LESSON 6
EXECUTION CONTEXT & CLOSURES

Actions Supported by All the Laws of Nature

Other References:
https://betterprogramming.pub/execution-context-lexical-environment-and-closures-in-javascript-b57c979341a5
https://medium.com/@happymishra66/execution-context-in-javascript-319dd72e8e2c
https://ui.dev/javascript-visualizer/

Wholeness: A fundamental aspect of function-oriented programming in JavaScript is the use of closures to store state information associated with a function when the function is passed to other objects. Science of Consciousness: Closures provide a protective wrapper for state information associated with a function. An analogy in consciousness is the supportive wrapper that transcendental consciousness provides to our own consciousness. At this level of consciousness we are connected to the home of all the laws of nature.

Main Points

1. Global Environment
2. Execution Context
3. Lexical Environment
4. Closure

The six global DOM objects

- Every JavaScript program can refer to the following global objects:

Name	Description
document	Current HTML page and its content
history	List of pages the user has visited
location	URL of the current HTML page
navigator	Info about the web browser you are using
screen	Info about the screen area occupied by the browser
window	The browser window

The window object

- *the entire browser window; the top-level object in DOM hierarchy*

- technically, all global code and variables become part of the window object
- properties:
  - document, history, location, name
- methods:
  - alert, confirm, prompt (popup boxes)
  - setInterval, setTimeout clearInterval, clearTimeout (timers)
  - open, close (popping up new browser windows)
  - blur, focus, moveBy, moveTo, print, resizeBy, resizeTo, scrollBy, scrollTo

The document object

- JavaScript representation of *the current web page and the elements inside it*

- properties:
  - anchors, body, cookie, domain, forms, images, links, referrer, title, URL
- methods:
  - getElementById
  - getElementsByName
  - getElementsByTagName
  - close, open, write, writeln
- complete list

Main Point

Javascript has a set of global DOM objects accessible to every web page. Every Javascript object runs inside the global window object. The window object has many global functions such as alert and timer methods.

At the level of the unified field, an impulse anywhere is an impulse everywhere.

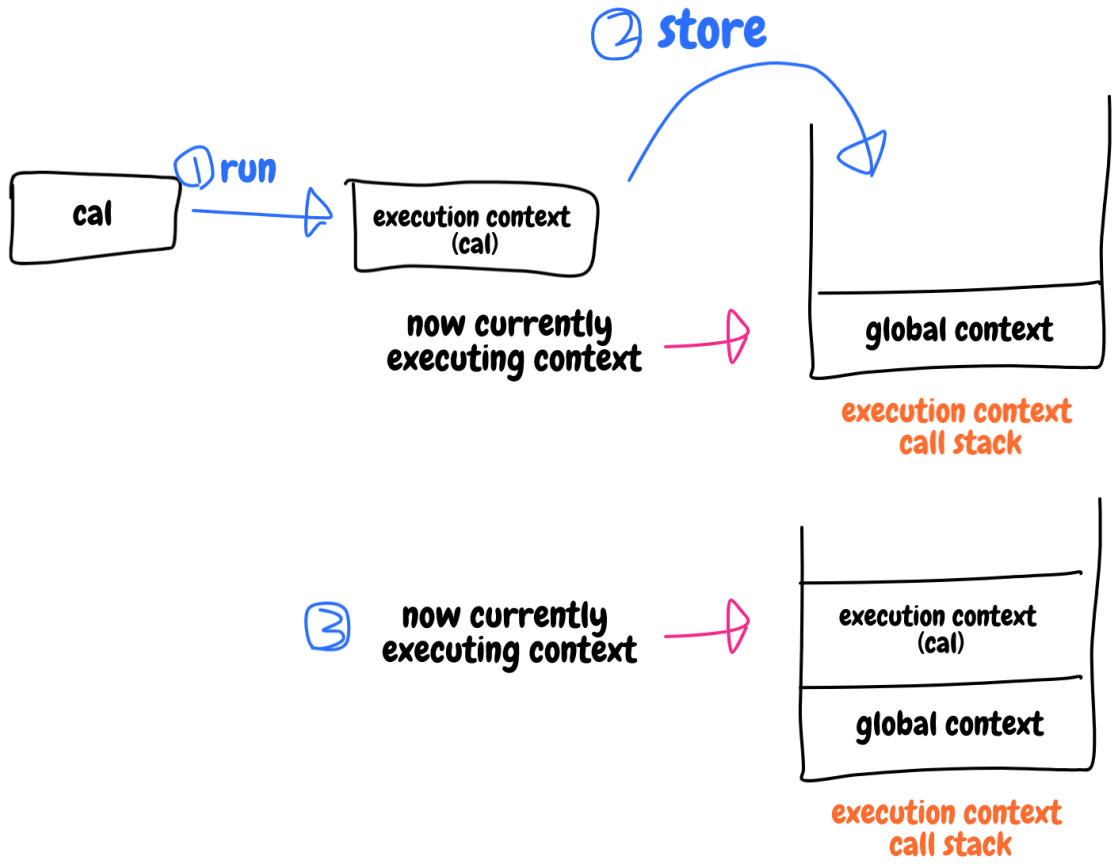
What is Execution Context?

