

Comp 324/424 - Client-side Web Design

Spring Semester 2019 - Week 7

Dr Nick Hayward

JS Core - objects - part 2

Arrays

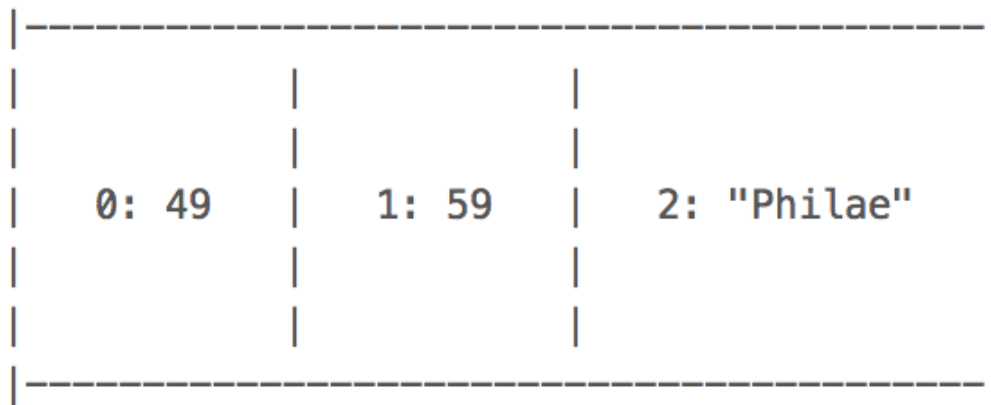
- JS array an object that contains values, of any type, in numerically indexed positions
 - *store a number, a string...*
 - *array will start at index position 0*
 - *increments by 1 for each new value*
- arrays can also have properties
 - *eg: automatically updated **length** property*

```
var arrayA = [  
  49,  
  59,  
  "Philae"  
];  
arrayA.length; //returns 3
```

- each value can be retrieved from its applicable index position,

```
arrayA[2]; //returns the string "Philae"
```

Image - JS Array



JS Array.

JS Core - objects - Arrays

examples

- Random Greeting Generator - Basic

JS Core - checking equality - part I

- JS has four equality operators, including two **not equal**
 - `==`, `===`, `!=`, `!==`
- `==` - checks for value equality, whilst allowing coercion
- `===` - checks for value equality but without coercion

```
var a = 49;  
var b = "49";  
  
console.log(a == b); //returns true  
console.log(a === b); //returns false
```

- first comparison checks values
 - *if necessary, try to coerce one or both values until a match occurs*
 - *allows JS to perform a simple equality check*
 - *results in `true`*
- second check is simpler
 - *coercion is not permitted, and a simple equality check is performed*
 - *results in `false`*

JS Core - checking equality - part 2

- which comparison operator should we use
- useful suggestions for usage of comparison operators
 - *use === if either side of the comparison could be true or false*
 - *use === if either value could be one of the following specific values,*
 - *0, "", []*
 - *otherwise, it's safe to use ==*
 - *simplify code in a JS application due to the implicit coercion.*
- **not equal** counterparts, ! and !== work in a similar manner

JS Core - checking inequality - part I

- known as **relational comparison**, we can use the inequality operators,
 - `<`, `>`, `<=`, `>=`
- inequality operators often used to check comparable values like numbers
 - *inherent ordinal check*
- can be used to compare strings

```
"hello" < "world"
```

- coercion also occurs with inequality operators
 - *no concept of **strict inequality***

```
var a = 49;  
var b = "59";  
var c = "69";  
  
a < b; //returns true  
b < c; //returns true
```

JS Core - checking inequality - part 2

- we can encounter an issue when either value cannot be coerced into a number

```
var a = 49;  
var b = "nice";  
  
a < b; //returns false  
a > b; //returns false  
a == b; //returns false
```

- issue for < and > is string is being coerced into invalid number value, NaN
- == coerces string to NaN and we get comparison between 49 == NaN

