# Scope, closures and encapsulation in JavaScript

## Maharishi University of Management - Fairfield, Iowa © 2019

## Main Point Preview

- JavaScript has global scope and local scope within functions when variables are declared with var, and now has block scope with const and let.

- **Science of Consciousness**: The experience of transcending opens our awareness to the expanded scope of unbounded awareness, at the same time that it promotes the ability to focus sharply within any local boundaries.

## The global object

- technically no JavaScript code is "static" in the Java sense
  - all code lives inside of some object
  - there is always a `this` reference that refers to that object

- all code is executed inside of a global object
  - in browsers, it is also called window;
  - global variables/functions you declare become part of it
    - they use the global object as `this` when you call them

- "JavaScript's global object [...] is far and away the worst part of JavaScript's many bad parts." -- D. Crockford

## Implied globals

- name=value;

```
function foo() {
 x = 4;
 console.log(x);
} // oops, x is still alive now (global)
```

- if you assign a value to a variable without `var`, JS assumes you want a new global variable with that name
  - hard to distinguish
  - this is a "bad part" of JavaScript (D.Crockford)

## Scope

- scope: The enclosing context where values and expressions are associated.
  - essentially, the visibility of various identifiers in a program

- lexical scope: Scopes are nested via language syntax; a name refers to the most local definition of that symbol.
  - most modern languages (Java, C, ML, Scheme, JavaScript)

- dynamic scope: A name always refers to the most recently executed definition of that symbol.
  - Perl, Bash shell, Common Lisp (optionally), APL, Snobol

## Lexical scope in Java

- In Java, every block ( {} ) defines a scope.

```
public class Scope {
    public static int x = 10;

    public static void main(String[] args) {
        System.out.println(x);
        if (x > 0) {
            int x = 20;
            System.out.println(x);
        }
        int x = 30;
        System.out.println(x);
    }
}
```

# Lexical scope in JavaScript (pre-ES6)

- In JavaScript, there are only two scopes:
  - global scope: global environment for functions, vars, etc.
  - function scope: every function gets its own inner scope

Global Scope

Function Scope

```
var x = 10;
function main() {
  console.log("x1: " + x);
  x = 20;
  if (x > 0) {
    var x = 30;
    console.log("x2: " + x);
  }
  var x = 40;
  var f = function(x) { console.log("x3: " + x); }
  f(50);
}
main();
```

# Lack of block scope

```
for (var i = 0; i < 10; i++) {
 console.log("i inside for loop: " + i);
}
console.log(i); // 10
if (i > 5) {
 var j = 3;
}
console.log("j: " + j);
```

- any variable declared lives until the end of the function
  - lack of block scope in JS leads to errors for some coders
  - this is a "bad part" of JavaScript (D. Crockford)

## `var` vs `let` (ES6)

- `var` scope – nearest function scope
- `let` scope – nearest enclosing block

```
function a() {
 for (var x = 1; x < 10; x++) {
 console.log(x);
}
console.log("x: " + x);
//10
}
```

```
function a() {
 for (let x = 1; x < 10; x++) {
 console.log(x);
 }
 console.log("x: " + x);
//ReferenceError: x is not defined
}
```

- Use `let` inside for loops to prevent leaking to Global Scope
  - never use var in new JS code

## Best Practices

- variables defined with `var` are hoisted and have value `undefined` until it is assigned a value in code
  - Do not use var assignments in new code
- When using `let` or const, there will be no hoisting and we will receive a reference `error` if used before they are declared
- Best practice is to use const or let and explicitly declare them before using
  - Makes code more obvious for humans to understand
  - Use `const` by default
  - Only use `let` if you need to update variable later
  - Don't use `var`
- But, millions of legacy programs use var and any competent JS programmer must understand hoisting

## Main Point

JavaScript has global scope and local scope within functions when variables are declared with var, and now has block scope with const and let.

**Science of Consciousness:** The experience of transcending opens our awareness to the expanded scope of unbounded awareness, at the same time that it promotes the ability to focus sharply within any local boundaries.

## Main Point Preview
## 2-pass compiler

JavaScript has a 2-pass compiler that hoists all function and variable declarations. These declarations are visible anywhere in the current function scope regardless of where they are declared. Variables have value 'undefined' until the execution pass and an assignment is made.

**Science of Consciousness:** The first pass sets up the proper conditions for the successful execution of the second pass. Similarly, when we set up the proper conditions for transcending then all we have to do is let go and nature will ensure that the experience is successful.

