**Javascript Library**

**Linking to JS Files**

* Add the code in the <head> section <script src=”director/filename.js” type=”text/javascript”></script>
  + type not needed for HTML5
* Best practice is that most scripts are added just before the closing </body> tag when possible
  + Sometimes JS is needed earlier in the page to load properly
* You can/might see JS interspersed in HTML code, but this is not best practice

**Ask Users to Enable Javascript Using <noscript>**

* <h2><noscript>To get the most out of this website, please enable Javascript.</noscript></h2>
  + A <noscript> tag will only display if JS is not enabled
  + They are also useful when you want to display something instead of JS result if it’s disabled

**General Formatting**

* “use strict”;
  + Code this at the beginning of every file to enforce variable declaration (helps to ensure correct scope of variables)
* Scripts run when they are encountered in HTML code (can affect where they display)
* Statements end in semicolons and each statement should start on a new line;
* Statements can be housed in code blocks { }
* Zero based counting (the first character in a string is character 0) just like with python
* Commenting (good practice to add comments to explain your code)
  + In-line comments are any text that follows // all text until the end of the line is a comment
  + Multiline comments are similar to php & css with /\* all text here is commented even over multiple lines of text in a document \*/
* Creating variables
  + Best practice is to write variable names is camelCase (where the first word is lowercase then every subsequent word is capitalized) ex. thisVariableNameIsTooLong
  + Variable names can have numbers, letters, and & or \_ but cannot contain spaces or start with a number
    - No dashes – or periods . in a variable name
  + You need to declare (create) a variable before you can assign it a value, but this can be done simultaneously
  + Use var in front of the variable you want to create
    - var firstName; will create the variable “firstName”
    - Multiple variables can be created on the same line (var price, quantity, total;)
  + Assign a value to an already created variable a = 7; or b = a;
  + Declare a variable and assign a value simultaneously var c = 16;
    - Can do multiple on same line var c = 16, a = 7;
  + Use local variables whenever possible to decrease memory needed (see functions section)
  + More variables mean more memory, which means the page will take longer to respond
* Let vs var (YOU SHOULD PRETTY MUCH ALWAYS USE LET!)
  + Using var to declare a variable can be overwritten with no side effects
    - sometimes this is what you want, other times you might want to know if overwriting a variable
  + Using let to declare a variable that has already been let declared will throw an exception
    - let name = ‘Ryan Tucker’;
* Constants
  + Use const to declare a constant that will now be read only
  + The names of constants should be in ALL CAPS
  + Constants that are objects can still have their values changed, as if they were not constants
    - You have to access the values within the objects using JSON to change them
    - They just can’t be completely reassigned to something else
* Freezing objects
  + Freezing objects will prevent any further change to the object
  + Object.freeze(*objectName*); // now you can no longer edit *objectName*

**Data Types**

* The typeof keyword will tell you which data type something is
* Undefined
  + Type for variables that are declared but have not been assigned values
* Null
* Boolean
  + true or false
* String
  + Strings should be in ‘ ‘ (var favTeam = ‘Liverpool’;)
    - You can use double or single quote marks as long as they match
    - I’m going to use single quotes for strings except in certain cases
  + Strings must be written on one line
  + Simply use + to concatenate strings or +=
  + To have “ or ‘ appear in a string you need to put \ before it so JS doesn’t think it’s code (quote = “she said, \”It’s time to go to the store .\””)
  + Nested quotes within a string do not need \ if they are different (string = ‘this is “ok” in JS’;)
    - You might want to use double quotes for a string for apostrophes (string = “Chuck’s computer”;)
  + \ is the escape character that tells JS to do (or not do) something
    - \’ output single quote mark
    - \” output double quote mark
    - \\ output backslash
    - \n new line
    - \r carriage return
    - \t tab
    - \b backspace
    - \f form feed
  + Use .length to return a number for the length of a string (spaces count) (“a string”.length or use varName.length)
  + Use brackets to return a character from a string (var firstName = “Ryan”; firstName[0]; will return “R”)
    - To always return the last character of a string use firstName[firstName.length – 1];
    - To always return the second to last character of a string use stringName[stringName.length – 2]; and this works for other characters etc.
* Symbol
* Number
  + Numbers do not go in “” (var myAge = 35;)
* Object
* Array
  + List of values

**Logical Operators**

* && is used to test two conditions (both need to be true to return value of true)
* || is used like an or statement and will return true if either operand is true (pipe symbol between backspace and enter key on backslash)
* ! means NOT

**Strings**

* Contain strings in ‘’ or in “”
* escape character is \
  + \n starts a new line in a string
  + \” inserts a double quote in a string
  + \’ inserts a single quote in a string
* + to concatenate two strings
* += adds the result to the end of the variable
* Properties of string objects
  + .length; // will return the length of the string (#char) or probably # items in an array/object
* Methods of string objects
  + concat(*string1, string2, …*)
    - returns a new string that is the concatenation of provided strings
  + .indexOf(*search*[*, position*])
    - Searches for first occurrence of search string starting at the position specified or zero if position is omitted
    - If found, it returns the value of the position (0 indexed), else it returns -1
  + .lastIndexOf(*search*[*, position*])
    - Returns the last position found
  + .substr(*start*[, *length*])
    - Returns the substring that starts at the specified position (0 indexed) and returns the specified number of characters (or to the end)
  + .substring(*start*[, *end*])
    - returns the substring that starts at the specified position and ends at the specified position (not inclusive (or at the end)
  + .toUpperCase(); // changes all string chars to uppercase
  + .toLowerCase(); // changes all string chars to lowercase
  + .charAt(*index*); // returns the character located at the specified index
  + .split(‘*char[s]*’); // splits a string each time ‘char[s]’ found and stores in an array (char[s] becomes eliminated
    - var parts = *string*.split(‘ ‘); // the spaces in *string* are removed, and each word is stored in *parts* array
  + .trim(); // removes whitespace from start/end of a string (spaces)
  + .replace(‘*char[s]1*’, ‘*char[s]2*’); // by default only replaces first match: replaces *char[s]1* with *char[s]2*
    - Can add g or i to be global (replace all instances) and to be case insensitive
  + parseInt(*string*) // window object methods (not necessary to type window.parseInt)
    - Converts a string to an integer (will truncate rather than round)
  + parseFloat(*string*) // window object method
    - Converts a string to a float (decimal)

**Regular Expressions**

* Search for matching patterns in strings
* Used heavily in data validation
* Creating a regular expression object
  + RegExp() constructor
    - var *varName* = new RegExp(“*expression”*[, “*flags”*]);
    - var pattern = new RegExp(“tucker”, “i”); // i for case insensitive flag
  + Regular expression literal
    - var *varName* = /*expression*/[*flags]*;
    - var pattern = /tucker/i; // i for case insensitive flag
* Method of regular expression
  + test(*string*) // searches for the reg exp in the string, returns true if found and false if not
* Performing a test
  + var name = “Chuck Tucker”;
  + var pattern = /Tucker/;
  + pattern.test(name); // returns true
  + You can also run a test without storing the pattern in a variable
    - /Tucker/i.test(name);
  + Validation often in this format
    - if (*inputVar* === “” | ! /*pattern*/.test(*inputVar*)) { isValid = false;

