Philosophy of learning code:

“Don’t imitate, UNDERSTAND!!!”

SYNTAX PARSERS, EXECUTION CONTEXTS, LEXICAL ENVIRONMENTS

* Syntax Parser:
  + A program that reads your code and determines what it does and if its grammar is valid. Your code isn’t magic. Someone else wrote a program to translate it for the computer.
* Lexical Environment:
  + Where something sits physically in the code you write. “Lexical” means having to do with words of grammar. A lexical environment 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. Which one is currently is managed via execution contexts. It can contain things beyond what you’ve written in your code.

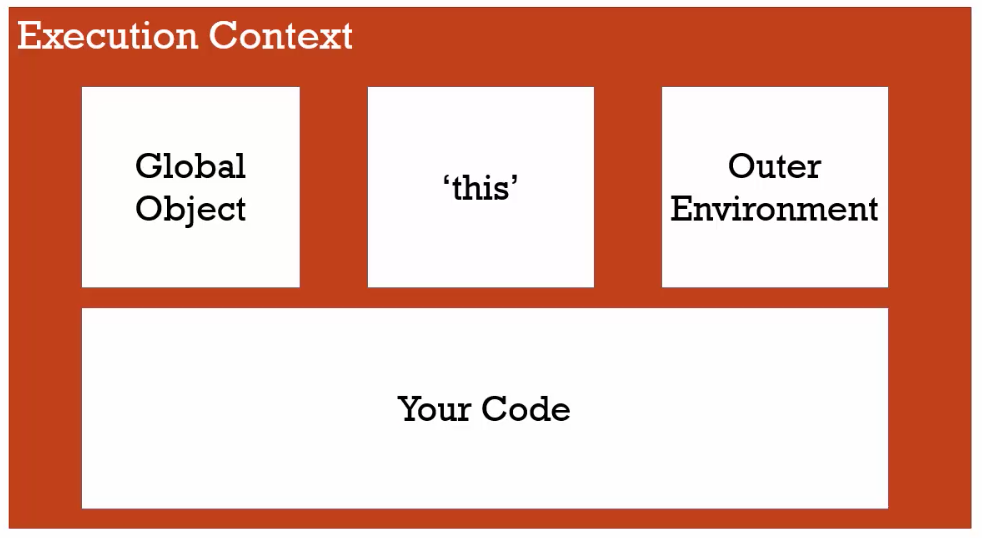
NAME/VALUE PAIRS AND OBJECTS

* Name/ Value Pair:
  + A name which maps to a unique value. The name may be defined more than once, but only can have one value in any given context. That value may be more name/value pairs.
* Object:
  + a collection of name value pairs. The simplest definition when talking about JS.
  + For example, I might have a name and a value for that name is another name with a name/value pairs inside, like a nesting doll.

THE GLOBAL ENVIRONMENT AND THE GLOBAL OBJECT

Whenever code is run its run in execution context (global). Meaning a wrapper that the JS engine, the program that the other people wrote that’s parsing and looking at and verifying and executing your code. That wraps that up, that code that you’ve written, it wraps the currently executing code in an execution context. So when we look at our very first JS program, we’re going to be looking at an execution context being created and run. The base execution context is your global execution context. That global context creates two things, global object and a special variable called “*this*”. This two things are created for you by JS engine whenever you run a JS program.

