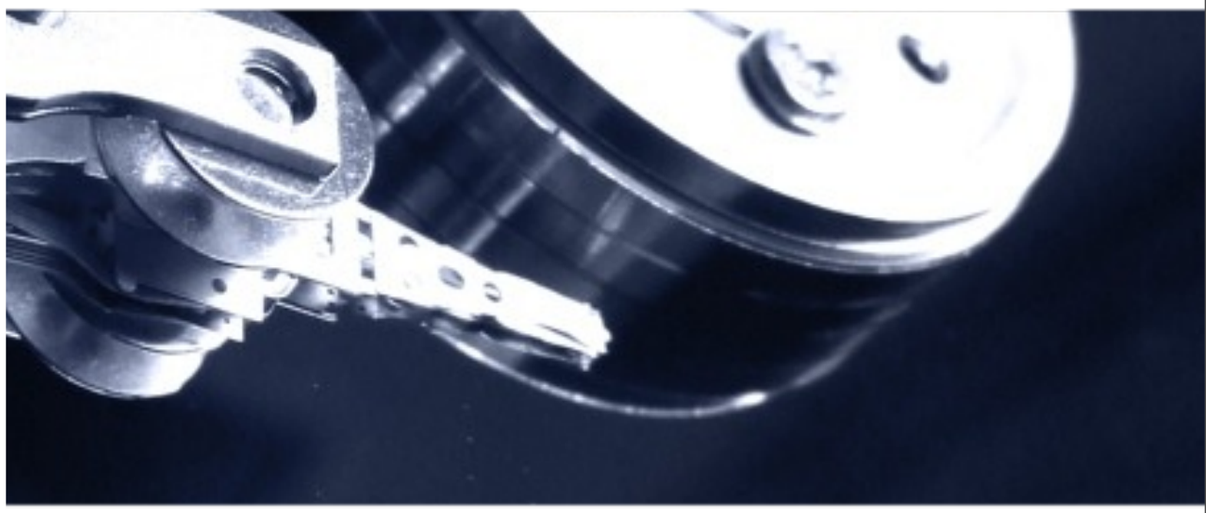
FULL SAIL
UNIVERSITY

web design and development



programming for web applications 1

PWA 1 / WDD244

courseDirector

lyndon.Modomo
lmodomo@fullsail.com

---

labSpecialist

JD.Benitez
ljdbenitez@fullsail.com

Vid.Strickland
jstrickland@fullsail.com

general.Info

- silence cell phones
- only bottled water allowed in classrooms
- all deadlines are final
- **11 day course**
  - you can miss 8 hours without documentation
  - you can miss a maximum of 16 hours *(documentation required)*
  - **no** *make-up assignment will be given for missing hours*

- either two 15-minute breaks or one 30-min during lecture
- in labs, no official break
- if you arrive 15 minutes after the start of lecture or lab, you are 2 hrs late
- if you will miss class, contact me ahead of time - **be proactive!**

general.**Info**

- **days:** tuesday, thursday, saturday
- **lecture:** 5:15 pm to 8:15 pm
- **lab:** 9:05 pm to 1:00 am (Changed Location: FS3C-115)
- **total days:** 11 days
  - 1st week: tuesday, thursday, saturday
  - 2nd week: tuesday, thursday, saturday
  - 3rd week: tuesday, thursday, saturday
  - 4th week: tuesday, thursday

about.**Course**

The **programming for web applications I** course trains students in the technologies used to create dynamic content for the Web using client-side programming. This course builds upon the coding and logic concepts learned in the Web Programming Fundamentals course, continuing the use of JavaScript. Students are also shown more advanced concepts such as data structures and key algorithms.

- client-side web programming
- client-side algorithms
- web data and validation
- programming for code libraries & reuse

- lecture
  - *exploration, practice, group coding*
- lab
  - *fine tune the concepts from lectures materials*

lecture.Tips

- lectures are fast-paced and concept-heavy
- keep coding terminology in mind
- most code examples in the lecture slides look like:

```
var name = function( variable1, variable2 ){ statements; }
```

