LECTURE 6: JAVASCRIPT PROGRAMMING ENVIRONMENT

Maharishi University of Management -Fairfield, Iowa © 2016

Main Point Preview

JavaScript has a set of global DOM objects accessible to every web page.

Every JavaScript object runs inside the global window object. The window object has many global functions such as alert and timer methods. At the level of the unified field, an impulse anywhere is an impulse everywhere.

Outline

- Global DOM Objects -- the global context all js programs run in
- Unobtrusive Javascript -- separation of content (HTML) and behavior (Javascript)
- DOM Element Objects -- the main operatives of js programs
- How HTML pages load JavaScript and when event handlers can be assigned
- JS Timers -- needed for today's lab, and common tool for js

The six global DOM objects

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

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

The window object

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

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

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

The document object

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

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

The location object

- *the URL of the current web page*

- properties:
  - host, hostname, href, pathname, port, protocol, search
- methods:
  - assign, reload, replace
- complete list

The navigator object

- *information about the web browser application*

- properties:
  - appName, appVersion, cookieEnabled, platform, userAgent
  - complete list
- Some web programmers examine the navigator object to see what browser is being used, and write browser-specific scripts and hacks:
  - `if (navigator.appName === "Microsoft Internet Explorer") { ...`
  - (this is poor style; you should not need to do this)

The screen object

- *information about the client's display screen*

- properties:
  - availHeight, availWidth, colorDepth, height, pixelDepth, width
  - complete list

The history object

- *the list of sites the browser has visited in this window*

- properties:
  - length
- methods:
  - Moving backward and forward through the user's history is done using the back, forward, go methods.
  - window.history.back();
  - HTML5 history API allows JavaScript to manipulate for SPA
- complete list
- sometimes the browser won't let scripts view history properties, for security

Main Point

JavaScript has a set of global DOM objects accessible to every web page. Every JavaScript object runs inside the global window object. The window object has many global functions such as alert and timer methods. At the level of the unified field, an impulse anywhere is an impulse everywhere.

Main Point Preview

- 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.

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 code?

Linking to a JavaScript file: script

- 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 tag should be placed in HTML page's head
- script code should be stored in a separate .js file (like CSS)

```
<script src="example.js" ></script>
```

When does my code run?

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

// global code
var x = 3;
function f(n) { return n + 1; }
function g(n) { return n - 1; }
x = f(x);
```

- 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

See example: lecture06_examples/runjs.html

A failed attempt at being unobtrusive

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

// code in myfile.js
document.getElementById("ok").onclick = okayClick; // error: cannot set
 property onclick of null
```

- problem: myfile.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
  - See lecture06_examples/failedattempt.html
- we need a way to attach the handler after the page has loaded...

The window.onload event

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

- 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

See example: lecture06_examples/unobtrusivehandler1.html (does it work?)

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;
```

Anonymous functions

```
function(parameters) {
  statements;
}
```

- JavaScript allows you to declare **anonymous functions**
- creates a function without giving it a name
- can be stored as a variable, attached as an event handler, etc.
- Important in JavaScript because of event handling nature and minimizing namespace clutter

Anonymous function example

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

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

or the following is also legal (though harder to read):

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

See example: lecture06_examples/anonymous.html

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.

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
- What about cssZenGarden?

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.

First-class functions

```
//Functions can be assigned to variables
const myfunc = function(a, b) {
 return a * b;
};

//Functions can be passed as parameters
function apply(a, b, myfunc) {
 const y = myfunc(a, b);
 return y;
}
const x = apply(2, 3, myfunc); // 6

//Functions can be return values
function getAlert(str) {
 return function() { alert(str); }
}
const whatsUpAlert= getAlert("What's up!");
whatsUpAlert(); // "What's up!"
```

Arrow functions (ES6)

- Arrow functions(ES6)
  - Syntactic sugar++ for anonymous functions (ala Java lambdas)

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

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



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

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

Arrow Functions Example

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

const multiply = (num1, num2) => num1 * num2;
console.log("output: " + multiply(5,5));
```

Arrow Functions Example – map

```
//creates a new array with the results of calling a provided function on every element in the
  calling array.
const a = [
 "Hydrogen",
 "Helium",
 "Lithium",
 "Beryllium"
];
const a2 = a.map(function(s) { return s.length });
console.log("a2: " + a2);

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

Arrow Functions Example – filter

//filter returns Array containing all elements that pass the test. If no elements pass returns 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 );

//find returns value of first element that satisfies test. Otherwise undefined is returned.
const a4 = a.find( s => s.length > 7 );
const a5 = a.findIndex( s => s.length > 7 );
```

