Lecture 6: JavaScript Programming Environment

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

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 top-level object in hierarchy
- **The `document` object** the root of HTML document and the "owner" of all other nodes.
- **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	

Popup windows with `window.open`

```
window.open("http://foo.com/bar.html", "My Foo Window",
    "width=900,height=600,scrollbars=1");
```

- `window.open` pops up a new browser window
- THIS method is the cause of all the terrible popups on the web!
- some popup blocker software will prevent this method from running

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);
```

Main Point

- JavaScript has a set of global 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*.

Unobtrusive JavaScript

- JavaScript event code seen yesterday was *obtrusive*, in the HTML; this is bad style
- now we'll see how to write *unobtrusive* JavaScript code
    - HTML with minimal JavaScript inside
    - uses the DOM to attach and execute all JavaScript functions
- allows separation of web site into 3 major categories:
    - **content** (HTML) - what is it?
    - **presentation** (CSS) - how does it look?
    - **behavior** (JavaScript) - how does it respond to user interaction?

Obtrusive event handlers(bad)

```
<button onclick="okayClick();">OK</button>

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

- this is bad style (HTML is cluttered with JS code)
- goal: remove all JavaScript code from the HTML body

Unobtrusive JavaScript

```
// where element is a DOM element object
element.onevent = function;

<button id="ok">OK</button>

var okButton = document.getElementById("ok");
okButton.onclick = okayClick;
```

- it is legal to attach event handlers to elements' DOM objects in your JavaScript code
  - notice that you do **not** put parentheses after the function's name
- this is better style than attaching them in the HTML
- Where should we put the above JS code?

JavaScript in a separate file

- JS code can be placed directly in the HTML file's body or head (like CSS)
  - but this is bad style (should separate content, presentation, and behavior)
- script code should be stored in a separate .js file
  - script tag in HTML should be used to link the .js files

```
<script src="filename" type="text/javascript"></script>
```

- When more than one script files are included, interpreter treats them as a single file; in the sense that they share global context.
  - The order in which file files are loaded still matters as interpreter starts executing code as soon as it hits the <script> tag.

When does my code run?

```
<html>
<head>
    <script src="myfile.js" type="text/javascript"></script>
</head>
<body> ... </body> </html>
```

- your file's JS code runs the moment the browser loads the script tag
  - any variables are declared immediately
  - any functions are declared but not called, unless your global code explicitly calls them
- at this point in time, the browser has not yet read your page's body
  - none of the DOM objects for tags on the page have been created yet

A failed attempt at being unobtrusive

```html
<html>
<head>
    <script src="myfile.js" type="text/javascript"></script>
</head>
<body>
<div><button id="ok">OK</button></div>
```

```javascript
// global code
document.getElementById("ok").onclick = okayClick; // null
```

- problem: global JS code runs the moment the script is loaded
- script in head is processed before page's body has loaded
  - no elements are available yet or can be accessed yet via the DOM
- we need a way to attach the handler after the page has loaded...

The `window.onload` event

- We want to attach our event handlers right after the page is done loading
  - There is a global **event** called `window.onload` event that occurs at that moment
- in `window.onload` handler we attach all the other handlers to run when events occur

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

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

An unobtrusive event handler

```html
<button id="ok">OK</button>   <!-- look Me, no JavaScript! -->
```

```javascript
// called when page loads; sets up event handlers
function pageLoad() {
    document.getElementById("ok").onclick = okayClick;
}

function okayClick() {
    alert("booyah");
}

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

Anonymous event handlers

- 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 (and memory management)

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

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

window.onload = attachHandlers;
```

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

Common unobtrusive JS errors

- many students mistakenly write () when attaching the handler

```
window.onload = pageLoad~~()~~;
window.onload = pageLoad;
okButton.onclick = okayClick~~()~~;
okButton.onclick = okayClick;
```

  - IMPORTANT FUNDAMENTAL CONCEPT !!!
    - Function reference versus evaluation

- event names are all lowercase, not capitalized like most variables

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

Ways an external script can be executed

- There are several ways an external script can be executed:
  - If `async="async"`: The script is executed asynchronously with the rest of the page (the script will be executed while the page continues the parsing)
  - If `async` is not present and `defer="defer"`: The script is executed when the page has finished parsing
  - If neither `async` or `defer` is present: The script is fetched and executed immediately, before the browser continues parsing the page

- Changes in HTML 5
  - The "type" attribute is optional
  - The "async" attribute is new in HTML 5