## Code Execution and Hoisting: 2 phase compilation

- When your code is being executed, the JS engine in the browser will create the global environment objects along with "`this`" object and start looking in your code for functions and variables.
- In **first phase**, JS engine looks through all global code for functions and variables (hoisting)
  - functions: saves entire function definition
  - variables: saves only variable name and value of 'undefined'
  - Only 'hoists' variable and function declarations
  - No variable initialization or function expressions are hoisted
- In **second phase**, JS engine will execute your code line-by-line and call functions and create execution context for every function(scope) in the execution stack.

## Variable Declarations Are Hoisted

- JavaScript hoisted all variable declarations – moves them to the beginning of their direct scopes(function).

```
function f(){
 console.log(bar); //undefined
 var bar = "abc";
 console.log(bar); //abc
}
```

- JavaScript executes f() as if its code were:

```
function f(){
 var bar;
 console.log(bar); //undefined
 bar = "abc";
 console.log(bar); //abc
}
```

## Function Declaration Hoisting

- Hoisting: moving to the beginning of a scope.
- Function declarations are hoisted completely

```
foo();
function foo(){ }
```

- JavaScript executes the code as if it looked like this:

```
function foo(){ }
foo();
```

## Function Expression are NOT hoisted

- Function expressions are not hoisted, so cannot use function expression functions before they are defined.

```
foo();
var foo = function (){
  ...
};
```

- JS Engine executes the code as:

```
var foo;
foo(); //TypeError: undefined is not a function
var foo = function (){
  ...
};
```

## Hoisting Example

```
var a = 5;
function b() {
 console.log("function is called!");
}
console.log("a: " + a); //5
b(); // function is called!
```

- Notice what will happen when we switch between the lines:

```
console.log("a: " + a); //undefined
b(); // function is called!
var a = 5;
function b() {
 console.log("function is called!");
}
```

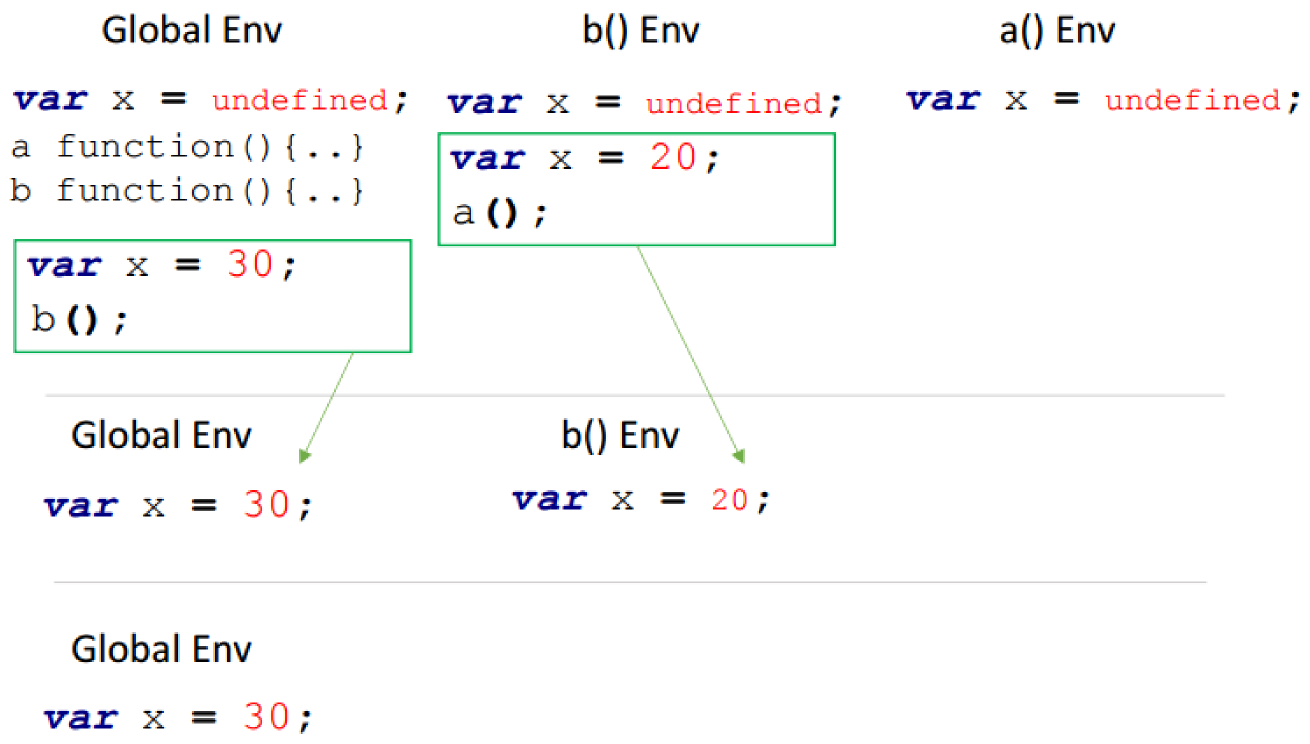
- What will happen if remove the variable `a` definition?