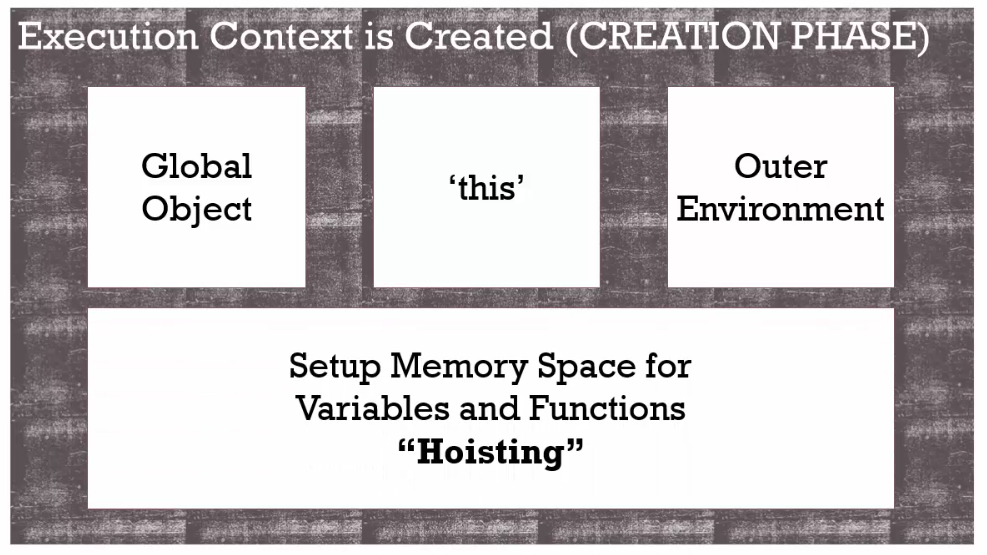
* Global:
  + Means “not inside of a function”.
  + When you create a variable outside of the function, the variable is created inside the global execution context.



THE EXECUTION CONTEXT: CREATION AND ‘HOISTING’

* Hoisting:

The reason that JS behaves the way it does, the variable and functions are to some degree available even though they’re written later in code, is because the execution context is created in two phases. The first phase is called the creation phase. In that phase we know we have the global object is set up within the memory. We have *this* which is set up in the memory. And again the global object only referring to when we’re dealing with the global execution context. We’ll see other execution context later. There’s an outer environment that’s create and in that creation phase, as the parser runs through your code and begins to set up what you’ve written for translation. It recognizes where you’ve created variables and where you’ve created functions. And so it sets up in this creation phase, the memory space for the variables and functions. And it’s that step that is somewhat confusingly called hoisting.



All this means that, before your code begins to be executed line by line, the JS 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.

However, it’s a little different with variables than with functions. The function in its entirety is placed into memory space, meaning that the function, its name and the code inside the function is being executed.

The next phase, the execution phase which actually executes your code line by line, that’s when these kind of assignments are set, where *a* equals to something. So the JS engine when it sets the memory for variable a, it doesn’t know that 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. It’s the same placeholder that we would have, if we never said it at all. All variables in JS are initially set to undefined.