- **Execution context** (**EC**) is created when the JavaScript code is executed.
- All ECs are pushed to Execution Context Stack
- Two types of EC:
  - **Global execution context** (**GEC**):
    - Default, loads first when run JS in browser
    - Only 1 GEC, JS is single threaded
    - Stores global objects: `window, document, history`, etc.
  - **Functional execution context** (**FEC**):
    - Created whenever any function is called
    - Each function has its own execution context
    - No `window` or other global objects
    - `arguments` object

Draw Execution Context Stack

```
function cal(type, a, b) {
    if (type === 'add') {
        return a + b;
    } else if (type === 'subtract') {
        return a - b;
    } else if (type === 'multiply') {
        return a * b;
    } else {
        return a / b;
    }
}

let four = 4;
let seven = 7;
cal('add', four, seven);
```

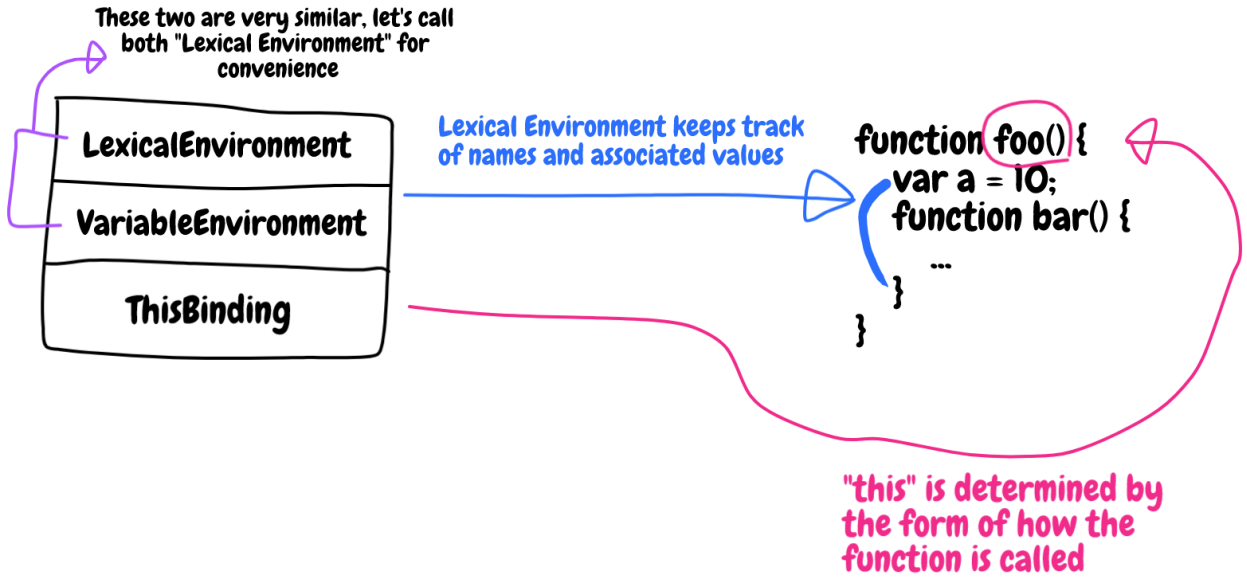


Two Stages Creating EC by JS Engine

1. Creation Phase
   - A function is called but its execution has not started
   - JS engine compile the code, doesn’t execute any code
     - Records variable declarations, functions – only variable declared with `var`, not `let` and `const`
2. Execution Phase
   - Scans through the function
   - Update the variable object with the values of the variables
   - Execute variables declared with `let` and `const`

Creation Phase: Lexical Environment

- A execution context is divided into three different areas
- LE is to keep track of variables, function names and associated values
  - `ThisBinding` determines how the function is called, will explain in later lecture.




```
function foo() {
    var a = 10;
    function bar() {}
}
foo();

// When foo is called, a new exec
ution environment
// might look like this below
execution_environment: {
    LexicalEnvironment: {
        a: undefined,
        bar: function() {}
    },
    ThisBinding: ...
}
```

Creation Phase:
What's inside Lexical Environment?

