Comp 324/424 - Client-side Web Design

Spring Semester 2024 Week 6

Dr Nick Hayward

JS Basics - variables - part 1

- **symbolic** container for values and data
- applications use containers to keep track and update values
- use a **variable** as a container for such values and data
  - allow values to vary over time
- JS can emphasize types for values, does not enforce on the variable
  - **weak typing** or **dynamic typing**
  - JS permits a variable to hold a value of any type
- often a benefit of the language
- a quick way to maintain flexibility in design and development

---

JS Basics - variables - part 2

- declare a variable using the keyword `var`
- declaration does not include **type** information

```
var a = 49;
//double var a value
var a = a * 2;
//coerce var a to string
var a = String(a);
//output string value to console
console.log(a);
```

- `var a` maintains a running total of the value of `a`
- keeps record of changes, effectively **state** of the value
- **state** is keeping track of changes to any values in the application

---

JS Basics - variables - part 3

- use variables in JS to enable central, common references to our values and data
- better known in most languages simply as **constants**
- JS is similar
  - creates a read-only reference to a value
  - value itself is not immutable, e.g. an object…
  - it's simply the identifier that cannot be reassigned
  - JS constants are also bound by scoping rules
- allow us to define and declare a variable with a value

    - not intended to change throughout the application
- **constants** are often declared together
    - uppercase is standard practice - although not a rule…
- form a store for values abstracted for use throughout an app
- JS normally defines constants using uppercase letters,

```
var NAME = "Philae";
```

- ECMAScript 6, ES6, introduces additional variable keywords
    - e.g. `const`

```
const TEMPLE_NAME = "Philae";
```

- benefits of abstraction, ensuring value is not accidentally changed
    - change rejected for a running app
    - in `strict` mode, app will fail with an error for any change

---

JS Basics - values and types

- JS has typed values, not typed variables
- JS provides the following built-in types
    - boolean
    - null
    - number
    - object
    - string
    - symbol (new in ES6)
    - undefined
- more help provided by JS's `typeof` operator
    - examine a value and return its type

```
var a = 49;
console.log(typeof a); //result is a number
```

- as of ES6, there are 7 possible return types in JS
- **NB:** JS variables do not have types, mere containers for values
    - values specify the type

```
var a = null;
console.log(typeof a); //result is object - known bug in JS...
```

JS Basics - comments

- JS permits comments in the code
- two different implementations

single line

```
//single line comment
var a = 49;
```

multi-line

```
/* this comment has more to say...
we'll need a second line */
var b = "forty nine";
```

JS Basics - logic - blocks

- simple act of grouping contiguous and related code statements together
  - known as **blocks**
- block defined by wrapping statements together
  - within a pair of curly braces, `{}`
- **blocks** commonly attached to other forms of control statement

```
if (a > b) {
...do something useful...
}
```

JS Basics - logic - conditionals - part 1

- conditionals, conditional statements require a decision to be made
- code statement, application, consults **state**
  - answer will predominantly be a simple **yes** or **no**
- JS includes many different ways we can express **conditionals**
- most common example is the `if` statement
  - if this given condition is true, do the following...

```
if (a > b) {
console.log("a is greater than b...");
}
```

- `if` statement requires an expression between the parentheses
  - evaluates as either *true* or *false*

JS Basics - logic - conditionals - part 2

- additional option if this expression returns false
  - using an **else** clause

```
if (a > b) {
console.log("a is greater than b...");
} else {
console.log("no, b is greater...");
}
```

- for an `if` statement, JS expects a `boolean`
- JS defines a list of values that it considers *false*
  - e.g. `0` ...
- any value not on this list of *false* values will be considered true
  - coerced to *true* when defined as a `boolean`
- conditionals in JS also exist in another form
  - the `switch` statement
  - more to come...

JS Basics - logic - loops

- loops allow repetition of sets of actions until a condition fails

- repetition continues whilst the requested condition holds
- loops take many different forms and follow this basic behaviour
- a loop includes the *test condition* as well as a *block*
  - normally within curly braces
  - block executes, an iteration of the loop has occurred
- good examples of this behaviour include `while` and `do...while` loops
- basic difference between these loops, `while` and `do...while`
  - conditional tested is before the first iteration (`while` loop)
  - after the first iteration (`do...while`) loop
- if the condition is initially false
  - a `while` loop will never run
  - a `do...while` will run through for the first time
- also stop a JS loop using the common `break` statement
- `for` loop has three clauses, including
  - initialisation clause
  - conditional test clause
  - update clause

JS Basics - logic - functions - part 1

- functions are a type of object
  - may also have their own properties
  - define once, then re-use as needed throughout our application
- **function** is a named grouping of code
  - name can be called, and code will be run each time
- JS functions can be designed with optional arguments
  - known as **parameters**
  - allow us to pass values to the function
- functions can also optionally return a value

```
function outputTotal(total) {
  console.log(total);
}
var a = 49;
a = a * 3; // or use a *= 3;

outputTotal(a);
```