- assignments are available in FSO
- course activities & presentation slides are available in GitHub.com - you will have access to this forever! - IF you have not used Git yet, we will review it later today
  - **https://github.com/lmodomo/pwa1**

lab.**Tips** / activity.**Tips**

- you will submit **all** assignments and activities via your Git repository - IF you have not used Git yet, we will review it later today:
  - **https://github.com/lmodomo/pwa1/lastNameFirstName**

month at a glance

week	day	lecture	lab
1	1	overview of this course / review ALL of WPF	assignment
	2	additional info for WPF topics (strings, numbers, booleans, operators, conditionals, functions) / new items: loops & arrays	assignment
	3	debugging / scope & closure	***practical***
2	4	basic objects / DOM introduction / events & callbacks	assignment
	5	regular expressions / math & date methods	assignment
	6	reflection / build something	***midterm practical***
3	7	object concepts / functions as objects / THIS / prototype	assignment
	8	custom library	assignment
	9	JSON / advanced arrays / data structural pattern	assignment
4	10	data visualization using canvas	assignment
	11	***final practical exam***	

grading at a glance

item	course %
labs 1, 2, 4, 5, 7, 8, 9, 10	10%
week 1's practical assessment	15%
midterm practical exam	30%
final practical exam	35%
professionalism	10%

