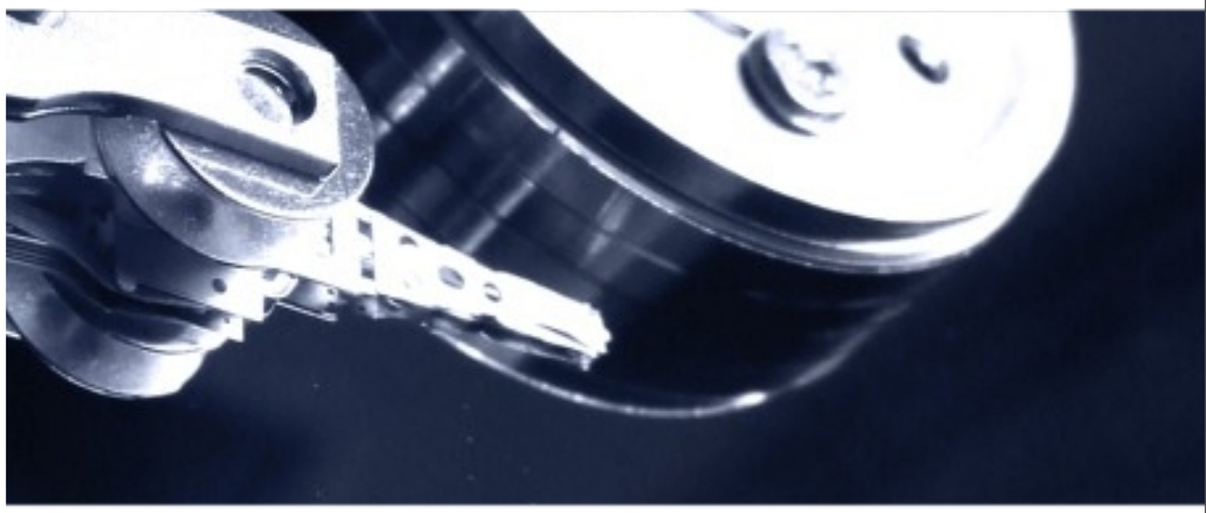
FULL SAIL
UNIVERSITY

# web design and development



## programming for web applications 1

courseMaterial.3

courseMaterial.**3**

# goal2.**Recap**

# goal2.**Recap**

- topics covered are as follows:

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

  - *new topics were:  self executing function & loops*

- hands on review of the previous 2 assignments

# courseMaterial.**Objective**

- course material
  - *debugging*
  - *scope / closure*
  - *practice all the new materials*
- assignment
  - *fine tune the concepts from the course materials*

courseMaterial.**3**

# debugging.**Errors**

# debugging.**Errors**

- debugging.**Tools**
  - always keep a browser debug window open
    - **firefox:** use both ***firebug*** and "***Inspect Element with Firebug***"
    - **chrome:** use both ***firebug*** and ***inspect***
  - **http://wddbs.com/jshero/**
  - **http://www.jshint.com/**
  - **http://www.jslint.com/**

# debugging.**Errors**

- types of errors
  - there are 3 categories of errors:
    - **syntax**
    - **runtime**
    - **logic**

# debugging.**Errors**

- syntax.Errors
  - these errors (also known as parsing errors) occur when the programmer types something incorrectly, such as:
    - forgetting to close a *string* with quotes, or escaping quotes with \
    - forgetting to separate array values with a comma
    - missing other necessary syntax characters such as ( ) , { }

# debugging.**Errors**

- syntax.Errors

```
var fsStudent = {
    age: 22,
    career: "Web Dev"
 ;
```

# debugging.**Errors**

- syntax.Errors

```
var fsStudent = {
    age: 22
    career: “Web Dev”
};
```

**Error:** missing } after property list

Source File: file:///Users/msmotherman/Desktop/SWA%20Courses/SWA-1/Lecture%20

```
career: "Web"
```