JS Core - more variables - part I

- a few rules and best practices for naming valid **identifiers**
- using typical ASCII alphanumeric characters
 - *an identifier must begin with a-z, A-Z, \$, _*
 - *may contain any of those characters, plus 0-9*
- property names follow this same basic pattern
- careful not to use certain keywords, or reserved words
- reserved words can include such examples as,
 - *break, byte, delete, do, else, if, for, this, while and so on*
 - *further details are available at the W3 Schools site*
- in JS, we can use different declaration keywords relative to intended scope
 - *var for local, global for global...*
- such declarations will influence scope of usage for a given variable
- concept of **hoisting**
 - *defines the declaration of a variable as belonging to the entire scope*
 - *by association accessible throughout that scope as well*
 - *also works with JS functions - hoisted to the top of the scope*

JS Core - more variables - part 2

- concept of nesting, and scope specific variables
- ES6 enables us to restrict variables to a block of code
- use keyword **let** to declare a block-level variable

```
if (a > 5) {  
  let b = a + 4;  
  
  console.log(b);  
}
```

- **let** restricts variable's scope to `if` statement
- variable `b` is not available to the whole function

ES6 - let variable

```
// function
var archiveCheck = function (level) {
  // add variable for archive
  var archive = 'waldzell';
  // specify purpose - default return
  var purpose = 'restricted';

  // check access level
  if (level === 'castalia') {
    let purpose = 'gaming';
    return purpose;
  }

  return purpose;
}

// log output - pass correct parameter value
console.log(`archive purpose is ${archiveCheck('castalia')}`);

// log output - pass incorrect parameter value
console.log(`archive purpose is ${archiveCheck('mariafels')}`);
```

JS Core - 1et

example

- Random Greeting Generator - A bit better

JS Core - more variables - part 3

- add **strict mode** to our code
- without we get a variable that will be hoisted to the top either
 - *set as a globally available variable, although it could be deleted*
 - *or it will set a value for a variable with the matching name*
- bubbled up through the available layers of scope
- becomes similar in essence to a declared global variable
- can create some strange behaviour in our applications
 - *tricky and difficult to debug*
- remember to declare your variables correctly and at the top

JS Core - more variables - example

```
var a;

function myScope() {
  "use strict";
  a = 49;
}

myScope()
a = 59;
console.log(a);
```

JS Core - functions and values

- variables acting as groups of code and blocks
- act as one of the primary mechanisms for scope within our JS applications
- also use functions as values
- effectively using them to set values for other variables

```
var a;

function scope() {
  "use strict";
  a = 49;
  return a;
}

b = scope() * 2;
console.log(b);
```

- useful and interesting aspect of the JS language
 - *allows us to build values from multiple layers and sources*

JS Core - more conditionals - part I

- briefly considered conditional statements using the `if` statement,

```
if (a > b) {  
  console.log("a is the best...");  
} else {  
  console.log("b is the best...");  
}
```

- Switch statements effectively follow the same pattern as `if` statements
 - *designed to allow us to check for multiple values in a more succinct manner*
 - *enable us to check and evaluate a given expression*
 - *then attempt to match a required value against an available `case`*
- addition of `break` is important, ensures only matched case is executed
 - *then the application breaks from the switch statement*
- if no `break` execution after that case will continue
 - *commonly known as **fall through***
 - *may be an intentional feature of your code design*
 - *allows a match against multiple possible cases*

JS Core - switch conditional - example

```
var a = 4;

switch (a) {
case 3:
    //par 3
    console.log("par 3");
    break;
case 4:
    //par 4
    console.log("par 4");
    break;
case 5:
    //par 5
    console.log("par 5");
    break;
case 59:
    //dream score
    console.log("record");
    break;
default:
    console.log("more practice");
}
```

JS Core - more conditionals - part 2

ternary

- a more concise way to write our conditional statements
- known as the **ternary** or **conditional** operator
- consider this operator a more concise form of standard `if...else` statement

```
var a = 59;  
var b = (a > 59) ? "high" : "low";
```

- equivalent to the following standard `if...else` statement

```
var a = 59;  
  
if (a > 59) {  
  var b = "high";  
} else {  
  var b = "low";  
}
```

JS Core - closures - part I

- important and useful aspect of JavaScript
- dealing with variables and scope
 - *continued, broader access to ongoing variables via a function's scope*
- closures as a useful construct to allow us to access a function's scope
 - *even after it has finished executing*
- can give us something similar to a private variable
 - *then access through another variable using relative scopes of outer and inner*
- inherent benefit is that we are able to repeatedly access internal variables
 - *normally cease to exist once a function had executed*