$( “#*idInputVar*” ).next().text( “*Error message*.” ); }

* + - this codes checks if the field is blank or doesn’t meet the pattern, then sets an isValid var to false if either validation fails, then it sets a <span> after the field to an error message
* Matching special characters
  + Must use escape character ‘\’ first
  + \\ backslash character
  + \/ forward slash
  + \[ open bracket
  + \] close bracket
  + \( open parentheses
  + \) close parentheses
  + \{ open brace
  + \} close brace
  + \^ caret
  + \t tab
  + \n new line
  + \r carriage return
  + \f form feed
  + \v vertical tab
  + [\b] backspace (only one that must be inside brackets
  + \$ dollar sign
  + \. period
  + \| vertical bar
  + \udddd Unicode character with four hexadecimal digits (dddd)
  + \xdd Latin-1 character with two hexadecimal digits (dd)
* Matching types of characters
  + . any character except a new line
  + [ ] any character in the brackets (use \[ or \] to match a bracket)
    - T[AB] would find TA or TB
  + [^ ] any character not in the brackets
  + [a-z] any character in the range of charcters when used inside brackets
  + \w any letter, number, or the underscore
  + \W any character that’s not a letter, number, or the underscore
  + \d any digit
  + \D any character that’s not a digit
  + \s any whitespace character (space, tab, newline, arriage return, form feed, or vertical tab)
  + \S any character that’s not whitespace
* Match positions
  + ^ the beginning of the string (\^ to match a caret) (^ goes at the beginning of the pattern)
  + $ the end of the string ($ goes at the end of the pattern)
  + \b word chars that aren’t followed or preceded by a word character
  + \B word chars that are followed or preceded by a word character
* Subpatterns
  + () create a subpattern within parentheses (must parentheses to match them)
  + | matches either the left or right subpattern (\| to match vertical bar)
  + \n matches the subpattern in the specified position (replace n is a number of the subpattern used)
    - example /^(Rob) \1/ // searches for Rob appearing twice (or multiple times?)
    - The 1 tells it to repeat the first subpatter, which is “Rob”
* Repeating patterns
  + {n} pattern must repeat exactly “n” times (n is a number)
    - example \d{3} searches for three digits
  + {n,} pattern must repeat n or more times
  + {n,m} pattern must repeat from n to m times
  + ? zero or one of the previous subpattern (same as {0,1})
  + + one or more of the previous subpattern (same as {1,})
  + \* zero or more of the previous subpattern (same as {0,})
* Examples
  + Try Googling if looking for a potentially widely used pattern if having trouble (library of patterns)
  + Phone
    - /^(\d{3}-)|(\(\d{3}\) ?)\d{3}-\d{4}$/
      * searches for either “three digits and a dash” or “three digits in parentheses (with or without a space because of the ‘ ?’”
      * “three digits”
      * dash
      * “four digits”
  + Credit Card
    - /^\d{4}-\d{4}-\d{4}-\d{4}$/
      * 4 digits dash 4 digits dash 4 digits dash 4 digits
      * may need to escape dashes
    - /^\d{4}-?\d{4}-?\d{4}-?\d{4}$/
      * dashes optional (zero or one of the previous subpattern)
      * may need to escape dashes \-
  + Zip Code
    - /^\d{5}(-\d{4})?$/
      * as 99999 or 99999-9999
      * -9999 optional (zero or one)
  + Email Address (more complete email validation function below)
    - /^[\w\.\-]+@[\w\.\-]+\.[a-zA-Z]+$/
      * as username@mailserver.domain
      * Search Pattern
        + any of the following characters at the beginning (^[\w\.\-])

\w letter number or underscore

\. a period

\- a dash

must have at least one character (+) before the “@”

* + - * + @ sign
        + any of the following chars after the “@” and before the “.”

\w letter number or underscore

\. period

\- dashes

must have at least one (+)

* + - * + \. a dot
        + domain name with a series of letters

[a-zA-Z] upper or lowercase

at least one (+)

* + SMTP complete email validation function
    - var isEmail = function(email) {

if (email.length === 0) { return false; }

var parts = email.split(“@”);

if (parts.length !==2) { return false; }

if (parts[0].length > 64) { return false; }

if (parts[1].length > 255) {return false; }

var address =

“(^[\\w!#$%&’\*+/=?^’{|}~-]+(\\.[\\w!#$%&’\*+/=?^’{|}~-]+)\*$)”;

var quotedText = “(^\”(([^\\\\\”])|(\\\\[\\\\\”]))+\”$)”;

var localPart = new RegExp( address + “|” + quotedText );

if ( !localPart.test(parts[0]) ) { return false; }

var hostnames =

“(([a-zA-Z0-9]\\.)|([a-zA-Z0-9][-a-zA-Z0-9]{0,62}[a-zA-Z0-9]\\.))+”;

var tld = “[a-zA-Z0-9]{2,6}”;

var domainPart = new RegExp(“^” + hostnames + tld + “$”);

if ( !domainPart.test(parts[1]) ) { return false; }

return true;

};

* + Date in mm/dd/yyyy or m/dd/yyyy
    - /^[01]?\d\/[0-3]\d\/\d{4}$/
      * Month
        + a 1 or a zero (optional)
        + a digit
        + \/ a forward slash
      * Day
        + a number 0-3
        + a digit
        + \/ a forward slash
      * Year
        + four digits at the end

**Mathematical Operators**

* Addition +
* Subtraction –
* Multiplication \*
* *x \*\* y*  // raises xy
* Division /
* ++ will add 1 so myVar++; is equivalent to myVar = myVar + 1;
* -- will subtract 1 so myVar--; is equivalent to myVar = myVar - 1;
* == will check if values are equivalent (returns value of true or false) (would say 3 is equiv to ‘3’)
  + Type conversion from string to number with == will still return true for above example
* === is strict equality operator (3 is strict equiv to 3 but not to ‘3’) (no type conversion)
* != checks to see if values are not equal (returns true if values are not equal and false if they are)
  + != will use type conversions to check for equality
* !== is the strict inequality checker and will not perform type conversions
* % returns the remainder of a division problem (does not work with negative numbers)
* += or -= or \*= or /= saves space (myVar = myVar + 12; is the same as myVar += 12;)
* < or > will return values of true or false
* <= and >= are less than or equal to and greater than or equal to respectively

