

Comp 324/424 - Client-side Web Design

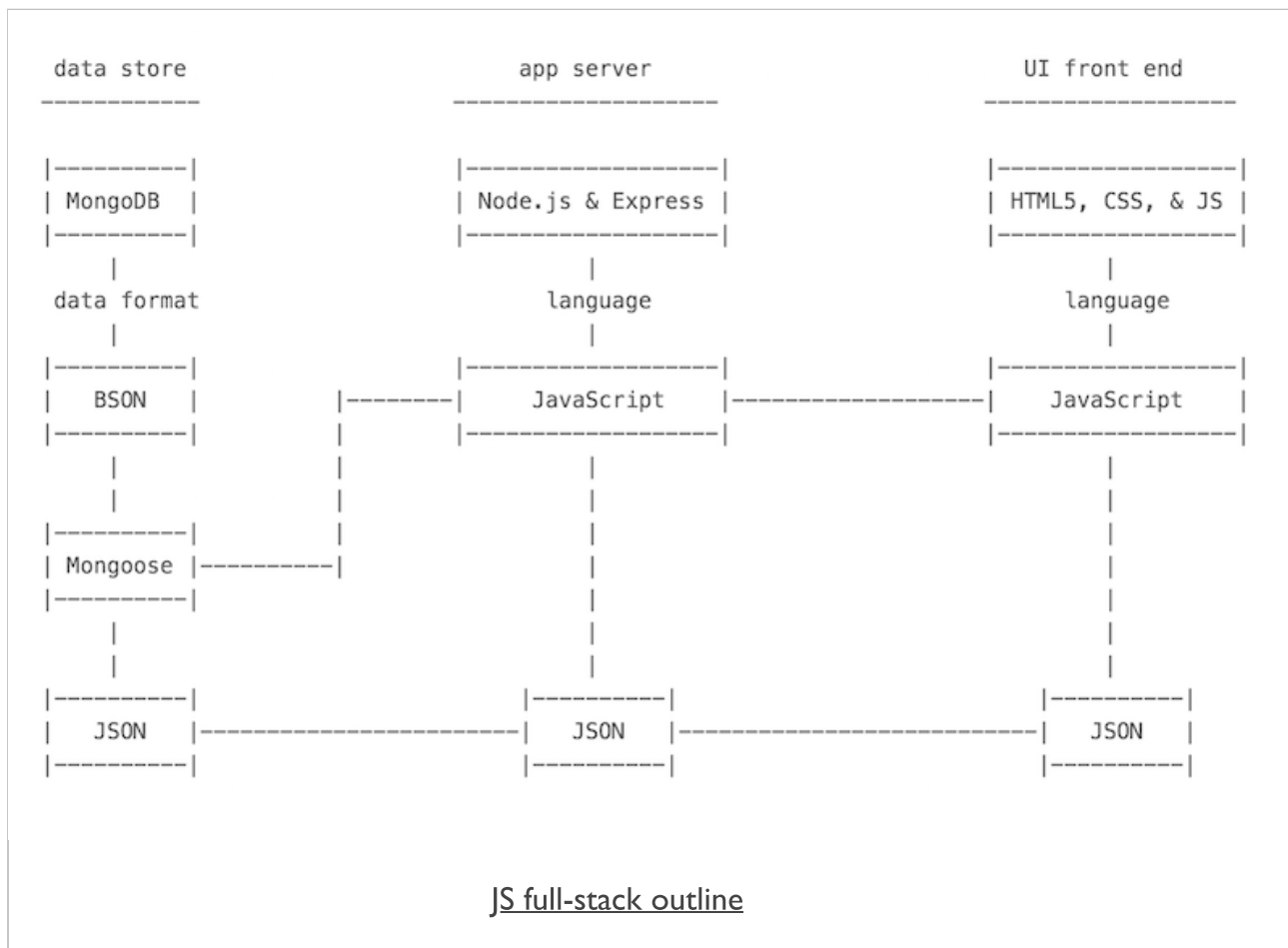
Fall Semester 2019 - Week 6

Dr Nick Hayward

building a web app - sample outline of underlying structure

- apps developed using a full JavaScript stack
- using and incorporating JS into each part of app's development
 - *UI front-end*
 - *app server and management*
 - *data store and management*
- Technologies will include
 - *front-end: HTML5, CSS, JS...*
 - *app server: Node.js, Express...*
 - *data store: MongoDB, Redis, Mongoose...*
- Data format is JSON

Image - building a web app - sample outline



n.b. I've explicitly omitted any arrows for flow within this diagram. This is something we'll return to as we start to work with Node.js, Mongoose, and MongoDB.

