Lecture 5: JavaScript for Modern Web Apps

CS472 Web Programming

Maharishi University of Management

Department of Computer Science

Assistant Professor Asaad Saad

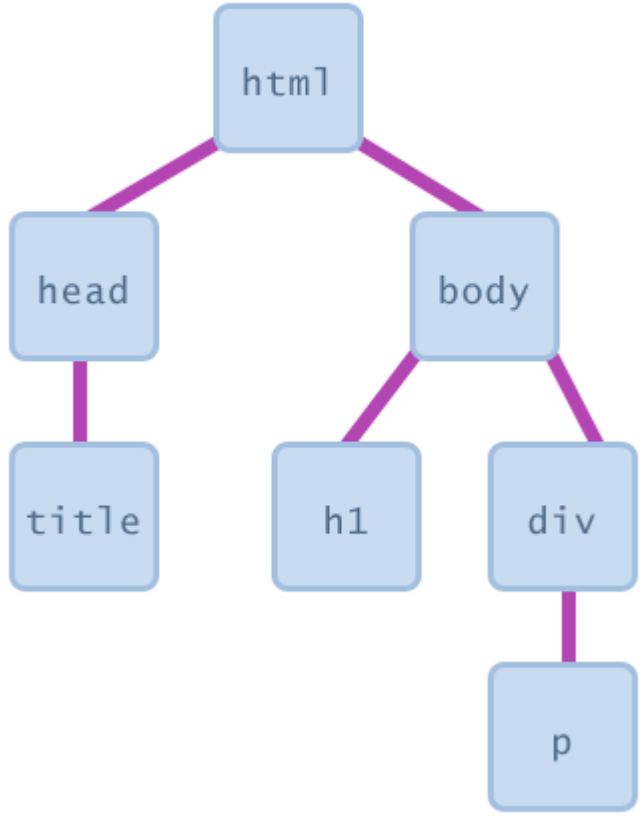
Maharishi University of Management - Fairfield, Iowa © 2016

Main Point

JavaScript programs have no main. They respond to user actions called events. Science of Consciousness: We respond most effectively to events in our environment if our awareness is settled and alert

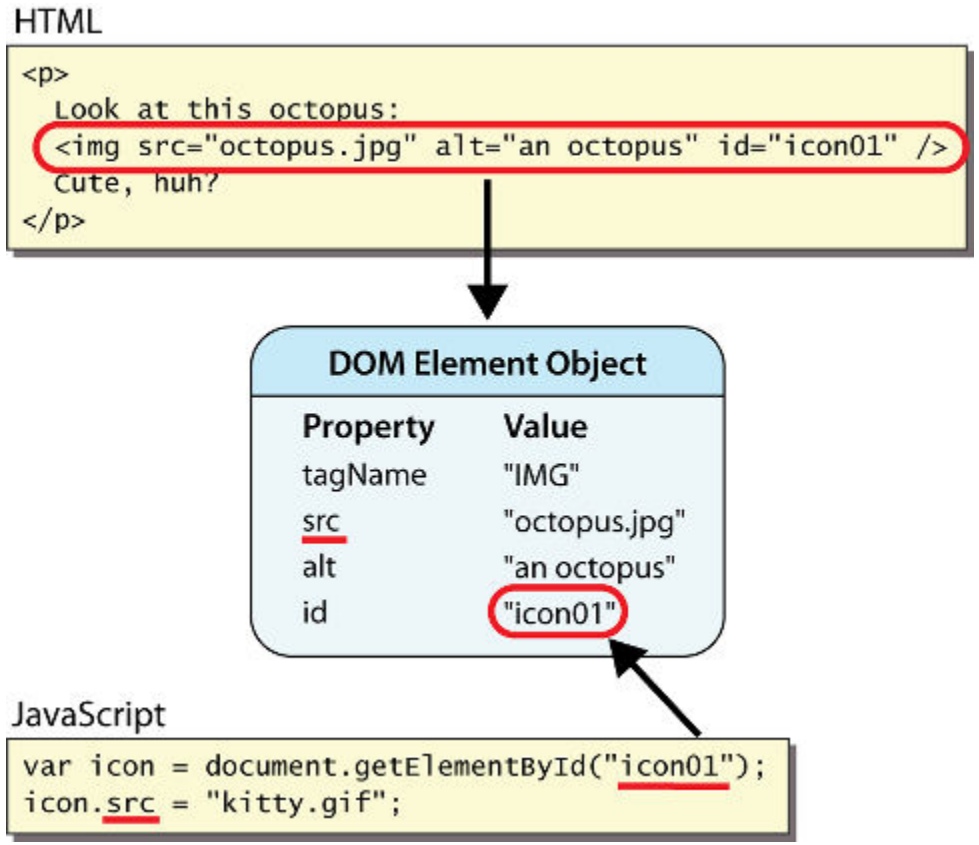
Document Object Model (DOM)

- All HTML elements are represented in browsers as objects
- All objects are nested together in one tree (DOM tree)
- Elements can have parents, siblings and children
- Most JS code manipulates elements (objects) on the DOM
  - we can examine elements' state (see whether a box is checked)
  - we can change state (insert some new text into a div)
  - we can change styles (make a paragraph red)



DOM element objects

- Every element on the page has a corresponding DOM object
- Access/modify the attributes of the DOM object with **`objectName.attributeName`**



JavaScript Simple Object

A collection of name/value pairs

```
var address = {
        street: 'Main Street',
        'number': 1000,
        apartment: {
                'floor': 3,
                'number': 301
        }
    }
```

Notice the object literal above, some names are with quotations!
To access an object property: `address.street` or `address['street']`

Lexical Environment and Execution Context

- In JavaScript, the lexical environment is where the code is sitting physically. Your code is going to be executed based on where it’s lexically located.
- Every lexical environment will have its own execution context (wrapper) in which your code will be running.

