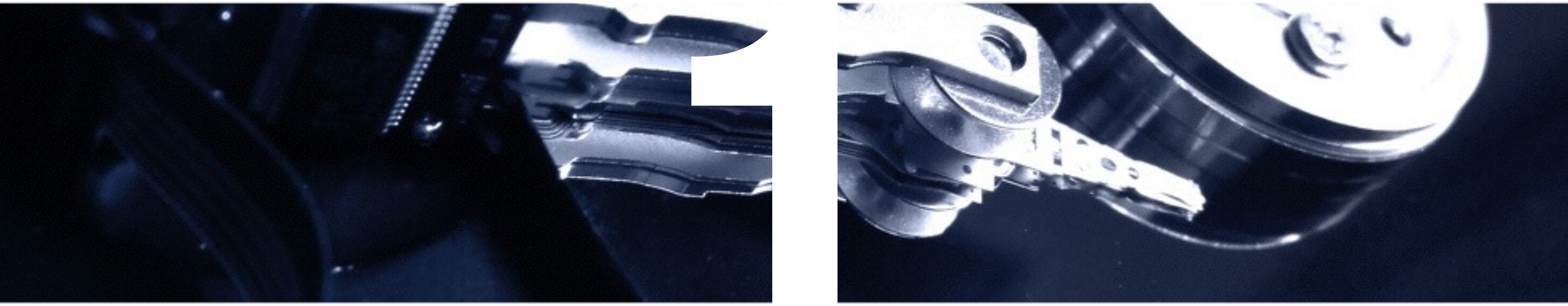
FULL SAIL
UNIVERSITY

# web design and development



## programming for web applications 1

## courseMaterial.2

**courseDirector**

Fialishia O’Loughlin

foloughlin@fullsail.com

---

**labSpecialist**

Eric Silvay

esilvay@fullsail.com

## courseMaterial.**2**

## goal1.**Recap**

## goal1.Recap

- questions from goal 1 which was a review of the items from WPF
    - variables & values
    - numbers
    - strings
    - arrays
    - conditionals
    - functions
- hands on review of assignment 1

## courseMaterial.**Objective**

- course material
  - *explore a little deeper the topics covered in course materials 1*

| | | |
|---|---|---|
| more.Strings | more.Numbers | more.Booleans |
| more.Arrays | more.Operators | more.Conditionals |
| more.Functions | | |

  - *new topic:  self executing function, loops*
  - *practice all the new materials*
- assignment
  - *fine tune the concepts from the course materials*

## courseMaterial.**2**

## more.**Strings**

*( more than what was in WPF :) )*

# more.Strings

| method | description |
|---|---|
| charAt() | Returns the character at the specified index. |
| charCodeAt() | Returns a number indicating the Unicode value of the character at the given index. |
| concat() | Combines the text of two strings and returns a new string. |
| indexOf() | Returns the index within the calling String object of the first occurrence of the specified value, or -1 if not found. |
| lastIndexOf() | Returns the index within the calling String object of the last occurrence of the specified value, or -1 if not found. |
| localeCompare() | Returns a number indicating whether a reference string comes before or after or is the same as the given string in sort order. |
| length | Returns the length of the string. |
| match() | Used to match a regular expression against a string. |

# more.Strings

| method | description |
|---|---|
| **replace()** | Used to find a match between a regular expression and a string, and to replace the matched substring with a new substring. |
| **search()** | Executes the search for a match between a regular expression and a specified string. |
| **slice()** | Extracts a section of a string and returns a new string. |
| **split()** | Splits a String object into an array of strings by separating the string into substrings. |
| **substr()** | Returns the characters in a string beginning at the specified location through the specified number of characters. |
| **substring()** | Returns the characters in a string between two indexes into the string. |
| **toLocaleLowerCase()** | The characters within a string are converted to lower case while respecting the current locale. |

# more.Strings

| method | description |
|---|---|
| toLocaleUpperCase() | The characters within a string are converted to upper case while respecting the current locale. |
| toLowerCase() | Returns the calling string value converted to lower case. |
| toString() | Returns a string representing the specified object. |
| toUpperCase() | Returns the calling string value converted to uppercase. |
| valueOf() | Returns the primitive value of the specified object. |

# more.Strings

- string methods

**.charAt(***index***)**

- returns the character at the given index position

```
“James Bond”.charAt(2)          //returns “B”
```

**.toLowerCase()** *or* **.toUpperCase()**

- converts all of the characters in the string

```
“James Bond”.toLowerCase()    //returns “james
bond”
```

