**3. Javascript Foundation II**

**2. Execution Context**

**STATEMENT** -- Whenever code is run in JS it is run inside of an execution context.

function() {

this is an execution context

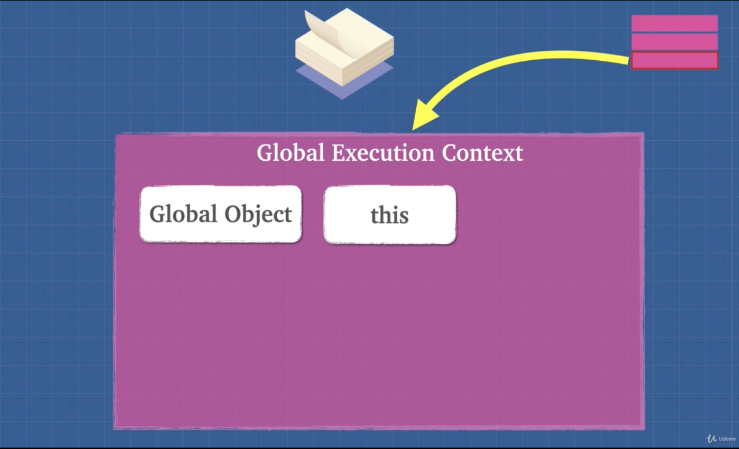
}

* Every function() creates a new execution context and it is pushed on to the stack.

**global() execution context**

* Underneath the hood there is a "global() execution context". After every line is read it gets popped of from the stack.
* global() execution context gives us 2 objects, Global object & this object.