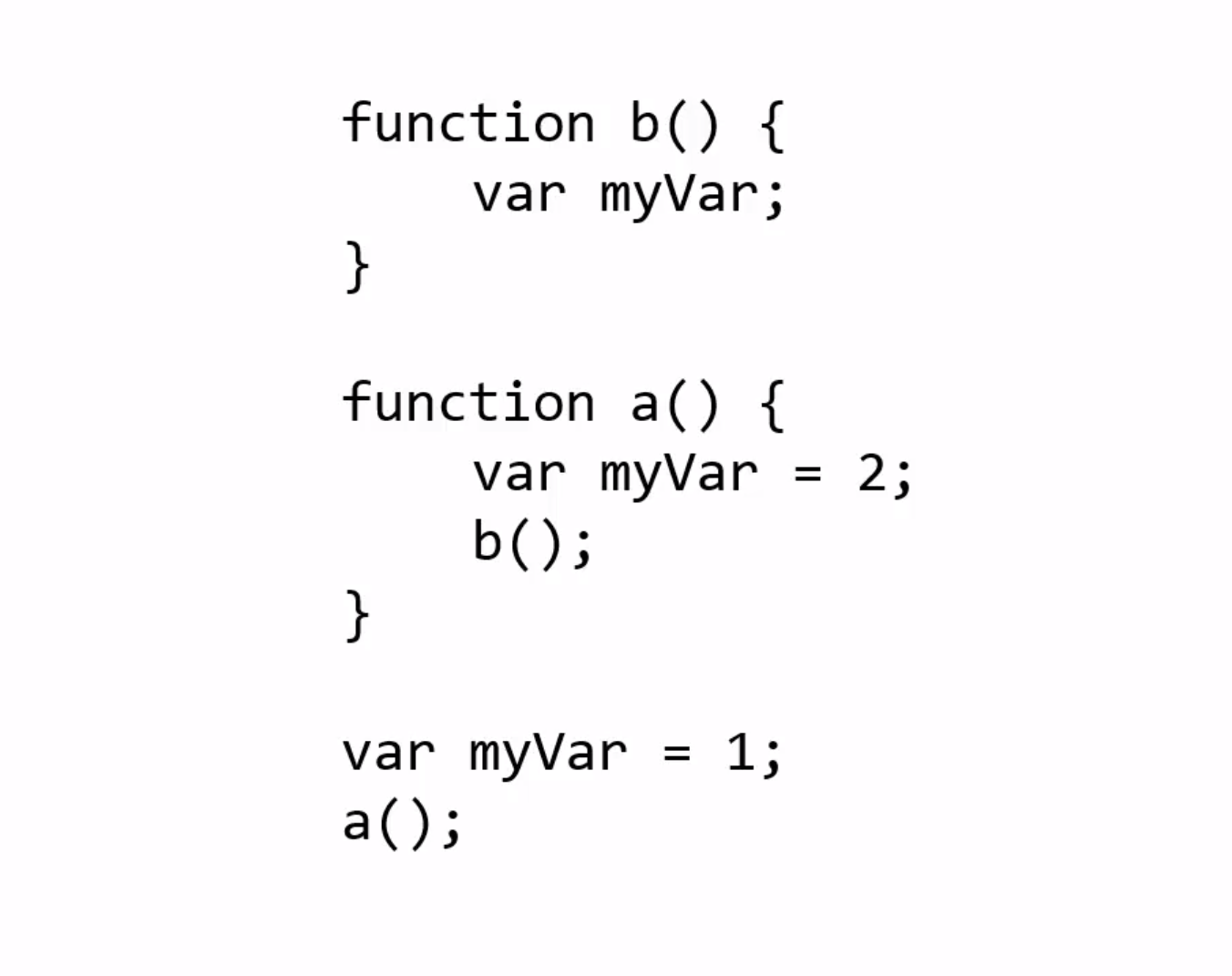
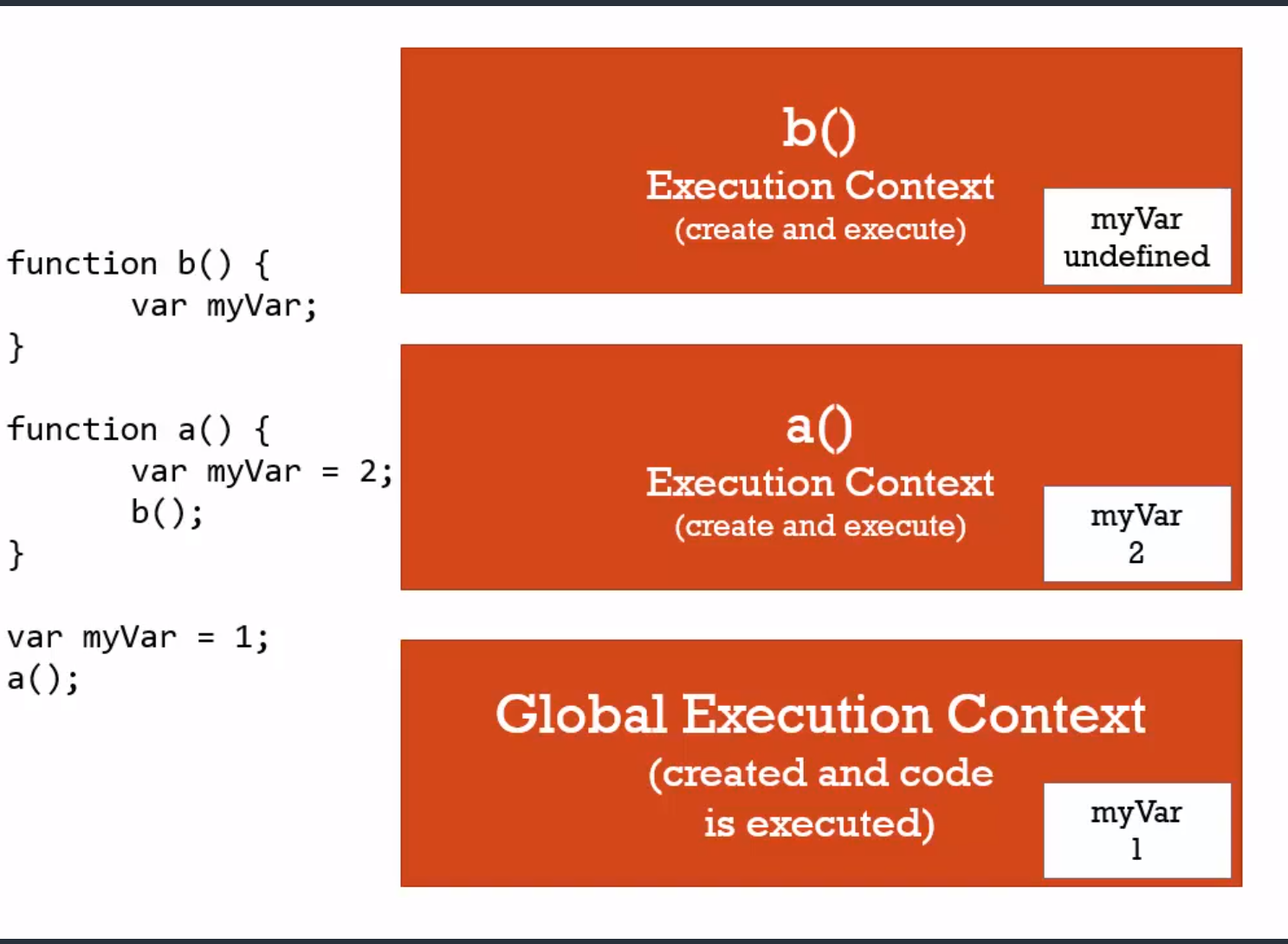
JS AND ‘UNDEFINED’