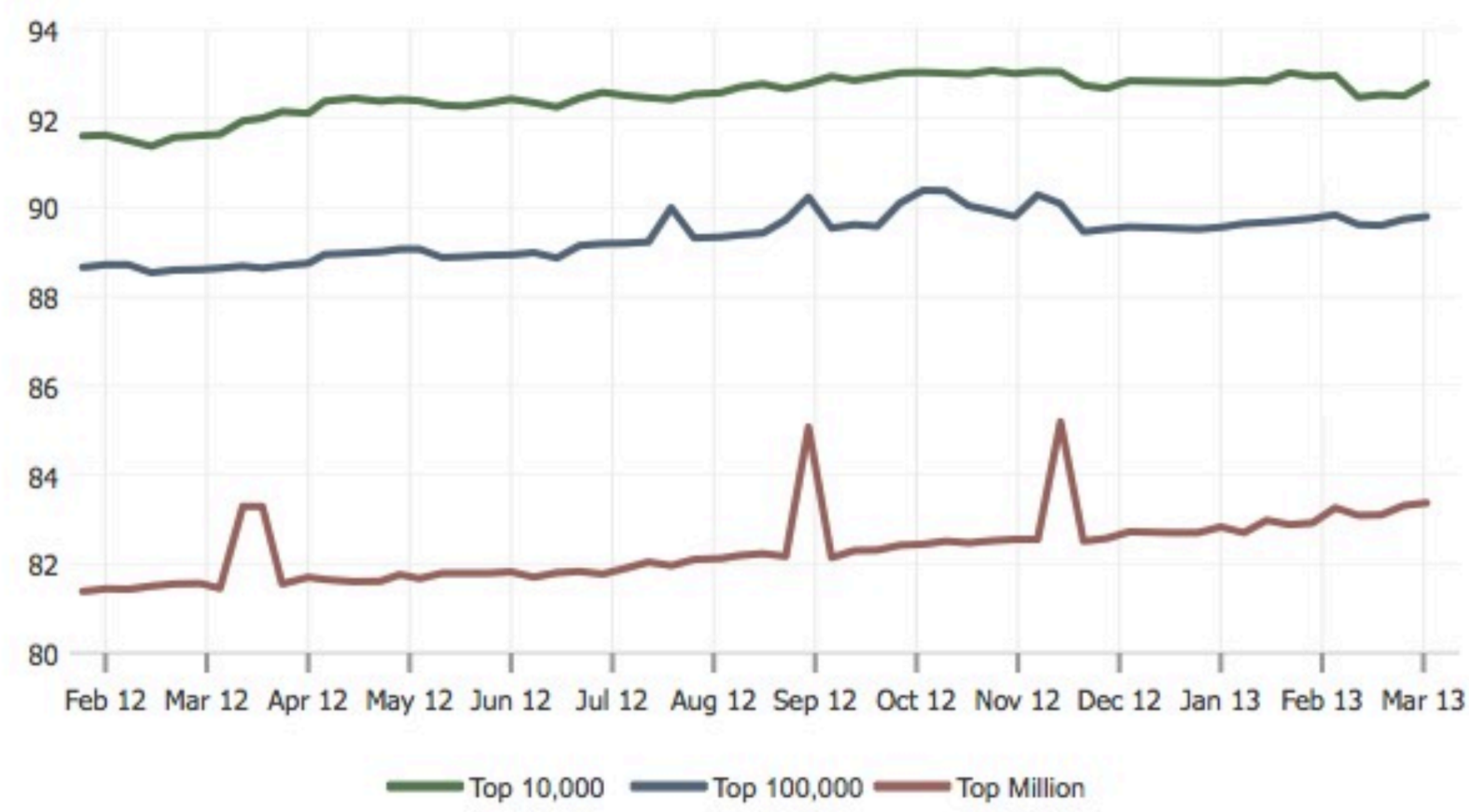
what is **javascript?**

javascript is the scripting language of the web

trends.builtwith.com

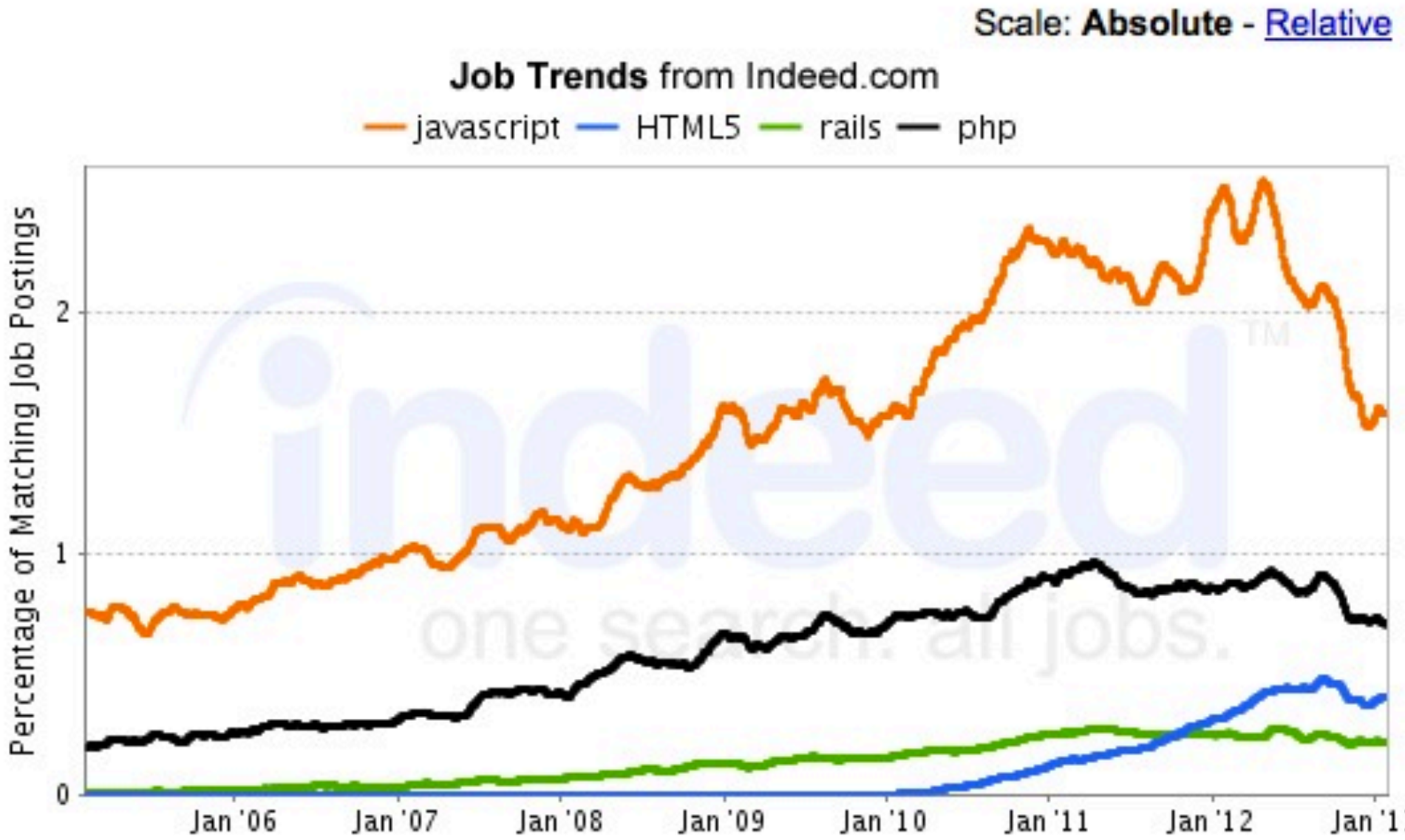
Javascript Usage Trends

JavaScript is a scripting language most often used for client-side web development. Its proper name is ECMAScript, though "JavaScript" is much more commonly used. The site uses JavaScript.



javascript, HTML5, rails, php Job Trends

Scale: **Absolute** - Relative



Indeed.com searches millions of jobs from thousands of job sites.
This job trends graph shows the percentage of jobs we find that contain your search terms.