- EnvironmentRecord
  - records declaration of variables, functions.
- Outer Lexical Environment – Scope
  - Links to parent LexicalEnvironment
  - Used when can't find a property in the current LexicalEnvironment
  - Global LexicalEnvironment doesn't have outer, `null`

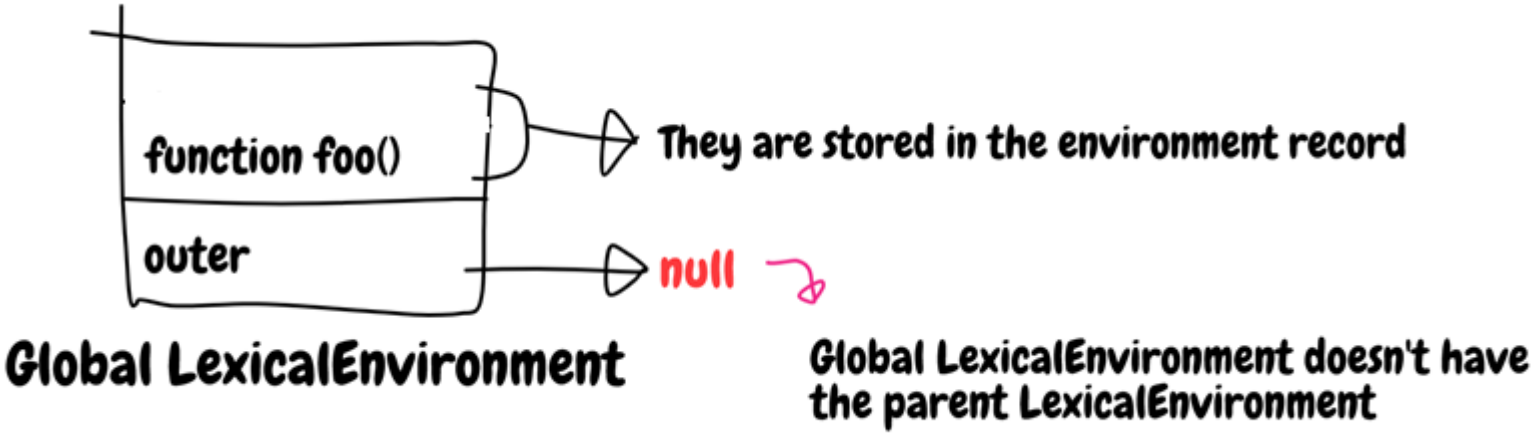
Creation Phase:
Example: The Global Lexical Environment

```
let x = 1;

function foo() {
    let y = 2;

    function bar() {
        let z = 3;

        function baz() {
            console.log(z);
            console.log(y);
            console.log(x);
            console.log(w);
        }
        baz();
    }
    bar();
}
foo();
```



Execution Phase: Scope Chain

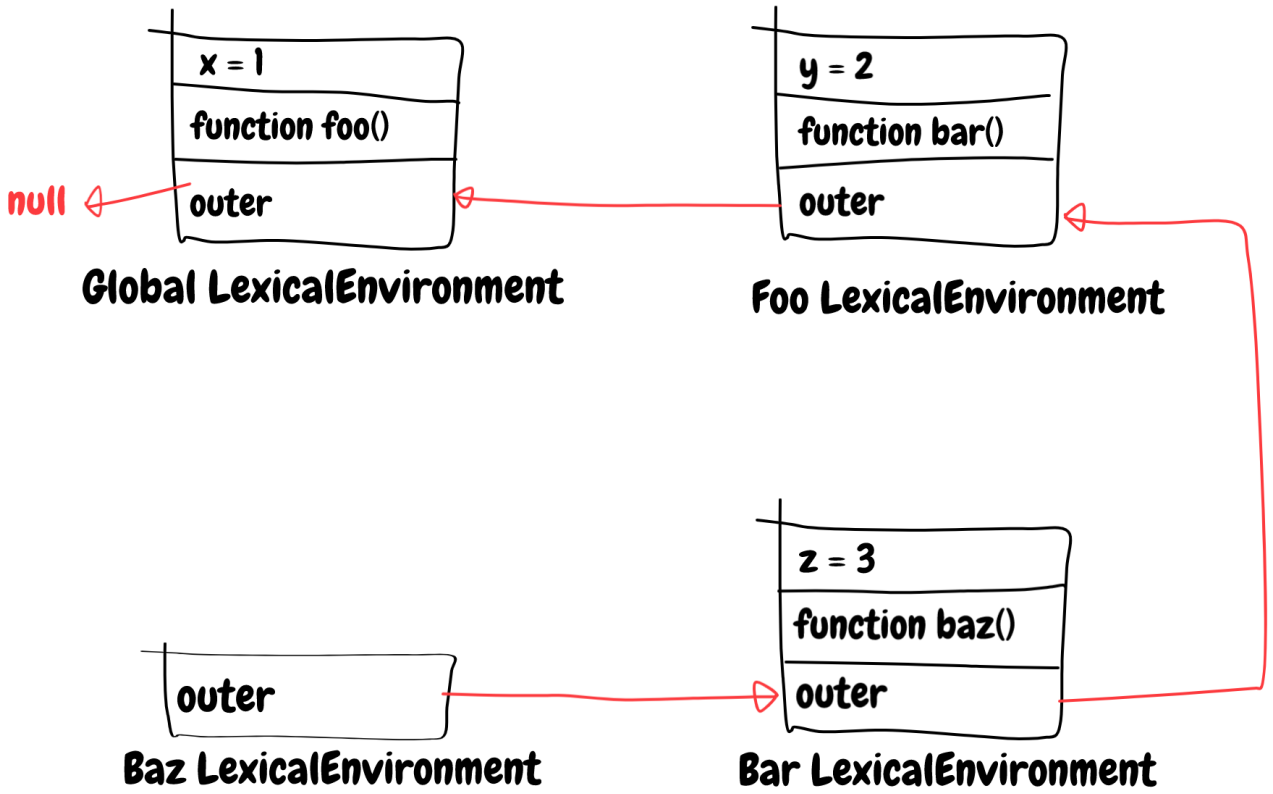
- A list of all the variable objects of functions inside which the current function exists
- The entire relationship amongst LexicalEnvironments

```
let x = 1;

function foo() {
    let y = 2;

    function bar() {
        let z = 3;

        function baz() {
            console.log(z);
            console.log(y);
            console.log(x);
            console.log(w);
        }
        baz();
    }
    bar();
}
foo();
```



