SCOPE, CLOSURES AND ENCAPSULATION IN JAVASCRIPT

Except where otherwise noted, the contents of this document are Copyright 2012 Marty Stepp, Jessica Miller, Victoria Kirst and Roy McElmurry IV. All rights reserved. Any redistribution, reproduction, transmission, or storage of part or all of the contents in any form is prohibited without the author's expressed written permission. Slides have been modified for Maharishi University of Management Computer Science course CS472 in accordance with instructors agreement with authors.

Maharishi University of Management - Fairfield, Iowa © 2016

All rights reserved. No part of this slide presentation may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying or recording, or by any information storage and retrieval system, without permission in writing from Maharishi University of Management.

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: Local and global scopes are similar to fine and broad awareness. 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

```
function foo() {
 x = 4;
 print(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
  - Any typo becomes an undetected ‘bug’ !!
  - 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. It searches through the dynamic stack of function calls for a variable declaration.
  - Perl, Bash shell, Common Lisp (optionally), APL, Snobol
  - See slide 24 (Scope Example 1)

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 (were) only two scopes:
  - global scope: global environment for functions, vars, etc.
  - function scope: every function gets its own inner 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();
```

Global Scope

Function Scope

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 OUTside block: " + 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

Best Practice: const, let, var, looping

"I personally recommend you always use const, as it leads to fewer bugs. I have yet to encounter a situation where I needed to use var. As a general rule, use let only for loop counters or only if you really need reassignment. Everywhere else, use const.

Personally, I've ditched loops for filter(), map() & reduce(). You should too."

(freecodecamp.org blog)

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: Local and global scopes are similar to fine and broad awareness. 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 code executes, JS engine first creates global environment objects and "`this`" object and looks for function and variable declarations
- In **first phase**, JS engine looks through all global code for functions and global variables (hoisting)
  - functions: saves entire function definition
  - variables: saves only variable name and value of 'undefined'
  - ➢ 'hoists' variable and function declarations
  - No variable initialization or function expressions are hoisted
- In **second phase**, JS engine
  - executes code line-by-line
  - for every function call adds new context and to execution context stack

Variable Declarations Are Hoisted

- JavaScript hoists 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
  - allows calling a function before it is declared.