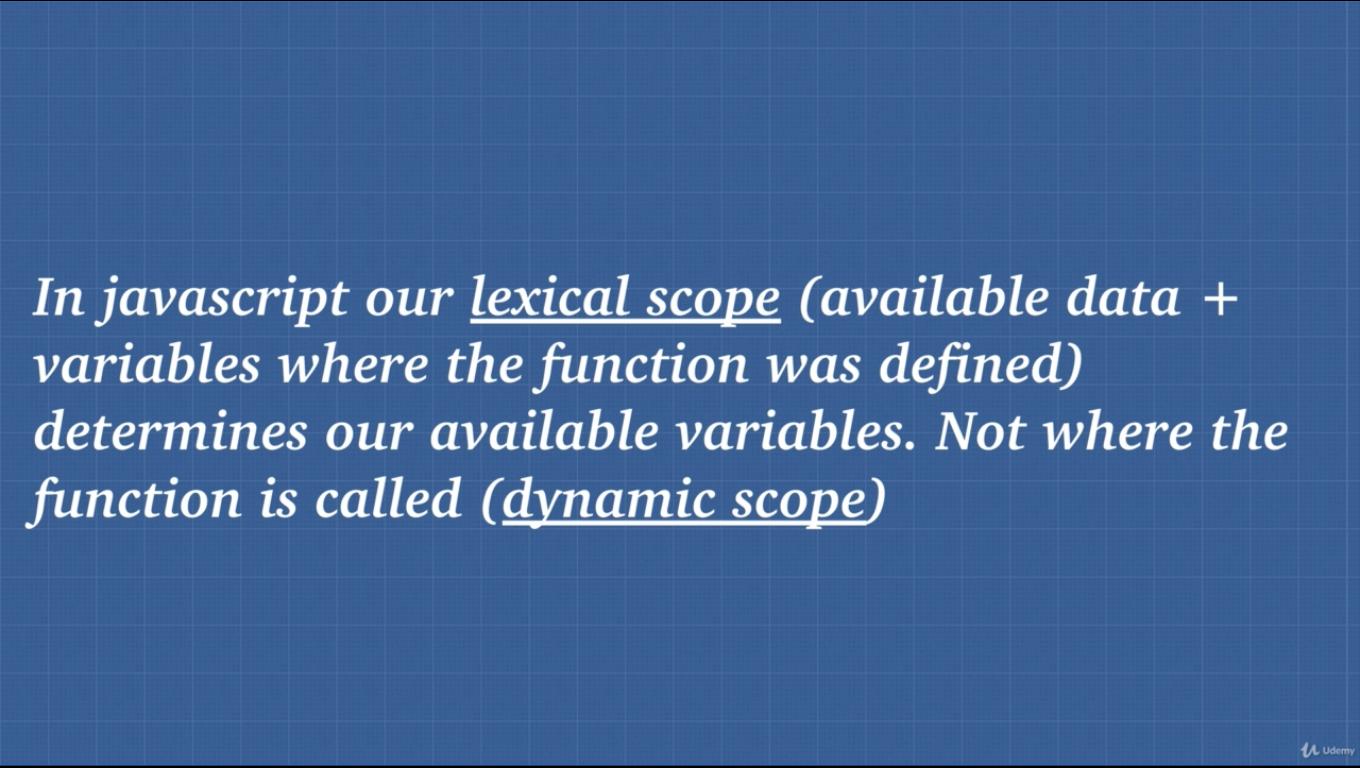
**3. Lexical Environment**

**Lexical Environment** is simply where we write our code.

**STATEMENT** – Execution context tells us which lexical environment is currently running or which planet is currently running.

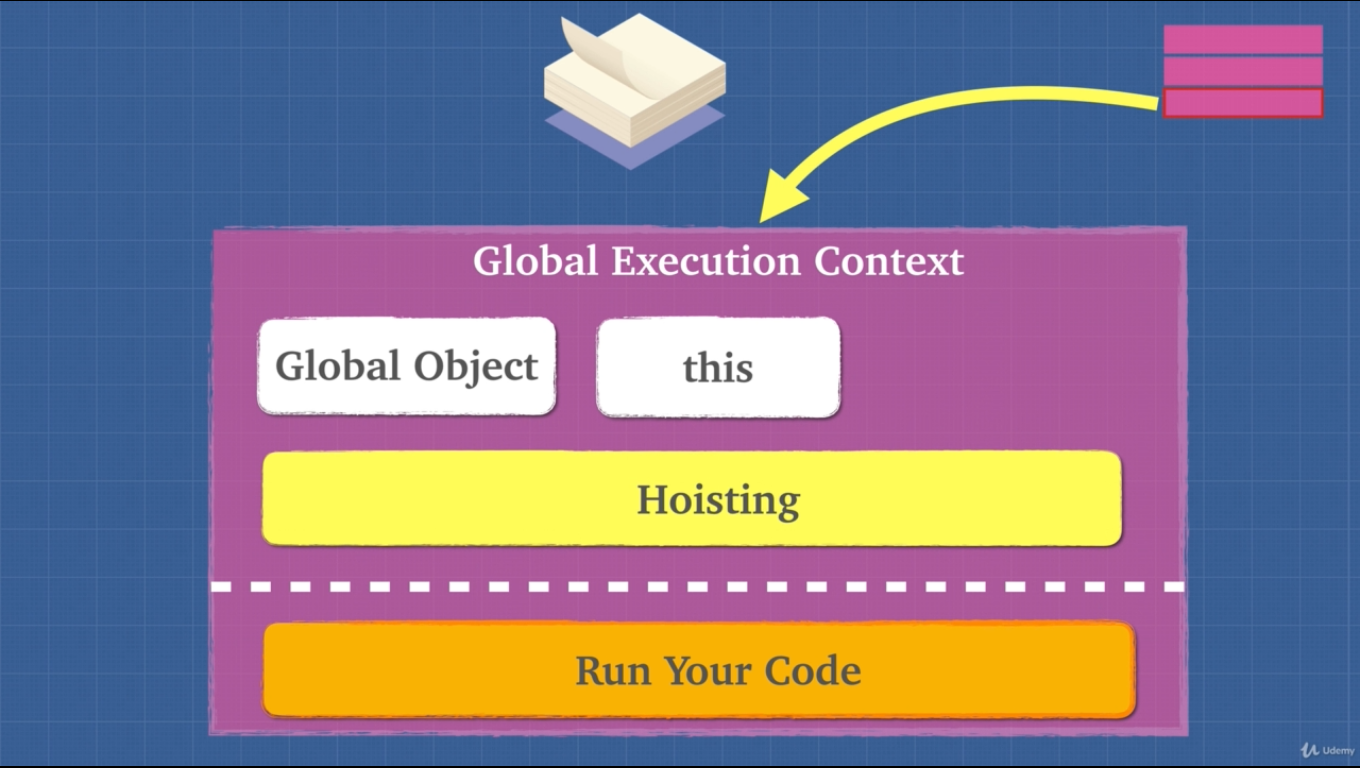


* Every function creates a new lexical environment.
* Lexical scope determines the available data, variables etc. not where the function is called.