## more.Strings

- string methods

  **.indexOf(*string*)**

  - returns the first found index position of the given character set.

```
"James Bond".indexOf("m")          //returns 2
"James Bond".indexOf("Bond")       //returns 6
```

  **.slice(*start_index, end_index*)**

  - returns a new string slice using index positions -  the 2nd value is a "stop"

```
"James Bond".slice(6,7)            //returns "B"
"James Bond".slice(6,8)            //returns "Bo"
"James Bond".slice(6)              //returns "Bond"
```

# more.Strings

- string methods

  `.split(separator, limit)`

  - creates an array by splitting the string using the given character set
  - *limit* is an optional parameter, limiting the amount of splits.

```
"a,b,c,d".split(",")          // ['a', 'b', 'c', 'd']
"a-b-c-d".split("-", 3)       // ['a', 'b', 'cd']
```

courseMaterial.**2**

# more.**Numbers**

*( more than what was in WPF :) )*

## more.Numbers

| method | description |
|---|---|
| constructor() | Returns the function that created this object's instance. By default this is the Number object. |
| toExponential() | Forces a number to display in exponential notation, even if the number is in the range in which Javascript normally uses standard notation. |
| toFixed() | Formats a number with a specific number of digits to the right of the decimal. |
| toLocaleString() | Returns a string value version of the current number in a format that may vary according to a browser's locale settings. |
| toPrecision() | Defines how many total digits (including digits to the left and right of the decimal) to display of a number. |
| toString() | Returns the string representation of the number's value. |
| valueOf() | Returns the number's value. |

## more.Numbers

- number methods

**.toFixed(*number*)**

- formats a number with a specific number of digits to the right of the decimal

```
var num = 45.7896
console.log(num.toFixed(2));      //returns 45.78
```

## courseMaterial.**2**

## more.**Booleans**

*( more than what was in WPF :) )*

## more.Booleans

- truthy / falsy
  - javascript always tries to interpret values if it can - this results in things that are equal to false *(falsy),* but are not the same thing as **false**

### falsy values

- false
- null
- undefined
- “ ”
- 0
- NaN

### truthy

- any other value is considered “truthy”

```
var myStr = “a string”;
if (myStr) {
   // evaluates to true
};
```

## more.Booleans

- truthy / falsy

```
var myNum = 0;
if (myNum) {
   // evaluates to false;
};
```

```
var myStr = “”;
if (myStr) {
   // evaluates to false
};
```

```
var myBool = false;
if (myBool) {
   // evaluates to false
};
```

# more.Booleans

- truthy / falsy
  - if you try to reference a variable that does not exist, javascript identifies missing values as **undefined**
  - this is also the default value for any variable, such as:

```
var a, b, c;
alert(a);   //alerts undefined
```

  - **null** exists in javascript, but incorrectly - instead, check for **undefined** values
  - IF you wanted a null you would have to specifically set a  variable to null

```
var a = null;
alert(a);   //alerts undefined
```

## courseMaterial.**2**

## more.**Arrays**

*( more than what was in WPF :) )*

## more.Arrays

| method | description |
|---|---|
| **concat()** | Returns a new array comprised of this array joined with other array(s) and/or value(s). |
| **every()** | Returns true if every element in this array satisfies the provided testing function. |
| **filter()** | Creates a new array with all of the elements of this array for which the provided filtering function returns true. |
| **forEach()** | Calls a function for each element in the array. |
| **indexOf()** | Returns the first (least) index of an element within the array equal to the specified value, or -1 if none is found. |
| **join()** | Joins all elements of an array into a string. |
| **lastIndexOf()** | Returns the last (greatest) index of an element within the array equal to the specified value, or -1 if none is found. |

# more.Arrays

| method | description |
|---|---|
| map() | Creates a new array with the results of calling a provided function on every element in this array. |
| pop() | Removes the last element from an array and returns that element. |
| push() | Adds one or more elements to the end of an array and returns the new length of the array. |
| reduce() | Apply a function simultaneously against two values of the array (from left-to-right) as to reduce it to a single value. |
| reduceRight() | Apply a function simultaneously against two values of the array (from right-to-left) as to reduce it to a single value. |
| reverse() | Reverses the order of the elements of an array -- the first becomes the last, and the last becomes the first. |
| shift() | Removes the first element from an array and returns that element. |