Find Javascript jobs, Html5 jobs, Rails jobs, PHP jobs

brief.History

- what is javascript?
  - **javascript is a client-side, object-oriented language.**
  - *it is executed from the user's browser, as part of a web document.*
  - 3 layers of a **client-side web document**:
    1. **content / structure:** the core html / xml-like structure
    2. **presentation:** CSS styling layer
    3. **behavior:** javascript layer is what we call the behavior layer. This allows us to attach interactivity to the document, using the browser

brief.History

- how does JavaScript work?
  - every language has a *medium*, the output graphics renderer
    - C++/Cocoa/etc: ***the operating system***
    - Flash: ***the Flash player plugin***
    - JavaScript's medium is the **browser**
      - **good**: *JavaScript needs no plugin, it is native to all browsers*
      - **bad**: *anyone here enjoy IE6?*
    - JavaScript code controls the document (such as html), and the browser it is being rendered by

brief.History

- what can JavaScript do for our sites?
  - let's look at some examples of JavaScript in action:
    1. http://tweetping.net/
    2. http://isotope.metafizzy.co/
    3. http://www.nihilogic.dk/labs/wolf/
    4. http://www.erpoyart.com/jsdyn/

Web Programming Fundamentals (WPF)

review

WPF review

version.**Control**

version.Control

- why version control?
  - change control - gives us a log of edits to a file and why (label / tag)
  - performs automatic backups
  - rollback to previous versions of a file or directory (long & short term)
  - allows us to create a ‘sandbox’ so we can experiment
  - team benefits
    1. keep team members up to date with synchronization
    2. lends to accountability - know who made changes and why
    3. allows for conflict detection and resolution

version.Control

- 6 version control concepts

1. **repository** - storage for your files and history (normally on a server)
2. **working set** - current state of the files on a **local machine**
3. **add** - adds the “new files” from the working set into the repository
4. **commit (check-in)** - adds the “changed files” from the working set to the repository
5. **update (check-out)** - adds changes from the repository to the working set
6. **tag / label** - tag the current state of the repository for the future

version.Control

- what is Git?
  1. a popular version control system (VCS)
  2. primarily a source code management system (SCM)
  3. **not** useful for tracking non source code files
     - pictures, videos, music, work documents, spreadsheets, PDFs

version.Control

- what is GitHub?

1. a web based hosting service for software development projects that use the Git revision control system
2. GitHub has both paid plans for private repositories, and free accounts
3. as of May 2011, GitHub was the most popular open source code repository site
4. GitHub is what you will be using throughout your journey in the WDD degree at Fullsail
5. if you do not have a GitHub account them please sign up for a free edu account at: **https://github.com/edu**
6. every student's account should be an edu **private** account - keep all class work repositories private at all times

activity 1.1: setting up & using Git with SmartGit

- for those who have not used Git yet:
  1. setup a GitHub account at: **https://github.com/edu**
  2. install Git
  3. create a repo on GitHub
  4. install and configure SmartGit
  5. pull remote repo and create a branch from the local master
  6. review “commit”, pull” and “push”
  7. review merge conflict
- Git setup for this course:
  1. in FSO: see and process items in the activity entitled “Wk1: Activity: GIT UP”

WPF review

getting.**Started**

getting.Started

- what is an editor?
- select a code editor or IDE for development
  - sublime2: http://www.sublimetext.com/
  - coda: http://panic.com/coda/
  - textWrangler: http://www.barebones.com/products/textwrangler/
  - adobe dreamweaver: http://www.adobe.com/products/dreamweaver.html
  - eclipse: https://eclipse.org
- DO NOT use a Word Processor!!!