Global Environment and Global Objects

- The global environment is a wrapper to your code
- Any object or variable sitting in the global environment is accessible everywhere to any part of the code
- JS Engine will create this global objects for us along with "**`this`**"
- Global objects are: **`window`**, **`document`**, **`history`**, **`location`** , **`navigator`**, **`screen`**
- By default JS Engine will create all variables and objects in the **`window`** global object
- All DOM objects will be sitting in **`document`** global object

Global Objects

- **The `window` object** the entire browser window, the top-level object in hierarchy
- **The `document` object** the current web page and the DOM elements inside it
- **The `location` object** the URL of the current web page
- **The `navigator` object** information about the web browser application
- **The `screen` object** information about the client's display screen
- **The `history` object** the list of sites the browser has visited in this window

Object	Properties	Methods
window	document, history, **location**, name	**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
document	anchors, **body**, cookie, domain, forms, images, links, referrer, title, **URL**	**getElementById**, **getElementsByName** **getElementsByTagName**, close, open, write, writeln
location	**host**, **hostname**, **href**, pathname, port, protocol, search	assign, reload, replace
screen	availHeight, availWidth, colorDepth, **height**, pixelDepth, **width**	
history	**length**	**back**, **forward**, **go**
navigator	**appName**, appVersion, browserLanguage, cookieEnabled, platform, userAgent	

Accessing the DOM

`document.getElementById` returns the DOM object for an element with a given id

```
var textbox = document.getElementById("MyText");
textbox.value = "Hello!";
textbox.disabled ="disabled";

var checkbox = document.getElementById("MyCheckBox");
if (checkbox.checked) { … }


var span = document.getElementById("output2");
span.innerHTML = "Hello!";
span.className = "highlight";
span.style.fontFamily = "Tahoma";  // Style properties in camelCasedNames


var img = document.getElementById("logo");
img.src = "cat.png";
```

Common DOM styling errors

Many students forget to write .style when setting styles

```
var clickMe = document.getElementById("clickme");
clickMe.color = "red";
clickMe.style.color = "red";
```

style properties are capitalized as camelCasedNames

```
clickMe.style.font-size = "14pt";
clickMe.style.fontSize = "14pt";
```

style properties must be set as strings, often with units at the end

```
clickMe.style.width = 200;
clickMe.style.width = "200px";
clickMe.style.padding = "0.5em";
```

The below example computes "200px" + 100 + "px" which would evaluate to "200px100px"

```
var top = document.getElementById("main").style.top;
top  = top + 100 + "px";
top = parseInt(top) + 100 + "px";
```

Sometimes we cannot read existing styles. E.g., font sizes are often determined by browser default settings, the solution is:

```
var currentSize = textAreaEl.style.fontSize || 8;
```

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. Science of Consciousness: We must understand the global DOM objects at the same time we develop the details of any web app. The experience of pure consciousness provides global awareness at the same time it enhances our ability to have fine focus and make careful discriminations.

Your first JS file

To get JS Engine starts in your browser, all you need is to add the following code:

```
<script src="script.js" type="text/javascript"></script>
```

- The JS Engine will create all the **global objects** along with "**this**"
- All your code (variables and functions) will be attached to the global object **window**

What will happened if we have two or more scripts included? They all run as one single file!

```
<script src="script1.js" type="text/javascript"></script>
<script src="script2.js" type="text/javascript"></script>
```

Code Execution and Hoisting

- 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 the **first phase**, JS engine will reserve special memory space for functions (as whole), while it reserves memory only to variables names. All variables are initially set to `undefined`. (Hoisting)
- In **second phase**, JS engine will execute your code line-by-line and call all functions and create execution context for every function (scope) in the execution stack.