Unobtrusive styling

```
function okayClick() {
 ~~this.style.color = "red";~~
 this.className = "highlighted";
}
.highlighted { color: red; }
```

- Well-written JavaScript code should contain as little CSS as possible
- Use JS to set CSS classes/IDs on elements
- Define the styles of those classes/IDs in your CSS file

Getting/Setting CSS classes

```
function highlightField() {
 // turn text yellow and make it bigger
 if (!document.getElementById("text").className) {
  document.getElementById("text").className = "highlight";
 } else if (document.getElementById("text").className.indexOf("invalid") < 0) {
   document.getElementById("text").className += " highlight";
 }
}
```

- JS DOM's className property corresponds to HTML class attribute
- somewhat clunky when dealing with multiple space-separated classes as one big string
  - which is what getElementById(..).className returns

Common bug: incorrect usage of existing styles

```
~~document.getElementById("main").style.top  =~~
~~document.getElementById("main").style.top + 100 + "px"; ~~// bad
```

- the above example computes e.g. "200px" + 100 + "px" , which would evaluate to "200px100px"

- a corrected version:

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

Main Point

- Unobtrusive JavaScript promotes separation of web page content into 3 different concerns: content (HTML), presentation (CSS), and behavior(JS) (ala MVC, knower, known, process of knowing)

- JavaScript code runs when the page loads it. Event handlers cannot be assigned until the target elements are loaded. In intelligent systems certain events must happen in a particular order. Creative intelligence proceeds in an orderly sequential manner.

Implied globals

- name=value;

```
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
    - hard to distinguish
    - this is a "bad part" of JavaScript (D.Crockford)

JavaScript `"use strict"` mode

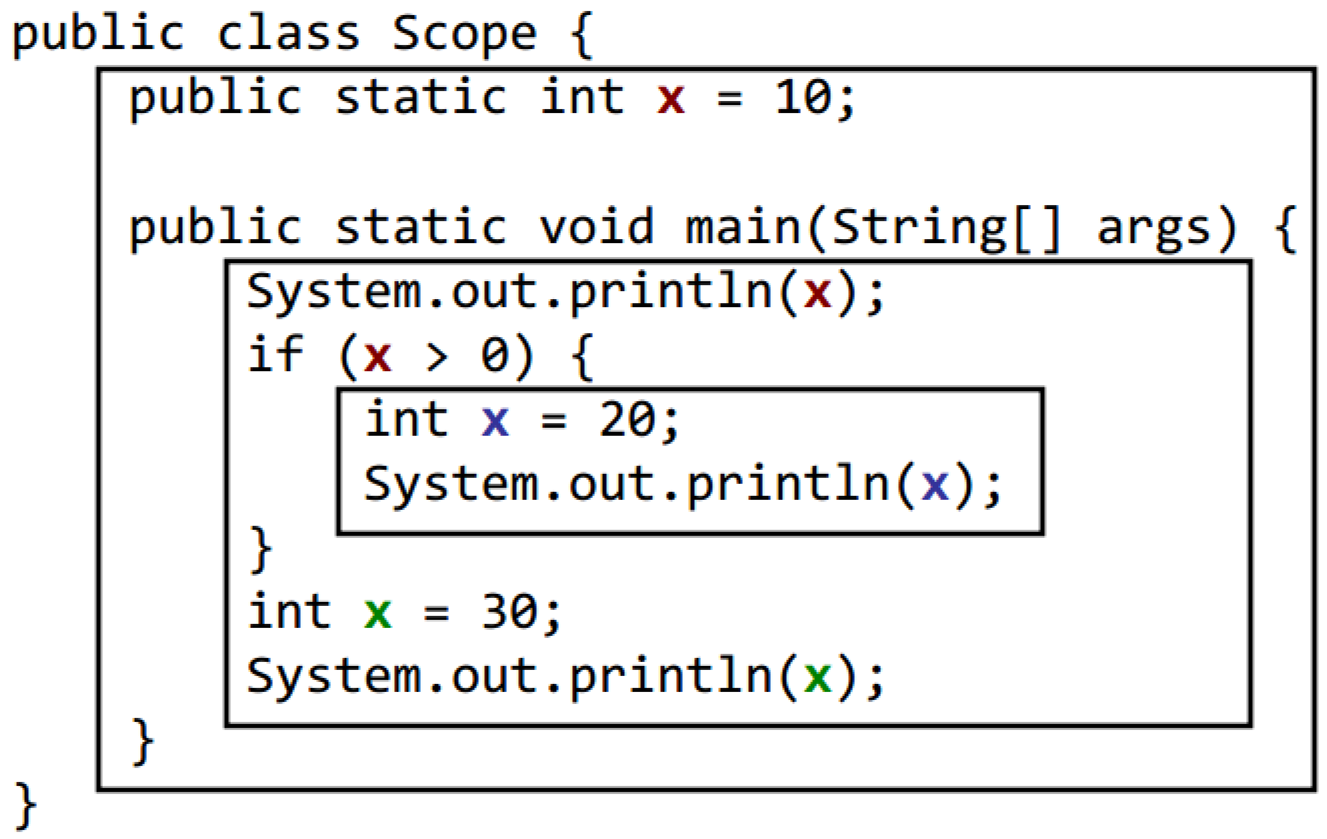
- Writing `"use strict";` at the very top of your JS file (or function) turns on strict syntax checking:
  - Shows an error if you try to assign to an undeclared variable
  - Stops you from overwriting key JS system libraries
  - Forbids some unsafe or error-prone language features
- ECMAScript 5 introduced strict mode to JavaScript. 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. (better than global, why?)
- You should always turn on strict mode!

Lexical Environment and Execution Context

- The lexical environment is where the code is sitting physically.
    - In JavaScript, 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.

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 (outside a function)
  - function scope: every function gets its own inner scope

Global Scope

Function Scope

No Block Scope