# Execution context & stack example

```
function a() {
 var x;
}

function b() {
 var x = 20;
 a();
}

var x = 30;
b();
console.log(x);
```



```
Global Env                   b() Env                      a() Env
var x = undefined;           var x = undefined;           var x = undefined;
a function(){..}             var x = 20;
b function(){..}             a();
var x = 30;
b();

Global Env                   b() Env
var x = 30;                  var x = 20;

Global Env
var x = 30;
```

## Main Point : 2-pass compiler

JavaScript has a 2-pass compiler that hoists all function and variable declarations. These declarations are visible anywhere in the current function scope regardless of where they are declared. Variables have value 'undefined' until the execution pass and an assignment is made.

**Science of Consciousness:** The first pass sets up the proper conditions for the successful execution of the second pass. Similarly, when we set up the proper conditions for transcending then all we have to do is let go and nature will ensure that the experience is successful.

## Main Point Preview
## Scope chain and execution context

When we ask for any variable, JS will look for that variable in the current scope. If it doesn't find it, it will consult its outer scope until we reach the global scope.

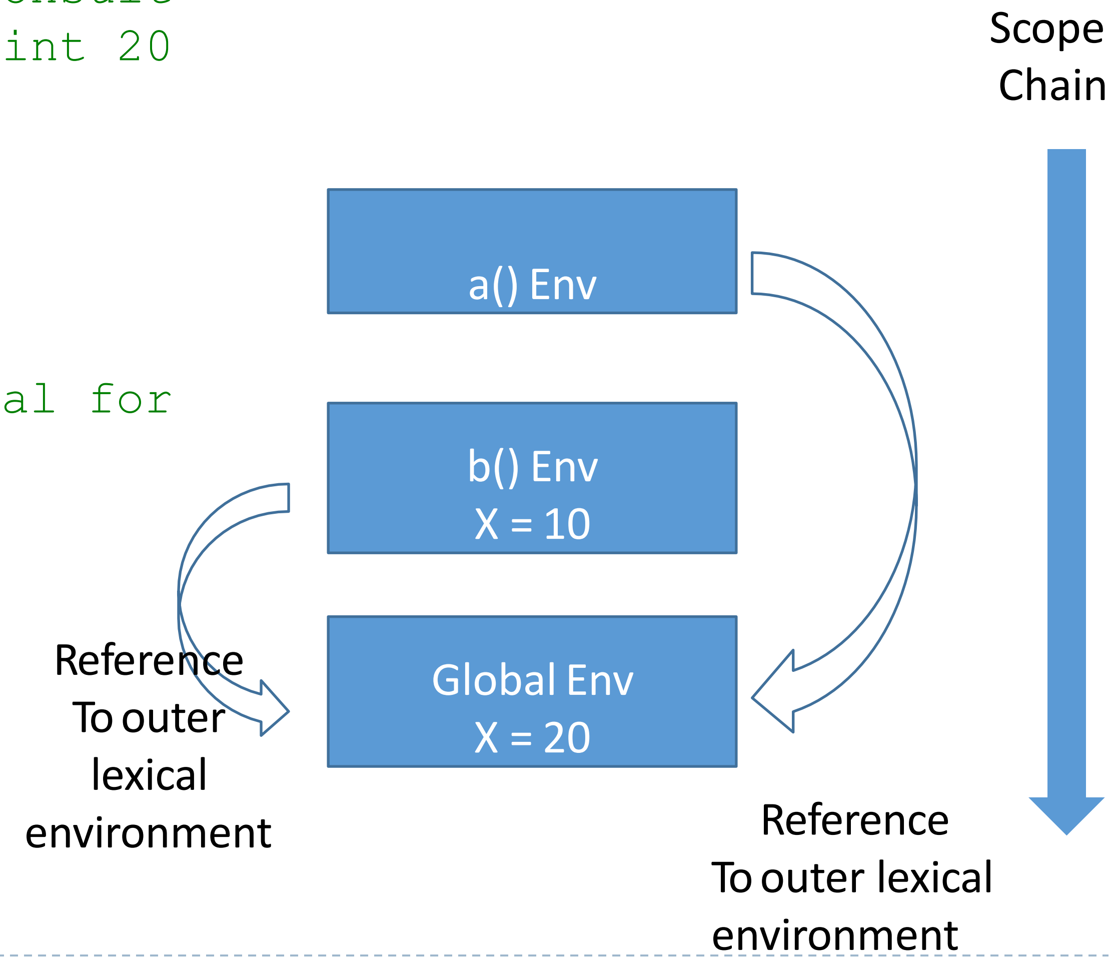
**Science of Consciousness:** During the process of transcending we naturally proceed from local awareness to more subtle levels of awareness to unbounded awareness.