**Numbers**

* General Functions
  + parseInt(*string*) // window object methods (not necessary to type window.parseInt)
    - Converts a string to an integer (will truncate rather than round)
  + parseFloat(*string*) // window object method
    - Converts a string to a float (decimal)
* Methods of a Number Object
  + Number objects are created whenever a number is stored in a variable
    - Use any variable that holds a number before the “.”
  + .isNan(*expression*)
    - returns true if the expression cannot be converted to a number, false otherwise
    - can use as isNaN(*variable*)
  + .toFixed(*digits*)
    - returns a string representation of the number after rounding to # of *digits*
    - need to parseFloat(*var.*toFixed(*digits*)) to return a number (or parseInt)
      * Note: parseFloat will cut off zeros that trail after the decimal
      * Use your number object and .toFixed to display the proper number of decimal places
  + .toString(*base*)
    - returns a string with the number in the given base
    - if the base is omitted, then 10 is used
  + .toPrecision(*value*); // rounds to total number of places (returns a string)
  + .toExponential(); // returns a string representing the number in scientific notation
* Properties of the Number Object
  + Methods are static methods and properties are static properties
    - Always use “Number” before the “.” not a variable storing a number
  + Number.MAX\_VALUE
    - largest positive number capable of being represented by JS (~1798 with 305 zeros)
  + Number.MIN\_VALUE
    - smallest (negative) value
  + Number.POSITIVE\_INFINITY
    - Infinity // shortcut
    - any number larger than MAX\_VALUE
  + Number.NEGATIVE\_INFINITY
    - -Infinity // shortcut
    - any number smaller than MIN\_VALUE
  + Number.NaN
    - NaN // shortcut
    - represents a value that isn’t a number
* Dividing by 0
  + 0 divided by 0 results in NaN
  + Dividing a nonzero number by 0 results in +/- infinity depending on the sign of the number
    - This is different than most languages that often result in runtime errors (not in JS)
    - Good idea to check if *result* == Infinity or -Infinity to display errors and avoid this issue
* Math Object
  + Static methods and properties
    - Always use “Math.”
  + Properties
    - Math.PI
      * returns π
  + Methods
    - Math.abs(*x*)
      * returns the absolute value of *x*
    - Math.round(*x*)
      * returns the value of *x* rounded to the closest integer value (0.5 rounds up)
    - Math.ceil(*x*)
      * returns *x* rounded up
    - Math.floor(*x*)
      * returns *x* rounded down
    - Math.pow(*x, pow*)
      * returns the value of *x* raised to the power specified
      * power may be a decimal number or a fraction
    - Math.sqrt(*x*)
      * returns the square root of *x*
    - Math.max(*x1, x2, …*)
      * returns the largest value from supplied parameters
    - Math.min(*x1, x2, …*)
      * returns the minimum value from the supplied parameters
    - Many trig functions
      * these functions use radians rather than degrees
    - Math.random()
      * generates a random number greater than or equal to 0 but less than 1
      * function to generate a random number between min and max values

var getRandomNumber = function (*min, max*) {

var random;

if (!isNaN(min) && !isNaN(max)) {

random = Math.random(); // 1 > random >= 0

random = min + Math.round(random \* (max - min));

// if random = 0, random will become the min value

// if random = 0.99999 will return min value plus the full range (max value)

}

return random;

};

**Arrays (lists)**

* Arrays can consist of strings, numbers, Boolean, etc. in the same array
* Arrays can be defined with values on separate lines
* You may see array constructors use var varName = new Array ( ); rather than the example above, but this is not best practice
* Creating arrays
  + var myArray = [‘Strings in quotes’, 42, 75, ‘numbers aren\’t’];
  + var myArray = new Array(*length*);
  + var myArray = [];
* Use [ ] and zero indexing to call an entry from an array
  + myArray[0]; returns “Strings in quotes” while myArray[2]; returns 75
* Modify an array entry (anArray = [5, 10, 15, 20, 25];)
  + anArray[2] = 12; will result in anArray = [5, 10, 12, 20, 25]
  + delete
    - delete *arrayName*[2]; // deletes the value at index 2 (undefined)
* Multidimensional arrays (nested arrays, or arrays within arrays within arrays etc.)
  + var bestArray = [[“Yankees”, 11], [“Red Sox”, 0]]
  + bestArray[0]; returns [“Yankees”, 11]
  + bestArray[0][1]; returns 11
  + bestArray[1][0]; returns “Red Sox”
* Associative arrays
  + use strings as indexes
  + creating associative arrays
    - var *arrayName* = [];
    - *arrayName*[“*indexName*”] = *value*;
    - can continue to add index/value pairs in this manner
    - can edit values the same way by calling an index that is already defined
    - can use existing index/value pairs to create a new entry that is a calculation of existing entries
      * var items = [];
      * items[“cost”] = 10;
      * items[“tax”] = 0.08;
      * items[“totalCost”] = (items[“cost”] + (items[“cost”] \* items[“tax”])).toFixed(2);
  + length of an associative array
    - using .length will only count numbered indexes, and *arrayName*.length; returns 0 for an associative array
    - Object.keys(*arrayName*).length; // results in the desired length of the associative array
      * uses the keys method of the Object object
      * this method returns an indexed array whose values are the keys of the associative array
  + keys of an associative array
    - Object.keys(*arrayName*); // returns an array with the keys of *arrayName*
* Properties of an array object
  + .length will return the number of items in an array
    - Delete all elements in an array
      * *arrayName.*length = 0; (sets array length to zero)
* Methods of an array object // *arrayName*.*method*(*params*);
  + .push(*element\_list*); will append data to an array (array name goes before the period)
    - var thisArray = [1,2,3]
      * thisArray.push(4); will result in thisArray = [1,2,3,4]
      * thisArray.push([5,6]); will result in thisArray = [1,2,3,4,[5,6]]
  + .unshift(*element\_list*); will insert data at the beginning of an array (similar to push)
    - thisArray.unshift([-1, 0]) will result in thisArray = [[-1, 0], 1, 2, 3, 4, [5, 6]]
  + .pop(); will remove an item from an array, and it can be stored in a variable
    - thisArray.pop(); will remove [5,6] from the array (the last entry)
  + .shift(); works the same way, except that it removes the first entry from an array
    - var removed = thisArray.shift(); return removed; would return the first entry that was removed
  + .reverse();
    - reverses the order of the elements in an array
  + .splice(*start, number*);
    - removes the *number* of elements specified starting at the index of *start*, returning removed elements
  + .splice(*start, number, element\_list*);
    - removes the *number* of elements specified starting at index of *start*, and replaces them with *element\_list*, returning the removed elements
  + .slice(*start, number*);
    - returns a new array that starts with the index *start* and returns the *number* of elements specified (does not alter original array)
  + .concat(*array\_list*);
    - returns a new array that consists of the original array concatenated with arrays in *array\_list*
    - (does not alter original array)
  + .join([*separator*]);
    - returns a string with all items concatenated into one string
    - when no parameter is passed:
      * converts all elements to strings and concatenates them separated by commas (into one string)
    - will be separated by the string you pass
    - example: can use *arrayName*.join( “\n” ); to display an array as single line entries in html
  + .toString();
    - same as .join with no parameter passed
  + .toLocaleString();
    - same as toString but uses a locale-specific separator
  + .isArray(*object*);
    - returns true if object is an array and false if isn’t
  + .indexOf(*value[, start]*);
    - returns the first index at which the first parameter is found, or -1 if the value isn’t found
    - option to define a starting index
  + .lastIndexOf(*value[, start]*);
    - returns the last index at which the first parameter is found, or -1 if the value isn’t found
    - option to define a starting index
* Method to create an array from a string
  + .split(*separator[, limit]*);
    - returns an array, split a string into an array based on the value of the separator (separator value determines where array items are split
    - a separator of “” will separate every single character
    - optional limit parameter specifies the max number of entries in the new array
* Methods of array objects that accept functions as parameters (callbacks)
  + .sort([*comparison\_function*]);
    - with no parameter passed, converts any numberic values to strings (i.e. 25 > 100)
    - comparison function to sort in numeric order
      * function(x, y) {return x-y;}
      * *arrayName.*sort(function(x, y) {return x-y;} // sorts numbers ascending
      * {return y-x;} would sort descending
      * can use *arrayName*[0]; to then return the max or min value
  + .forEach(*function[, this]*)
    - accepts a function that is executed once for each element (returns a value of undefined)
      * function syntax
        + function(*currentValue[, currentIndex][, array]*);
        + *currentValue* is required
    - example:
      * var items = [2, 4, 6, 8, 10];
      * var double = [];
      * items.forEach(function(item){