Execution Phase: The workflow when `z` and `w` are looked for

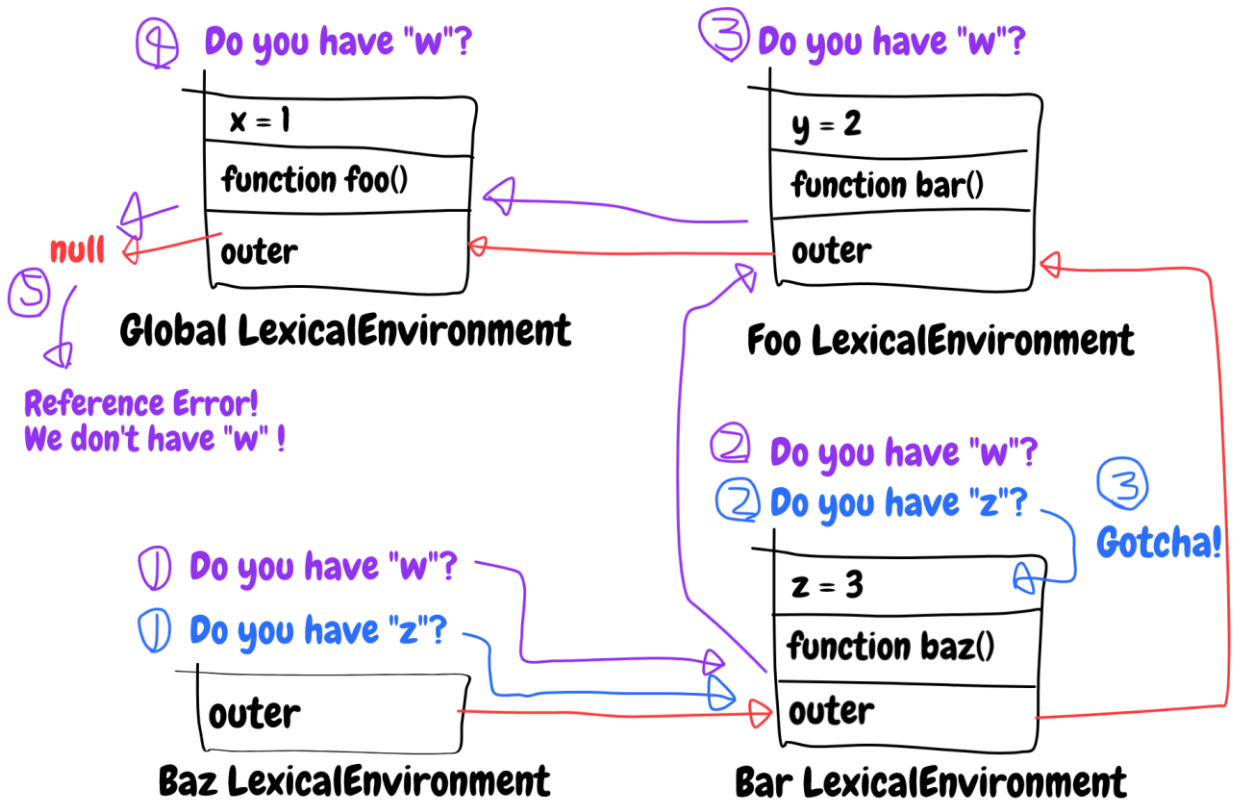
- When `baz()` is called, it looks for `z`, `y`, `x`, and `w`.

```
let x = 1;

function foo() {
    let y = 2;

    function bar() {
        let z = 3;

        function baz() {
            console.log(z);
            console.log(y);
            console.log(x);
            console.log(w);
        }
        baz();
    }
    bar();
}
foo();
```



Nested functions

- A function is called “nested” (inner) when it is created inside another function
- Nested functions are quite common in JavaScript.
- What’s much more interesting, a nested function can be returned:
  - as a property of a new object
    - if the outer function creates an object with methods
  - as a result by itself.
    - can then be used somewhere else.
    - No matter where, it still has access to the same outer variables

Example

```
let a = 1;
let b = 2;

function foo() {
    let a = 3;
    let b = 4;
    bar(a);
    console.log(a, b);

    function baz(arg1, arg2, arg3) {
        console.log(arg1, arg2, arg3);
        b = arg1 + arg2;
        a = arg1 + arg2;
    }

    const f = () => {console.log(a, b)};

    baz(5, 10, f, 40, 50);
    console.log(a, b);
}

function bar(arg1, arg2) {
    console.log(arg1, arg2);
    a = arg1 + 40;
}

console.log(a, b);
```

Main Point Preview: Closures

Closures are created whenever an inner function with free variables is returned or assigned as a callback. Closures provide encapsulation of methods and data. Encapsulation promotes self-sufficiency, stability, and re-usability.