getting.Started

- document behaviors
  - JavaScript is included as part of a web document *(such as an html file)*
  - the code is placed in a **<script></script>** tag element
  - the script tag will also require a mime-type declaration of “text/javascript”

getting.Started

- linking JavaScript to html
  - in the head, the script's code is run before the body is rendered

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

  - in the body, the script's code is run in the order of the elements around it

```
<body>
   <script type="text/javascript"></script>
</body>
```

getting.Started

- linking JavaScript to html : external scripts
  - the method we'll be using most often is to have all your script information in an external document (i.e. **file.js**) with a .js extension
  - makes archiving project code easier and separates your code from the document
  - add an **src** attribute to the script tag pointed to the file's url
  - common practice is to use a "js" folder to contain your script files

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

getting.Started

- loading order
  - in our documents, loading order is crucial...
    - *<head> runs before <body>*
    - *scripts are executed in the order they are placed*
  - inside the body, all document elements load sequentially (line by line)
  - a common rule of thumb is to put the <script> tags at the end of the <body> tag

activity 1.2: setting up your dev. environment

- lecture activity
  - pwa1 repo: navigate to the directory “Lecture Activities/Wk1”
  - copy the directory “wk1_activity_setup” in your local repo to directory “firstNameLastName/wk1/activities/”
  - ignore all the html code in the index.html file - create a script tag (<script>) at the end of the body tag (</body>), using an external file:
    `<script type="text/javascript" src="js/main.js"></script>`
  - in the main.js file, add an alert: **`alert(“Hello World!”);`**
  - if the “Hello World!” alert box pops up then your development environment is setup correctly
  - then add: **`console.log(“Hello World”);`**

WPF review

code.**Review:** reminders1

code.Review: reminders1

- printing to the “screen” - alert method
  - a common method for testing values in javascript is the **alert(***expression***)** method
  - this will create a modal dialog box in the browser window containing whatever *literal* you pass:

```
alert(“hello, world!”);
```

  - because this is *modal*, it will halt the execution of your code until the dialog box is closed
  - for this reason, it is one of the most commonly used value testing methods
  - alert is “old school”

code.**Review:** reminders1

- printing to the “console” - console.log
  - because of the nature of javascript, I recommend constant rigorous testing of your code as you write it
  - ideally, test after finishing a code block that has an impact on the application

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

WPF review

code.**Review**: problem solving

problem.**solving**

- problem-solving is the core skill of every developer

  step 1: define the problem inside and out

  a. what **assumptions** are being made

  b. what **constraints** are there

  c. what are the **goals** we are trying to meet?

  d. **visualize** and draw out the problem

  e. **describe** the problem in your own words

problem.**solving**

step 2: choose the part of the problem to solve

a. break the main problem into smaller problems (**sub-problems**)

b. **identify the constraints** for each of the sub-problems

step 3: identify potential solutions

a. there are normally more than one way to solve a problem, **write them down**

b. **create potential solutions** for all the sub-problems

c. will this potential solution work in the real world?

problem.**solving**

step 4: evaluate the solutions and select the best one

a. does the solution **meet the goal**?

b. does the solution **work for all cases**?

c. **rank** the solutions if there are more then one solution

step 5: develop an action plan to implement the best solution

a. how will the implementation process work?

b. crash test the solution (**test, test, test, test, test**....)

code.**Review** : variables & values

code.Review

- declaring and defining variables
  - variable declaration begins with the **var** keyword
  - variable assignment uses the **assignment operator** (**=**)
  - javascript does use ***loose typing***.
    - *this means the coding engine will determine the data type automatically vs declaring what the variable will be (string, number, boolean, etc).*
  - *example: (declaring and defining on one line)*

```
var lastName = “bond”;
```

code.Review

- identifiers
  - the name of a variable is the ***identifier*** (i.e. like lastName) in the previous slide
    - *can have: letters, numbers, underscore, or $*
    - *cannot have spaces*
    - *begin with a number*
    - *is case-sensitive*
    - *should use camelCasing*
    - *make the name meaningful*

```
legal   var jamesBond;
legal   var _007;
legal   var $;
legal   var james007;
```

code.Review

- multiple variables
  - can be declared on the same line, separated by comma
  - can also be declared without values *(meaning they are **undefined**)*

```
var name = “james”, course=”PWA1”,
month=3;
var name, course, month;
var name, course=”PWA1”, month;
var a = b = c = d = 10;
```

