

Comp 324/424 - Client-side Web Design

Spring Semester 2018

Dr Nick Hayward

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 I

- 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*
 - eg: `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 I

- 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

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
 - eg: 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
 - eg: check against auto-create for global variables
 - or missing `var` keyword for variables...

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

JS Core - 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 Core - objects - part I

Objects

- **object** type includes a compound value
 - *JS can use to set properties, or named locations*
- each of these properties holds its own value
 - *can be defined as any type*

```
var objectA = {  
  a: 49,  
  b: 59,  
  c: "Philae"  
};
```

- access these values using either **dot** or **bracket** notation

```
//dot notation  
objectA.a;  
//bracket notation  
objectA["a"];
```

JS Core - objects - example

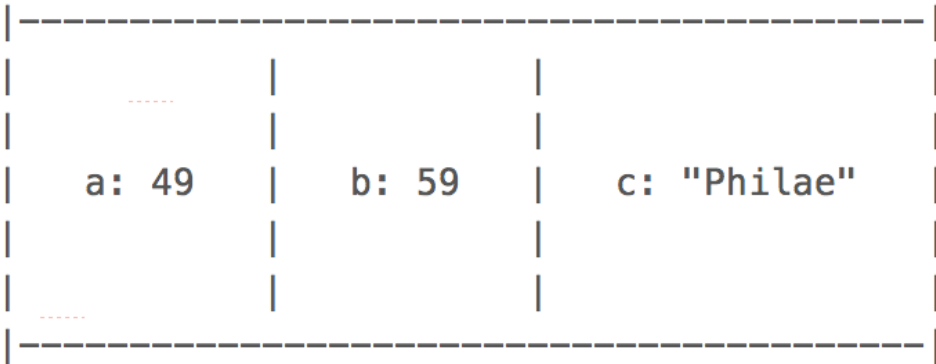
```
// create object
var object = {
  archive: 'waldzell',
  access: 'castalia',
  purpose: 'gaming'
};

// log output with dot notation
console.log(`archive is ${object.archive}`);

// log output with bracket notation - returns undefined
console.log(`access is restricted to ${object[1]}`);

// log output with bracket notation
console.log(`purpose is ${object['purpose']}`);
```

Image - JS Object



JS Object

ES6 - template literals

```
// create object
var object = {
  archive: 'waldzell',
  access: 'castalia',
  purpose: 'gaming'
};

// log output with template literals
console.log(`archive is ${object.archive}`);

// log output
console.log('archive is ' + object.archive);

// log output all object properties with template literals
console.log(`archive = ${object.archive}, access = ${object.access}, purpose = ${object.purpose}`);

// log output all object properties
console.log('archive = ' + object.archive + ', access = ' + object.access + ' purpose = ' + object.purpose);
```

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

0: 49	1: 59	2: "Philae"

JS Array

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 - 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 Fiddle - Strict Mode

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();
```

- JS Fiddle - Closures - Example I

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 Fiddle - Closures - Example 2

Demos

- JSFiddle tests - JS
- JSFiddle - Closures - Example 1
- JS Fiddle - Closures - Example 2
- JSFiddle Demo
- JSFiddle - Functions I
- JSFiddle - Scope I
- JS Fiddle - Strict Mode

References

- Apache Cordova
- Electron
- jQuery
- JSLint - JavaScript Validator
- JSONLint - JSON Validator
- MDN
 - *MDN - JS*
 - *MDN - JS Const*
 - *MDN - JS Data Types and Data Structures*
 - *MDN - JS Grammar and Types*
 - *MDN - JS Objects*
- W3 - JS Object