Never set a variable to “undefined”, because it is dangerous practice. It is hard to debug down the road, because it will not be clear, if the coder set the value to undefined or the value was never set in the first place. It is always better to let “undefined” mean, I have never set this value of a variable.

FUNCTION INVOCATION AND THE EXECUTION STACK

* INVOCATION:
  + 
  + Running a function. In JS, by using parenthesis().
  + After Global Execution Context created and code is executed, a new execution context is created and placed on what’s called the execution stack. And a stack is just what is sounds like, one on top of the other on top of the other.
  + And whichever one is on top is the one that’s currently running.
  + So anytime you execute or invoke in JS, a new execution stack is created and put on the execution stack. However if I have another function invocation, it will stop at that line and create another execution context and run that code.
  + **Every function creates a new execution context which runs through the create phase then executes the code line by line within the function.**

FUNCTIONS, CONTEXT, AND VARIABLE ENVIRONMENTS

* Variable Environments:
  + Where the variables live and how they are related to each other in memory.
  + 
  + At first GEC(Global Execution Context) is created and so myVar put into Global Environment. When code is executed myVar would be 1.
  + Then a() is executed and creates an EC where myVar = 2. Th EC of a() is not the same as GEC, but the one that is put on top of GEC.
  + The nit invokes b() and the new execution context is run. There the myVar is undefined.
  + 
  + Each of the myVar is sitting inside its own execution context.

SCOPE CHAIN

* When we request a variable, or do something with that variable, JS does more than just look in the variable environment of the currently executing context.
  + Remember that each execution context has some special things that get created for you, like the variable *this.*
  + Remember we also mentioned one other thing that we have not used yet, the reference to the outer environment. Every execution context has a reference to its outer environment.
  + Lexical environment: where something is written, physically is important, it determines certain things as far as how the JS engine will decide and interpret where things live, and where things will live in memory, and how they’ll connect to each other.
  + So it is important to understand where functions sit lexically, to know its outer environment.
  + So JS looks at outer environment to look for variables if they are not declared inside the function’s EC.

SCOPE, ES6, AND *LET*

So we’ve talked about execution context, execution environment, variable environment, lexical environment. And all of those things ultimately are defining what is called scope. Especially when we talked about the scope chain that outer reference that any running function has.