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

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

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

Function expressions are not hoisted

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

```
foo(); //TypeError: undefined is not a function

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 move lines:

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

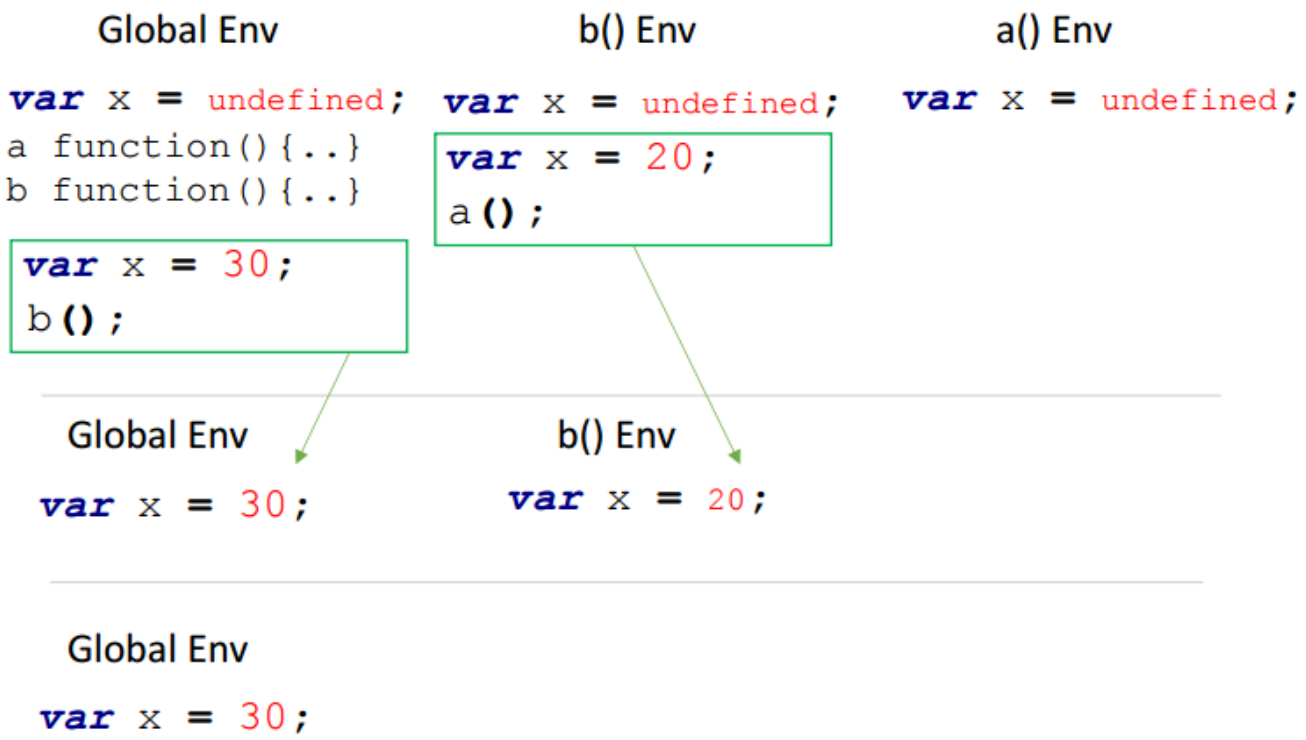
- What will happen if remove the variable `a` declaration line?

Execution context & stack example