Hoisting Example

```
var a = 5;
function b(){
    console.log('function is called');
};

console.log (a);   // 5
b();   // function is called
```

Notice what will happen when we switch between the lines:

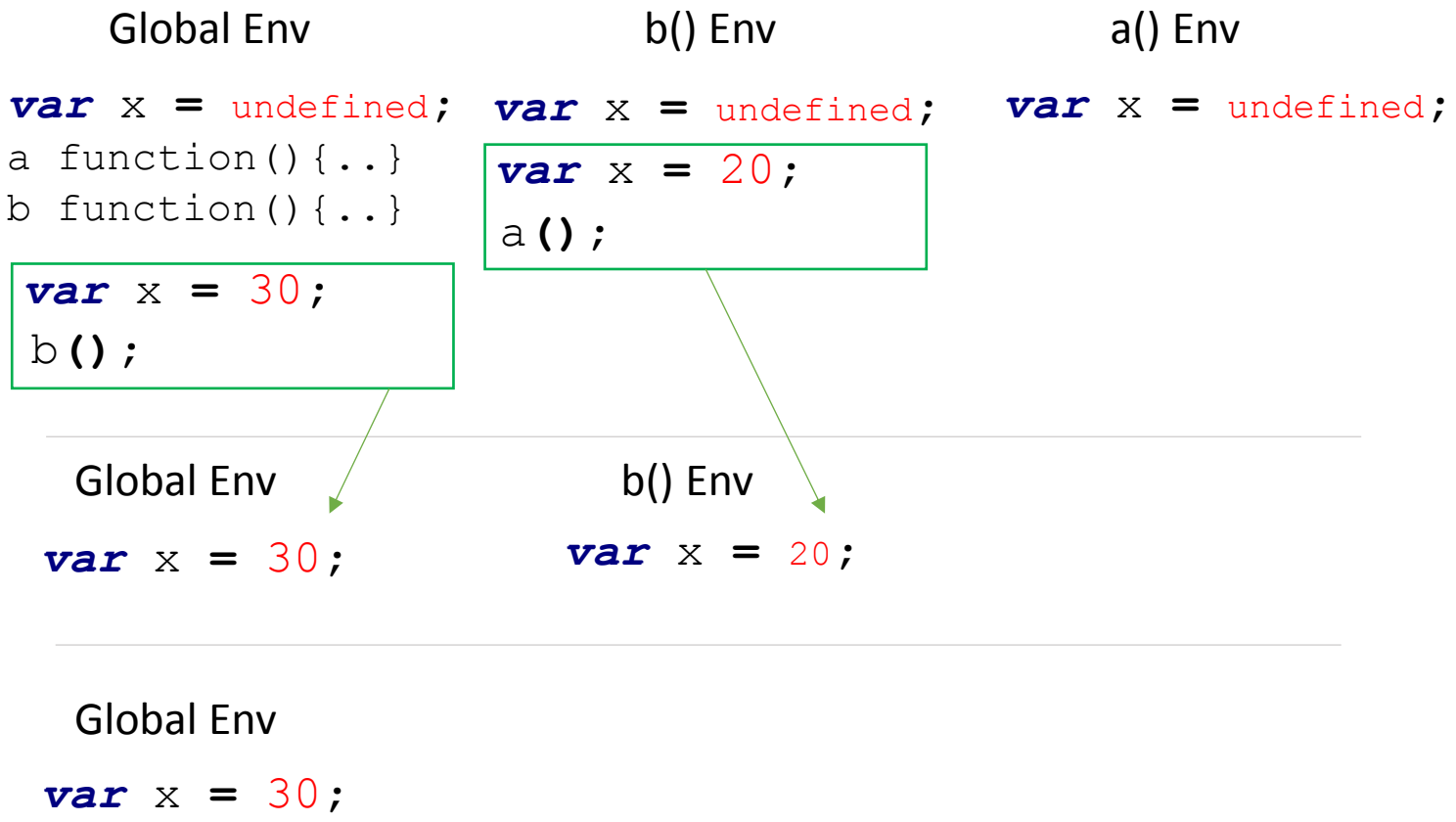
```
console.log (a);   // undefined
b();   // function is called

var a = 5;
function b(){
    console.log('function is called');
};
```

What will happen if I remove the variable **a** definition?

```
// singleline comment
/* multiline comment */
```

Execution context & stack example

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

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

var x = 30;
b();

