Comp 324/424 - Client-side Web Design

Fall Semester 2024 - Week 5

Dr Nick Hayward

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

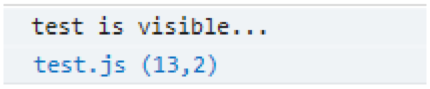
JS Core - closures - example - 1

```
//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

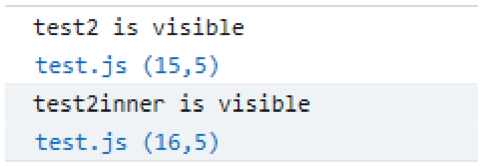
Source - JavaScript Closures - YouTube

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



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

Figure 2: 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
    - 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();
```

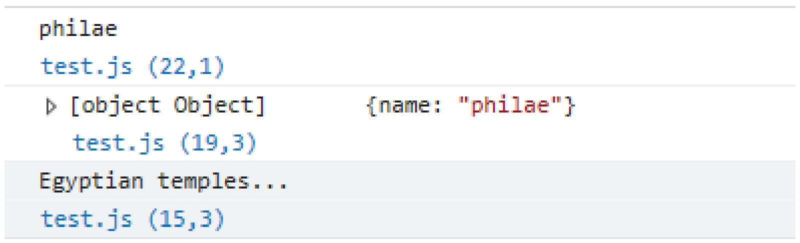
object literals

Image - Object literals console output

JS core - `this` - part 4

events

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



```
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

```
<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

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

```
// 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

JS Core - checking equality - part 1

- 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 1

- 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`

HTML5, CSS, & JS - example - part 14

interaction - add a note - abstract code

- need to create a new function to abstract
  - creation and output of a new note
  - manage the input field for our note app
- moving logic from button click function to separate, abstracted function
- then call this function as needed
  - for a button click or keyboard press
  - then create and render the new note

```
// create a note
// - input = value from input field
// - output = DOM node for output of new note
function createNote(input, output) {
    // create p node
    let p = document.createElement('p');
    // get value from input field for note
    let inputVal = input.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
      output.appendChild(p);
      // clear input text field
      input.value = '';
    }
}
```

HTML5, CSS, & JS - example - part 15

```
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', () => {
      createNote(inputNote, noteOutput);
  });

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

}

// load app
travelNotes();
```

interaction - add a note - plain JS

- DEMO - travel notes - series 1

HTML5, CSS, & JS - example - part 16

interaction - add a note - animate

- JavaScript well-known for is its simple ability to animate elements
- many built-in effects available in various JS animation libraries
  - build our own as well
- to `fadeIn` an element, effectively it needs to be hidden first
- we hide our newly created note
- then we can set it to `fadeIn` when ready
  - ...
- DEMO - travel notes - series 1

CSS Basics - complex selector - part 1

- our DOM will often become more complicated and detailed
- depth and complexity will require more complicated selectors as well
- lists and their list items are a good example

```
<ul>
  <li>unordered first</li>
  <li>unordered second</li>
  <li>unordered third</li>
</ul>
<ol>
  <li>ordered first</li>
  <li>ordered second</li>
  <li>ordered third</li>
</ol>
```

- two lists, one unordered and the other ordered
- style each list, and the list items using rulesets

```
ul {
  border: 1px solid green;
}
ol {
  border: 1px solid blue;
}
```

Demo - Complex Selectors - Part 1

- Demo - Complex Selectors Part 1

CSS Basics - complex selector - part 2

- add a ruleset for the list items, `<li>`
- applying the same style properties to both types of lists
- more specific to apply a ruleset to each list item for the different lists

```
ul li {
  color: blue;
}
ol li {
  color: red;
}
```

- also be useful to set the background for specific list items in each list

```
li:first-child {
  background: #bbb;
}
```

- pseudoclass of `nth-child` to specify a style for the second, fourth &c. child in the list

```
li:nth-child(2) {
  background: #ddd;
}
```

Demo - Complex Selectors - Part 2

- Demo - Complex Selectors Part 2

CSS Basics - complex selector - part 3

- style odd and even list items to create a useful alternating pattern

```
li:nth-child(odd) {
  background: #bbb;
}
li:nth-child(even) {
  background: #ddd;
}
```

- select only certain list items, or rows in a table &c.
  - e.g. every fourth list item, starting at the first one

```
li:nth-child(4n+1) {
  background: green;
}
```

- for **even** and **odd** children we're using the above with convenient shorthand
- other examples include
  - `last-child`
  - `nth-last-child()`
  - many others...

Demo - CSS Complex Selectors - Part 3

- Demo - Complex Selectors Part 3

HTML5, CSS, & JS - example - part 17

style and render notes

- we have some new notes in our app
- add some styling to help improve the look and feel of a note
- can set background colours, borders font styles...
- set differentiating colours for each alternate note
- allows us to try some pseudoclasses in the CSS
  - specified paragraphs in the `note-output` section

```
.note-output p:nth-child(even) {
  background-color: #ccc;
}
.note-output p:nth-child(odd) {
  background-color: #eee;
}
```

- DEMO - travel notes - series 1

HTML5, CSS, & JS - final thoughts

- a basic app that records simple notes
- many additional options we can add

- some basic functionality is needed to make it useful
  - autosave - otherwise we lose our data each time we refresh the browser
  - edit a note
  - delete a note
  - add author information
- additional functionality might include
  - save persistent data to DB, name/value pairs…
  - organise and view collections of notes
  - add images and other media
    - local and APIs
  - add contextual information
    - again, local and APIs
  - structure notes, media, into collection
  - define related information
  - search, sort…
  - export options and sharing…
- security, testing, design patterns

Video - Scotoma - Da Vinci Code

Scotoma - The Da Vinci Code - Source: YouTube

Demos

CSS

- CSS - Complex Selectors Part 1
- CSS - Complex Selectors Part 2
- CSS - Complex Selectors Part 3

Travel Notes - series 1

- travel notes - demo 6
- travel notes - demo 7
- travel notes - demo 8

References

- CSS Selectors
- JS
  - MDN - JS
  - JS Info - DOM Nodes
    - MDN - JS Objects
    - W3 Schools - JS