double.push(item\*2);

});

// double = [4, 8, 12, 16, 20]

* + - * In php it was foreach(items as item) and the above works similarly
  + .every(*function[, this]*);
    - accepts a function that tests each element to see if it meets a specified condition and returns true only if every element does (returns false otherwise)
    - example:
      * write function to test a condition
      * function must return something
      * var test = function(element, index, array) {

return x > 5;

}

*array*.every(test); // returns true only if every element is greater than 5

* + .some(*function[, this]*);
    - works like every, but returns true if at least one element meets the condition of the function
  + .map(*function[, this]*);
    - accepts a function that is executed once for each element, and returns a new array containing the result
    - example:
      * var square = function(element) { return element \* element; }
      * *arrayName*.map(square);
        + would return a new array with all of the elements squared
      * can also pass methods of the Math object
        + *arrayName.*map(Math.sqrt); // would return square roots for each element
  + .filter(*function[, this]*);
    - accepts a function that is executed once for each element and returns a new array containing elements that meet the requirement of the condition
    - example:
      * var numbers = [1, 2, 3, 4,…];
      * var isOdd = function(value) { return value % 2 == 1; }
      * number.filter(isOdd); // returns only the odd values as a new array
  + .reduce(*function[, init]*);
    - not exactly sure
    - spits out one list of values (looks like it turns a multidimensional array into a single dimension)
  + .reduceRight(*function[, init]*);
    - not exactly sure
    - spits out one list of values (looks like it reduces multidimensional array into single dimension, but starts with the last index (entry or array) and works towards the front
* Working with arrays
  + Loops
    - for (var i = 0; i < *arrayName*.length; i++)
    - iterate using *arrayName*[i] to do it for each entry
  + For/In loops (works kinda like foreach in php)
    - for (var *elementIndex* in *arrayName*) { code to execute }

**Dates**

* Always validate and process dates that users enter before using
  + some unexpected results can occur using 2-digit years or if using - instead of / in dates
* var now = new Date(); // nothing specified in parentheses returns current date/time
  + new Date(YYYY, MM, DD, HH, mm, ss, ms); // months are 0-11, (i.e. 11 is December) (annoying, I know)
  + new Date(‘Mon DD, YYYY HH:mm:ss’); // use three letter month
  + supply date info as a string
* Methods of Date Objects (use “.” before the method to use on a specific date object)
  + getDate(); // returns current day of the month
  + setDate(*value*); // sets the day of the month
  + getDay(); // returns day of the week (0-6)
  + getFullYear(); // returns 4-digit year
  + setFullYear(*value*); // sets 4-digit year
  + getHours(); // returns hour (0-23)
  + setHours(*value*); // sets hour (0-23)
  + getMilliseconds(); // returns milliseconds (0-999)
  + setMilliseconds(*value*); // sets milliseconds
  + getMinutes();
  + setMinutes(*value*); (0-59)
  + getMonth();
  + setMonth(*value*); (0-11)
  + getSeconds();
  + setSeconds(*value*); (0-59)
  + getTime();
  + setTime(*value*); (UTC)
  + getTimezoneOffset(); // returns time zone offset in mins for locale
  + toDateString(*value*); // returns “date” as a string in local time
  + toTimeString(*value*); // returns “time” as a string in local time (+/- GMT, doesn’t say EST for example)
  + toString(*value*); // returns a string for the specified date using the client’s timezone

**Functions**

* Declaring a function
  + return *value*; should be the last statement in a function if it is used, as nothing will be processed after it
  + Parameters needed to run the function are included in the parentheses next to the function name
  + function functionName() {

document.write(‘Hello!’);

statement2;

}

* + function getArea(width, height) {

return width \* height;

}

* + - The code above needs (width, height) parameters in parentheses because the function uses these parameters in the script
* Calling a function
  + Type the function name followed by ();
  + Using the example above, typing functionName(); would call the function
  + To call the getArea function, you type getArea(3, 5); which would use 3 for width and 5 for height
  + You could use variables for the getArea function like so var wallWidth = 3; var wallHeight = 5; getArea(wallWidth, wallHeight);
  + It is possible to call a function before declaring it
  + Examples using getArea above
    - var kitchenArea = getArea(30, 12); kitchenArea would equal 360
    - var brArea = getArea(10, 16); brArea would equal 160
* It is possible to get multiple values from a function and store them in an array
  + function getSize(width, height, depth) {

var area = width \* depth;

var volume = width \* height \* depth;

var sizes = [area, volume];

return sizes;

}

var areaOne = getSize(3, 2, 3)[0];

var volumeOne = getSize(3, 2, 3)[1];

* + - The code above declares the function getSize
    - This function has three parameters (width, height, depth)
    - The variable area is created and assigned the value width \* height
    - The variable volume is created and assigned the value width \* height \* depth
    - The variable sizes is created and assigned the array with the area value and the volume value
    - The sizes array is returned as the output
    - The variable areaOne has a value of 9 and the variable volumeOne has a value of 18
* IIFE (immediately invoked function expression)
  + In some cases, you don’t want to store a function to be called later, you just want it to run and store the value in a variable immediately upon being encountered in code
  + These functions are not declared and are called anonymous functions
  + Uses
    - As an argument (value) within another function
    - To assign the value of a property to an object
    - In event handlers and listeners to perform a task when an event occurs
    - To prevent conflicts between scripts that might use the same variables
  + var area = (function() {

var width = 3;

var height = 2;

return width \* height;

} () );

* + - The yellow parentheses make sure this is treated like an expression
    - The green parentheses tell the script to call the function immediately
  + Scope (important to IIFE’s) 🡪 where a variable can be used
    - Local (function-level) variables 🡪 have local scope, are declared within a function, and cannot be accessed outside of that function
      * Local variables can have different values each time a function runs
      * Two different functions can use the same variable name without conflict
    - Global variables can be used anywhere and have global scope
      * Take up more memory than local variables (stored as long as the page is loaded)
      * Could lead to naming conflicts
      * Use local variables whenever possible
      * Forgetting to use *var* will cause local variables to become global variables (bad practice)