Science of Consciousness: Closures provide a supportive wrapper for actions that will occur in another context. Transcendental consciousness provides a supportive wrapper for our actions that will occur outside of meditation.

What is a Closure?

- (def) A **closure** is the combination of
  - function bundled together (enclosed) with references to its surrounding state (the **lexical environment**).
  - some define closures to be (only) when there is an inner function with free outer variable
    - “free” variables (not defined in the local function and global)

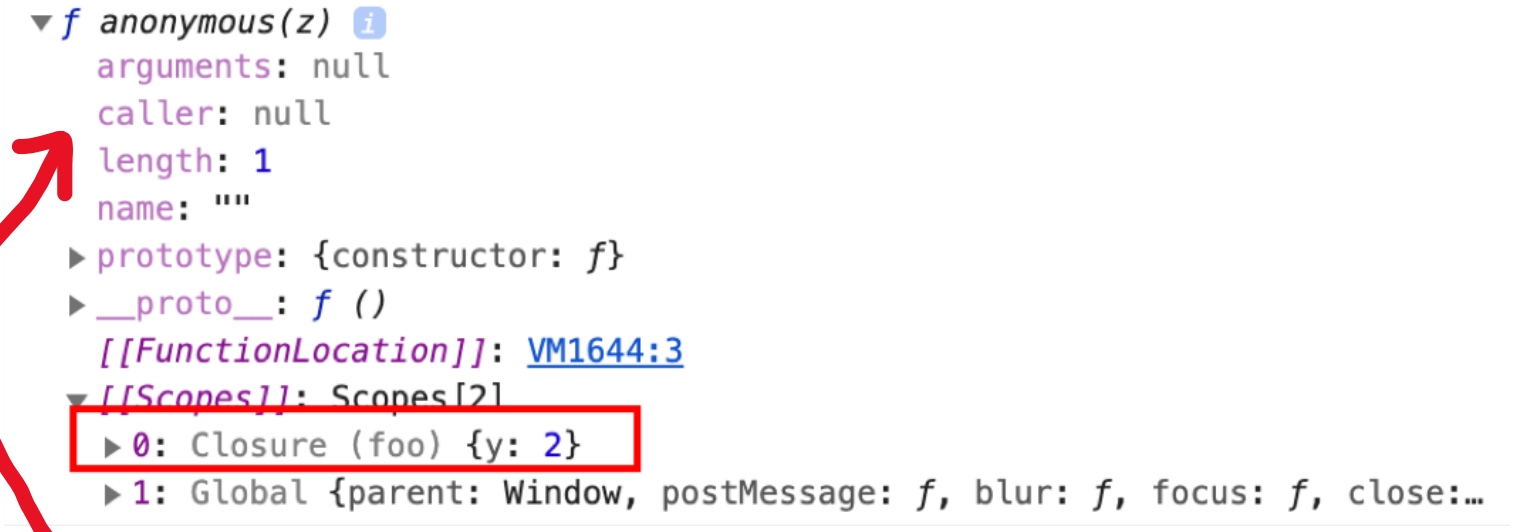
- A closure is created:
  - To expose a function, return it or pass it to another function.
  - inner function will have access to variables in the outer function scope,
    - even after outer function has returned (removed from execution context stack).

Clourse Details

```
let x = 1;

function foo(y) {
    return function(z) {
        return x + y + z;
    }
}

let f = foo(2); // f is clourse
console.dir(f);
```