**4. Hoisting**

**Hoisting**: Hoisting is moving the var or function declaration to the top of their respective environment during compilation phase.



* Hoisting is unique only to javascript.
* var and function expression is partially hoisted and function declaration is hoisted.

**5. Exercise Hoisting**

* For var hoisting ignores same variable name if found, only takes the first one because it is partially hoisted.
* For function, hoisting happens fully. If same function is found it overrides the previous function in memory.

**6. Exercise Hoisting 2**

* When we run a function a new execution context is created.

function() {

Here is a new execution context.

}

**STATEMENT** – Hoisting is bad practice. We can avoid hoisting using “const” or “let”.

**8. Function Invocation**

Function invocation/Call/Execution are the same thing

function\_name()

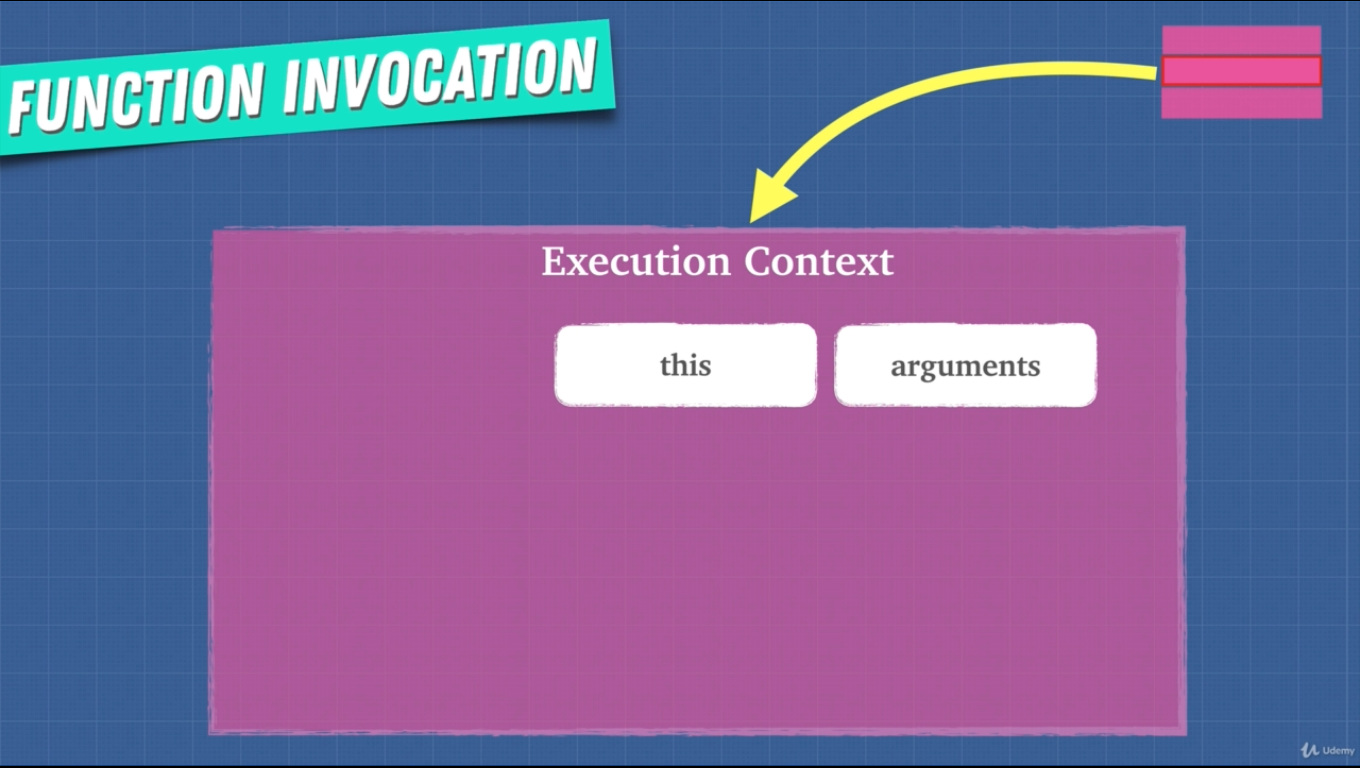
* Function expression is defined at runtime.
* Function declaration is defined at parse time (reading the code and allocating mamory).

function() {

Here is a new execution context.

