES5 documentation, the execution context *“contains whatever state is necessary to track the execution progress of its associated code”.*

Typically, this “environment” is either: (1) the default **global** environment or (2) the **function** environment, code inside a function. *Note: eval code is a distinct environment; however, for the scope of this post, only the global and function environment will be discussed below.*

When the JavaScript interpreter “compiles” the code line-by-line before it is executed, it creates a **call stack**. Because JavaScript is a single threaded language, only one thing can happen in the browser at a time, the rest are in this call stack waiting to be run.

*Imagine a typical stack with the****default global execution context****at the bottom and any number of****function execution contexts****stacked on top. The same is true is for function calls inside another function. Simply put, function-level execution contexts can be stacked on top one after another until the code is run through.****Think: box on top/within a box.***

Going back to the line-by-line compilation of JavaScript code *before* the code is executed, there is actually something happening behind-the-scenes!

This is called the **creation phase**.

**Phase 1: The Creation Phase**

During the microseconds before JavaScript code is ever executed, three things happen during the creation phase:

1. The **ThisBinding** State Component is determined.
2. The **LexicalEnvironment** State Component is defined.
3. The **VariableEnvironment** State Component is defined.

https://cdn-images-1.medium.com/max/640/1*4877k4Hq9dPdtmvg9hnGFA.jpeg

First, the **ThisBinding** State Component…

This is simply the value of the this keyword in the current execution context.

In the *global execution context*, the this keyword is bound to the Global Object (in the browser, this is the Global Window Object).

In *function code*, however, the this keyword depends on how the function is called.

For example:

let foo = {  
 bar: function () {  
 console.log(this);  
 }  
};

foo.bar(); // 'this' is bound to the 'foo' object because 'bar' was   
 // called with reference to 'foo'

let baz = foo.bar;

baz(); // 'this' is bound to the Global Window Object because no   
 // reference was given when called

In summary, if an object reference is given as the function is called, the this value will point to that object. However, if no object is given (or that reference is null or undefined), the this value will point to the Global Window Object.

https://cdn-images-1.medium.com/max/640/1*4877k4Hq9dPdtmvg9hnGFA.jpeg

Second, the **LexicalEnvironment** State Component…

I think it is helpful to think about why this is important. In short, one has to wonder, how does the machine know where to look for all the variables declared in the code? The LexicalEnvironment is where this "identifier resolution" happens!

The official ES5 docs defines Lexical Environment (an abstract concept really) as the place where “the association of Identifiers to specific variables and functions based upon the lexical nesting structure of ECMAScript code” is stored. **Jargon aside**, there are two main takeaways from this definition:

1. **Identifier associations** are simply the *binding* between variable and function declarations with their values. (e.g., let x = 10, 'x' is bound to '10').
2. **Lexical structure** is just describing the actual location where the code was written. *See below*.

let foo = function (a) {   
 let b = 10; // 'b' and 'bar' is in the lexical structure of 'foo'   
 let bar = function(a) {  
 let c = 20; // however, 'c' would only be in the lexical   
 return a + b + c; // structure of 'bar' because 'c' is written   
 }; // inside the 'bar' function.  
};

This about *where* the code is written and that’s how you know which variable and function declarations go into which LexicalEnvironment.

Now, *within* this Lexical Environment are two components: (1) the **environment record** and (2) a **reference to the outer environment**.

1. The *environment record* is the place where the variable and function declarations are stored.
2. The *reference to the outer environment* is the way in which the machine conducts identifier resolution (scope).

* In the *global code environment*, you can expect the built-in Object/Array/etc. prototype functions inside this environment record as well as any user-defined global variables. And the reference to the outer environment would be set to null (as it is the outermost environment).
* In *function code*, the user-defined variables inside the function AND the lexical structure (see above!) are stored in the *environment record*. And the reference to the outer environment can be the global environment, or whatever outer function that wraps around the inner function.

To make matters even more confusing, there are two **types** of *environment records*…

1. **Declarative environment record** handles variables, functions, and parameters. (similar to the ES3 ‘activation object’ concept)
2. **Object environment record** describes the way in which identifier associations are defined in the *global context* (and with statements).

In short,

* In *global code*, the environment record is the *object environment record*.
* For *function code*, it is the *declarative environment record*.

**A quick, but important note:** for function code, the *declarative environment record* also contains an arguments object that has indexes mapped to arguments as key:value pairings and the *length* of the arguments passed into the function.

* As implied above, for *global code*, the LexicalEnvironment is set to the *Global Environment* (with the *object environment record* and the reference to the outer environment set to null).
* For *function code*, the LexicalEnvironment will contain all the variable and function declarations. This is actually where we see the "hoisting" phenomenon -- where declarations are brought to the top of its lexical scope prior to any assignment to a value.

https://cdn-images-1.medium.com/max/640/1*4877k4Hq9dPdtmvg9hnGFA.jpeg

And finally, the **VariableEnvironment** State Component…

This is actually a copy of the **LexicalEnvironment**. There is actually no difference between the two EXCEPT when working with with statements. with statements actually modify the LexicalEnvironment and not the VariableEnvironment. Using with statements are outside the scope (no pun intended) of this discussion; in fact, some developers argue that the with statement should be avoided altogether for performance reasons.

As such, assume the two are the same thing in most cases.

**Phase 2: The Execution Phase**

This is the most straightforward section of this entire post. After the V8 Engine has gone through the code, bound the this keyword and defined the LexicalEnvironment and the VariableEnvironment, finally the code is executed!

Here, assignments to those variable/function declarations are finally done (i.e., those “hoisted” variables are now properly defined!), code is run and we see the end result of our code!

https://cdn-images-1.medium.com/max/640/1*ZrJKJqBsksWd-8uKM9OvgA.png

**Why does any of this stuff matter?**

Chances are, you don’t need to know any of this to demonstrate understanding of concepts like “hoisting” and scope/identifier resolution for most developers. However, if you are anything like me, and desire to know exactly what is happening at a deeper level (but still more abstract than machine code), then now you can explain exactly what is happening according to the ECMAScript specifications.

For personal learning, understanding where identifier resolution actually happens (LexicalEnvironment) and why "hoisting" occurs (Creation Phase vs. Execution Phase) was illuminating. A lot of articles won't dig this deep because at a certain point, you might as well read the actual documentation. But here, I tried to digest that in a way so I can understand it. And I hope it is helpful to any readers out there, who like me, loves digging deep.

*Read more:*

* [Dmitry Soshnikov’s post on the Theory of Lexical Environments](http://dmitrysoshnikov.com/ecmascript/es5-chapter-3-1-lexical-environments-common-theory/)
* [Dmitry Soshnikov’s post on ECMAScript Implementation of Lexical Environments](http://dmitrysoshnikov.com/ecmascript/es5-chapter-3-2-lexical-environments-ecmascript-implementation/)
* [ES5 Documentation on Code Evaluation and Execution Context](http://es5.github.io/#x10.4)
* [ES5 Documentation on Lexical Environments](http://es5.github.io/#x10.2)