**JS Items**

* console.log(); will display info in parentheses in the developer log
* return *value*; will return whatever you ask it to (not using return will result in an undefined value)
* parseInt(); converts a string into an integer (unless string characters prevent that)
  + You can specify the base system for numbers using radix parseInt(string, radix);
  + The radix value (2-36) tells the number of characters allowed (2 is binary code, 16 is hexadecimal)
* Math.random(); (supposed to be capitalized) will return a random decimal number (n) where 0 <= n < 1
* Math.floor(); will round down to the nearest whole number (rounds down every time)
* Use Math.random() and Math.floor() together to generate random whole numbers within a range

function randomRange(myMin, myMax) {

return Math.floor(Math.random() \* (myMax - myMin + 1)) + myMin;

}

* + This line multiplies a random decimal less than 1 to the value of your range + 1 then adds it to your minimum value (assuring your min value is the lowest number you can get)
  + If Math.random returns 0, then you return your min value
  + If Math.random returns a number near one, you could get close to myMax + 1 but never all the way there, then you round down to myMax

**Objects (Murach)**

* Native object types (hierarchy)
* Every object will inherit the properties and methods of objects at higher levels
  + Object (the Object object)
    - String
    - Number
    - Boolean
    - Date // must use new Date() because no literal way to create a date object
    - Array
    - Function
* Creating Objects
  + var *varName* = new *objectType*(*arguments*);
    - var today = new Date();
    - var lastName = “Hopper”; // same as new String(“Hopper”);
      * other native objects work the same way
      * this is “literal” object creation (can’t do this for date objects)
      * best practice to do this rather than always using the “new” keyword
    - var *objectName* = {}; // creates a new Object object or new Object();
* Accessing properties and methods
  + Properties
    - var *varName* = *objectName.property;*  // same as *objectName[“property”];*
  + Methods
    - var *varName = objectName.method(arguments);* // same as *objectName[“method”](args);*
* Initialize New Object with Properties and Methods
  + Example below will create a new Object object
  + var *objectName* = {

*methodName:* function*(arguments) { … },*

*methodTwoName:* function*(arguments) { … },*

*propertyName: propertyValue,*

*propertyTwoName: propertyTwoValue*

};

* + Example below that uses the “this” keyword
    - var invoice = {

taxRate: 0.08;

getSalesTax: function(subtotal) {

return (subtotal \* this.taxRate);

},

getTotal: function(subtotal) {

return (subtotal + this.getSalesTax(subtotal));

},

terms: {

initialRate: 0.07;

finalRate: 0.09;

dueDays: 30;

}

};

* + Using properties/methods of an object using example for *invoice* object above
    - var salesTax = invoice.getSalesTax(100).toFixed(2); // salesTax = 8.00
    - var invoiceTotal = invoice.getTotal(100).toFixed(2); // invoiceTotal = 108.00
    - var dueDays = invoice.terms.dueDays; // dueDays = 30
* Extending an object (adding properties/methods after object creation)
  + var *objectName*{}; // create an object
  + *objectName.propertyName* = *value;* // add a new property or modify the value of an existing property
  + *objectName.methodName* = function(*arguments*) { … };
* Deleting an object’s property
  + delete *objectName.propertyName*; // calling the property will return “undefined”
* Multiple variables can refer to the same object
  + *newObjectReference = oldObjectName;* // will result in two names referencing the same object
    - changes to one reference will cascade to all references because the object has changed
* Constructor Functions (constructor)
  + Special kind of function that creates an object type
    - You specify the object type (invoice, scuba shop, flounder, etc.)
  + Can then call function to create multiple instances of that object type: inherits the methods of that type
  + Names of constructor functions should begin with capital letters
  + Use *this* to refer to the new object that is created
  + Use *new* to create an instance of that object type
  + Constructors can accept parameters or accept no parameters
  + Examples:
    - var O*bjectTypeName* = function() {

this.*property1*;

this.*property2*;

}; // this object type accepts no parameters and will have two properties w/ no values

* + - var *ObjectTypeName* = function(*property1, property2*) {

this.*property1* = *property1*;

this.*property2* = *property2*;

}; // this version accepts two parameters and assigns them to properties

* + Adding methods to a constructor function (works for properties too)
    - Must use the *prototype* keyword, then use chaining to define the method
      * *ObjectTypeName*.protoype.*methodName* = function([*args*]) {*statements*};
    - Can also add methods just like you added properties in the original constructor function using the *this* keyword
    - Methods added to the protoype will be available to existing objects as well as objects that are created later of that object type
    - Not recommended to add methods to native object types, but this is possible
  + Adding methods/properties to instances
    - Once an object is created, you can add methods/properties to that instance using dot notation
    - These properties/methods are only available to that instance
    - These are called “direct or own properties/methods” and will override a prototype property
    - Assign the new property/method using dot notation, and it will replace the prototype version
  + Using objects
    - Creating an instance of an object type and assigning it to a variable
      * var *objectName* = new *ObjectTypeName*([*args*]);
    - Accessing properties or methods (after creating an instance)
      * var *varName* = *objectName*.*propertyName*; // or can not assign to variable, and just use it in a calculation for example
      * return *varName.methodName*([*args*]);
  + Big Example illustrating use of *this*
    - var Invoice = function(subtotal, taxRate) {

this.subtotal = subtotal;

this.taxRate = taxRate;

}; // constructor

Invoice.prototype.getTaxAmount = function() {

return (subtotal \* this.taxRate);

}; // add method to the Invoice object type for all new instances of that object

Invoice.prototype.getInvoiceTotal = function() {

return subtotal + this.getTaxAmount(this.subtotal);

}; // method that uses the inputted subtotal to calculate the total cost

var invoice1 = new Invoice(1000.00, 0.075); // creates an instance

alert (invoice1.taxRate); // displays 0.075

invoice1.getTaxAmount; // returns 75

invoice1.getInvoiceTotal; // returns 1075.00

* Factory Function (using the create() method of the Object object)
  + Preferred by some programmers (not always supported in older ES3, but supported by ECMAScript5)
  + create(*prototype*[, *properties*]); // it’s the optional “properties” arg that’s not supported by ES3
  + Objects created in this way are of the “object” type rather than defining a new type of object
  + Example:

var getTrip = function(destination, miles, gallons) {

var tripPrototype = {

isValid: function() {

if (this.destination === “” || isNaN(this.miles) || isNaN(this.gallons) ) {

return false;

} else if (this.miles <= 0 || this.gallons <= 0) {

return false;

} else {

return true;

}

},

calculateMpg: function() {

return this.miles / this.gallons;

},

toString: function() {

var mpg = this.calculateMpg().toFixed(1);

return this.destination + “: Miles - “ + this.miles + “; MPG - “ + mgp;

}

};

var trip = Object.create(tripPrototype);

trip.destination = destination;

trip.miles = parseFloat(miles);

trip.gallons = parseFloat(gallons); // destination, miles, gallons collected from a form

return trip;