## more.Arrays

| method | description |
|---|---|
| slice() | Extracts a section of an array and returns a new array. |
| some() | Returns true if at least one element in this array satisfies the provided testing function. |
| toSource() | Represents the source code of an object |
| sort() | Sorts the elements of an array. |
| splice() | Adds and/or removes elements from an array. |
| toString() | Returns a string representing the array and its elements. |
| unshift() | Adds one or more elements to the front of an array and returns the new length of the array. |

## more.Arrays

- array methods

`.join(string)`

  - converts all the elements of an array into strings, and concatenates those strings together.

```
[‘a’, ‘b’, ‘c’].join(“,“);          //returns “a,b,c”
```

`.reverse()`

```
[‘a’, ‘b’, ‘c’].reverse();          // [‘c’,’b’,’a’]
```

## more.Arrays

`.slice( start_index, end_index )`

- if only the **start** position is provided, it will begin there and continue to the end of the whole array.
- if an **end** is specified, it will go up to but not include the end position
- negative numbers can be used to target the end of the array

```
myArr = [1,2,3,4,5];
myArr.slice(0, 3);          // returns [1,2,3]
myArr.slice(3);             // returns [4,5]
myArr.slice(1, -1);         // returns [2,3,4]
```

## more.**Arrays**

**.push( )**

- array.**push()** will append one or more new elements to the **end of an array** - if you assign this to a variable, the variable will equal the length of the array

```
myArr = [“Joe”, “Kid”];
myArr.push(“Mike”, “Tony”);   //returns [“Joe”, “Kid”, “Mike”,
“Tony”]
```

**.pop( )**

- array.**pop()** will **delete the last element** inside the array

```
myArr.pop( );     //returns [“Joe”, “Kid”, “Mike”]
```

## more.**Arrays**

**.unshift( )**

- array.**unshift( )** will append one or more new elements to the ***beginning* of an array** - similar to how push( ) works

```
myArr = [“Joe”, “Kid”];
myArr.unshift(“Mike”, “Tony”);    //returns [“Mike”, “Tony”, “Joe”,
“Kid”]
```

**.shift( )**

- array.**shift( )** will **delete the *first* element** inside the array, the opposite of pop( )

```
myArr.shift( );     //returns [“Tony”, “Joe”, “Kid”]
```

## more.Arrays

`.sort( )`

- array.**sort( )** sorts the elements of the array based on a *comparison function* - if no comparison is provided, it is sorted alphabetically (**a** to **z**)

```
myArr.sort( );
```

- the below will not sort numbers correctly!
- for more complex sorting, we'll need a *comparison function*

```
myArr.sort( function(){} );
```

# more.Arrays

`.sort( )`

- with a comparison function, we'll pass "a" and "b" as arguments - the comparison function will store 2 elements of the array into **a** and **b** and use a **return** comparison, that is either **< 0**, **> 0**, or **= 0**.

```
myArr = [3, 1, 5, 4];
myArr.sort( function(a,b){
    return a-b;
} );
```

- **< 0** will put a before b
- **> 0** will put b before a
- **0** will not change their position from each other.

- in this example, it will sort the array by ascending numbers (a - b) // 1,3,4,5
- **return b - a** would return descending instead

## courseMaterial.**2**

## more.**Operators**

*( more than what was in WPF :) )*

## more.Operators

- typeof operator
  - **typeof** is a unique operator that takes a variable and returns its data type as a lowercase text string - the data types are:
    - **“undefined”**
    - **“boolean”**
    - **“number”**
    - **“string”**
    - **“function”**
    - **“object”** *(matches arrays too)*

```
var myStr = “jamesBond;
alert( typeof myStr );
//returns ”string”
```

```
var myNum = 5;
alert( typeof myNum );
//returns ”number”
```

## more.Operators

- typeof operator
  - if we need to determine if a variable exists, we can do the following:

```
if ( typeof myNum === “undefined” ){
   //myNum is not set
};
```

```
if ( typeof myString === “undefined” ){
   //myString is not set
}else{
   //myString is set
};
```

courseMaterial.**2**

## more.**Conditionals**

*( more than what was in WPF :) )*

## more.Conditionals

- switch statement
  - “**if**” conditionals are perfect for comparing several sets of varying conditions
  - switch statements will evaluate a single *conditional expression* and then perform an equality check against possible **cases**
  - let’s look at an “if” statement being performed by a “switch” instead...