JS Core - closures - example - I

```
//value in global scope
var outerVal = "test1";

//declare function in global scope
function outerFn() {
  //check & output result...
  console.log(outerVal === "test1" ? "test is visible..." : "test not visible...");
}

//execute function
outerFn();
```

Image - JS Core - closures - global scope

```
test is visible...  
test.js (13,2)
```

JS Core - Closures - global scope

JS Core - closures - example - 2

```
"use strict";

function addTitle(a) {
  var title = "hello ";
  function updateTitle() {
    var newTitle = title+a;
    return newTitle;
  }
  return updateTitle;
}

var buildTitle = addTitle("world");
console.log(buildTitle());
```

JS Core - closures - part 2

Why use closures?

- use closures a lot in JavaScript
 - *real driving force behind Node.js, jQuery, animations...*
- closures help reduce amount, complexity of code necessary for advanced features
- closures help us add otherwise impossible features, e.g.
 - *any task using callbacks - event handlers...*
 - *private object variables...*
- closure allows us to work with a function that has been defined within another scope
 - *still has access to all variables within the defined outer scope*
 - *helps create basic encapsulated data*
 - *store data in a separate scope - then share it where needed*

JS Core - closures - part 3

```
function count(a) {  
  return function(b) {  
    return a + b;  
  }  
}  
  
var add1 = count(1);  
var add5 = count(5);  
var add10 = count(10);  
  
console.log(add1(8));  
console.log(add5(8));  
console.log(add10(8));
```

- using one function to create multiple other functions, add1, add5, add10, and so on.

JS Core - closures - example - 3

```
// variables in global scope
var outerVal = "test2";
var laterVal;

function outerFn() {
  // inner scope variable declared with value - scope limited to function
  var innerVal = "test2inner";
  // inner function - can access scope from parent function & variable innerVal
  function innerFn() {
    console.log(outerVal === "test2" ? "test2 is visible" : "test2 not visible");
    console.log(innerVal === "test2inner" ? "test2inner is visible" : "test2inner is not visible");
  }
  // inner function now added to global scope - now able to access elsewhere & call later
  laterVal = innerFn;
}
// invokes outerFn, innerFn is created, and its reference assigned to laterVal
outerFn();
// THEN - innerFn is invoked using laterVal - can't access innerFn directly...
laterVal();
```

Image - JS Core - closures - inner scope

```
test2 is visible  
test.js (15,5)  
test2inner is visible  
test.js (16,5)
```

JS Core - Closures - inner scope

JS Core - closures - part 4

- how is the `interval` variable available when we execute the inner function?
 - *this is why **closures** are such an important and useful concept in JavaScript*
 - *use of closures creates a sense of persistence in the scope*
- closures help create
 - *scope persistence*
 - *delayed access to functions and variables*
- closure creates a safe wrapper around
 - *the function*
 - *variables that are in scope as a function is defined*
- closure ensures function has everything necessary for correct execution
- closure wrapper persists whilst function exists

n.b. *closure usage is not memory free - there is an impact on app memory and usage...*

JS core - this

- `this` keyword - correct and appropriate usage
 - *commonly misunderstood feature of JS*
- value of `this` is not inherently linked with the function itself
- value of `this` determined in response to how the function is called
- value itself can be dynamic, simply based upon how the function is called
- if a function contains `this`, its reference will usually point to an **object**

JS core - this - part I

global, window object

- when we call a function, we can bind the `this` value to the window object
- resultant object refers to the root, in essence the global scope

```
function test1() {  
  console.log(this);  
}  
  
test1();
```

- **NB:** the above will return a value of `undefined` in strict mode.
- also check for the value of `this` relative to the global object,

```
var a = 49;  
  
function test1() {  
  console.log(this.a);  
}  
  
test1();
```

- JSFiddle - this - window
- JSFiddle - this - global

JS core - this - part 2

object literals

- within an object literal, the value of `this`, thankfully, will always refer to its own object

```
var object1 = {  
  method: test1  
};  
  
function test1() {  
  console.log(this);  
}  
  
object1.method();
```