JS Intro

- JavaScript (JS) a core technology for client-side design and development
- now being used as a powerful technology to help us
 - *rapidly prototype and develop web, mobile, and desktop apps*
- libraries such as jQuery, React, AngularJS, and Node.js
- helps develop cross-platform apps
 - *Apache Cordova*
 - *Electron*
- Embedded systems
 - *Espruino - <http://www.espruino.com/>*
 - *Tessel - <https://tessel.io/>*

JS Basics - operators

- operators allow us to perform
 - *mathematical calculations*
 - *assign one thing to another*
 - *compare and contrast...*
- simple * operator, we can perform multiplication

```
2 * 4
```

- we can add, subtract, and divide numbers as required
- mix mathematical with simple assignment

```
a = 4;  
b = a + 2;
```

JS Basics - some common operators - part I

Assignment

- `=`
- eg: `a = 4`

Comparison

- `<`, `>`, `<=`, `>=`
- eg: `a <= b`

Compound assignment

- `+=`, `-=`, `*=`, `/=`
- compound operators are used to combine a mathematical operation with assignment
- same as `result = result + expression`
- eg: `a += 4`

Equality

operator	description
<code>==</code>	loose equals
<code>===</code>	strict equals
<code>!=</code>	loose not equals
<code>!==</code>	strict not equals

- eg: `a != b`

JS Basics - some common operators - part 2

Increment/Decrement

- increment or decrement an existing value by 1
 - `++`, `--`
 - eg: `a++` is equal to `a = a + 1`

Logical

- used to express compound conditionals - **and**, **or**
 - `&&`, `||`
 - eg: `a || b`

Mathematical

- `+`, `-`, `*`, `/`
 - eg: `a * 4` or `a / 4`

Object property access

- properties in objects are specific named locations for holding values and data
- effectively, values within values
 - `.`
 - eg: `a.b` means object `a` with a property of `b`

JS Basics - values and types

- able to express different representations of values
 - *often based upon need or intention*
 - *known as **types***
- JS has built-in types
 - *allow us to represent **primitive** values*
 - *eg: **numbers, strings, booleans***
- such values in the source code are simply known as **literals**
- **literals** can be represented as follows,
 - *string literals use double or single quotes eg: "some text" or 'some more text'*
 - *numbers and booleans are represented without being escaped eg: 49, true;*
- also consider arrays, objects, functions...

JS Basics - type conversion

- option and ability to convert types in JS
 - in effect, **coerce** our values and types from one type to another
- convert a number, or coerce it, to a string
- built-in JS function, `Number ()`, is an explicit coercion
 - explicit coercion, convert any type to a number type
- implicit coercion, JS will often perform as part of a comparison

```
"49" == 49
```

- JS implicitly coerces left string to a matching number
 - then performs the comparison
- often considered bad practice
 - convert first, and then compare
- implicit coercion still follows rules
 - can be very useful

JS Basics - variables - part I

- **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 - 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 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