## more.Conditionals

```
switch ( myArray[6] ){
  case 9:
    // code
    break;
  case 8:
    // code
    break;
  case 7:
    // code
    break;
  default:
    // code
    break;
};
```

courseMaterial.**2**

## more.**Functions**

*( more than what was in WPF :) )*

## more.Functions Notes

- **function** returning a boolean, using conditionals
  - only one return statement in a function will ever be executed - but this doesn't restrict functions to only having a single return
  - if a function has **no return** statement, or uses a return without a value, the function automatically returns the value *undefined*

```
function functionName(){
  if(condition){
    return true;
  }else{
    return false;
  }
};
```

# more.Functions Notes

- **function** returning multiple values **using an array**

```
function functionName(){
  return [“ferrari”, “lambo”, “vwBug”]
};
var myList = functionName();  //will return the array of
values
```

- **function** directly within some other code

```
function functionName(){
  return “ferrari”;
};
var msg = ‘jamesBond drives a ‘ + functionName()’;
//will return the “jamesBond drives a ferrari”
```

## more.Functions

- self.Executing function
  - an anonymous function that is run automatically as soon as it is defined

```
(function(){    //this is a basic function which includes () at end
   //code goes here
})();               //the () tells the function to run immediately
```

courseMaterial.**2**

## javascript.**Loops**

# javascript.**Loops**

- loops - general information
  - javascript will be utilizing the same loops that most programming languages share: **while** and **for**
  - loop/repeat a block of code until a *condition* is met
  - the most common use of loops is to cycle through all the values of an **array** or other forms of data set *(such as objects)*
  - a **counter** is needed which is a simple numeric variable that increases or decreases
  - the *condition* that a loop checks for is usually against the **counter** variable
  - common variable names for counters are **i** and **x** - most developers reserve these names for this purpose

## javascript.**Loops**

- while ( )
  - the **while** loop is the simplest conditional loop

```
while ( condition ) {
    //code goes here
}
```

  - before the code is executed, the condition is checked
  - if it evaluates to *true*, the code is run, and the loop returns to the condition check again
  - it will repeat this process until the condition becomes *false*, which will then skip the code and exit the loop

# javascript.**Loops**

- for ( )
  - a **for** loop is an increment-based loop, where the increment is part of the condition
  - there are 3 statements inside the condition of a **for** loop:

```
for ( var i = 0;  i < 5;  i++ ) {
   //code goes here
};
```

  - first a counter variable is initialized
  - second the "condition check"
  - and third, increment the variable and perform the code till the end of the loop
  - each is separated by semicolon, just like normal statements

# javascript.**Loops**

- array.**Length**
  - if we wanted to cycle through all the values in an array, we need to know how many values are in the array
  - this is where the **.length** property comes in - this will return a **number** value, equal to the number of elements in the array
  - an array's numeric **index** begins at 0
  - in the example below, the last index of the array would be 4 - the **.length** property returns the *count*, which would be 5

```
var myNums = [1, 2, 3, 4, 5];
alert( myNums.length );    //returns a "5"
```

# javascript.**Loops**

- for ( )
  - the for loop is the most commonly used in programming, since the increment makes it easy to cycle through arrays or objects.
  - to cycle through each index of an array, we could use the **.length** property, and use the counter **i** as the index, such as:

```
for ( var i=0; i<myNums.length; i++){
  alert( myNums[i] );
};
```

  - however, this is not the most efficient way...

## javascript.**Loops**

- for ( )
  - depending on the size of an array, it can be more efficient to save the array length in a variable, inside the first statement

```
for (var i=0, j=myArr.length; i<j; i++){
  alert( myNums[i] );
};
```

# javascript.**Loops**

- breaking.**Loops**
  - in some situations, you may need to force a loop to stop
  - by using the **break** statement, any loop will stop running at the break point, and perform no more iterations

```
for (var i=0, j=myArr.length; i<j; i++) {
  if (condition) {
    break;
  };
  //...;
};
```

# javascript.**Loops**

- continue.**Loops**
  - while the **break** statement will stop a loop and exit it, the **continue** statement will stop a loop's current iteration, and continue on to the next iteration

```
for (var i=0, j=myArr.length; i<j; i++) {
  if (condition) {
    continue;
  };
  //...;
};
```

## Assignment / Goal 2

- Goal2: Assignment: JavaScript Practice
  - Log into FSO.  This is where all your assignment files will be located as well as Rubrics and assignment instructions.
- Goal2: Assignment: The Duel - Part II
  - You will use the same files you used for the Duel - Part 1, for this assignment. See FSO for the assignment instruction.
- Commit your completed work into GitHub
  - As part of your grade you will need at least 6 reasonable GIT commits for each assignment.
- In FSO there is an announcement with “Course Schedule & Details” in the title, in that announcement you will see a “Schedule” link which has the due dates for assignments.