- return value for `this` will be the object itself
- we get the returned object with a property and value for the defined function
- other object properties and values will be returned and available as well
- [JSFiddle - this - literal](#)
- [JSFiddle - this - literal 2](#)

JS core - this - part 3

object literals

```
var sites = {};  
sites.name = "philae";  
  
sites.titleOutput = function() {  
  console.log("Egyptian temples...");  
};  
  
sites.objectOutput = function() {  
  console.log(this);  
};  
  
console.log(sites.name);  
sites.objectOutput();  
sites.titleOutput();
```

Image - Object literals console output

```
philae
test.js (22,1)
> [object Object]      {name: "philae"}
test.js (19,3)
Egyptian temples...
test.js (15,3)
```

JS - this - object literals output

JS core - this - part 4

events

- for events, value of `this` points to the owner of the bound event

```
<div id="test">click to test...</div>
```

```
var testDiv = document.getElementById('test');

function output() {
  console.log(this);
};

testDiv.addEventListener('click', output, false);
```

- element is clicked, value of `this` becomes the clicked element
- also change the context of `this` using built-in JS functions
 - such as `.apply()`, `.bind()`, and `.call()`
- JSFiddle - this - events

ES6 JS - Arrow functions

basic

```
/**
  js-plain - definitions and arguments
  - basic example for arrow function
**/

// define array for planets
planets = ['mars', 'jupiter', 'venus'];
// use for each loop with array, and create arrow function for output to console
planets.forEach(planet => console.log(planet));
```

■ Demo

ES6 JS - Arrow functions

function context

```
/**
  js-plain - definitions and arguments
  - example of arrow function with function context
  **/

// button constructor
function Button() {
  this.clicked = false;
  // arrow function in function context
  this.click = () => {
    this.clicked = true;
    var message = `button clicked - ${this.clicked}`;
    console.log(message);
    document.getElementById("output").append(message);
  };
}

// create button object
var button = new Button();
var element = document.getElementById("test");
element.addEventListener("click", button.click);
```

■ Demo

ES6 JS - Arrow functions

example

- Random Greeting Generator - A bit better - v0.2

JS - Closures - *private* object property

A brief demo of getters and setters with private object property.

- FN: constructor function
 - *'private variable' - not directly accessible*
 - *define properties on object*
 - *add getter and setter methods*
- Use:
 - *instantiate object using constructor*
 - *log output of check against getter method for value of 'private' variable*
 - *use 'setter' method to update value of 'private' variable*
 - *log output for check of value update of 'private' variable*

JS - closures - *private* object property - example

```
// define constructor
function Archive() {
  // private variable - accessible through function closures
  let _catalogue = 'glass bead';
  // define catalogue property access
  Object.defineProperty(this, 'catalogue', {
    get: () => {
      console.log(`catalogue requested...`);
      return _catalogue;
    },
    set: value => {
      console.log(`catalogue updated`);
      _catalogue = value;
    }
  });
}

// instantiate object from Archive constructor
const archiveCheck = new Archive();

// check access to constructor variable - returns 'undefined' without getter method
console.log(`direct access against private variable = ${archiveCheck._catalogue}`);
// check access using getter method - returns variable value
console.log(`getter access against private variable = ${archiveCheck.catalogue}`);

// update catalogue value - uses 'setter' method
archiveCheck.catalogue = 'history';

// check update catalogue variable
console.log(`updated catalogue = ${archiveCheck.catalogue}`);
```

- Demo - private object property

JS extras - best practices - part I

a few best practices...

variables

- limit use of global variables in JavaScript
 - *easy to override*
 - *can lead to unexpected errors and issues*
 - *should be replaced with appropriate local variables, closures*
- local variables should always be declared with keyword `var`
 - *avoids automatic global variable issue*

declarations

- add all required declarations at the top of the appropriate script or file
 - *provides cleaner, more legible code*
 - *helps to avoid unnecessary global variables*
 - *avoid unwanted re-declarations*

types and objects

- avoid declaring numbers, strings, or booleans as objects
- treat more correctly as primitive values
 - *helps increase the performance of our code*
 - *decrease the possibility for issues and bugs*