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

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

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



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.

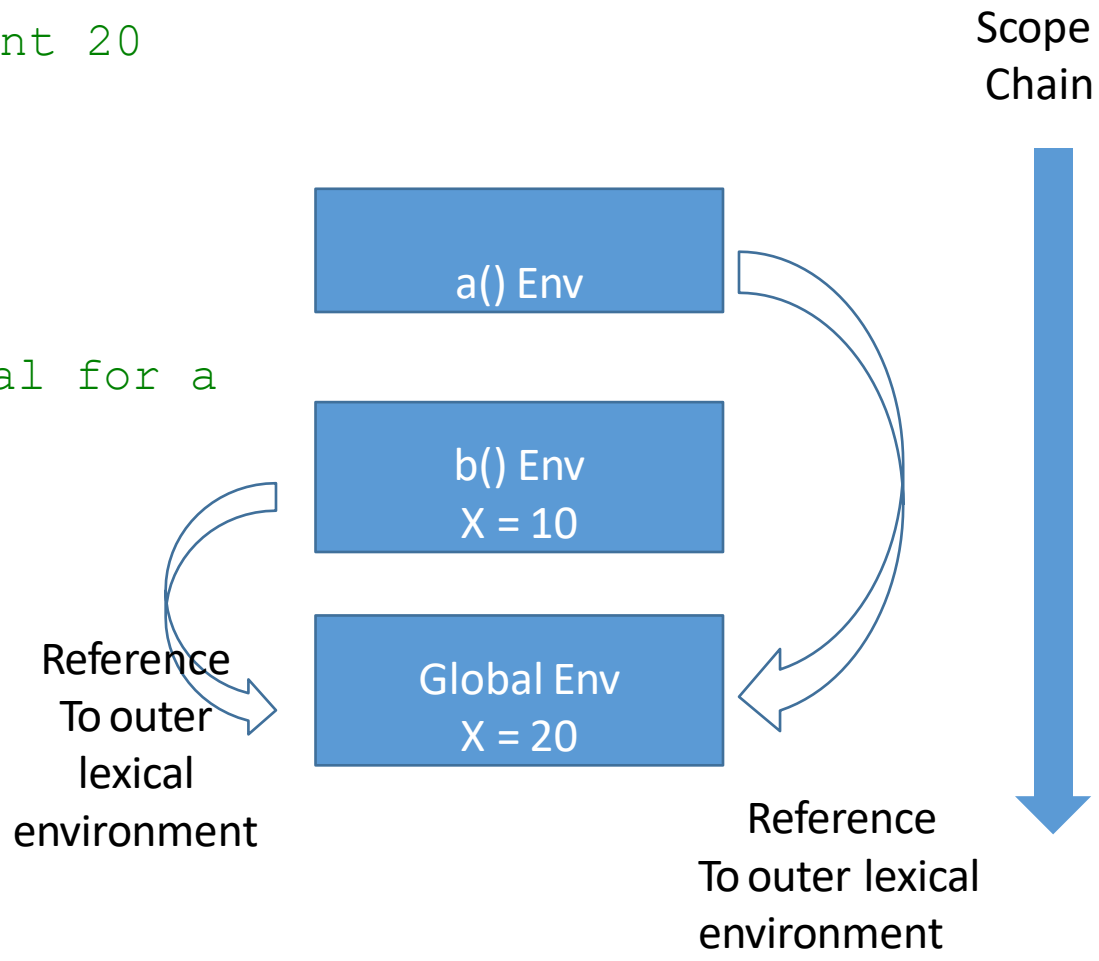
Science of Consciousness: **T**he scope chain proceeds from local awareness to more global awareness. During the process of transcending we naturally proceed from local awareness to more subtle levels of awareness to unbounded awareness.

Scope Chain

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

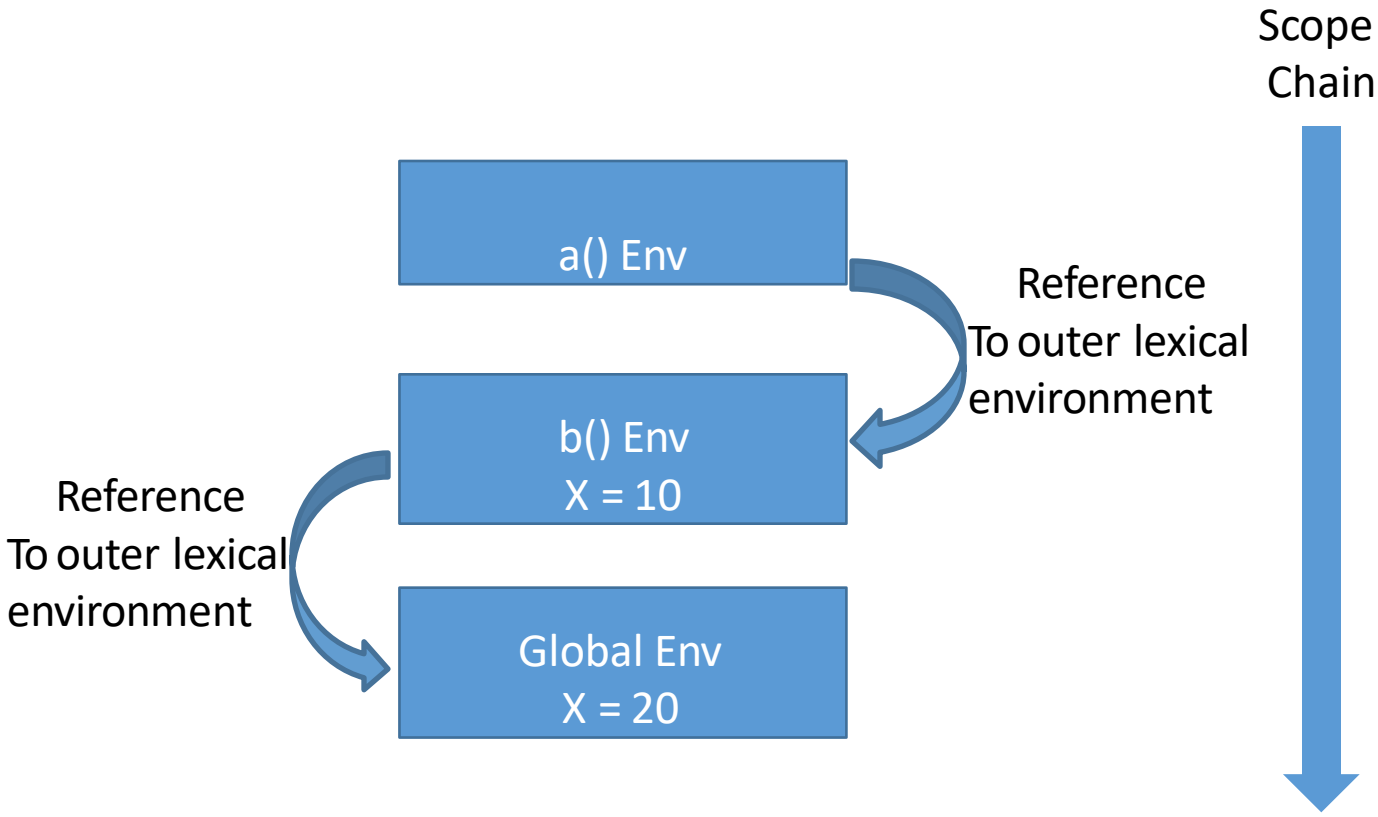
Scope Example1