We get this and arguments objects.

}



**9. arguments Keyword**

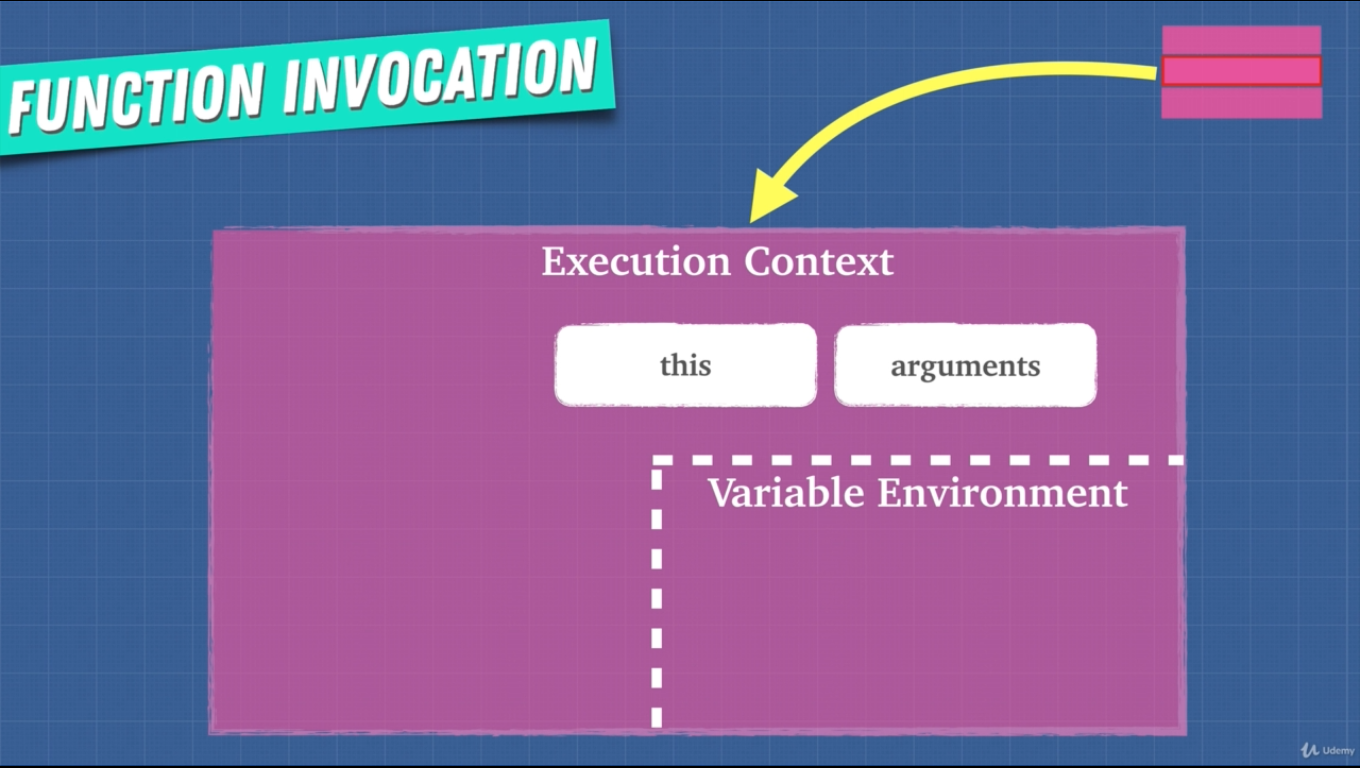
* Using arguments is bad practice. JS engine or compiler may not be able to optimize our code.
* arguments is not really an array, it’s an object.
  + Work around to this is