code.Review

- values
  - programming produces results and each result value is represented by a **literal**:
    - **string literal:** “any character set can go here”
    - **numeric literal:** 0, 103, 42, 3.145, 20.1, 6.02e23
    - **boolean literal:** true *or* false
    - **array literal:** `[1, true, “3”]`
    - **function literal:** `function(){`*`statements;`*`}`

code.Review

- string literal
  - remember to escape quote characters if the outside quotes are the same:
    - var myStr = “I need some \”quotes\” to be here.”;
  - to combine two strings together (***concatenate)***, use + sign (**the + is an expression**)
    - var name = “james” + “ bond”; *//returns “james bond”*
    - var name = “james” + “ “ + “bond”; *//returns “james bond”*
  - if you add a number to a string, the resulting format is always a string:
    - var myStr = “6” + 2; *//results = “62”*
  - other math operations will result in a number:
    - var myStr = “6” / 2; *//results = “3”*

code.Review

- numeric literal
  - **arithmetic operators: +, -, /, *, % (modulo)**
    - keep in mind **order of operations**: ( ), *, /, +, -
    - what will this equal ?: 4 + (6 * 5 - 10) / 5 = ?
  - you can shorten quick operations with an **assignment operator**: +=, -=, *=, /=
    - myNum = myNum + 5;
    - myNum += 5;
  - you can increment or decrement numbers by 1 with ++ or --
    - myNum++; myNum--;   //**increment / decrement operators**
  - modulo, returns the remainder of an equation
    - 32 / 10 = 2 (can use modulo to determine if a number is an even of odd number)

code.Review

- array literal
  - **arrays** are a unique data type that can hold a collection of *values*
    - in javascript, arrays are constructed by the brackets [and]

```
var myArr = [ ];   //empty array
```

    - arrays can hold any value type, separated by comma

```
var myArr = [“bond”, 007, function(){}, true];
```

code.Review

- array literal
  - arrays are indexed numerically, beginning at 0

```
var myArr = [“bond”, 007, function(){}, true];
            0     1        2             3
```

  - to access or set a value, use an index number inside **[]**

```
alert( myArr[3] );    //returns true (getter)
```

```
myArr[4] = “jamesBond”; //setting a value (setter)
```

code.Review

- arrays with expressions
  - i.e. add all the students up in each classroom

```
var student = [ 36, 5, 100];
```

  - results

```
var total = student[0] + student[1] + student[2]
```

WPF review

code.**Review** : conditionals

code.**Review**: conditionals

- **if** statement
  - the standard conditional of any programming language
  - a way of splitting a linear path of options into multiple possibilities by testing a condition
  - if a specified **condition** evaluates to **true** (*it's either **true** or **false***), then a block of code or a group of **actions** are executed

```
if (condition) {
  //block of code or'actions'
};
```

```
if (true) {
  //I get to run!
};
```

code.**Review**: conditionals

- comparison operators
  - a **condition** is a block of 2 values being compared with a **comparison operator** between them - the *comparison operator* evaluates the comparison and **returns a *true* or *false* boolean**
  - conditions can be used just about anywhere, not just in conditionals and loops

```
var myVar =  5 > 3 ;
alert(myVar);                //alerts true
```

  - most common **comparison operator**:

> (greater than)	**>=** (greater than or equal to)	**!=** (NOT equal to)
< (less than)	**<=** (less than or equal to)	**==** (equal to)

code.**Review**: conditionals

- **if** statement examples
  - the overall *condition* must evaluate to ***true*** for the code block to be run

```
if (1 == 2) {
  alert("ahh never!");
};
```

- a common mistake is to use the **assignment operator** = instead of the ==
- == used for comparing items

code.**Review**: conditionals

- == versus ===
  - == **equality operator**, this performs a loose check to see if the two values can be interpreted as the same value
    - "1" == 1 **returns *true***, because these values can be considered "equal", even though one is an *integer* and the other is a *string*.
  - === **identity operator**, this checks to see if both values are exactly identical, not only in value, but also in datatype
    - "1" === 1 **returns false**, because one is a *string* and the other is not.
    - "1" !== 1 **returns true**
  - we will use the **===** for javascript versus the **==**

code.**Review**: conditionals

- if - else statements
  - if the condition of an **if** statement is not met, then run the code block in the **else** statement
  - used to choose between two blocks of code

```
if (a < b){
   //execute this block of code if a is less than b
}else{
   //execute this block of code
};
```