Closure Scope

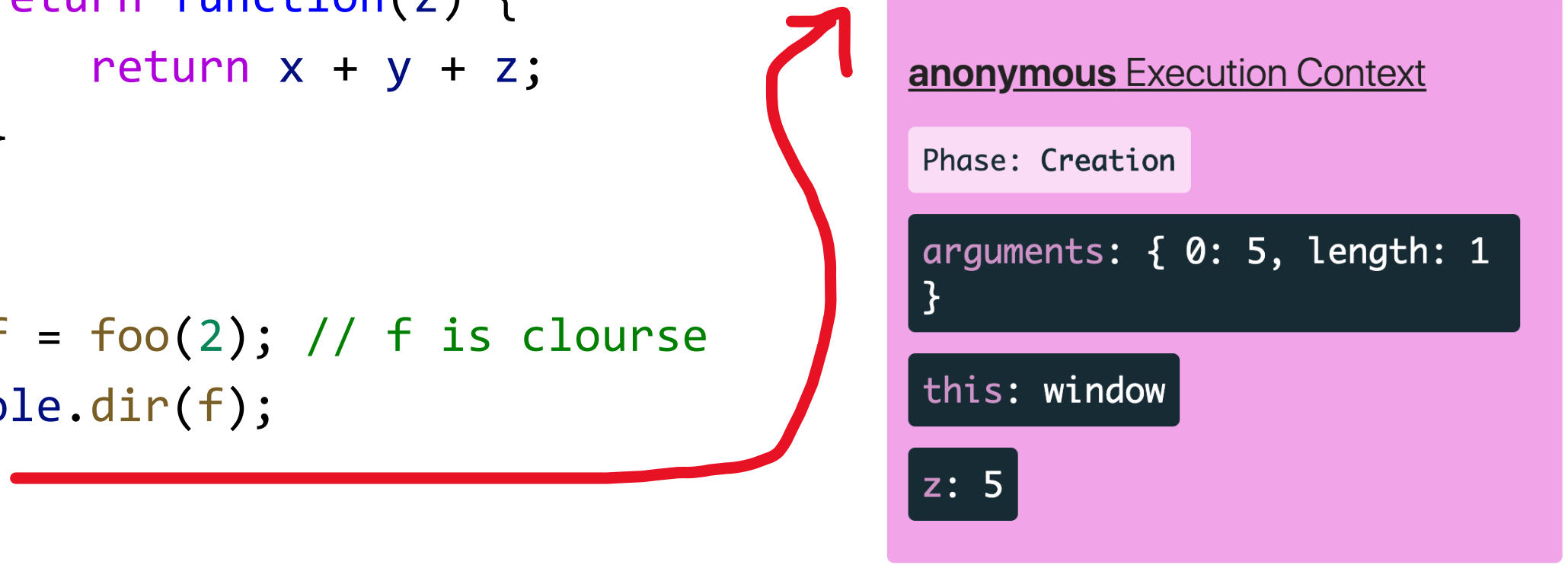
```
arguments: { 0: 2, length: 1 }
this: window
y: 2
```

Execute Clourse Details

```
let x = 1;

function foo(y) {
    return function(z) {
        return x + y + z;
    }
}

let f = foo(2); // f is clourse
console.dir(f);
f(5);
```



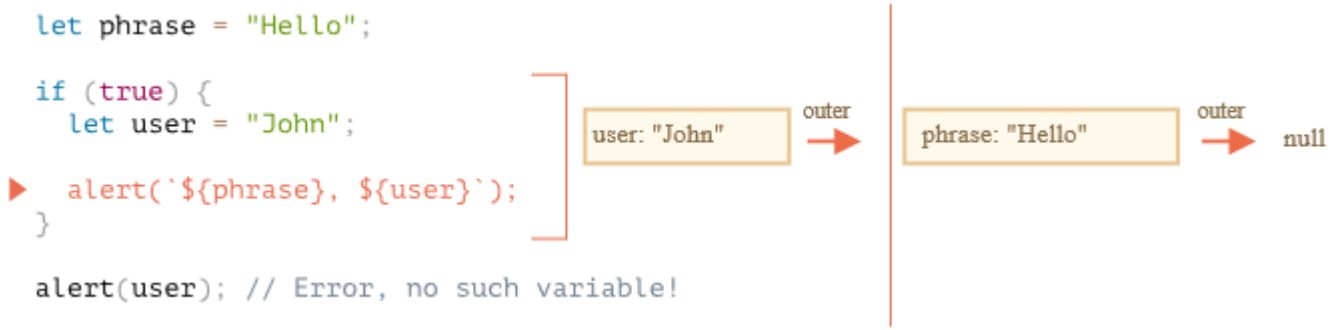
Main Point: Closures

Closures are created whenever an inner function with free variables is returned or assigned as a callback. Closures provide encapsulation of methods and data. Encapsulation promotes self-sufficiency, stability, and re-usability.

Science of Consciousness: Closures provide a supportive wrapper for actions that will occur in another context. Transcendental consciousness provides a supportive wrapper for our actions that will occur outside of meditation.

Code blocks and scope

- a Lexical Environment exists for any code block `{...}`
  - created when a code block runs and contains block-local variables.



- When execution gets to the `if` block,
  - new "`if-only`" Lexical Environment is created for it
  - has the reference to the outer one, so phrase can be found.
  - all variables and Function Expressions declared inside `if` reside in that Lexical Environment
    - can't be seen from the outside
    - after if finishes, the alert below won't see the user, hence the error.

For, while

- For a loop, every iteration has a separate Lexical Environment.
  - If a variable is declared in `for(let ...)`, then it's also in there:

```
for (let i = 0; i < 10; i++) {
    // Each loop has its own Lexical Environment
    // {i: value}
}
alert(i); // Error, no such variable
```

- for `let i` is visually outside of `{...}`.
  - The `for` and `while` constructs are special
  - each iteration of the loop has its own Lexical Environment with the current `i` in it.
- like `if`, after the loop `i` is not visible.