console.log(x); // 30
```



Implied Globals

If you assign a value to a variable without `var`, JS assumes you want a new global variable with that name (this is bad practice, it's disallowed by jsLint and strict mode)

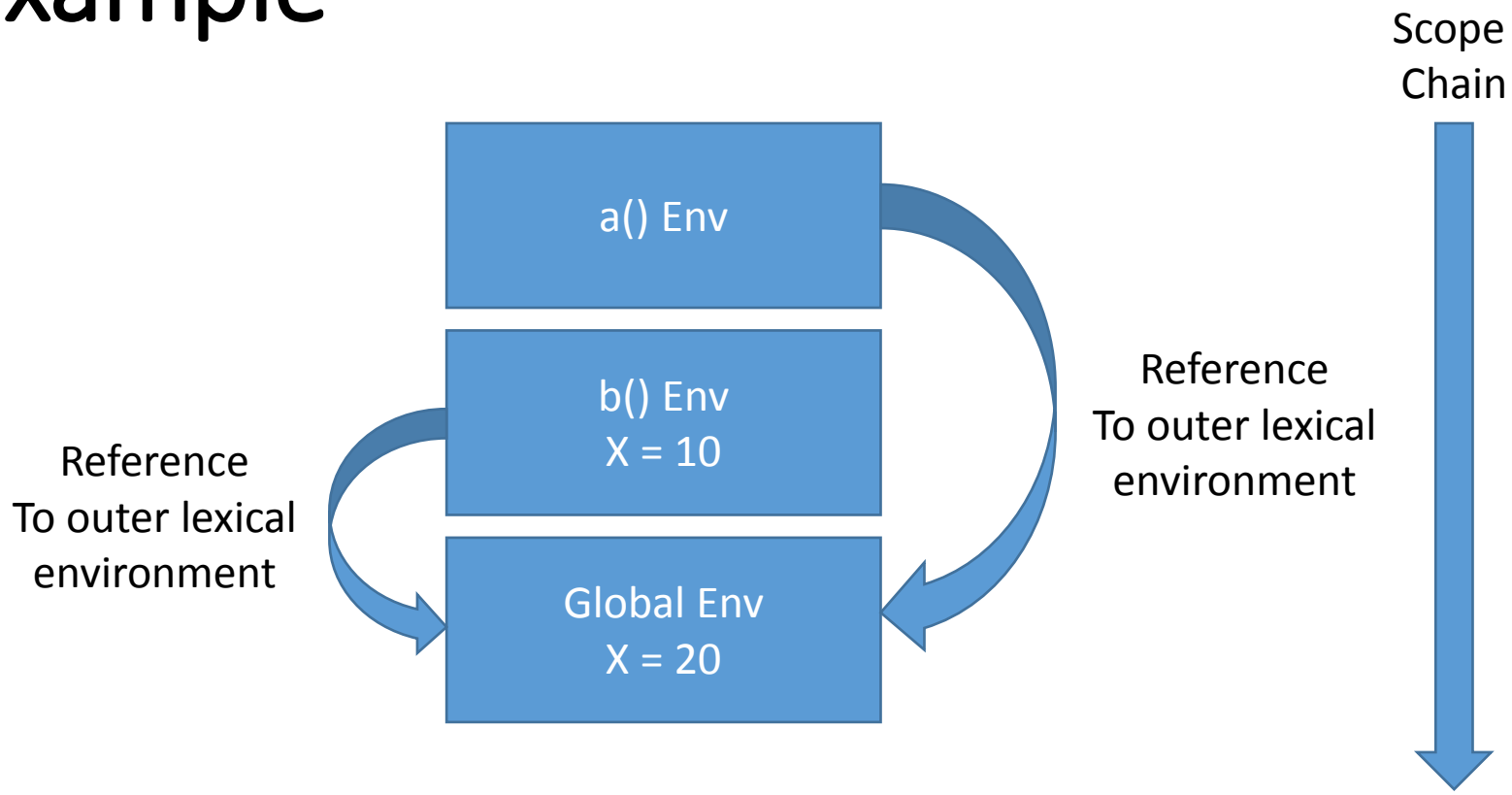
```
function foo() {
        x = 2;
        var y = 3;
        print(x); // 2
        print(y); //3
}
foo();
print(x); // still 2 even outside the loop
print(y); //error
```

Scope and Scope Chain

- Scope: The enclosing context where values and expressions are associated.
- Functions in JS define a new scope
- When we ask for any variable, JS Engine will look for that variable in the current scope, if it doesn’t find it.. Then it will consult its outer scope until we reach the global scope

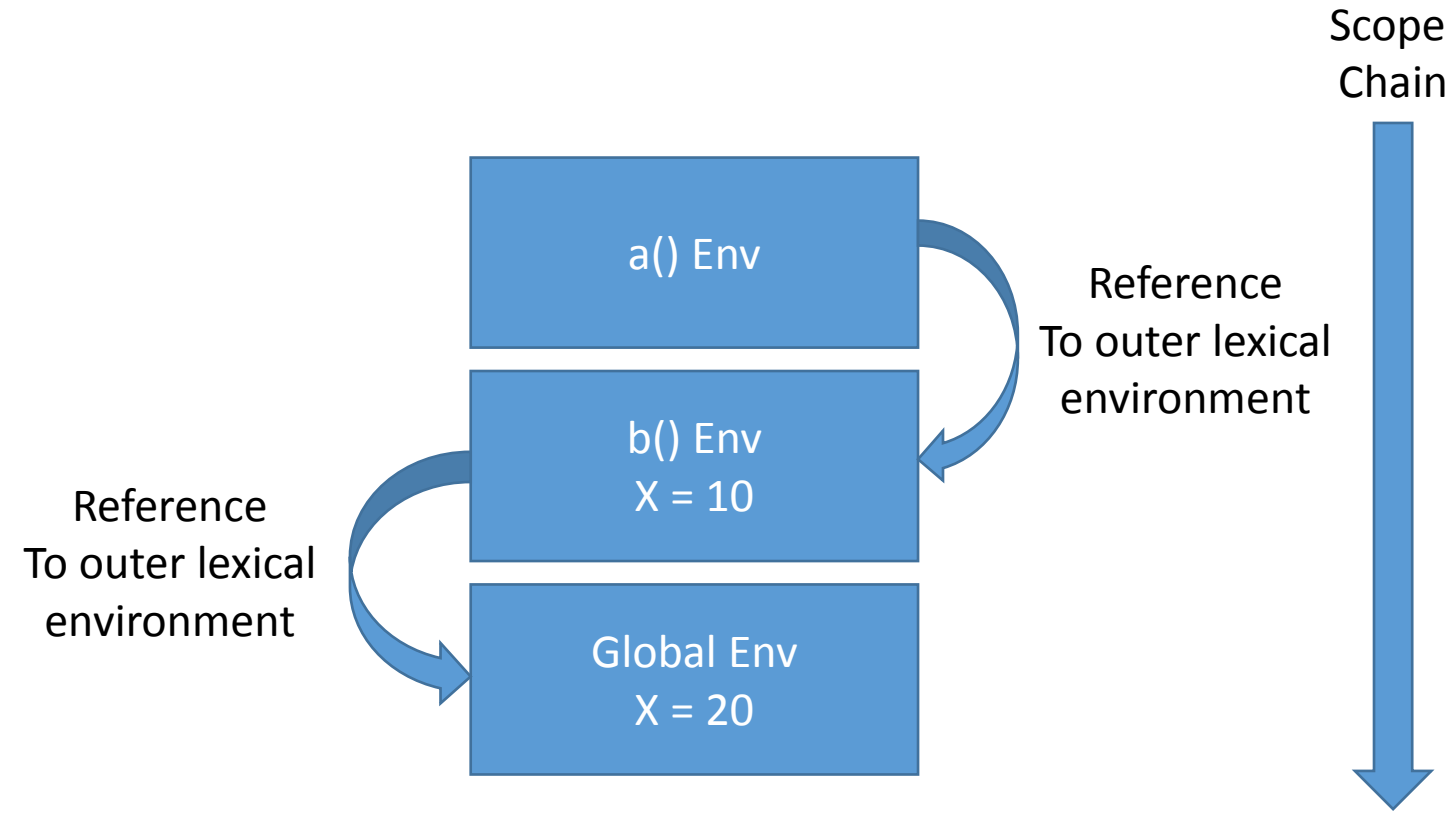
Simple Scope Example