```
var x = 10;

function main() {
var x;
 console.log("x1: " + x);
 if (x > 0) {
   var x = 30;
   console.log("x2: " + x);
 }
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 inside a function 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 (and const)

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(){
        for (let x = 1; x < 10; x++){
                console.log(x);
        } // different x every loop
         console.log(x); // error
}
```

- Use `let` instead of `var` inside `for` loops to prevent leaking to the outer scope.
- `let`, unlike `var`, does not create a property on the global object.
- `let`, unlike `var`, does not hoist

```
var x = 'global';
let y = 'global';
console.log(this.x); // "global"
console.log(this.y); // undefined
console.log(window.y); // undefined
console.log(y); // "global"
```

`const` (ES6)

The **`const`** declaration scoped to block, cannot be updated, **NOT** immutable, objects cannot change its structure but we can change the values. To make an object immutable we call: `Object.freeze();`

```
const MY_NUM = 7;
MY_NUM = 20; // this will fail
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!
```

Main Point

- JavaScript code written outside of a function are in global scope. Functions in JS define a new scope (local scope)
- **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.

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 **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’
  - 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.

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?

ⓘ

```
// single line comment
/* multi lines comment */
```

Function expressions are not hoisted

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

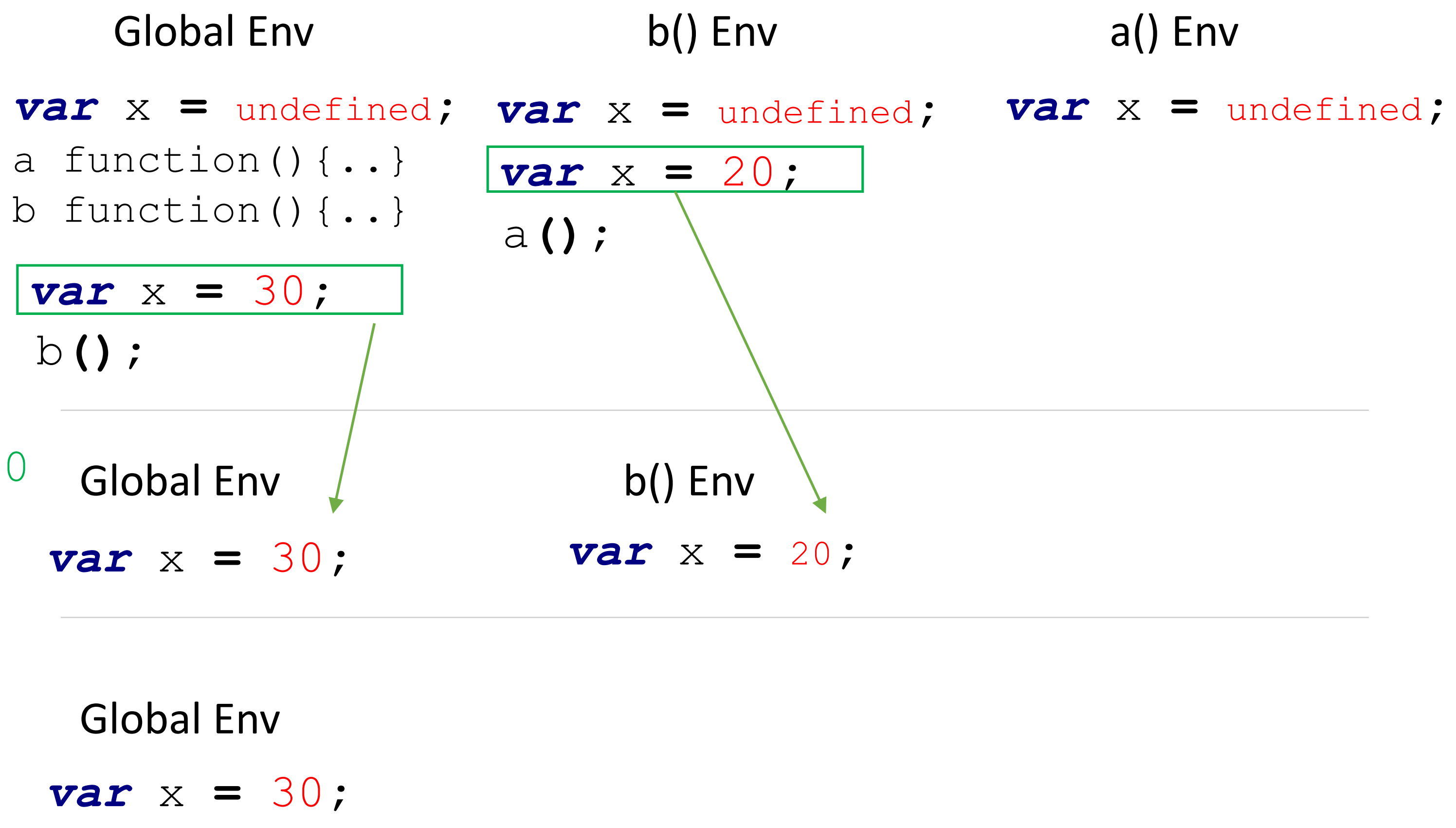
```
foo(); //TypeError: undefined is not a function

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

- JS Engine executes the code as:

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

Execution context & stack example

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

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

var x = 30;
b();

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



```
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;
```

Notes and best practices

- When using a variable defined with **`var`** before the declaration, it's usually hoisted and we usually get no error with value **`undefined`**.
- When using **`let`** or **`const`**, there will be no hoisting and we will receive an **`error`** if used.

> Rule of thumb:
> - Use **`const`** by default
> - Only use **`let`** if you need to update your variable later
> - Don't use **`var`**

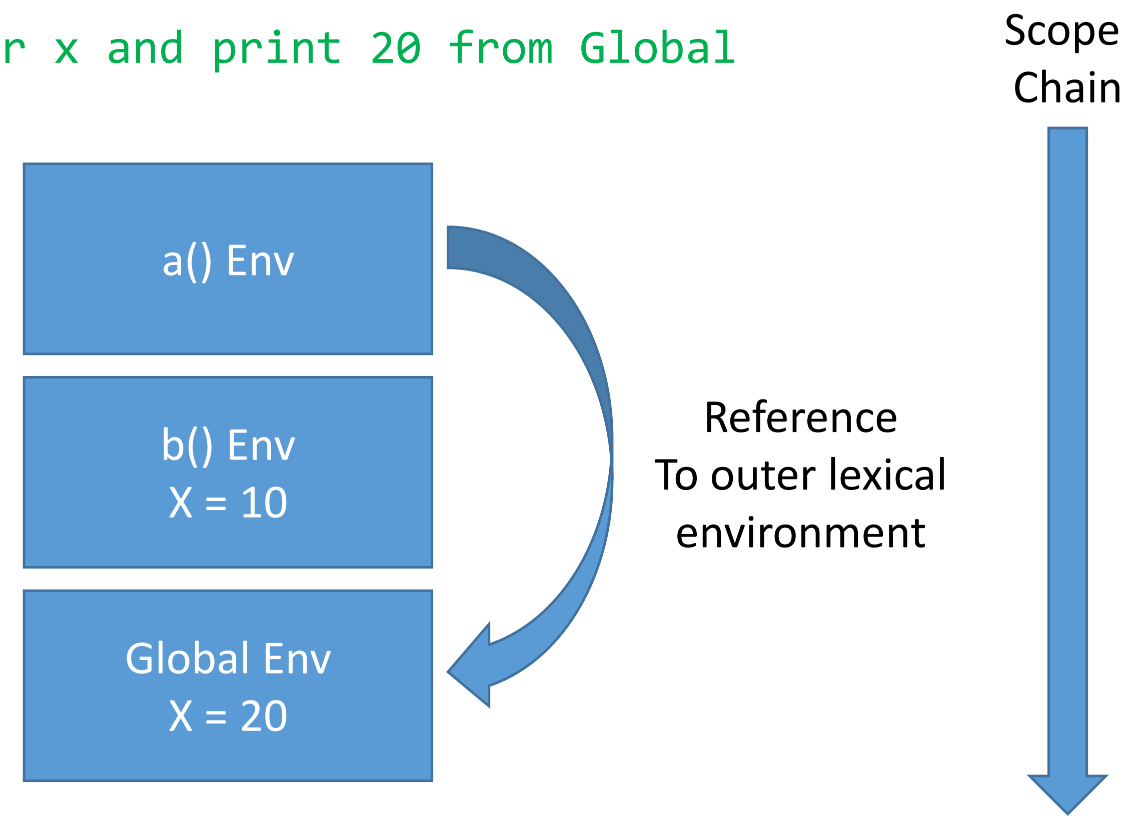
- But, millions of legacy programs use `var` and any competent JS programmer must understand hoisting