JS extras - best practices - part 2

type conversions and coercion

- weakly typed nature of JS
 - *important to avoid accidentally converting one type to another*
 - *converting a number to a string or mixing types to create a NaN (Not a Number)*
- often get a returned value set to NaN instead of generating an error
 - *try to subtract one string from another may result in NaN*

comparison

- better to try and work with `===` instead of `==`
 - *`==` tries to coerce a matching type before comparison*
 - *`===` forces comparison of values and type*

defaults

- when parameters are required by a function
 - *function call with a missing argument can lead to it being set as **undefined***
 - *good coding practice to assign default values to arguments*
 - *helps prevent issues and bugs*

switches

- consider a `default` for the switch conditional statement
- ensure you always set a `default` to end a switch statement

JS extras - performance - part I

loops

- try to limit the number of calculations, executions, statements performed per loop iteration
- check loop statements for assignments and statements
 - *those checked or executed once*
 - *rather than each time a loop iterates*
- for loop is a standard example of this type of quick optimisation

```
// bad
for (i = 0; i < arr.length; i++) {
  ...
}
// good
l = arr.length;
for (i = 0; i < l; i++) {
  ...
}
```

- source - W3

JS extras - performance - part 2

DOM access

- repetitive DOM access can be slow, and resource intensive
- try to limit the number of times code needs to access the DOM
- simply access once and then use as a local variable

```
var testDiv = document.getElementById('test');  
testDiv.innerHTML = "test...";
```

JavaScript loading

- not always necessary to place JS files in the <head> element
 - *check context, in particular for recent mobile and desktop frameworks*
 - *Cordova, Electron...*
- adding JS scripts to end of the page's body
 - *allows browser to load the page first*
- HTTP specification defines browsers should not download more than two components in parallel

JS - initial usage

fun exercise

Choose one of the following app examples,

- **sports** website for latest scores and updates
 - *e.g. scores for current matches, statistics, team data, player info &c.*
- **shopping** website
 - *product listings and adverts, cart, reviews, user account page &c.*
- **restaurant** website
 - *introductory info, menus, sample food images, user reviews &c.*

Then, consider the following

- where do you need JavaScript in the app?
 - *why?*

JS extras - JSON - part I

- JSON is a lightweight format and wrapper for storing and transporting data
- inherently language agnostic, easy to read and understand
- growing rapidly in popularity
 - *many online APIs have updated XML to JSON for data exchange*
- syntax of JSON is itself derived from JS object notation
 - *text-only format*
- allows us to easily write, describe, and manipulate JSON in practically any programming language
- **JSON syntax** follows a few basic rules,
 - *data is recorded as name/value pairs*
 - *data is separated by commas*
 - *objects are defined by a start and end curly brace*
 - *{ }*
 - *arrays are defined by a start and end square bracket*
 - *[]*

JS extras - JSON - part 2

- underlying construct for JSON is a pairing of name and value

```
"city": "Marseille"
```

JSON Objects

- contained within curly braces
- objects can contain multiple name/value pairs

```
{  
  "country": "France",  
  "city": "Marseille"  
}
```

JS extras - JSON - part 3

JSON Arrays

- contained within square brackets
 - *arrays can also contain objects*

```
{
  "cities": [
    {
      "name": "Marseille",
      "region": "Provence-Alpes-Côte d'Azur"
    },
    {
      "name": "Paris",
      "region": "Île-de-France"
    }
  ]
}
```

- use this with JavaScript, and parse the JSON object.
 - *JSFiddle - Parse JSON*

Demos

- ES6 (ES2015)
 - *let usage - Random Greeting Generator v0.2*
- JS Arrays
 - *Random Greeting Generator - v0.1*
- JSFiddle
 - *Basic logic - functions*
 - *Basic logic - scope*
 - *this - events*
 - *this - global*
 - *this - literal*
 - *this - literal 2*
 - *this - window*
 - *Parse JSON*

Resources

- [MDN - JS](#)
- [MDN - JS Data Types and Data Structures](#)
- [MDN - JS Grammar and Types](#)
- [MDN - JS Objects](#)
- [W3 Schools - JS](#)