JS Basics - logic - functions - part 2

```
function outputTotal(total) {
  console.log(total);
}

function calculateTotal(amount, times) {
  amount = amount * times;
  return amount;
}

var a = 49;
```

```
a = calculateTotal(a, 3);
outputTotal(a);
```

- JSFiddle Demo

JS Basics - logic - scope

- scope or **lexical scope**
  - collection of variables, and associated access rules by name
- in JS each function gets its own scope
- variables within a function's given **scope**
  - can only be accessed by code inside that function
- variable name has to be unique within a function's scope
- same variable name could appear in different scopes
- nest one scope within another
  - code in inner scope can access variables from either inner or outer scope
  - code in outer scope cannot, by default, access code in the inner scope

---

JS Basics - logic - scope example

```
function outerScope() {
  var a = 49;
  //scope includes outer and inner
  function innerScope() {
    var b = 59;
    //output a and b
    console.log(a + b); //returns 108
  }
  innerScope();

  //scope limited to outer
  console.log(a); //returns 49
}

//run outerScope function
outerScope();
```

- JSFiddle Demo

CSS Basics - selectors

- **selectors** are a crucial part of working with CSS, JS...
- basic selectors such as

```
  p {
    color: #444;
  }
```

- above ruleset adds basic styling to our paragraphs
  - sets the text colour to HEX value *444*
- simple and easy to apply

    - applies the same properties and values to all paragraphs
- specificity requires classes, pseudoclasses...

HTML5, CSS, & JS - example - part 8

```
function travelNotes() {
  "use strict";

  // get a reference to `.note_output` in the DOM
  // n.b. these can be combined as well...
  let noteOutput = document.querySelector('.note-output');
  noteOutput.innerHTML = '<p>first travel note for Marseille...</p>';

}

// load app
travelNotes();
```

travel.js - plain JS

- a simple JS function to hold the basic logic for our app
- call this function any reasonable, logical name
- in initial function, we set the `strict` pragma
- many different ways to achieve this basic loading of app logic
- DEMO - travel notes - series 1

JS Basics - strict mode

- intro of ES5 - JS now includes option for **strict** mode
    - ensures tighter code and better compliance...
    - often helps ensure greater compatibility, safer use of language...
    - can also help optimise code for rendering engines
- add **strict** at different levels within our JS code
    - e.g. single function level or enforce for whole file

```
function outerScope() {
  "use strict";
  //code is strict

  function innerScope() {
  //code is strict

  }
}
```

- if we set **strict** mode for complete file - set at top of file
    - all functions and code will be checked against **strict** mode
        - e.g. check against auto-create for global variables
        - or missing `var` keyword for variables...

```
function outerScope() {
  "use strict";
  a = 49; // `var` missing - ReferenceError
}
```

Video - JavaScript

strict mode JavaScript Strict Mode - UP TO 4:32

Source - JavaScript - Overview of Strict Mode

HTML5, CSS, & JS - example - part 9

interaction - add a note - plain JS

- added and styled our input and button for adding a note
- use JavaScript to handle click event on button
- update `travel.js` file for event handler

```
let addNoteBtn = document.getElementById('add-note');
addNoteBtn.addEventListener('click', () => {
  console.log('add button clicked...');
});
```

JS Core - more variables - part 1

- 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 - `let`

example

- Random Greeting Generator - A bit better

Video - Variables

`let` **and** `const` JavaScript scope and variable usage - UP TO 2:30

Source - JavaScript scope and variables

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

```javascript
var a;

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

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

HTML5, CSS, & JS - example - part 10

interaction - add a note - output - plain JS

- update code to better handle and output the text from the input field
- what is this handler actually doing?
  - attached an event listener to an element in the DOM
  - uses standard CSS selectors to find the required element
- JavaScript can select and target DOM elements using standard CSS selectors
  - then manipulate them, as required

```javascript
function travelNotes() {
  "use strict";

  // get a reference to `.note_output` in the DOM
  let noteOutput = document.querySelector('.note-output');
  // add note button
  let addNoteBtn = document.getElementById('add-note');

  // add event listener to add note button
  addNoteBtn.addEventListener('click', () => {
    // create p node
    let p = document.createElement('p');
    // create text node
    let noteText = document.createTextNode('sample note text...');
    // append text to paragraph
    p.appendChild(noteText);
    // append new paragraph and text to existing note output
    noteOutput.appendChild(p);
  });

}
```

- DEMO - travel notes - series 1

HTML5, CSS, & JS - example - part 11

```
function travelNotes() {
  "use strict";

  // get a reference to `.note_output` in the DOM
  let noteOutput = document.querySelector('.note-output');
  // add note button
  let addNoteBtn = document.getElementById('add-note');
  // input field for add note
  let inputNote = document.getElementById('input-note');

  addNoteBtn.addEventListener('click', () => {
    // create p node
    let p = document.createElement('p');
    // get value from input field for note
    let inputVal = inputNote.value;
    // create text node
    let noteText = document.createTextNode(inputVal);
    // append text to paragraph
    p.appendChild(noteText);
    // append new paragraph and text to existing note output
    noteOutput.appendChild(p);
  });