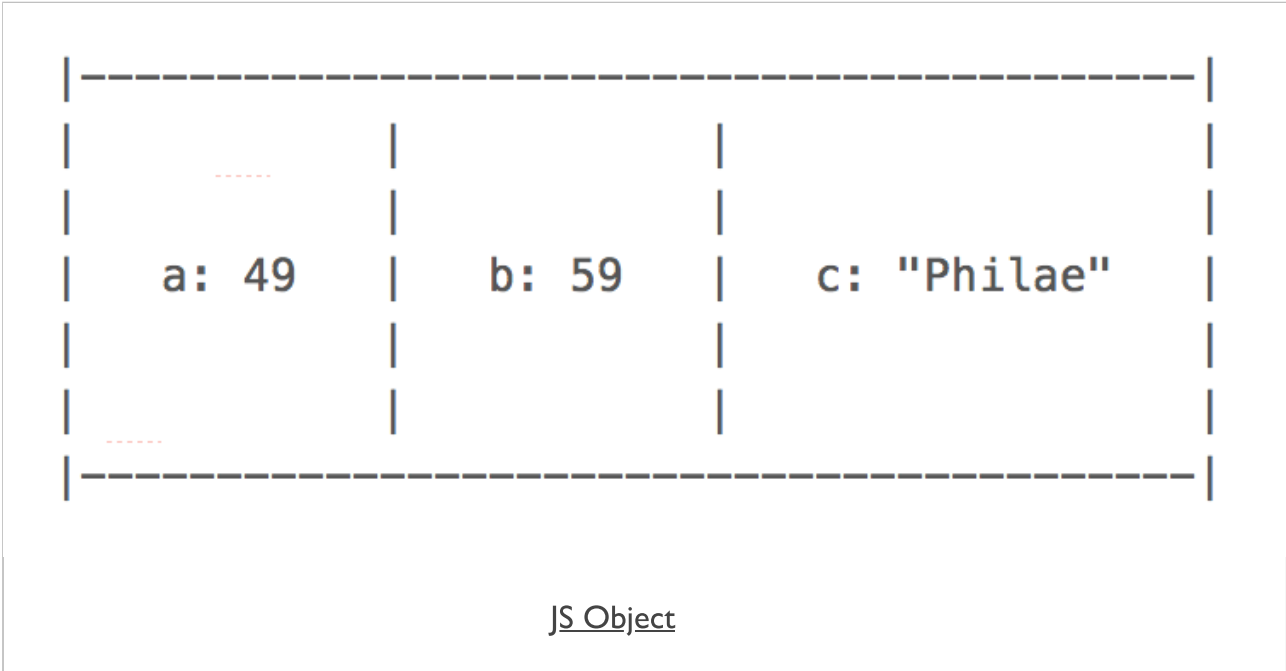
```
// create object
var object = {
  archive: 'waldzell',
  access: 'castalia',
  purpose: 'gaming'
};

// 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



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

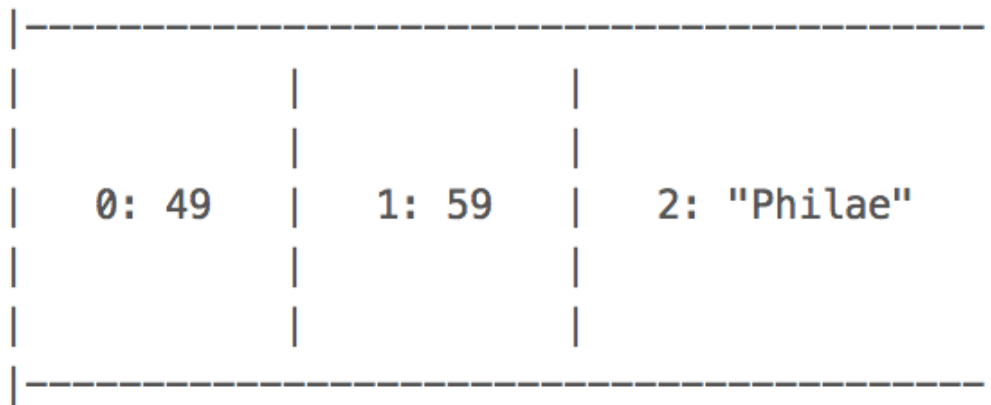
- JS array an object that contains values, of any type, in numerically indexed positions
 - *store a number, a string...*
 - *array will start at index position 0*
 - *increments by 1 for each new value*
- arrays can also have properties
 - *eg: automatically updated **length** property*

```
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



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";  
}
```

Demos

- ES6 (ES2015)
 - *let usage - Random Greeting Generator v0.2*
- JS Arrays
 - *Random Greeting Generator - v0.1*
- JSFiddle
 - *Basic logic - functions*
 - *Basic logic - scope*

Resources

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