};

within the document ready function…

$(“#add\_trip”).click(function() {

var trip = getTrip ($(“#destination”).val(), $(“#miles”).val(), $(“#gallons”).val() );

if (!trip.isValid()) {

alert(“Please complete all fields. Miles and gallons “

+ “must be numeric and greater than 0.”);

} else {

// code to execute otherwise

}

});

* Function objects contain an “arguments property”
  + Similar to an array but not a true array object (so props and methods of array objects don’t work)
  + This information is stored in *arguments* and it is 0 indexed
    - return arguments[0]; // would return the first argument passed to a function
    - arguments.length; // returns the number of arguments passed
    - Can use arguments.length to create functions with variable amounts of arguments
      * Example for creating an invoice prototype that first accepts a tax rate followed by 0 or more item codes:
        + var Invoice = function(taxRate) {

this.items = [];

this.taxRate = taxRate;

if (arguments.length > 1) {

for (var i = 1; i < arguments.length; i++) { // skip taxRate 1st arg

this.items.push(arguments[i]);

}

}

};

* + - Can create optional arguments using technique like in the function below to accept an optional separator or set to use a comma by default if no separator is passed
      * Invoice.prototype.listItems = function(separator) { // add method to Inv prototype

var sep = (arguments.length === 0) ? “, “ : separator;

// sep is “, “ if arguments.length = 0 else it is the value of *separator*

return this.items.join(sep);

}; // output is a list of all the Invoice items in one line separated by a comma or other inputted separator

* Cascading Methods (fluent coding)
  + With methods that modify an object or property of an object it is best practice to returnin the object at the end of the method (return this;)
  + When you do this, you can chain multiple methods together
    - tasklist.load().display(*$(“tasks”)*); // $(“tasks”) is an arg passed to the .display method
      * This code runs the load() method of the tasklist object, then runs the the display method immediately after
      * You must have *return this;* in each method for chaining to work here

**Javascript Libraries**

* + Libraries
    - are simply JS files that contain related functions, objects, or both
    - are easy to reference and reuse
    - encourage separation of concerns
  + Reference libraries in the correct order
    - Reference in the <head> of html just like all other JS <script> files
    - Reference a library before any files that use it
    - jQuery must be referenced before any script files that use jQuery
    - Library files must be referenced before a file that uses that library

**Objects (old book)**

* Objects are similar to variables, but they group multiple pieces of information
  + An object can have many different **properties** that consist of keys that have values
  + A function within an object is referred to as a **method**
  + **Literal notation** for object creation
    - Properties/methods are separated by commas, but not after the last entry

var scubaShop {

name: ‘Charleston Scuba’,

brands: [‘Scuba Pro’, ‘AquaLung’, ‘Cressi’],

pool: false,

numStaffSpots: 4,

numStaffFilled: 3,

checkAvailability: function() {

return this.numStaffSpots – this.numStaffFilled;

} //no comma after the last entry

};

* + scubaShop is the object; name, brands, pool etc. are properties (keys with values); and checkAvailability is a method (totalSpots and takenSpots would be global variables)
  + This object could be used to update and display info for many different scuba shops
  + The *this* before the dot tells the function to use numStaffSpots etc. from *this* particular object
  + **Constructor notation** for object creation
    - The object is declared, then the properties and methods are assigned using either dot notation or bracket notation

var scubaShop = new Object();

scubaShop[‘name’] = ‘Charleston Scuba’; //bracket notation

scubaShop.checkAvailability = function() { //dot notation

return this.numStaffSpots – this.numStaffFilled;

};

* + Creating multiple objects at once using **constructor notation** within a function
    - Use a function as a template that can create many similar objects (like scuba shops)

function Shop(name, brands, pool, numStaffSpots, numStaffFilled, checkAvailability) {

this.name = name;

this.brands = brands;

this.pool = pool;

this.numStaffSpots = numStaffSpots;

this.numStaffFilled = numStaffFilled;

this.checkAvailability = function() {

return this.numStaffSpots – this.numStaffFilled;

};

* + - The *this* keyword indicates that those properties/methods belong to the object that *this* function creates
    - Names of **constructor functions** usually begin with uppercase letters to remind developers to use the *new* keyword when creating new objects using the constructor function
    - Use constructor functions to create new objects of that type and store them in variables
      * var chasScubaShop = new Shop(‘Charleston Scuba’, [‘ScubaPro’, ‘Aqualung’, ‘Cressi’], false, 4, 4);
      * That function would create chasScubaShop with all of the info stored as keys/values
      * This is good if you only need to access some stored information at a time using a general format or set of calculations or using complex information that needs to be available sometimes
      * chasScubaShop is defined by the code, but it is only created if needed
* **Accessing information from an object**
  + **Dot Notation**
    - scubaShop.name; would return ‘Charleston Scuba’
    - scubaShop.checkAvailability(); would call that method
    - Use *this* to reference keys from that object in methods
  + Bracket Notation
    - Commonly used when:
      * A variable is used in place of the property name
      * The name of a property contains special characters (spaces, dashes etc.)
      * The property is a number (not good practice to do that)
    - scubaShop[‘pool’]; would return false
    - scubaShop[‘checkAvailability’](); would call that method
* Adding or deleting information in an existing object
  + Just add new info (scubaShop.compressor = true;)
  + Use the keyword delete to remove a property/method 🡪 delete scubaShop.numStaffFilled;
* Use .hasOwnProperty(); to check if an object has a property with the specified key (returns true/false)

**JavaScript Object Notation (JSON)**

* Collection of key/value pairs
* Values can be different formats including arrays
* Multiple objects are often stored in arrays
* Access information in this situation by using a combination of dot and bracket notation (use bracket for the special cases like names with spaces) and combining with array accessing syntax for nested arrays within objects
  + Combination of these properties can take this form mainArray[3].key4[5];
  + This would access the fourth object in the mainArray, access the key named “key4”, then return the 6th entry of the array associated with that key

**Conditional Operators**

* if is a keyword that’s value is Boolean (either true or false)
* The condition is only executed when the if statement is true (or you can return true/false by simply specifying return testingcondition; (testingcondition could be a < b; or c === d; etc.)
* Format for if statements within a function:

function testThis(val) {

if (condition involving val) {

code to execute;

} else if (condition to evaluate) {

code to execute;

} else {

code to execute;

}

}

* Single line if else statements can be written return condition involving val ? true : false;
  + Other syntax for that is return condition involving val ? “return this” : “else return this”;
  + Can do this for multiple scenarios
    - return num > 0 ? "positive" : num === 0 ? "zero" : "negative";
* Switch statements (case values are tested as strict === comparisons)

function caseInSwitch(val) {

var answer = "";

switch(val) {

case 1:

answer = "alpha";

break;

case 2:

answer = "beta";

break;

case 3:

answer = "gamma";

break;

case 4:

answer = "delta"

break;

}

return answer;