* + Or using spread operator and args

**10. Variable Environment**

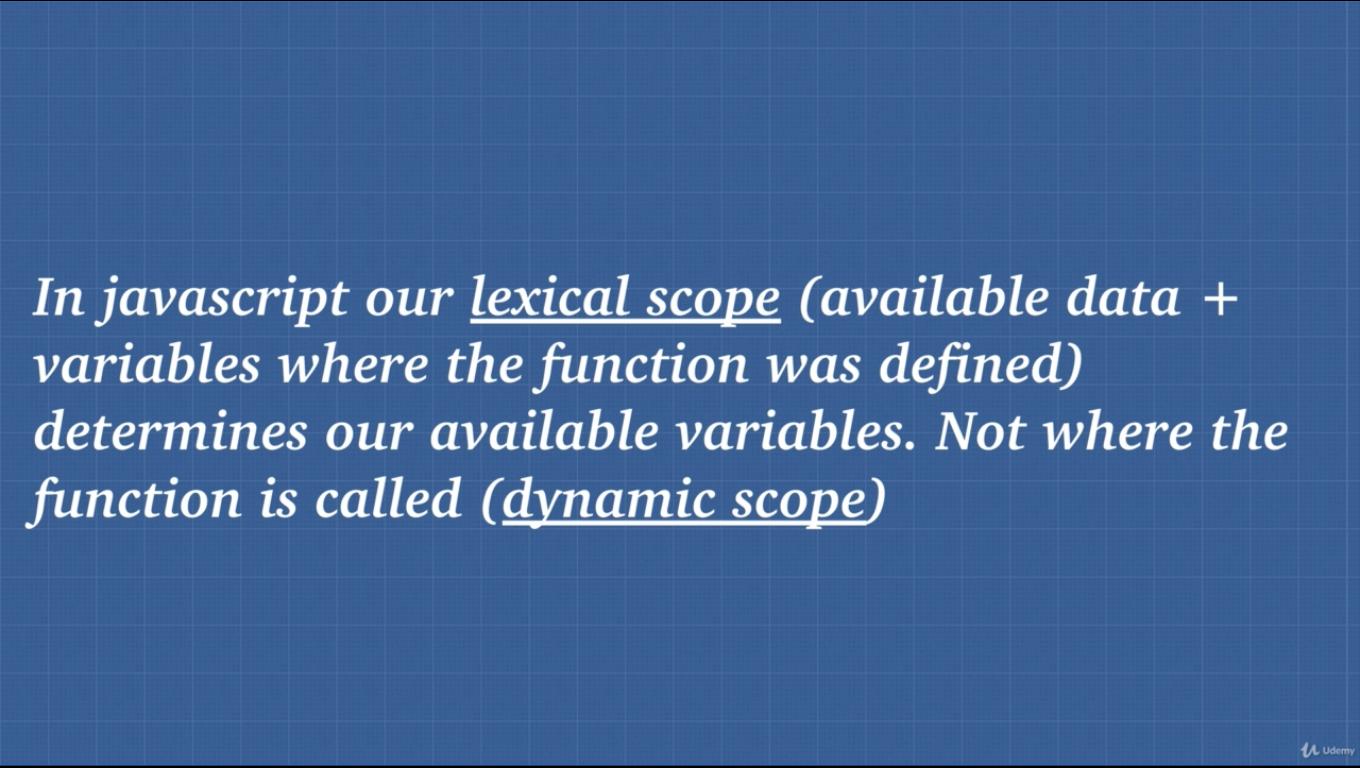
* Variable inside of a function or execution context/little universes.



* On a new execution context/world their own variable will only work in their world/execution context.
* Whenever any function/execution context is popped from the stack their variable will also be removed.
* Reference variable and stuff will be in future lessons.