code.**Review**: conditionals

- if - else if - else statements
  - if the condition of an "**if**" statement is not met, we can use "**else if**" statements to check alternative conditions, and an "**else**" statement if all others return false
  - used to choose between three of more blocks of code

```
if (a < b){
  //execute this block of code if a is less than b
}else if (a < c){
  //execute this block of code if a is less than c
}else if (a < d){
  //execute this block of code if a is less than d
}else{
  //execute this block of code if all the others return false
};
```

code.**Review**: conditionals

- nested conditional statement
  - if the condition of an **if** statement is not met, we can use **else if** statements to check alternative conditions, and an **else** statement if all others return false

```
if (a === b){      //1st IF statement
  //execute this block of code if a is equal to b
  if (a === c){   //nested IF statement : 2nd IF statement
     //execute this block of code if a is equal to c
  }else{          //nested ELSE statement
    //execute block of code if the matching "IF" statement returns false
  }
}else{
    //execute this block of code if the 1st "IF" statement returns false
};
```

code.**Review**: conditionals

- logical operators
  - to test more than one condition at the same time, we'll use **logical operators**
  - logical operators are parsed from left to right, therefore more complex comparisons should always be at the right-side of the equation
    - **&&** is the **AND operator** - used to compare two conditions, and if both result in true, the overall condition is true.
      - ("1" == 1 && "2" == 2) ... *returns true*
    - in an **AND operator**, if the left side returns *false*, then the whole condition is false, and the right side is completely skipped

code.**Review**: conditionals

- logical operators
  - || is the **OR operator** - used to compare two conditions, and if ***either*** results in true, the overall condition is true
    - (“1” == 1 || “5” == 2) ... *returns true*
  - with the **OR operator**, if the left-side evaluates to *true*, then the whole condition is true and the right-side is skipped
  - **!** is the logical **NOT operator** - used to invert a condition

```
if ( !( “2”==2 && “5”==1 ) ) { }    //returns true because of !
```

code.**Review**: conditionals

- name that conditional
  - identify the boolean result (***true*** *or* ***false***) for the following conditions:

```
( ! false )
```

```
( 5 <= 4  ||  0==true )
```

```
( ! ( “5” === 5 && 4 !== “4” ) )
```

```
( 5 < 2  ||  (1 === 1 && ! (2 == “2” || 5 < 2 ) )  )
```

code.**Review**: conditionals

- ternary operator : short-form IF conditional operator ? :
  - the **conditional operator ? :** is a condensed version of an IF statement, which *returns* a statement based on a condition being *true* or *false*

```
condition ? true : false;
```

? *statement* is run on true

: *statement* is run on false

```
if (a<b){
  myVar = a;
}else{
  myVar = b;
};
```

becomes

```
myVar = a<b ? a : b;
```

code.**Review**: conditionals

- ternary operator : short-form IF conditional operator ? :
  - one use would be to determine the **value of a variable**, based on a condition, like:

```
var mood = (smile == true) ? 'happy' : 'sad';
```

  - but the conditional operator can be used almost anywhere to insert a *result* into a block of code...

```
console.log( smile === true ? "happy" : "sad" );
```

```
console.log("I am " + ( smile === true ? "happy" : "sad" ) + "!!");
```

code.**Review**: conditionals

- ternary operator : short-form IF conditional operator ? :
  - it can also be used to perform single statements

```
b < 5 ? i++ : i--;
```

```
(halt == true) ? alert("error") : console.log("error");
```

WPF review

code.**Review** : functions (reusable code)

code.Review

- the 4 types of procedural code

1. **sequential -** some element of instruction that goes step by step in sequence
2. **conditional** - decision making, the ability to branch from sequential code
3. **reusable** - code that is self contained, and can be implemented at different spots in the code (i.e procedure & methods)
4. **repetitive** - code that is written once and then we can tell the system how many times to run the code (i.e loops)

code.**Review**: functions

- functions
  - **functions** provide us a way to save a block of code so we can execute it later and it can be invoked as many times as we want - **reusable code**
  - in javascript, functions are values like all our other data types - **function literal**
  - 2 types for function structures
    - 1. **basic** function structure
    - 2. **anonymous / closure** function structure - named this because there is not a name associated with the function

code.**Review**: functions

- **basic** function structure (also called a **named/defined** function)
  - basic functions are created, then called when it is needed to run
  - the parenthesis allows us to pass parameters/arguments into the function

```
function functionName(){
  //reusable code goes here
};
```