}

caseInSwitch(1); returns alpha

* If you want to check several cases using switch, but then return something else (like a final else statement) use default as the last case

function switchOfStuff(val) {

var answer = "";

switch (val) {

case "a":

answer = "apple";

break;

case "b":

answer = "bird";

break;

case "c":

answer = "cat";

break;

default:

answer = "stuff";

break;

}

return answer;

}

switchOfStuff(1); would return “stuff”

* Switch statements where different cases should execute the same statement can be combined by listing each case, followed by the statement to run, with only one break after each set of cases

function sequentialSizes(val) {

var answer = "";

switch (val) {

case 1:

case 2:

case 3:

answer = "Low";

break;

case 4:

case 5:

case 6:

answer = "Mid";

break;

case 7:

case 8:

case 9:

answer = "High";

break;

}

return answer;

}

sequentialSizes(1); would return low (4-6 mid & 7-9 high)

* Using a function to lookup info in an object

function phoneticLookup(val) {

var result = "";

var lookup = {

"alpha": "Adams",

"bravo": "Boston",

"charlie": "Chicago",

"delta": "Denver",

"echo": "Easy",

"foxtrot": "Frank"

}

result = lookup[val];

return result;

}

phoneticLookup("charlie"); returns “Chicago”

* Conditional Operator
  + (*conditional expression*) ? *do\_if\_true* : *do\_if\_false*;
  + Examples
    - var message = (age >= 18) ? “Can vote” : “Cannot vote”;
    - var overtime = (hours > 40) ? ((hours - 40) \* rate \* 1.5) : 0;
* Using conditional operators
  + || or
  + && and

**Loops**

* **Break**
  + Use a break; statement to end a loop
  + This can help prevent infinite looping (use if() break; )
* **Continue**
  + Use continue to skip back to the beginning of a loop
  + Can test a condition and skip an action/restart the loop until a condition is met (for example)
* **While loops** will run while a certain condition is true, then stop once it’s not
  + The code below will run until n is greater than 4, so once completed myArray = [0, 1, 2, 3, 4]

var myArray[];

var n = 0;

while(n <= 4) {

myArray.push(n);

n++;

}

* **For loops** have three optional expression for ([initialization]; [condition]; [final-experession])
  + The initialization statement is executed only once at the very beginning (define and set up loop variable)
  + The condition is evaluated at the beginning of every loop and the loop will run while the condition has a value of true
  + The final-expression is executed at the end of each loop (often used to increment or decrement loop iterations/counter)
  + The code below accomplishes the same thing as the while loop code above

var myArray = [];

for (var n = 0; n <= 4; n++) {

myArray.push(n);

}

* + There are lots of possibilities (count by odd numbers by initialize n = 1 then final-expression of n += 2)
  + Have a counter work backwards by odd numbers (myArray = [9, 7, 5, 3, 1])

var myArray = [];

for (var n = 9; n > 0; n -= 2) {

myArray.push(n);

}

* + The following code will multiply every number in an array and the first subarray (the i and j variables are the index of the arrays. additional variables would be needed for subarrays of the j array) (example array for this multiplyAll([1, 2], [3, 4], [5, 6]); and the code would multiply 1\*2\*3\*4\*5\*6

function multiplyAll(arr) {

var product = 1;

for (var i = 0; i < arr.length; i++) {

for (var j = 0; j < arr[i].length; j++) {

product = product \* arr[i][j];

}

}

return product;

}

* A **do…while** loop will work like a while loop, except it will run once even if the specified condition fails on the first try (the code below will run once even if i < 5 on the first run, then it will stop if i < 5 after that)

do {

myArray.push(i);

i++;

} while (i < 5);

\*See Comp Sci folder for additional examples of codes

Much of this code can be written very similarly to the php syntax you are used to using.

**Exception Handling**

* Error Objects
  + Properties
    - name (type of error)
    - message (message describing the error)
  + Creating error objects
    - new Error(*message*);
  + Throwing an error
    - throw *errorObject*;
  + Example of creating and throwing errors
    - var *myFunction* = function (*parameters*) {

if (*error\_conditions*) {

throw new Error(*“error message here*”);

} else {

// statements if no error

}

return *anything to return;*

};

try {

*myFunction*(*parameters*);

}

catch (error) {

alert (error.name + “: “ + error.message);

}

finally { $(“#*id\_to\_focus\_on”.*focus(); }

* + Special types of errors
    - RangeError // a numeric value has exceeded the allowable range
    - ReferenceError // a variable is read that hasn’t been define
    - SyntaxError // a runtime syntax error
    - TypeError // type is different than expected
      * Creating these
        + new RangeError(*message*);
* Try/Catch
  + Syntax
    - try {

// statements to try

}

catch(*errorName*) { // *errorName* could just be error

// what to do in case of an error (example below)

$(“#message”).text(*errorName*.name + “: “ + *errorName*.message);

}

[ finally { /\* statements \*/ } ] // optional block, executed no matter what (ex. set focus)

**Browser Object Model**

* Model of the current window or browser tab
* Structure
  + Window (or tab) // parent object, and below are the child objects
    - Document (current web page)
    - History
    - Location (url of current page)
    - Navigator (information about browser)
    - Screen (device’s display information)

**Document Object Model (DOM)**

* Model of the current web page (part of the browser object model)
* Structure
  + Each node is an object with methods/properties
    - Start with Document, then <html> and other HTML element nodes (tags)
    - Element nodes can have child element nodes (like <ul> and <li> or <div> and <p>)
    - Each element node can have an attribute (Attr) node (like class)
      * These can be modified using JS to trigger new css styles under certain cases
    - Element nodes can have text nodes
    - Node Interface
      * Properties
        + .nodeValue // returns the text stored in the node (or null)
        + .parentNode // returns the parent node of a node if one exists (or null)
        + .childNodes // returns an array of child nodes (or empty array)
        + .firstChild // returns node object for first child (or null)
        + .lastChild // returns node object for last child (or null)
        + .nextElementSibling // returns a node object for the next sibling (or null)
  + Must access document, then navigate through element nodes to desired element, then you can access its attributes or text
* Properties
  + document.title // title of current doc
  + document.lastModified // date on which the doc was last modified
  + document.URL // string containing URL of current doc
  + document.domain // returns domain of current doc
* Methods (select and update parts of a page)
  + document.querySelectorAll(); // returns list of elements that match CSS selector specified
  + document.createElement(); // creates the new element
  + document.createTextNode(); // creates a new text node
  + document.getElementsByTagName(“*tagName*”); // returns an array of all elements of specified tag
  + document.getElementsByName(“*name*”); // returns an array of all elements that have the specified name
  + document.getElementsByClassName(“*class*”); //returns an array of all objects with speficied class name (can include multiple class names, separate each with a space)
  + document.write(*string*)
    - Writes the string that’s passed to the document
    - Can use document.write(“<br>”); to code a new line
    - Can write html elements using document.write
  + document.writeln(*string*) // that’s a lowercase L
    - Ends with a new line character, but this is ignored unless coded within HTML pre element
      * document.writeln(“<pre>This is the first line. “);
      * document.writeln(“This is the second line.</pre>”);
  + document.getElementById(*id*)
    - Gets the HTML element that has the id that’s passed and returns that element
    - A common function to do this:
      * function $(id) {

return document.getElementById(id);