}
```

interaction - add a note - output - plain JS

- DEMO - travel notes - series 1

ES6 JS - Arrow functions

```
/**
  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));
```

basic

- Demo

ES6 JS - Arrow functions

```
/**
  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);
```

function context

- Demo

ES6 JS - Arrow functions

example

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

HTML5, CSS, & JS - example - part 12

```
function travelNotes() {
  "use strict";

  // get a reference to `.note_output` in the DOM
  let noteOutput = document.querySelector('.note-output');
  // add note button
  let addNoteBtn = document.getElementById('add-note');
  // input field for add note
  let inputNote = document.getElementById('input-note');

  // add event listener to add note button
  addNoteBtn.addEventListener('click', () => {
    // create p node
    let p = document.createElement('p');
    // get value from input field for note
    let inputVal = inputNote.value;

    // check input value
    if (inputVal !== '') {
      // create text node
      let noteText = document.createTextNode(inputVal);
      // append text to paragraph
      p.appendChild(noteText);
      // append new paragraph and text to existing note output
      noteOutput.appendChild(p);
      // clear input text field
      inputNote.value = '';
    }
  });

}
```

interaction - add a note - clear input - plain JS

- DEMO - travel notes - series 1

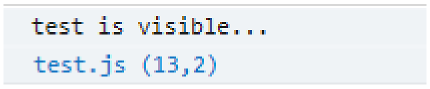
JS Core - closures - part 1

- 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 - 1

```
//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)
```

Figure 1: JS Core - Closures - global scope

Video - JS Core

closures - part 1 Closures in JavaScript - UP TO 3:17

Source - JavaScript Closures - YouTube

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.

Video - JS Core

closures - part 2 Closures in JavaScript - UP TO 5:21

Source - JavaScript Closures - YouTube

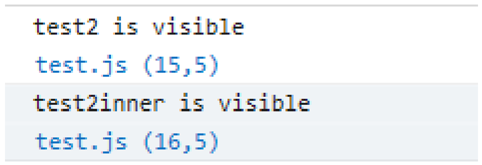
JS Core - closures - example - 3

```
// 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

JS Core - closures - part 4

- how is the `innerVal` 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



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

Figure 2: JS Core - Closures - inner scope

  - 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…

Video - JS Core

closures - part 3 Closures in JavaScript - UP TO 6:20

Source - JavaScript Closures - YouTube

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 1

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();
```

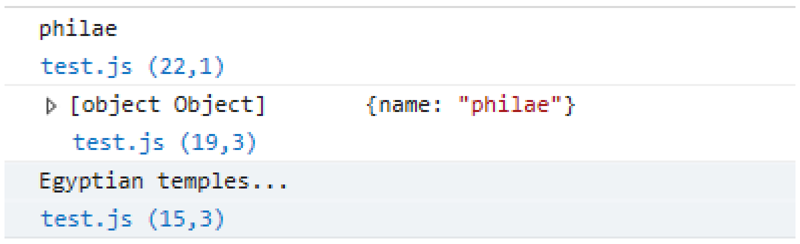
- return value for `this` will be the object itself
- we get the returned object with a property and value for the defined function
- other object properties and values will be returned and available as well
- JSFiddle - this - literal
- JSFiddle - this - literal 2

JS core - `this` - part 3

```

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();
```

object literals

Image - Object literals console output



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

Figure 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

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

```javascript
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

HTML5, CSS, & JS - example - part 13

interaction - add a note - keyboard listener - plain JS

- need to consider how to handle keyboard events
- listening and responding to a user hitting the return key in the input field
- similar pattern to user click on button

```javascript
// add event listener for keypress in note input field
inputNote.addEventListener('keypress', (e) => {
  // check key pressed by code - 13 - return
  if (e.keyCode === 13) {
    console.log('return key pressed...');
  }
});
```

- need to abstract handling both button click and keyboard press
- need to be selective with regard to keys pressed
- add a conditional check to our listener for a specific key
- use local variable from the event itself, e.g. `e` , to get value of key pressed

- compare value of `e` against key value required
- example recording keypresses
  - Demo Editor

Video - Users and interaction

digital accessibility What is digital accessibility?

Source - Digital Accessibility - YouTube

Demos

Travel Notes - series 1

- travel notes - demo 1
- travel notes - demo 2
- travel notes - demo 3
- travel notes - demo 4
- travel notes - demo 5

JavaScript

- Basic logic - functions
- Basic logic - scope
  - Basic logic - arrow functions
  - Basic logic - arrow function context

random greeting generator

- Random Greeting Generator - v0.1
- `let` usage - Random Greeting Generator
- Random Greeting Generator - A bit better - v0.2

Resources

- JS
  - MDN - JS
  - JavaScript Closures - YouTube
  - JavaScript - Scope and variables - YouTube
  - JS Info - DOM Nodes
  - MDN - JS Const
    - MDN - JS Data Types and Data Structures
    - MDN - JS Grammar and Types
    - MDN - JS Objects
    - W3 Schools - JS