## Scope Chain

- When we ask for any variable, JS Engine will look for what variable in the current scope. If it doesn’t find it, it will consult its outer scope until we reach the global scope.

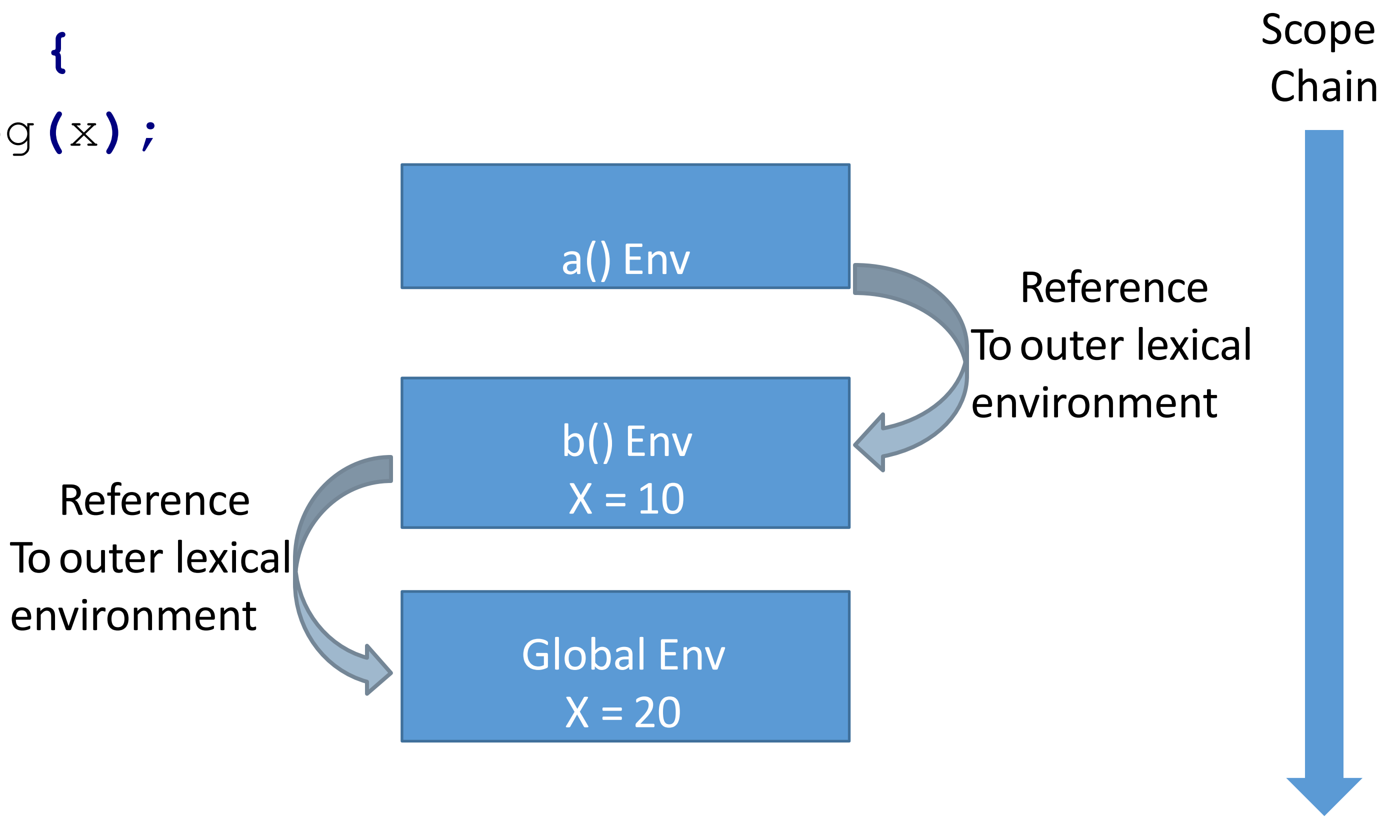
# Scope Example1

```
function a() {
 console.log(x); // consult
  Global for x and print 20
  from Global
}
function b() {
 var x = 10;
 a(); // consult Global for
  a
}
var x = 20;
b();
```


Scope Chain
a() Env
b() Env
X = 10
Global Env
X = 20
Reference To outer lexical environment
Reference To outer lexical environment