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.

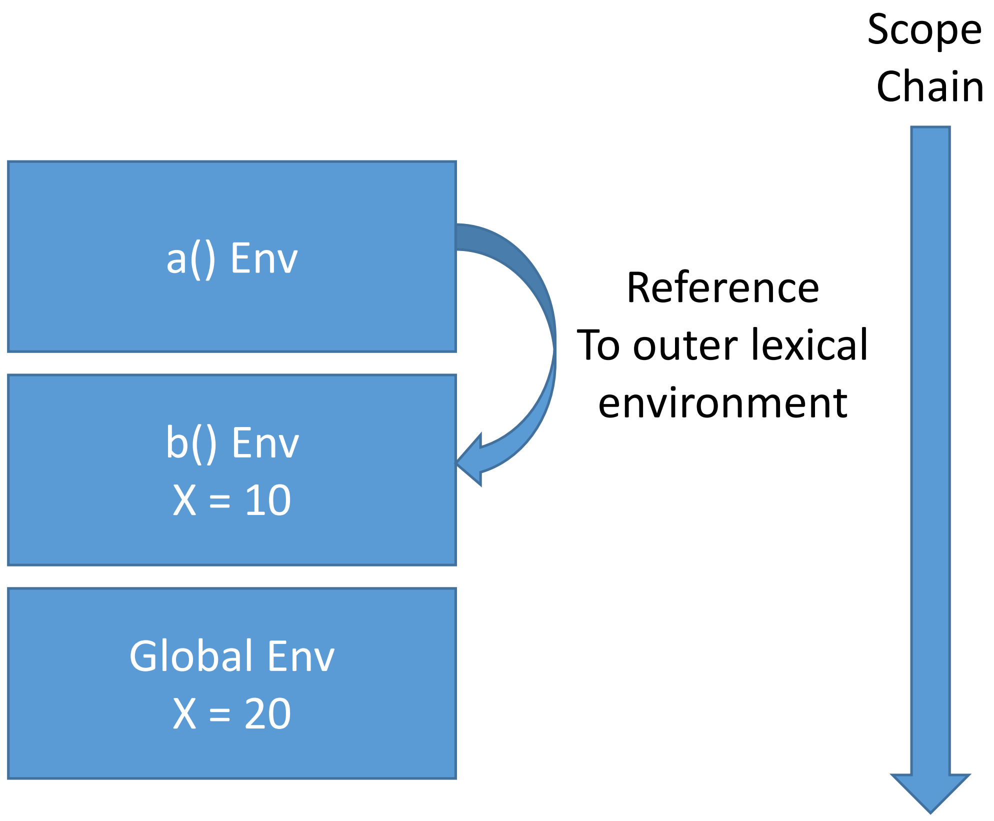
Simple Scope Example

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

function b(){
        var x = 10;
        a(); // consult Global for a
        console.log(x);
}