```
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();
```



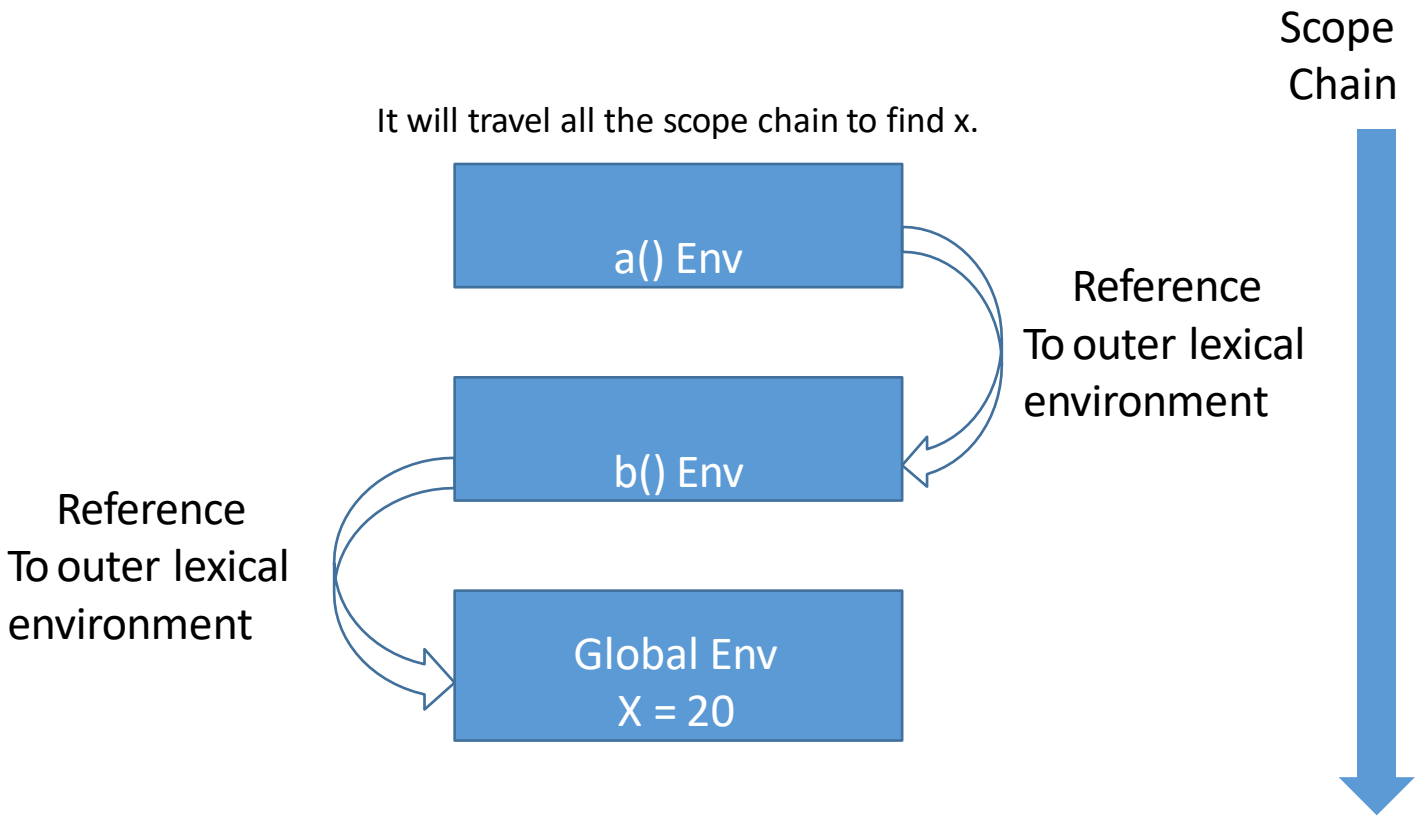
Scope Example 2: Inner function