# Scope Example 2: Inner function

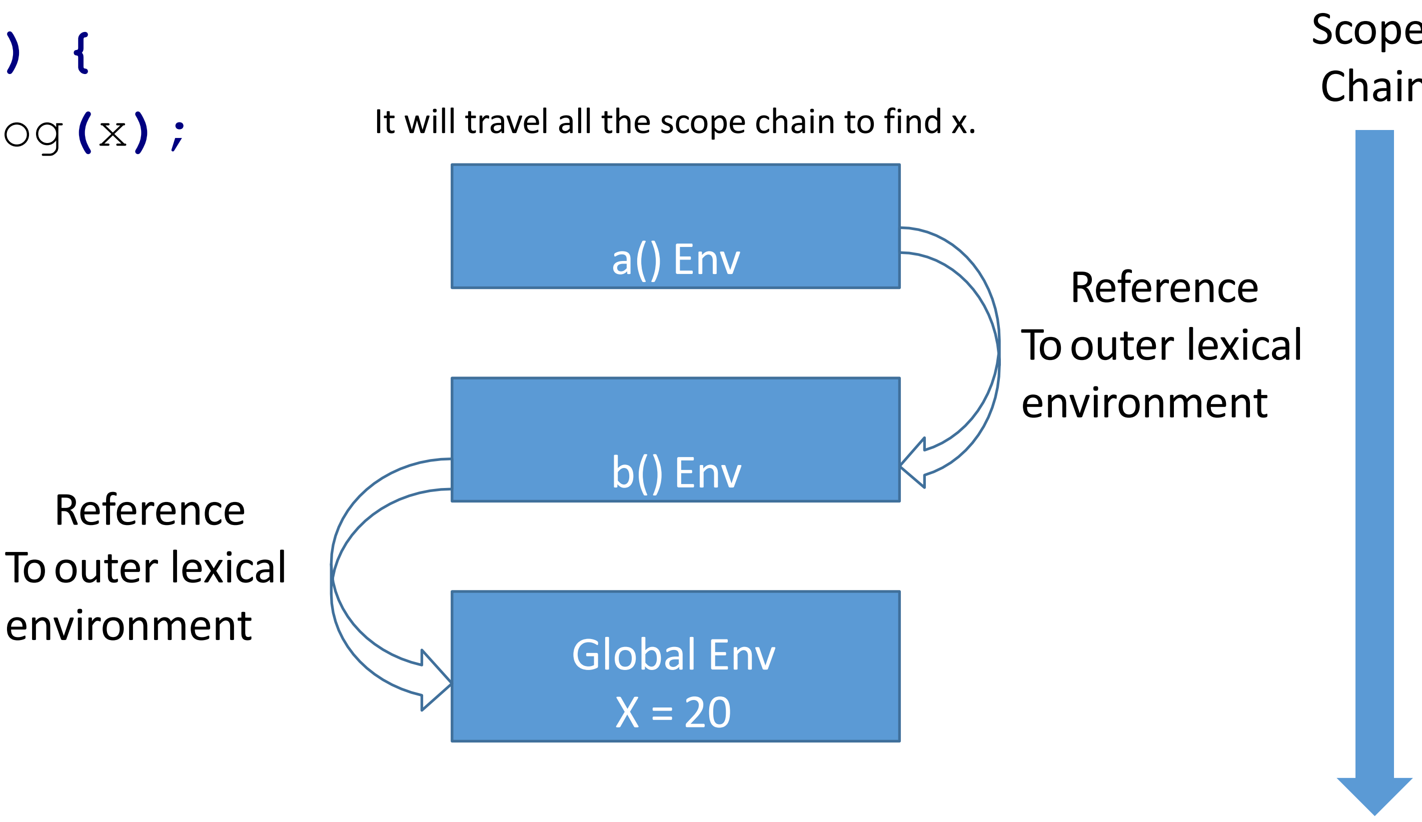
```
function b() {
 function a() {
   console.log(x);
 }
 var x = 10;
 a();
}
var x = 20;
b();
```



## Scope Example 3

```
function b() {
 function a() {
   console.log(x);
 }
 a();
}
var x = 20;
b();
```

It will travel all the scope chain to find x.