  - invoking the function is as simple as calling the identifier using ( ) - can place this above or below the function above (unlike the anonymous function)

```
functionName();
```

code.**Review**: functions

- **anonymous** function structure
  - anonymous functions are created and given a name as the code is run
  - **this structure is more common** and used more when we get to objects, therefore we will use this structure for this course, but it is important to know both forms
  - **must** define/declare the function before calling/invoking the function - unlike the basic named structure where you can call a function before or after the code

```
var functionName = function(){
   //reusable code goes here
};
```

```
functionName();     //to call or invoke the function
```

code.**Review**: functions (reusable code)

- function literal - example
  - example: create a function to increment a number literal
    - first, we need to initialize a numeric variable to store our counter, then create our function to increment our numeric variable by 1, then execute our function

```
var myctr = 1;                          //initialize our variable to 1
var myCounter = function(){
  myctr++;                             //increments our number by 1
  alert(myctr);                         //outputs the new result
};

myCounter();                           //will output "2" to the page
myCounter();                           //will output "3" to the page
```

code.**Review**: functions (reusable code)

- function literal: **passing data into a function**
  - functions would be very limited if they could only execute the same thing every time
  - we can create dynamic results by passing ***identifiers / variableNames*** into our function - more commonly know as **parameters**
    - if we want to pass more than 1 argument, separate the arguments w/ a comma... - can have any number of parameters passed into a function

```
var identifier = function( var1, var2 ) { //code goes here };
```

  - two important rules will apply to these new variables..
    1. if a variable of the same name already exists globally, the global will be ignored,
    2. the variables will only exist inside this function

code.**Review**: functions (reusable code)

- function literal: **passing data into a function**
  - **variable arguments** will store data that is passed to the function, so if we wanted to fill ***var1*** and ***var2*** with data, we would invoke the function, and pass some data
  - **value1** and **value 2** are referred to as **arguments** for the function

```
myFn("value1", "value2");
```

  - **var1** and **var2** are referred to as **parameters** of the function
  - now inside our function we call on these variables.. such as:

```
var myFn = function(var1, var2) {
  alert( var1 + var2 );
};
```

code.**Review**: functions (reusable code)

- function literal : **passing data to a function - example**

```
var myctr = 1;
var myCounter = function(newct){
  myctr += newct;
  alert(myctr);
};

myCounter(5);   //will output “6” to the page
myCounter(2);   //will output “3” to the page
```

code.**Review**: functions (reusable code)

- function literal: **returning values**
  - **returning values** is information coming out of a function
  - functions **do *not*** have to return any data (this is referred to as a **procedure**)
  - the **return** statement **ends a function's execution**, and can gives a value (of any type) back to the point of call (referred to as a **method**)
  - the called function should be assigned to a variable (i.e **name** - see below)

```
var myFn = function(){
   return "jamesBond";
};

var name = myFn();    //name is now equal to "jamesBond"
```

code.**Review**: functions (reusable code)

- function literal: **returning values**
  - if a return gives back no value, it instead returns “undefined”
  - this technique is commonly used to *end* a function early, during execution

```
var myFn = function(){
  return;
  alert(“No.”);          //this line will never run
};

var name = myFn();    // name is now equal to “undefined”
```

WPF review

code.**Review:** reminders2

code.Review: reminders2

- tab your code cleanly - http://jsbeautifier.org/
- comment your code
  - tips: http://bit.ly/wtNfY
  - mordernizr: http://bit.ly/s5M57i
- compression (minification) (semicolons matter)
  - http://closure-compiler.appspot.com
- NEW item: protects javascript code from being stolen and compresses the code
  http://www.javascriptobfuscator.com/

code.Review: reminders2

- comments
  - single-line

```
// this is a single line comment.
```

  - multi-line

```
/* This is a multiline comment.
comment ends with */
```

Lab 1

lab resume in 50 min

- **Wk1: Assignment: The Duel - Part 1**
  - Log into FSO. This is where all your assignment files will be located as well as Rubrics and assignment instructions.
- **due at the end of lab - commit your completed work to your GitHub account**
  - as part of your grade you will need at least 6 reasonable Git commits while working on your assignment