JavaScript - Understanding the Weird Parts

# Execution Contexts and Lexical Environments

## Conceptual Aside: Syntax Parsers, Execution Contexts, and Lexical Environments

### Syntax parser

* A program that reads your code and determines what it does and if its grammar or syntax is valid.
* You have your code that you've written and there's a program that is going to convert what you've written into a real set of computer instructions, something the hardware can physically understand.
* There's a compiler or an interpreter between those two things (your code and computer instructions) and part of that is a syntax parser.

### Lexical environment

* Where something sits physically in the code you write.
* The word ‘lexical’ means ‘having to do with words or grammar’.
* Lexical environment exists in programming languages in which ‘where’ you write something is important.

### Execution context

* A wrapper to help manage the code that is running.
* There are lots of lexical environments, areas of the code that you are looking at physically. But which one is currently actually running, is managed via what's called execution contexts.
* And an execution context contains your code, the running code. It's running your code, but it also can contain things beyond what you've written in your code because remember, your code is being translated, being processed by a whole other feature, a whole other set of programs that someone else wrote. And so, it's executing your code and it can do other things as well.

## Conceptual Aside: Name/Value Pairs and Objects

* A **Name/Value** pair is a name which maps to a unique value. And the value can contain other name/value pairs (nested name/value pairs).
* An **object** is a collection of Name/Value pairs. That's the simplest possible definition of an object when you're talking about JavaScript.

## The Global Environment and The Global Object

* Whenever code is run in JavaScript, its run inside an execution context.
* Meaning a wrapper that the JavaScript engine, the program that other people wrote, that's parsing and looking at and verifying and executing your code. That wraps the code that you've written, it wraps the currently executing code in an execution context.
* The base execution context is your **global execution context** and it has a couple of special things that come along for the ride.
* When we say global, we're talking about the thing that's accessible everywhere to everything in you're code, it's global.
* So the global execution context creates two things for you. It creates a **Global Object**. And it creates a special variable called **'this'**.
* The JavaScript engine is creating these two things for you whenever your code is run, because your code is wrapped inside an execution context.
* There is always a **global object** when you're running JavaScript. In the case of browsers, it's the **window** object. Each browser tab has its own global execution context.
* In case of browser the global execution context creates global object called window and ‘this’ variable. And at the global level those two things are equal so window object = this
* **When we say global, in JavaScript that means not inside a function.**
* In JavaScript, when you create variables and functions, and if you're not inside a function, those variables and functions get attached to the global object.
* Refer – <https://github.com/sameerbhilare/Javascript/tree/master/JavaScript%20-%20Understanding%20the%20Weird%20Parts/01-global-environment>

## The Execution Context - Creation and Hoisting

* In JavaScript, the variables and functions are to some degree available even though they're written later in the code.
* The reason JavaScript behaves this way is the execution context is created in two phases.
* The first phase is **creation phase**. In this phase, it creates global execution context (global object and this variable) and also creates and outer environment.
* Also it sets up the memory space for the variables and functions used in your code and this is called as “**Hoisting**”.
* All this means is that before your code begins to be executed line by line, the JavaScript engine has already set aside memory space for the variables that you've created in that entire code that you've built, and all of the functions that you've created as well.
* So those functions and those variables exist in memory. So when the code begins to execute line by line, it can access them.
* So the JavaScript engine when it sets up the memory space for a variable, it doesn't know what its value will ultimately end up being until it starts executing its code. So instead, it puts a placeholder called **undefined**. That placeholder means “oh, I don't know what this value is yet”.
* **All variables in JavaScript are initially set to undefined, and functions are sitting in memory in their entirety.**
* So when the code begins executing, those things are actually already sitting in memory because it looked at your code and already preset things up to be ready for the code to start executing.
* Refer – <https://github.com/sameerbhilare/Javascript/tree/master/JavaScript%20-%20Understanding%20the%20Weird%20Parts/02-executation-context-hoisting>

## Conceptual Aside: JavaScript and 'undefined'

* 'undefined' is actually a special value that JavaScript has within it internally that means that the variable hasn't been set.
* **Never set yourself a variable equal to undefined. It is meant to be set by JavaScript in creation phase.**
* It's better to let ‘undefined’ mean that I, the programmer, never set this value. That will really help you when debugging code.
* The error "Uncaught ReferenceError: b is not defined" simply means the variable is not present in the memory space which was created in the creation phase.
* Refer – <https://github.com/sameerbhilare/Javascript/tree/master/JavaScript%20-%20Understanding%20the%20Weird%20Parts/03-javascript-and-undefined>

## The Execution Context - Code Execution

* And the second phase, is the **execution phase**.
* In the execution phase we already have all those things set up that we had before. And now it runs your code line by line, interpreting it, converting it, compiling it, executing it on the computer into something the computer can understand.
* Refer – <https://github.com/sameerbhilare/Javascript/tree/master/JavaScript%20-%20Understanding%20the%20Weird%20Parts/04-executation-context-execution-phase>

## Conceptual Aside: Single Threaded, Synchronous Execution

* Single Threaded – That means that one command is being executed at a time.
* Synchronous means, in the purposes of programming, one at a time. So one line of code being executed for synchronous execution at a time and for our purposes, in order that it appears.
* **JavaScript is single threaded, synchronous execution in its behavior.**

## Function Invocation and the Execution Stack

* Invocation – That just means running a function or calling a function.
* Anytime you execute or invoke a function in JavaScript, a new execution context is created and put on the execution stack. Once the function execution is one, the execution context is popped off the stack. And whichever one is on top is the one that's currently running.

## Functions, Context, and Variable Environments

* **Variable environment** is where the variables live that you've created and how they relate to each other in memory.
* Every execution context has its own variable environment.
* Refer – <https://github.com/sameerbhilare/Javascript/tree/master/JavaScript%20-%20Understanding%20the%20Weird%20Parts/05-variable-environment>

## The Scope Chain

* **Variable environment** is where the variables live that you've created and how they relate to each other in memory.
* Every execution context has its own variable environment.
* However when we request a variable or when we do something with the variable, JavaScript does more then just look in the variable environment of the currently executing context.
* Every execution context has a reference to its **outer (lexical) environment**.
* JavaScript does something special, it cares about the lexical environment when it comes to the outer (lexical) environment reference that every execution context gets. And when you ask for a variable while running a line of code inside any particular execution context, and if it can't find that variable, it will look at the outer (lexical) environment reference and go look for variables there somewhere down below it in the execution stack (**scope chain**). And that outer environment reference where that points is going to depend on where the function sits **lexically**.
* That whole chain is called the Scope Chain. Scope means, where I can access a variable. And the chain is those links of outer (lexical) environment references.
* Refer – <https://github.com/sameerbhilare/Javascript/tree/master/JavaScript%20-%20Understanding%20the%20Weird%20Parts/06-scope-chain>