## Scope Example 3

```
function f() {
 var a = 1, b = 20, c;
 console.log(a + " " + b + " " + c);
 function g() {
   var b = 300, c = 4000;
   console.log(a + " " + b + " " + c);
   a = a + b + c;
   console.log(a + " " + b + " " + c);
 }
 console.log(a + " " + b + " " + c);
 g();
 console.log(a + " " + b + " " + c);
}
f();
```

## Scope Exampe 4

```
var x = 10;
function main() {
 console.log("x1 is:" + x);
 x = 20;
 console.log("x2 is:" + x);
 if (x > 0) {
    var x = 30;
    console.log("x3 is:" + x);
 }
 console.log("x4 is:" + x);
 var x = 40;
 var f = function(x) {
   console.log("x5 is:" + x);
 };
 f(50);
 console.log("x6 is:" + x);
}
main();
console.log("x7 is:" + x);
```

## Main Point
## Scope chain and execution context

When we ask for any variable, JS will look for that variable in the current scope. If it doesn't find it, it will consult its outer scope until we reach the global scope.

**Science of Consciousness:** During the process of transcending we naturally proceed from local awareness to more subtle levels of awareness to unbounded awareness.

## Main Point Preview

Closures are created whenever an inner function is defined and it closes over its free variables. Closures provide encapsulation of methods and data. Encapsulation promotes self-sufficiency, stability, and re-usability.

**Science of Consciousness:** Transcendental consciousness provides encapsulation of thoughts, perceptions and actions by our Self, pure awareness. This experience provides a common stable positive blissful environment for all point value thoughts, perceptions and actions.

## Recall: function as first-class citizen

- we created a function named invokeMe that invokes whatever we pass to it (line 1, 2, 3)
- Upon calling the function invokeMe we passed to it an anonymous function (line 4, 5, 6)
- The anonymous function will be invoked (line 2)

```
1. function invokeMe(x) {
2.       x();
3. }

4. invokeMe(function() {
5.       console.log('Hi');
6. });
```

## Recall: Calling an inner function

```
function init() { //function declaration
 var name = "Mozilla";
 function displayName() {
      alert(name);
 }
 displayName();
}
init();
```

## Returning an inner function

```
function makeFunc() {
 var name = "Mozilla"; //local to makeFunc
 function displayName() {
  alert(name);
 }
 return displayName;
}
var myFunc = makeFunc();
myFunc(); //is the local variable still
  accessible by myFunc?
```

## Master the JavaScript Interview: What is a Closure?

- "most competent interviewers will ask you what a closure is, and most of the time, getting the answer wrong will cost you the job."
  - "Be prepared for a quick follow-up: Can you name two common uses for closures?"
- first and last question in my JavaScript interviews.
  - can't get very far with JavaScript without learning about closures.
- You can muck around a bit, but will you really understand how to build a serious JavaScript application? Will you really understand what is going on? Not knowing the answer to this question is a **serious red flag**.
- Not only should you know the mechanics of what a closure is,
  - you should know why it matters,
  - should know several possible use-cases for closures.

## Closures

- closure: A first-class function that binds to free variables that are defined in its execution environment.

- free variable: A variable referred to by a function that is not one of its parameters or local variables.
  - bound variable: A free variable that is given a fixed value when "closed over" by a function's environment.

- A closure occurs when a(n inner) function is defined and it attaches itself to the free variables from the surrounding environment to "close" up those stray references.

## Closures in JS

```
const x = 1;
function f() {
    let y = 2;
    const sum = function () {
        const z = 3;
        console.log(x + y + z);
    }
    y = 10;
    return sum;
}
const g = f();
g();
```

- inner function closes over free variables as it is declared
  - grabs references to the names, not values

# Common closure bug with fix

```
var funcs = [];
for (var i = 0; i < 5; i++) {
 funcs[i] = function() {
   return i;
 };
}
console.log(funcs[0]());
console.log(funcs[1]());
console.log(funcs[2]());
console.log(funcs[3]());
console.log(funcs[4]());
```

- Closures that bind a loop variable often have this bug.
- Why do all of the functions return 5?

```
var helper = function(n) {
 return function() {
   return n;
 }
}
var funcs = [];
for (var i = 0; i < 5; i++)
   {
 funcs[i] = helper(i);
};
console.log(funcs[0]());
console.log(funcs[1]());
console.log(funcs[2]());
console.log(funcs[3]());
console.log(funcs[4]());
```

## Common closure bug with fix (ES6)

```
var funcs = [];
for (var i = 0; i < 5; i++) {
 funcs[i] = function() {
   return i;
 };
}
console.log(funcs[0]());
console.log(funcs[1]());
console.log(funcs[2]());
console.log(funcs[3]());
console.log(funcs[4]());
```

```
var funcs = [];
for (let i = 0; i < 5; i++) {
 funcs[i] = function() {
   return i;
 };
}
console.log(funcs[0]());
console.log(funcs[1]());
console.log(funcs[2]());
console.log(funcs[3]());
console.log(funcs[4]());
```