Bare code blocks

- can use “bare” code block `{…}` to isolate variables into a “local scope”.
  - in web browser all scripts share the same global area.
  - create a global variable in one script, it becomes available to others.
  - source of conflicts if two scripts use the same variable name
    - Last one loaded overwrites and becomes source of values for early script too
    - if the variable name is a widespread word, e.g., lat, long

- to avoid that, can use a code block to isolate the whole script or a part of it:

```
{
    // do some job with local variables that should not be seen outside
    let message = "Hello";
    alert(message); // Hello
}
alert(message); // Error: message is not defined
```

- code outside block doesn’t see variables inside the block
  - Every block has own Lexical Environment.

The old "var"

- In the very first chapter about variables, we mentioned three ways of variable declaration:
  1. `let`
  2. `const`
  3. `var`
- `let` and `const` behave exactly the same way in terms of Lexical Environments.
  - `var` is a very different beast,
  - originates from very old times.
  - generally not used in modern scripts, but still lurks in the old ones.
- If you don't plan on meeting such scripts you may even skip this chapter or postpone it
  - But is used in millions of programs – anything prior to ES6

- function scope
  - Ignores blocks
  - Are always 'hoisted', which means they are visible throughout the function
    - Even if defined at the end! (like a function declaration)
    - Only the declaration is hoisted, not the assignment
    - Undefined until assignment reached

IIFE

- Before ES6 no block-level lexical environment in JavaScript.
  - Only function scope
- “immediately-invoked function expressions” (abbreviated as IIFE).
  - Special syntax to wrap code and protect global namespace
  - declare a Function Expression and run it immediately
  - nowadays there’s no reason to write such code

```
(function() {
    var message = "Hello";
    alert(message); // Hello
})();
```

JavaScript “strict” mode

```
"use strict"; your code...

(function() {
 "use strict";
 your code...
})();
```

- writing "use strict"; at the very top of your JS file turns on strict syntax checking:
  - shows an error if you try to assign to an undeclared variable
  - stops you from overwriting key JS system libraries
  - forbids some unsafe or error-prone language features
- "use strict" also works inside of individual functions
- You should *always* turn on strict mode for your code in this class!

Main Point Preview: Timout callbacks

The asynchronous global methods setTimeout and setInterval take a function reference as an argument and then callback the function at a specified time. Science of Consciousness: Accepting an assignment and carrying it out at a designated time is a fundamental capability required for intelligent behavior. A clear awareness and mind promotes good memory and the ability to successfully execute tasks.

Timers

- `setTimeout` allows to run a function once after the interval of time.
- `setInterval` allows to run a function regularly with the interval between the runs.

setTimeout

```
let timerId = setTimeout(func, [delay], [arg1], [arg2], ...)
```

- Func: Function or a string of code to execute.
- Delay: delay before run, in milliseconds (1000 ms = 1 second), by default 0.
- arg1, arg2… : Arguments for the function

```
function sayHi() {
    alert('Hello');
}
setTimeout(sayHi, 1000);
```

- With arguments:

```
function sayHi(phrase, who) {
    alert(phrase + ', ' + who);
}
setTimeout(sayHi, 1000, "Hello", "John"); // Hello, John
```

Pass a function, but don't run it

- Novice developers sometimes make a mistake by adding brackets ()

```
// wrong!
setTimeout(sayHi(), 1000);
```

- doesn't work,
  - `setTimeout` expects a reference to a function.
  - here `sayHi()` runs the function,
  - result of its execution is passed to `setTimeout`.
  - result of `sayHi()` is `undefined` (the function returns nothing), so nothing is scheduled

- function call versus function binding
  - `sayHi()` versus `sayHi`
  - execute the function versus reference to the function
  - fundamental concept!!

Canceling with `clearTimeout`

A call to `setTimeout` returns a “timer identifier” `timerId` that we can use to cancel the execution.

```
let timerId = setTimeout(...);
clearTimeout(timerId);
```

schedule the function and then cancel it

```
let timerId = setTimeout(() => alert("never happens"), 1000);
alert(timerId); // timer identifier
clearTimeout(timerId);
alert(timerId); // same identifier (doesn't become null after canceling)
```

setInterval

*

The `setInterval` method has the same syntax as `setTimeout`:

```
let timerId = setInterval(func, [delay], [arg1], [arg2], ...)
```

Repeatedly calls the function after the given interval of time.

To stop further calls, we should call `clearInterval(timerId)`.

```
// repeat with the interval of 2 seconds
let timerId = setInterval(() => alert('tick'), 2000);
// after 5 seconds stop
setTimeout(() => { clearInterval(timerId); alert('stop'); }, 5000);
```

Zero delay setTimeout

*

- There's a special use case: `setTimeout(func, 0)`, or just `setTimeout(func)`.