Arrow Functions Example - reduce

- executes a **reducer** function (that you provide) on each member of the array resulting in a single output value.
  - will execute the callback function starting at index 1, using the first value as the initial 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.

```
const array1 = [1, 2, 3, 4];

const reducer = (accumulator, currentValue) => accumulator + currentValue;

console.log(array1.reduce(reducer));// expected output: 10
```

Reduce with initial 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.
- To sum up values contained in an array *of objects* you **must** supply an initial value so that the first item is also processed by your function

```
var initialValue = 0;

var sum = [{x: 1}, {x:2}, {x:3}].reduce(
  function (accumulator, currentValue) {
        return accumulator + currentValue.x;},initialValue);

console.log(sum); // logs 6
```

map/filter/find/reduce are “pure” functions

- **Important principle of “functional” programming**
- **Pure functions have no side effects**
  - **Do not change state information**
  - **Do not modify the input arguments**
- **Take arguments and return a new value**
- **Valuable benefits for automated program verification, parallel programming, reuse, and readable code**

‘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
  - 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 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

Good things to know

- No function overloading in JavaScript
  - Extra arguments ignored and missing ones ignored
- Arguments object and Rest (ES6) parameters
- Spread operator

Function Signature

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

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

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? JavaScript ignores extra arguments and uses undefined for missing arguments. Last declaration overwrites earlier ones.

`arguments` Object

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

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

Exercise: write a function that can be called with any number of arguments and returns the sum of the arguments.

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
- ES6 compatible code, then rest parameters should be preferred.

```
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”]
```

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—i.e., the details are handled automatically like the convenience looping mechanisms.

Timer events

method	description
setTimeout(*function*, *delayMS*);	arranges to call given function after given delay in ms
setInterval(*function*, *delayMS*);	arranges to call function repeatedly every *delayMS* ms
clearTimeout(*timerID*); clearInterval(*timerID*);	stops the given timer so it will not call its function

- both setTimeout and setInterval return an ID representing the timer
  - this ID can be passed to clearTimeout/Interval later to stop the timer

setTimeout Example

```
<button onclick="delayMsg();">Click me!</button>
<span id="output"></span>

function delayMsg() {
 setTimeout(booyah, 5000); document.getElementById("output").innerHTML
 = "Wait for it...";
}
function booyah() {
 // called when the timer goes off
 document.getElementById("output").innerHTML = "BOOYAH!";
}
```

setInterval example

```
timer = null; // stores ID of interval timer
function delayMsg2() {
 if (timer === null) {
      timer = setInterval(rudy, 1000);
 } else {
      clearInterval(timer); // cancel the timer
      timer = null;
 }
}

function rudy() { // called each time the timer goes off
 document.getElementById("output").innerHTML += " Rudy!";
}
```

Passing parameters to timers

```
function delayedMultiply() {
 // 6 and 7 are passed to multiply when timer goes off
 setTimeout(multiply, 2000, 6, 7);
}
function multiply(a, b) {
 alert(a * b);
}
```

- any parameters after the delay are eventually passed to the timer function
- why not just write this? setTimeout(**multiply(6, 7)**, 2000);

Common timer errors

- many students mistakenly write () when passing the function

```
setTimeout(booyah(), 2000);
setTimeout(booyah, 2000);
setTimeout(multiply(num1 * num2), 2000);
setTimeout(multiply, 2000, num1, num2);
```

- what does it actually do if you have the ()? -- e.g., first and 3rd examples above
  a) Does nothing
  b) It is an error to call a function as a parameter
  c) It is an error if the function returns a function
  d) Calls the function after 2 seconds
  e) it calls the function immediately
  f) Calls the function immediately and attempts to call the return value of the function after 2 seconds

IMPORTANT!!!

JavaScript “strict” mode

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

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

- writing "use strict"; at the very top of your JS file turns on strict syntax checking:
  - shows an error if you try to assign to an undeclared variable
  - stops you from overwriting key JS system libraries
  - forbids some unsafe or error-prone language features
- You should always turn on strict mode for your code in this class
  - Important for ES6
- IIFE syntax will be discussed further in next lesson

CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

Actions in Accord with the Laws of Nature

1. Client-side JavaScript is attached to HTML pages and is executed on the browser when the script loads.

2. JavaScript reacts to browser events and manipulates the web page using the HTML DOM API

---

3. **Transcendental consciousness** is the source of thought and the home of all the laws of nature.

4. **Impulses within the transcendental field** spontaneously are in accord with all the laws of nature.

5. **Wholeness moving within itself:** In unity consciousness, one enjoys all perceptions and knowledge in terms of the source of all the laws of nature, pure bliss consciousness. This is parallel to the good feeling and confidence that we experience when we have a deep appreciation and understanding of the environment in which JavaScript runs.