```
function a(){
        console.log(x);
}

function b(){
        var x = 10;
        a();
}

var x = 20;
b(); // 20
```



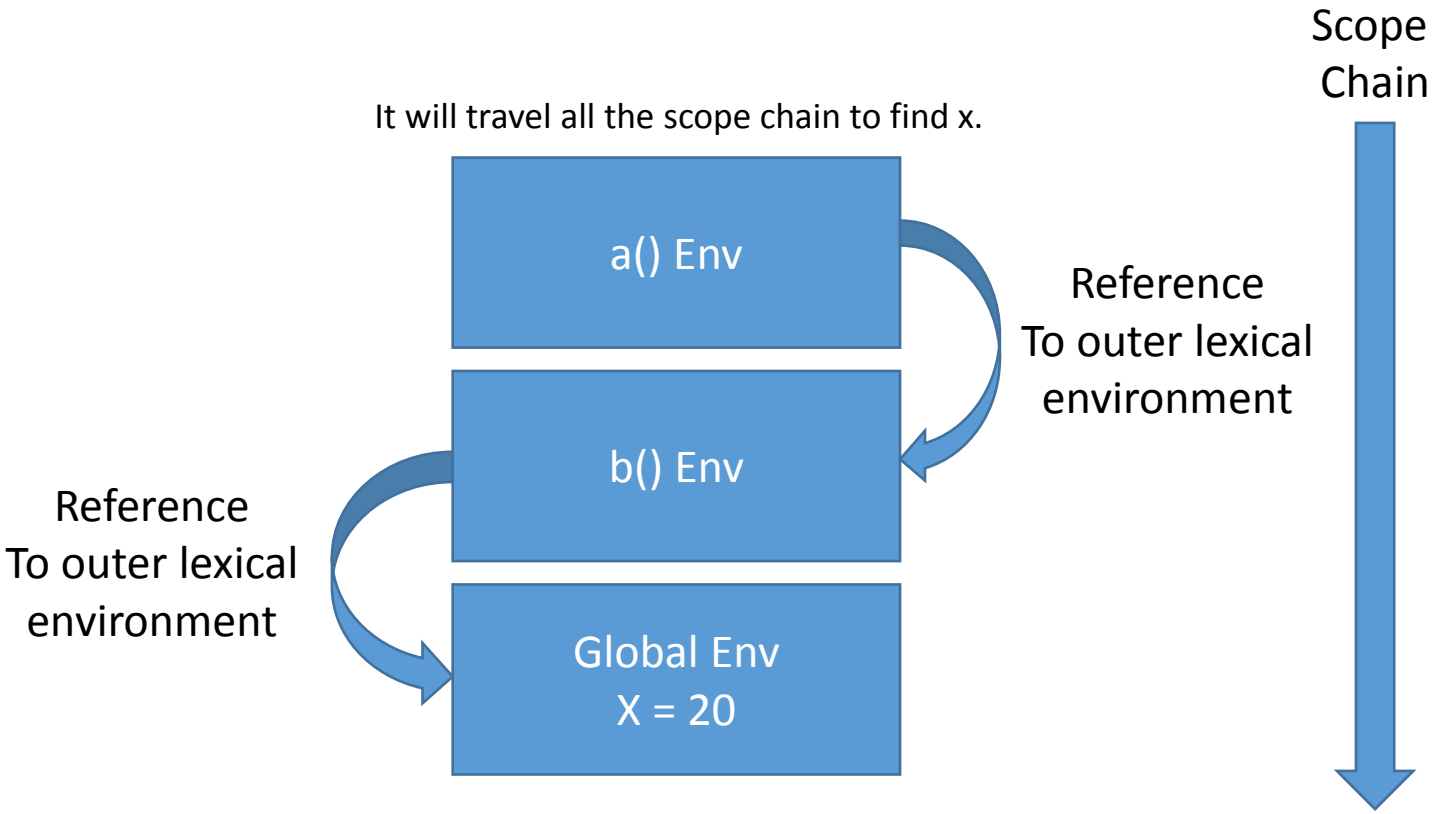
Simple Scope Example

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

var x = 20;
b(); // 10
```