## Practical uses of closures

- A closure lets you associate some data (the environment) with a function—parallel to properties and methods in OOP.
- Consequently, you can use a closure anywhere you might use an object with a single method.
- Situations like this are common on the web.
  - an event handlers is a single function executed in response to an event.
    - – e.g., DOM and timer event handlers
  - closures also very useful in Javascript for encapsulation and namespace protection

## Function factory with closures

example of closures being helpful with event handling

```
<a href="#" id="size-12">Size 12</a>
<a href="#" id="size-16">Size 16</a>
<a href="#" id="size-18">Size 18</a>

function makeSizer(size) {
 return function() {
 document.body.style.fontSize = size + "px";
 };
}
document.getElementById("size-12").onclick =
  makeSizer(12);
document.getElementById("size-16").onclick =
  makeSizer(16);
document.getElementById("size-18").onclick =
  makeSizer(18);
```

# Function factory with closures (cont)

- have a function that sets the fontsize, and want to have some state info (about the environment)
  - state info is the font size that are wanting
  - normally could make this a parameter,
  - but have to add parameter without executing the function
  - also, the click event will not pass any parameters to the callback function
  - hence, if want to save some state info along with the function, the way to do it in javascript is to use a closure because it creates a function and can save the parameter to be used later when the event fires
- in java, could have a "functor object" to do the same sort of thing.
  - could also do this in javascript if the client of the callback was expecting an object instead of a function.
  - Then, also need the client and object to have a common understanding of what the method name will be in the "functor object". With callbacks, the method does not have to have a name.

## Main Point

Closures are created whenever an inner function is defined and it closes over its free variables. Closures provide encapsulation of methods and data. Encapsulation promotes self-sufficiency, stability, and re-usability.

**Science of Consciousness:** Transcendental consciousness provides encapsulation of thoughts, perceptions and actions by our Self, pure awareness. This experience provides a common stable positive blissful environment for all point value thoughts, perceptions and actions.

## Main Point Preview

Functional programming methods map, filter, reduce make code more understandable and error free by automating details of general-purpose looping mechanisms, indicating their intent by their name, and not changing the state of the original array.

**Science of Consciousness:** Growth of consciousness makes behavior simpler and more error free because intentions that arise from deep levels are spontaneously in accord with and supported by all the laws of nature.

## Good things to know

- No function overloading in JavaScript
  - Extra arguments ignored and missing ones ignored
- Arguments object and Rest (ES6) parameters
- Arrow functions(ES6)
  - Syntactic sugar for anonymous functions (ala Java lambdas)
- Functional programming via map, filter, reduce

## Function Signature

- If a function is called with missing arguments(less than declared), the missing values are set to : `undefined`

```
function f(x) {
 console.log("x: " + x);
}
f();
f(1);
f(2, 3);
```

## No overloading!

```
function log() {
 console.log("No Arguments");
}
function log(x) {
 console.log("1 Argument: " + x);
}
function log(x, y) {
 console.log("2 Arguments: " + x + ", " + y);
}
log();
log(5);
log(6, 7);
```

- Why? Functions are objects!

## arguments Object

The **arguments** object is an Array-like object corresponding to the arguments passed to a function.

```
function findMax() {
 var i;
 var max = -Infinity;
 for (i = 0; i < arguments.length; i++) {
   if (arguments[i] > max) {
   max = arguments[i];
   }
 }
 return max;
}
var max1 = findMax(1, 123, 500, 115, 66, 88);
var max2 = findMax(3, 6, 8);
```

## Rest parameters (ES6)

rest parameters are only the ones that haven't been given a separate name, while the arguments object contains all arguments passed to the function

```
function sum(x, y, ...more) {
 //"more" is array of all extra passed params
 let total = x + y;
 if (more.length > 0) {
   for (let i = 0; i < more.length; i++) {
   total += more[i];
   }
 }
 console.log("Total: " + total);
 return total;
}
sum(5, 5, 5);
sum(6, 6, 6, 6, 6);
```

## Spread operator (ES6)

The same … notation can be used to unpack iterable elements (array, string,
object) rather than pack extra arguments into a function parameter.

```
var a, b, c, d, e;
a = [1,2,3];
b = "dog";
c = [42, "cat"];

// Using the concat method.
d = a.concat(b, c); // [1, 2, 3, “dog”, 42, “cat”]

// Using the spread operator.
e = [...a, b, ...c]; // [1, 2, 3, “dog”, 42, “cat”]
```

## Arrow functions (ES6)

- Arrow functions can be a shorthand for an anonymous function in callbacks

```
(param1, param2, …, paramN) => { statements }
(param1, param2, …, paramN) => expression
          // equivalent to: => { return expression; }

// Parentheses are optional when there's only one
  parameter:
(singleParam) => { statements }
singleParam => { statements }

// A function with no parameters requires parentheses:
() => { statements }
```