```
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();
```



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 Example 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: The scope chain proceeds from local awareness to more global awareness. 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 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.

Calling an inner function

```
function init() { //function declaration
 const name = "Mozilla";
 function displayName() {
      console.log(name);
 }
 displayName();
}
init();
```

- like in Java, nested functions have access to variables declared in their outer scope

Returning an inner function

```
function makeFunc() {
 const name = "Mozilla"; //local to makeFunc
 function displayName() {
 console.log(name);
 }
 return displayName;
}
const myFunc = makeFunc();
myFunc();
```

- Q:is the local variable still accessible by myFunc?
- A: yes. Example of saving local state inside a JavaScript closure.

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 is defined when a(n inner) function is defined that has free variables
  - a closure instance is created when the inner function is returned or assigned to a variable 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;
} //end of f

const g = f();
g();
```

- inner function closes over free variables when it is returned
  - Saves 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?

```
/* return a function with no parameters
  that has an 'embedded parameter' */
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)

//buggy version with var

```
var funcs = [];
for (var i = 0; i < 5; i++) {
 funcs[i] = function() {
   return i;
 };
}
```

```
//ES6 solution:  let vs var
const 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, use a closure anywhere you might use an object with a single method.
  - objects have properties to capture state info
  - JavaScript closures capture state info by saving references to free variables
- Situations like this are common on the web.
  - an event handler is a single function executed in response to an event.
    - e.g., DOM and timer event handlers
      - .. in 30 seconds print out whatever is in the currentQuestion variable
    - E.g., factory function that sets state information in reusable code (next slide)
  - closures for encapsulation and namespace protection (module pattern)
- Event handlers must be functions without parameters
  - If you need to pass parameter information with an event handler
    - callback with no parameters but include free variables from the lexical environment.
    - JavaScript engine will create closure over bound variables when assign callback to event handler.

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);

//what is the free variable?
//why is the closure necessary?
```

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 associated with different buttons
  - normally could make this a parameter,
  - but must 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 common way to do it in JS is to use a closure because it creates a function and can save enclosing state info in the ‘free’ variables

Main Point

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.

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 provide a lexical scoping mechanism for JavaScript inner functions. Let and const provide this for blocks of ES6 code. These mechanisms promote encapsulation, layering, and abstraction 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