Simple Scope Example

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

var x = 20;
b(); // 20
```



Scope Example

```
function f() {
        var a = 1, b = 20, c;
        console.log(a + " " + b + " " + c); // 1 20 undefined

        function g() {
            var b = 300, c = 4000;
            console.log(a + " " + b + " " + c); // 1 300 4000
            a = a + b + c;
            console.log(a + " " + b + " " + c); // 4301 300 4000
        }

        console.log(a + " " + b + " " + c); // 1 20 undefined
        g();
        console.log(a + " " + b + " " + c); // 4301 20 undefined
}
f();
```

Scope Example

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

`var` vs `let` (ES6)

`var` scope is defined by the nearest function block

`let` scope is defined by the nearest enclosing block

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

```
function a(){
        'use strict';
        for (let x = 1; x < 10; x++){
                        console.log(x);
        }
         console.log(x); // error
}
```

JavaScript `"use strict"` mode

- 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
  - ECMAScript 5 introduced strict mode to JavaScript. The intent is to allow developers to opt-in to a “better” version of JavaScript, where some of the most common and egregious errors are handled differently.
  - Best way to help developers debug is to throw errors when certain patterns occur, rather than silently failing or behaving strangely. Strict mode code throws far more errors, and that’s a good thing, because it quickly calls to attention things that should be fixed immediately.
- "use strict" also works inside of individual functions
- You should always turn on strict mode!

Primitive types

Primitive type: is a type that's represented as a single value (not object)

- **`undefined`**
- **`null`**
- **`boolean`**: true or false
- **`number`**: float
- **`string`** (with single or double quotation)
- **`symbol`** (ES6) immutable type for objects (object wrapper)

Types are dynamic and JS Engine will automatically change between types or wrap a primitive type when needed (coercion)

```
var a = 1 + '2' // '12' concatenation in JS with +
```

JS Wrapper Objects for Primitive Types

The primitive types **`boolean`**, **`string`** and **`number`** can be wrapped by **`Boolean`**, **`String`** and **`Number`** constructors respectively.

You can find out a variable's wrapping type by calling **`typeof`**()

`length` is always a property in JS (not a method as in Java)

```
var s = "Hello"; // string primitive
var firstLetter = s[0]; // JS coerce between primitives and objects
var firstLetter = s.charAt(0);
var lastLetter = s.charAt(s.length - 1);
```

Coercion examples

- Boolean(0) = false
- Number(“3”) = 3
- Number(false) = 0
- Number(true) = 1
- Number(undefined) = NaN
- Number (null) = 0
- 3>2>1 = false // 3>2 = true, true>1 = false
- 3<2<1 = true // 3<2= false, false<1 = true
- 0<1 = true

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence

Falsey and Truthy

Any value can be used as a boolean

- "**falsey**" values: 0, 0.0, NaN, "", null, and undefined
- "**truthy`**" values: anything else

```
var iLikeWebApps = true;
var ieIsGood = "IE6" > 0;   // NaN > 0 false
if ("web dev is great") {  /* true */ }
if (0) {  /* false */ }
```

`null` and `undefined`

`undefined` : has been declared, but no value assigned

- Declare vars without giving them a value
- vars that are "hoisted" to beginning of a function

`null` : var exists, and was specifically assigned an value of null

```
var a = null; // a is null
var b = 9; // b's 9
var c; // c is undefined
```

`const` (ES6)

The **`const`** declaration creates a read-only reference to a value. It does **NOT** mean the value it holds is immutable, just that the variable identifier cannot be reassigned.