# debugging.**Errors**

- syntax.Errors
  - syntax errors will usually be displayed immediately ***before*** a script's execution
  - any single script is an individual set of code using <script> tags - the code below indicates two different sets of sequential scripts
  - *note that both scripts are part of the same document, and share variables*

```
<head>
    <script type="text/javascript" src=".."></script>
    <script type="text/javascript" src=".."></script>
</head>
```

# debugging.**Errors**

- syntax.Errors
  - syntax errors will also completely prevent the execution of that **script**'s code *(we'll only see the first syntax error, there could be more to follow)*, however, other scripts will still attempt to run as normal

```
<script type="text/javascript">
    alert("Error here! );
    // this entire script will be skipped because of 1 typo
</script>

<script type="text/javascript">
    alert("I'm error free!");
</script>
```

# debugging.**Errors**

- runtime.Errors
  - runtime errors are exceptions that occur ***during*** the code's execution
  - because these are not syntax issues, they will not cause an error until the problematic line of code is executed
  - once the error occurs however, script execution will stop
  - the most common cause of runtime errors is when a variable or function does not exist (or the reference is misspelled), or when an object being accessed is invalid
    - *most often, if you think your logic is correct, then it is a spelling typo*

**Error:** this.doThis is not a function

Source File: file:///Users/msmotherman/Desktop/SWA%20Courses/SWA-1/Labs/lab2/c

# debugging.Errors

- runtime.Errors

```
<script type="text/javascript">
    var myVar = "yay";
    alert(mvar);
    // this line of code will not be run
</script>
```

# debugging.**Errors**

- logic.Errors
  - logic “errors” are the apparent lack of success *(the desired effect doesn’t happen)*
  - this is the most difficult type of error to find - this type of problem does not return an actual error because no syntax or runtime exception has occurred
  - the problem is simply in the programmer’s logic - can you find the mistake?...

```
var myArr = [“1”, 2, true];
for(var i = 0; i < myArr.length; i++){
  if(typeof myArr[i] = “number”){
    alert(“Yay, a number!”);
  };
};
```

# debugging.**Errors**

- debugging techniques
  - use a good text editor or IDE
  - use multiple browsers (they display different error messages)
  - keep the browsers console open at all times
  - use console.log EXTENSIVELY!!!!!!!
  - write JavaScript in external files
  - take a break!

# debugging.**Errors**

- suggestion - using console.log
  - because of the nature of JavaScript, I recommend constant rigorous testing of your code ***as*** you write it
  - test after finishing a *code block that has an impact on the application* - use console.log EXTENSIVELY

```
console.log(value);
```

  - console.log a string to determine if your script is following the program's logical flow

```
console.log("string");
```

# debugging.Errors

- example - using console.log

```
var myTest = function(){
   //console.log a string to determine if the script gets into this function
   console.log ('in myTest Function');
   return "returning.This";
};
var output = myTest();
//console.log the output variable:
//1. to see if the function ran & exited correct
//2. to see if the output is correct
console.log(output);
```

# debugging.**Errors**

- error.Handlers
  - in javascript, we have an additional statement for error handling
    - the **try, catch, throw** statement
    - **try** - use to test a block of code for errors
    - **catch** - use to let you handle the error
    - **throw** - use to let you create custom errors
  - this technique will catch **runtime errors** *only.*
  - the primary purpose is in production-use - when in development mode, it is usually easier to refer to an error console (such as firefox), or use alerts to find problems

# debugging.**Errors**

- try.. catch.. throw..
  - a **try** statement allows you to try out a block of code. If anything causes an exception during that block, the code is aborted, and the **catch** statement block is executed instead

```
try {
   // code to try  throw “error message”;
} catch (error) {
   // code to run on error throw “error message”;
}
```

  - this can be useful in preventing your end users from seeing errors or disruptions