**11. Scope Chain**

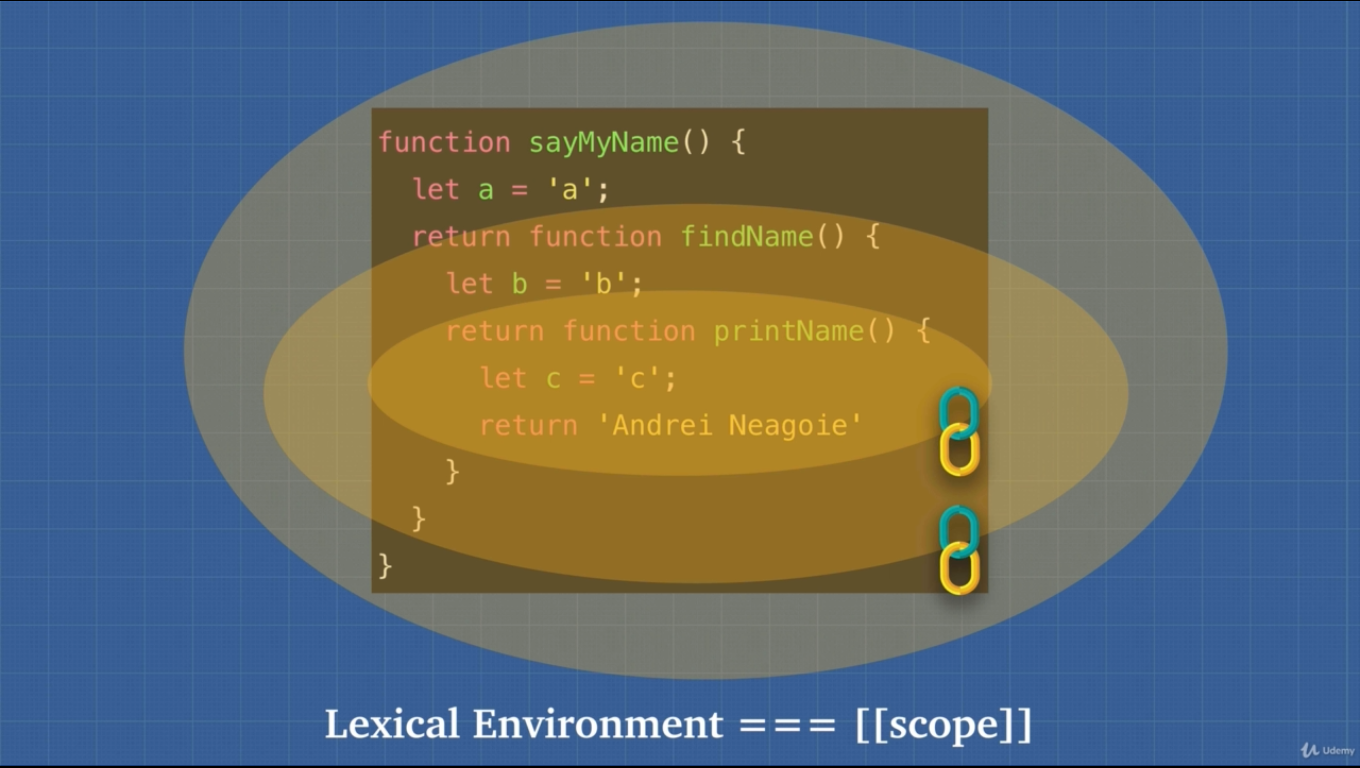
* Each context has a link to its outside world/parent.
* Outer environment/world is defined where the function sits lexically. We may call a function in another function but that doesn’t mean that we can access their variable, again it is defined lexically.
* **Scope chain** gives us access to the parent environment.
* Function firstly looks for any variable inside its own variable environment if it doesn’t find any value then it looks at the parent environment.



(Lexical scope or static scope)

* Lexical scope in a language means that only looking at the source code we can determine which environment the variables and data are available.
* Children has ink to all the ancestors.

**12. [[scope]]**



**13. Exercise JS is Weird**

* We cannot use function expression with its function name. It’ll throw a reference error. It is enclosed in its own scope 😐. It is added to its own execution context variable environment 😐.

**14. Function Scope vs Block Scope**

* Scope means what variable we have access to.
* Previously JS has functional scope using that var ☹ keyword. Now with the introduction of const 😊 and let 😊 in ES6 we can use blocked scope.
* In functional scope we could use other scoped variables 😐.
* With blocked scope variable remains to its own scope nowhere else.

**16. Global Variables**

* Using global variable is bad practice. We should avoid using global variable.
* Variable collision may occur as web pages pack all the script files and run them on a global execution context.
* Memory heap can be filled with data. Browser gets slower.