```
const MY_NUM = 7;
MY_NUM = 20; // this will fail
console.log(MY_NUM); // will print 7
const MY_NUM = 20; // trying to redeclare a constant throws an error
var MY_NUM = 20; // this will fail

const FOO; // SyntaxError: missing = in const declaration

const MY_OBJECT = {"key": "value"}; // const also works on objects
MY_OBJECT = {"OTHER_KEY": "value"}; // this will fail

// object attributes are not protected, so the following statement is
executed without problems
MY_OBJECT.key = "otherValue"; // will work!
```

JS Syntax - (almost same as Java)

`for` loop , **`if/else`** statement, **`while`** loops

```
for (initialization; condition; update) {
        statements;
}
if (condition) {
        statements;
} else if (condition) {
        statements; }
else {
        statements;
}
while (condition) {
        statements;
}
do { statements; } while (condition);
```

Semi-colon ;

Semi-colons in JS are optional, JS automatically add them to our code

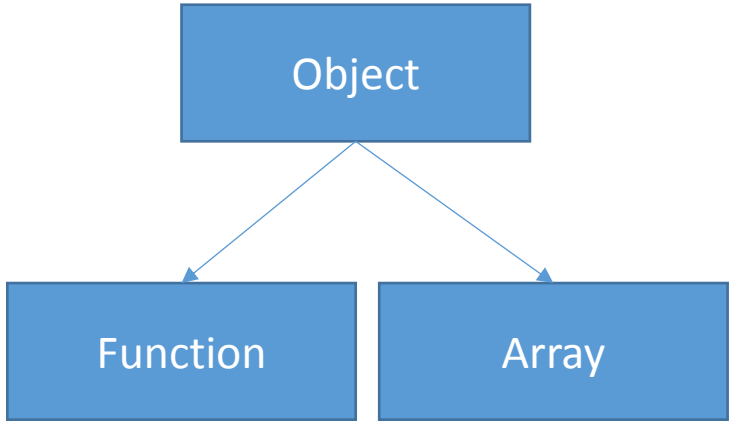
```
function a(){
        return {
                name: 'George';
        }
} // return an object

function a(){
        return
        {
                name: 'George';
        }
}  // return null, why?
```

Main Point

JavaScript is a loosely typed language. It has types, but does no compile time type checking. Programmers must be cautious of automatic type conversions, including conversions to Boolean types. It has a flexible and powerful array type as well as distinct types of null and undefined. **Science of Consciousness:** If our awareness is established in the source of all the laws of nature then our actions will spontaneously be in accord with the laws of nature for a particular environment.

Other JS Objects

- **`Object`, `Array`, `Function`**



There are two ways to initialize an array

```
var name = []; // empty array
var name = new Array(); // empty array
var name = [value, value, ..., value]; // pre-filled
var name = new Array(value, ..., value); // pre-filled
```

Remember **`length`** is a property not a method (it grows as needed when elements are added)

Array methods: `concat, join, pop, push, reverse, shift, slice, sort, splice, toString, unshift`

Arrays

Each element inside an array can be from a different type, it can even contain an object or a function.

An easy way to loop through Arrays or Objects:

```
var names = ['George', 1, {val: 3}, function calc(){}];

names.forEach(function(name){
                    console.log(name);
});
```

Functions

- Functions define a new scope
- Function are objects
- Function are first-class citizens
  - Assign them to variables
  - Pass them around as parameters
  - Create them of the fly
- Function can be anonymous (name property is empty)
- Function are invocable

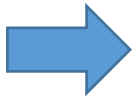
```
function eat(){
        console.log("Yummy!");
}
eat.meal = "pizza";
```

Notice how we added a property to the function

Anonymous functions

- JavaScript allows you to declare anonymous functions
- Can be stored as a variable, attached as an event handler, etc.
- Keeping unnecessary names out of namespace for performance and safety

```
function okayClick() {
        alert("Hi");
}

window.onload = function() {
  var okButton = document.getElementById("ok");
  okButton.onclick = okayClick;
};
```



```
window.onload = function() {
   var okButton = document.getElementById("ok");
   okButton.onclick = function() {
                              alert("Hi");
                              };
};
```

Function Statement & Function Expression

Function Statement

```
function sayHi() {
        console.log('Hi');
}

sayHi();
```

Function Expression

```
var sayHi = function() {
        console.log('Hi');
}

sayHi();
```

Functions are objects, in the example of Function Expression, we are creating an anonymous function object and we have the variable sayHi in memory pointing to it.

What will happen if I try to call the function before its definition in both cases? (Remember hoisting)

Flexibility of Function Expressions

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

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

We created a function named `log` that invokes whatever we pass to it (lines 1, 2, 3).

Upon calling the function `log` we passed to it an anonymous function (lines 4,5,6)

The anonymous function will be invoked (line 2)

Arrow functions (ES6)

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

(<arguments>) => <return statement>

```
function multiply (num1, num2) {
        return num1 * num2;
}

var output = multiply(5, 5);
```

```
var multiply = (num1, num2) => num1 * num2;
var output = multiply(5, 5);
```

By Value vs by Reference

```
// by value (primitives)
var a = 1;
var b;

b = a;
a = 2;

console.log(a); // 2
console.log(b); // 1
```

```
// by reference (all objects)
var a = { fname: 'George' };
var b;

b = a;
a.fname = 'Mike';
console.log(a.fname); // Mike
console.log(b.fname); // Mike

a.fname = 'Asaad';
console.log(a.fname); // Asaad
console.log(b.fname); // Asaad

a = { fname: 'George' };
console.log(a.fname); // George
console.log(b.fname); // Asaad
```

= operator always sets a new memory space

Function Signature

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

```
function a(x) {
        console.log(x);
}

a(); // undefined
a(5); // 5
a(5, 10); // 5
```

arguments Object

JavaScript functions have a built-in object called the **arguments** object. The **argument** object contains an array of the arguments used when the function is called (invoked).