# debugging.Errors

- try.. catch.. throw..

```
try
  {
    //getElementByID will be reviewed later in the course
     var x=document.getElementById("demo").value;
    if(x=="")     throw "empty";
    if(isNaN(x)) throw "not a number";
    if(x>10)      throw "too high";
    if(x<5)       throw "too low";
  }
catch(err)
  {
    var y=document.getElementById("mess");
    y.innerHTML="Error: " + err + ".";
  }
}
```

courseMaterial.3

# variable.Scope

# variable.**Scope**

- what is scope?
  - scope controls the visibility and lifetime of variables and parameters - it defines where the variable is created, and where it can be accessed
  - JavaScript uses **lexical scoping**
    1. ***a "scope level" will check parent levels for variables,*** *and*
    2. ***a new "scope level" in javascript can only be made with a function***
  - if a variable doesn't exist in the current scope, through lexical scoping it will try to find the variable in a parent block
  - local variables override higher scoped variables

# variable.**Scope**

- what is scope?
  - all variables defined in a code block using {...} are not visible from outside of the block - the code block is normally defined in a function
  - variables defined in the code block are released when execution of the block is finished
- example using a 'global variable'

```
var globalVar = "jamesBond";
function functionName(){
  var localVar = "maryBond";    //only available within the function
  //you can use globalVar in this function
}
 //you can use globalVar outside of the function
```

# variable.**Scope**

- what is scope?
  - the 1st **num** is a **global** variable & will live for the life of the program
  - the 2nd **num** is a **local** variable & will last only as long as the function

```
var num = 1;          //this is the 1st num

function myCounter ( ) {
    var num = 50;   //this is the 2nd num
};
```

# variable.**Scope**

- examples of ‘local variable’ (inside a function)

```
function functionName(var1){
   //you can use var1 in this function
}
//you cannot use var1 after the function
```

```
function functionName(){
   var localVar = “jameBond”;
   //you can use localVar in this function
}
//you cannot use localVar after the function
```

# variable.Scope

- scope - common variable scope mistake

```
function functionName(){
  shouldBeLocalVar = “jamesBond”;

  //without the var, this is actually a Global variable!
  //you can use shouldBeLocalVar in this function
}
//you can use shouldBeLocalVar outside of the function
```

# variable.**Scope**

- scope - global & local variable with the same name
  - local variable takes precedence
  - global variable become invisible

```
var name = “james”;
console.log (‘name: = ‘, name);      //name: = “james”

function functionName(name){
  name = “Bond”;
  console.log (‘name: = ‘, name);   //name: = “Bond”
}

functionName(name);              //passing “james” into the function
console.log (‘name: = ‘, name);    //name: = “james”
```

courseMaterial.**3**

# javascript.**Closure**

# javascript.Closure

- what is closure?
  - think of closure as a function call with a memory
  - **a closure is a function tied to the scope in which it was created** - a closure function can make use of the variables that existed (in the same scope) when the function was created
  - you have a closure when:

1. one function is defined within another function
2. the inner function references variables that exist in the outer function (including the outer function's parameters)
3. the inner function will be called after the outer function has stopped executing

# javascript.**Closure**

- what is closure?
  - **the hallmark of closure**: the local variables that were available to the function when the function was defined are still available to the function even when it is called at a later time
  - in a normal, non-closure situation, the function bar (the example on the following slide) will no longer be available once the function execution has completed - without closure the bar function will cease to exist after the onload function has executed all its command

# javascript.Closure

- what is closure ?

```
function foo(x) {
    var tmp = 3;
    function bar(y) {
        var barVar = 100;
        console.log ('Closure :', x + y + (++tmp));
    }
    bar(10);
}
foo(2)
```

# javascript.Closure

- what is closure?
  - the previous slide will always console.log 16, because bar can access the x which was defined as an argument to foo, and it can also access tmp from foo
  - that is a closure - a function doesn't have to return in order to be called a closure - simply accessing variables outside of your immediate lexical scope creates a closure
  - the inner function will close the variables of foo before leaving

# Assignment / Goal 3

- Goal3: Assignment: Debug
  - Log into FSO.  This is where all your assignment files will be located as well as Rubrics and assignment instructions.
- Commit your completed work into GitHub
  - As part of your grade you will need at least 6 reasonable GIT commits while working on the 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.