## Arrow Functions Example

```
function multiply(num1, num2) {
 return num1 * num2;
}
var output = multiply(5, 5);
console.log("output: " + output);

var multiply2 = (num1, num2) => num1 * num2;
var output2 = multiply2(5, 5);
console.log("output2: " + output2);
```

## Arrow Functions Example – filter

Returns an Array containing all the array elements that pass the test. If no elements pass the test it returns an empty array.

```
const a = [
 "Hydrogen",
 "Helium",
 "Lithium",
 "Beryllium"
];
const a2 = a.filter(function(s) {
            return s.length > 7 });
const a3 = a.filter( s => s.length > 7 );

const a4 = a.find( s => s.length > 7 );
const a5 = a.findIndex( s => s.length > 7 );
```

## Arrow Functions Example - map

```
var a = [
 "Hydrogen",
 "Helium",
 "Lithium",
 "Beryllium"
];
var a2 = a.map(function(s) { return s.length });
console.log("a2: " + a2);

var a3 = a.map(s => s.length);
console.log("a3: " + a3);
```

## Arrow Functions Example - reduce

- executes a **reducer** function (that you provide) on each member of the array resulting in a single output value.
- returned value is assigned to the accumulator, whose value is remembered across each iteration throughout the array and ultimately becomes the final, single resulting value.
- initialValue:  optional value to use as the first argument to the first call of the callback. If no initial value is supplied, the first element in the array will be used.

```
function calc(multiplier, base, ...numbers) {
 var temp = numbers.reduce((accum, num) => accum + num, base);
 return multiplier * temp;
}
var total = calc(2, 6, 10, 9, 8);
console.log("total: " + total);
```

## ‘for in’ over object literal/Arrays –ES6

```
//for in over Object
//returns property keys (index)
   of object in each iteration –
   arbitrary order

var things = {
 'a': 97,
 'b': 98,
 'c': 99
};

for (const key in things) {
 console.log(key + ', ' +
   things[key]);
}

a, 97
b, 98
c, 99
```

```
// for in over Arrays
//should not be used to iterate over an
   Array where the index order is important

var things = [97,98,99];

for (const key in things) {
 console.log(key + ', ' +
   things[key]);
}

//0, 97
//1, 98
//2, 99
```

## ‘for of’ vs ‘for in’ –ES6

- Both for..of and for..in statements iterate over arrays;
- for..in returns keys and works on objects as well as arrays
- for..of returns values of arrays but does not work with object properties

```
let letters = ['x', 'y', 'z'];

for (let i in letters) {
    console.log(i); // "0", "1", "2"
}

for (let i of letters) {
    console.log(i); // “x", “y", “z"
}
```

## Summary ‘for’ loops

- ‘for’ is the basic for loop in JavaScript for looping
  - Almost exactly like Java for loop
  - If not sure what need, use this
- ‘for in’ is useful for iterating through the properties of objects, and can also be used to go through the indices of an array
- ‘for of’ is a new convenience (ES6) method for looping through values of ‘iterable’ collections (e.g., Array, Map, Set, String )
- ‘forEach’ is a another convenience method that executes a provided function once for each Array element.
  - forEach returns undefined rather than a new array
  - Intended use is for side effects, e.g., writing to output, etc.
- Best practice to use convenience methods when possible
  - Avoids bugs associated with indices at end points
  - map, filter, find, reduce best practice when appropriate

## Main Point

Functional programming methods map, filter, reduce make code more understandable and error free by automating details of general-purpose looping mechanisms, indicating their intent by their name, and not changing the state of the original array.

**Science of Consciousness:** Growth of consciousness makes behavior simpler and more error free because intentions that arise from deep levels are spontaneously in accord with and supported by all the laws of nature.

## CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

Life Is Found in Layers

1. JavaScript is a functional OO language that has a shared global namespace for each page and local scope within functions.
2. Closures and objects are fundamental to JavaScript best coding practices, particularly for promoting encapsulation, layering, and abstractions in code.

---

3. **Transcendental consciousness** is the experience of the most fundamental layer of all existence, pure consciousness, the experience of one's own Self.

4. **Impulses within the transcendental field:** The many layers of abstraction required for sophisticated JavaScript implementations will be most successful if they arise from a solid basis of thought that is supported by all the laws of nature.

5. **Wholeness moving within itself:** In unity consciousness, one appreciates that all complex systems are ultimately compositions of pure consciousness, one's own Self.

## References

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Closures