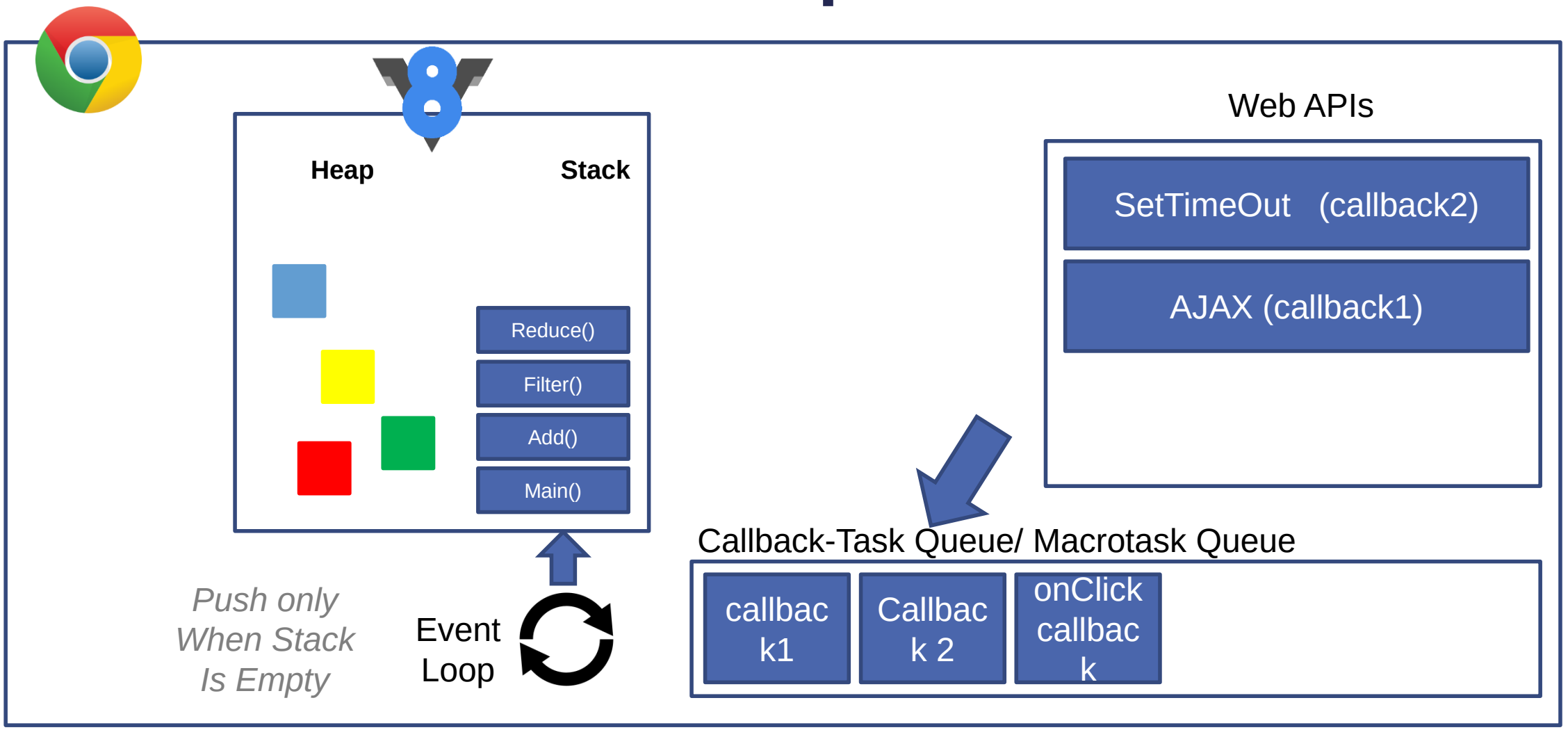
- schedules the execution of func as soon as possible.
  - after the current code is complete.

```
setTimeout(() => alert("Hello"), 0);
alert("World");
```

- The first line "puts the call into calendar after 0ms"
  - scheduler will only "check the calendar" after the current code is complete
  - `"Hello"` is first, and `"World"` – after it.

- There are also advanced browser-related use cases of zero-delay timeout, that we'll discuss in the chapter Event loop: microtasks and macrotasks.

Chrome – The Event Loop



If you block the stack, browser can't run the render queue

Main Point: Timout callbacks

The asynchronous global methods setTimeout and setInterval take a function reference as an argument and then callback the function at a specified time. Science of Consciousness: Accepting an assignment and carrying it out at a designated time is a fundamental capability required for intelligent behavior. A clear awareness and mind promotes good memory and the ability to successfully execute tasks.

CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

Actions Supported by All the Laws of Nature

1. Closures are a feature of functional programming languages that allow state information to be encapsulated with functions when they are passed among objects.
2. The scope of variables in modern JavaScript is defined by the lexical environment.

---

3. **Transcendental consciousness**. Is the home of all the laws of nature.
4. **Impulses within the transcendental field:** Thoughts encapsulated by this deep level of consciousness will result in actions in accord with all the laws of nature.
5. **Wholeness moving within itself:** In unity consciousness all thoughts and perceptions are enhanced by this supportive experience.