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



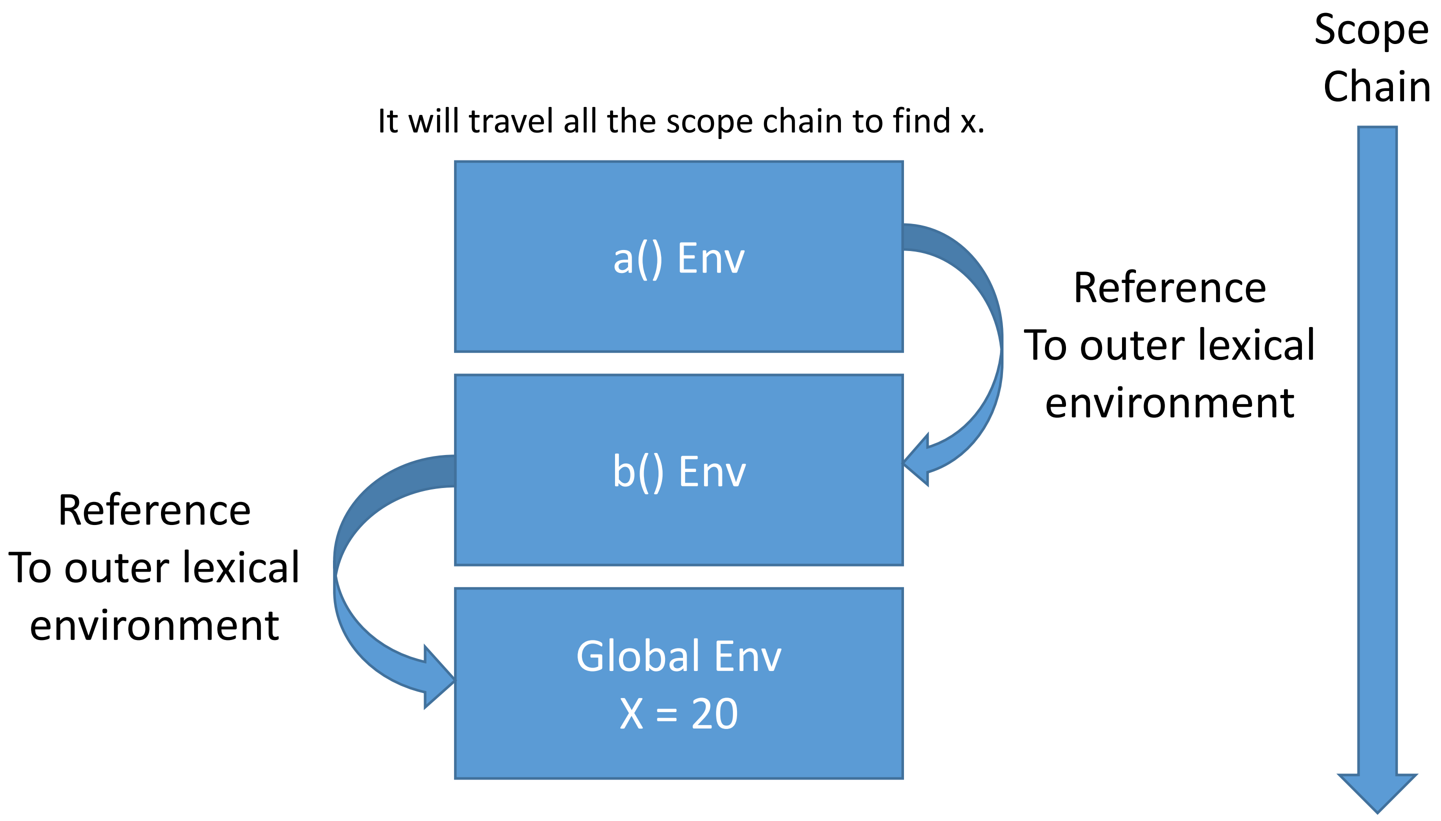
Simple Scope Example

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

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



Simple Scope Example

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

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


Scope Chain
It will travel all the scope chain to find x.
a() Env
Reference To outer lexical environment
b() Env
Reference To outer lexical environment
Global Env
X = 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("x1 is " + x);
       x = 20;
       console.log("x2 is " + x);
       if (x > 0) {
              x = 30; // x=30;
              console.log("x3 is " + x);
       }
       console.log("x4 is " + x);
       var x = 40; // 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.