* Scope:
  + Where a variable is available in your code. And if ti is truly the same variable or a new copy. And sometimes it has to do with it’s really even tha same variable or a new copy, like if you called the same function twice, it each gets its own execution context and thought it looks like the same variable, it’s actually two different variables in memory.
* Let:
  + So you can declare a variable, just like you would with var and during the execution phase where it’s created, the variable is still placed into memory and set to undefined.
  + However, you are not allowed to use let until the line of code is run during the execution phase that actually declares the variable.
  + So if you try to use a let variable before declaring it, you’d get an error. Now it’s still in memory but the engine just won’t allow it.
  + The other important thing is that it’s declared inside a block. A block’s in general defined by curly braces, so inside if statement or a for loop or something like that. When that variable is declared inside that block, it’s only available inside that block at that period of time for the running code. This is true even for for loops. So if you have a loop and are running the same code over and over but you have a let statement you’ll actually get a different variable in memory each time the loop is running. So this allows for what’s called block scoping as we mentioned, which other who are used to other programing languages may be familiar with.

WHAT ABOUT ASYNCHTONOUS CALLBACKS?

* Asynchronous:
  + More than one at a time. So we amy be dealing with code that’s executing and that starts off some other code to execute, and that may start other executing and all of those pieces of code are actually executing within the engine at the same time. But JS is synchronous.

TYPES AND JAVASCRIPT

* Dynamic typing:
  + You don’t tell the engine what type of data a variable holds, it figures it out while your code is running. Variables can hold different types of values because it’s all figured out during execution.
  + And so a single variable can, at different times during the execution of your code, hold different types of values, because it’s all figured out during execution.

PRIMITIVE TYPES

* Primitive type:
  + a type of data that represents a single value. That is not an object.
  + 1. Undefined:
    - Undefined represents lack of existence(you shouldn’t set a variable to this).
  + 2. Null:
    - Null represents lack of existence(you can set a variable to this).
    - This one is better for you to use when you want to say that something doesn’t exist, that the variable has no value. So leave udefined for the engine , and you can use null, the keyword for your variables you want to set them equal to nothing.
  + 3. Boolean:
    - This is either true or false.
  + 4. Number:
    - In JS, there is only one numeric type, and it’s called number. It’s a floating point number.
  + 5. String:
    - A sequence of characters (both single and double quotes can be used).
  + 6. Symbol:
    - Used in ES6.

COERCION

* Coercion:
  + Converting a value from one type to another. This happens quite often in JS because it’s dynamically typed.

OBJECTS AND OBJECT LITERALS

* Instead of writing *new Object();* , there is a shortcut called object literal where you assign {} to a variable.
* What is happening is that JS engine, when it’s parsing the syntax, and it comes across this curly brace, and it’s not part of something like an if statement or a for loop, or something like that, it assumes that you are creating an object.

FAKING NAMESPACES

* NAMESPACE:
  + A container for variables and functions. Typically to keep variables and functions with the same name separate.

FUNCTIONS ARE OBJECTS

* First Class Functions
  + Everything you can do with other types you can do with functions. Assign them to variables, pass them around, create on the fly.
* So when we say that functions are objects in JS, what does the function object look like?
* Well, just like any objest in JS, it resides in memory. It’s a special type of object, though, because it has all the features of a normal object and has some other special properties.
* In JS, the function object has some hidden special properties. Two important ones:
  + Name
    - Optional, can be anonymous. Now a function in JS doesn’t have to have a name.
  + Code property
    - Where the actual lines of code that you’ve written sit.
* So it isn’t like the code that you write is the function. The function is an object with other properties. And the code the you write is just one of those properties that you’re adding onto it. What’s special about that property is it’s invokable, meaning you can say run this code.
* When the function was created, it was put in memory, in this case, onto the global object, and it had a name. so the name property is greetm because that’s what I named my function. And it has a code property that contains the code that I wrote, the body of my functions. And so if I to call greet using the parentheses, that invokes that code, causes it to run, causes that execution context to be created, etc.

FUNCTIONS STATEMENT VS FUNCTION EXPRESSIONS

* Expression
  + A unit of code that results in a value. It doesn’t have to save to a variable.