```
function findMax() {
        var i;
        var max = -Infinity;
        for (i = 0; i < arguments.length; i++) {
                if (arguments[i] > max) {
                        max = arguments[i];
                }
        }
        return max;
}

var x = findMax(1, 123, 500, 115, 44, 88); // 500
var x = findMax(5, 32, 24); // 32
```

Spread Operator (ES6)

A **spread** takes all extra parameters and wrap them into an array.

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

sum(4,4); // 8
sum(4,4,4); // 12
```

Spread Advanced Example (ES6)

```
function calc (multiplier, base, ...numbers) {
        var val = numbers.reduce((accumulator, num) => accumulator + num, base);
        return multiplier * val;
}

var total = calc(2, 6, 10, 8, 9);
console.log(total);
```

Note that: arr.reduce(callback[, initialValue])

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(); // 2 Arguments: undefined, undefined
log(5); // 2 Arguments: 5, undefined
log(5, 10); // 2 Arguments: 5, 10
```

Why? Remember functions are objects!

The `window.onload` event

- We want to attach our event handlers right after the page is done loading (Why?)
  - There is a global **event** called `window.onload` event that occurs at that moment

```
// this will run once the page has finished loading
function functionName() {
        element.event = functionName;
        element.event = functionName;
        ...
}

window.onload = functionName; // global code
```

Common unobtrusive JS errors

Many students mistakenly write () when attaching the handler

```
window.onload = pageLoad();
window.onload = pageLoad;

okButton.onclick = okayClick();
okButton.onclick = okayClick;
```

Event names are all lowercase, not capitalized like most variables

```
window.onLoad = pageLoad;
window.onload = pageLoad;
```

Obtrusive & Unobtrusive JavaScript

Obtrusive event handlers - this is bad style (HTML is cluttered with JS code)

```
<button onclick="sayHi();">Say Hi</button>

// called when OK button is clicked
function sayHi() {
        alert("Hi");
}
```

Unobtrusive event handlers - this is better style than attaching them in the HTML

```
<button id="sayHi">Say Hi</button>

var sayHiButton = document.getElementById("sayHi");
sayHiButton.onclick = sayHi; // Assign Function name to the Click Event

function sayHi() {
        alert("Hi");
}
```

Main Points

- “Unobtrusive” JavaScript promotes separation of web page content into 3 different concerns: content (HTML), presentation (CSS), and behavior(JS). Science of Consciousness: This is another example of knower (model), known (view), and process of knowing (controller) which is found throughout computer science and nature. When we experience transcendental consciousness our Self is the knower, and it is also the known, and the experience is itself the process of the knower knowing herself.
- JavaScript code runs whenever appears on the HTML page. Event handlers cannot be assigned until after the target elements are loaded. Event handlers often use anonymous functions because the code is specific to that event and element. Science of Consciousness: Creative intelligence proceeds in an orderly sequential manner. When our thoughts and actions arise from the source of all the laws of nature then we spontaneously act in accord with the natural order of events in nature.

Asynchronous & Callbacks

A callback function is a function you give to another function , to be invoked later when the other function is finished (desired).

Callback functions are queued in the browser **Event Queue**.

Since JS in the browser executes code synchronously, how can we handle asynchronous calls and callbacks?

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

JS Timers

```
setTimeout(function, delayMS); // arranges to call given function after given delay in ms
setInterval(function, delayMS); // arranges to call function repeatedly every delayMS ms
```

Both **setTimeout** and **setInterval** return an ID representing the timer, this ID can be passed to **clearTimeout(timerID)** and **clearInterval(timerID)** to stop the given timer.

Note: If **function** has parameters: **setTimeout(function, delayMS, param1, param2 ..etc);**

```
setTimeout(hideBanner, 5000);

function hideBanner() { // called when the timer goes off
        document.getElementById("banner").style.display = "none";
}
```

Exercise: Alarm clock example.

Common Timer Errors

```
function multiply(a, b) {
        alert(a * b);
}
```

```
setTimeout(hideBanner(), 5000); // what will happen?
setTimeout(hideBanner, 5000);

setTimeout(multiply(num1 * num2), 5000);
setTimeout(multiply, 5000, num1, num2);
```

CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

JavaScript

1. It is a best to understand JS deep functionalities before using it
2. Unobtrusive JavaScript promotes separation of web page content into 3 different concerns: content (HTML), presentation (CSS), and behavior(JS)

---

1. **Transcendental consciousness** is the ultimate in robustness and simplicity, as it is the unchanging silence at the basis of creation.
2. **Impulses within the Transcendental field:** all actions are achieved by the interplay of unity, by understanding the principles by which unity becomes diversity we can gain complete understanding of the relative.
3. **Wholeness moving within itself:** In Unity Consciousness, one experiences that both the unchanging silence of the absolute and the never ending diversity of the relative are the Self.