} // supply a “string” as the html id, and return its element

* + - * Example: var emailAddress = $(“email\_address1”).value;
      * jQuery builds this function in, but you must use # (css selector for id)
        + This function in jQuery also lets you select classes, children, etc.
* **Get inputs from an HTML form (currently, not using form/action, just input fields)**
  + var *variableName* = $(“*html\_id\_name*”).value;
    - Need to nest these variables within a function that is attached to an event handler (like a submit button onclick)
    - Usually, all functions that deal with the data are defined first, then a processEntries() function can get and validate the entries before executing necessary functions using them
* DOM Objects
  + Textbox
    - Properties
      * .value //returns a string that represents the contents of the text box
        + var firstName = document.getElementById(“first\_name”).value;

stores value of HTML element with id=”first\_name” into the JS variable firstName

* + - * .disabled // returns Boolean that controls whether the box is disabled
    - Methods
      * .focus() // moves the cursor into the text box
* Common element methods (specify the element before the dot by storing the element in a variable using getElement method)
  + .hasAttribute(“*name*”); // returns true if the element has the attribute specified
  + .getAttribute(“*name*”); // returns the value of the specified attribute or an empty string
  + .setAttribute(“*name*”, “*value*”); // sets the attribute with “name” to the specified “value”, creating the attribute if it doesn’t already exist, changing it otherwise
  + .removeAttribute(”*name*”); // removes the specified attribute
* DOM HTML specification (can simplify code)
  + Store element in a variable using getElement code then use the following to set/modify:
  + Elements
    - All
      * .id // accesses the id attribute
      * .title // accesses the title attribute
      * .className // accesses the class attribute (can set multiple by separating with spaces)
      * .tagName // name of the tag (i.e. h2, a, p…)
      * .innerHTML // gets/sets the HTML of an element (see below for more info)
    - <a>
      * .href
    - img
      * .src
      * .alt
    - input
      * .disabled // set to true or false
  + Examples:
    - * var target = getElementById(“link\_id”).href; // stores the link with “link\_id” as id attr
      * var div1 = getElementById(“div1”);

div1.className = “align-right blue-text”; // set classes for element id=”div1”

* + - * getElementById(“btnClear”).disabled = true; // disable button with id=”btnClear”
* .innerHTML
  + Will insert the following HTML code into that element
    - Example
      * $(“*table\_id*”).innerHTML = …;
      * Above code uses *table\_id* as the id of a <table></table> element
      * You need to include <tr></tr> in the additions with <th> or <td> info in between
      * The “innerHTML” is added between the <table> & </table> tags
        + No firstChild or other identifiers needed
  + Set the HTML for an element
    - $(“*element\_id*”).innerHTML = “”;
      * Will change the innerHTML if you use = // set to value or empty string to remove
      * Can append several pieces of info if you use +=
  + Get the HTML for an element
    - var *varName* = $(“*element\_id*”).innerHTML;

**Built-In Objects**

* **Browser Object Model**
  + Window Object (current browser window/tab, topmost object in BOM)
    - Not necessary to code ‘window.’ because the window is the global object in JS
    - window.innerheight (height of actual viewing window)
    - window.innerwidth (width of actual viewing window)
    - window.pageXOffset (distance page has to be scrolled horizontally in px)
    - window.pageYOffset (distance page has to be scrolled vertically in px)
    - window.screenX (X-coordinate of cursor, relative to top-left of screen in px)
    - window.screenY (Y- coordinate of cursor, relative to top-left of screen in px)
    - window.location (current URL of window object or local file path)
    - window.document (ref to document object, current page contained in window)
    - window.history (ref to history object for window/tab containing history from that tab/window)
    - window.history.length (# items in history object for particular window/tab)
    - window.screen (ref to screen object)
    - window.screen.width (width property in px from screen object)
    - window.screen.height (height property in px from screen object)
    - window.alert(*string*) (creates dialogue box with message/must click ok to exit, like a flag)
    - window.prompt(*string, default*)
      * This will open a dialogue box that displays *string* and has a text block that displays the value of *default*, with cancel and ok boxes to click
      * The user can enter a value and click ok, and the value is returned as a string
      * If the user hits cancel, then null is returned
    - window.open() (opens new window with specified URL/popup blockers can affect this)
    - window.print() (acts like the browser’s print option/prints whole window)
    - parseInt
    - parseFloat
    - window.confirm(*string*)
      * Displays a dialogue box that contains the string in the parameter, an OK button, and a cancel button
      * If the user clicks ‘ok’ true is returned, if the user clicks ‘cancel’ false is returned

**Event Handling**

* Events occur when certain actions happen like user clicks
* Common events
  + Window Object
    - load // the document is loaded in the browser
  + Button Object
    - click // a button click
  + Control/Link Object
    - focus() // the object receives the focus
    - blur // object loses the focus
  + Control
    - change // user changes the value in the control
    - select // user sleects text in a text box or text area
  + Element
    - click // user clicks on the element
    - dblclick // user double clicks the element
    - mouseover // user moves the mouse over the element
    - mousein // user moves the mouse into the element
    - mouseout // user moves the mouse out of the element
* The **window.onload** event handler is responsible for any events that need to be handled after the page has loaded and the DOM has been built
  + Generally added at the end of a script file
  + window.onload = function() {

events to execute;

};

* Syntax for attaching an event handler
  + *objectVariable.oneventName = eventHandlerName;*
    - object name . event name (preceded by ‘on’) = variable for the function expression to handle the even (*eventHandlerName* is a function that processes the event)
    - Example (using the $ function for getElementbyId)
      * $(“submit\_button”).onclick = joinList;
        + where joinList is an example of a function that was defined earlier in your code
    - Example to create/attach in one step:
      * $(“submit\_button”).onclick = function() { code to execute; };
* Cancel the default action for an event
  + To be compatible with all browsers, use the following code:
    - var *eventName* = function(evt) {

// if the event object is not sent, get it from the window object

if (!evt) { evt = window.event; } // for older IE

// cancel the default action

if (evt.preventDefault) { // checks if the method exists (does in most browsers)

evt.preventDefault(); // for most browsers

} else {

evt.returnValue = false; // for older IE

}

};

**Handling Errors**

* Good idea to nest error handling within a function that could produce errors
* Within a function
  + initialize error message with empty string
  + use if, else if, else (or switch) statements to check for validity etc. and store any errors in error message variable
  + at the end of the function, check if error message is still empty string, if so, execute code, otherwise display error message using something like “alert(errorMessage);” // p. 135 Murach for example

**Forms and Controls**

* Examples use the $ function for getElementById presented earlier
  + Recall that jQuery would need the # to get an element by ID
* To set the value of entries, you can add window.onload events
  + $(“*idName*”).value = “”; // for text, textarea, & select
  + $(“*idName*”).checked = *true/false*; // for radio or checkboxes
* To get the value of entries you can use the same syntax the other way around
  + Text, textarea, & select
    - var *varName* = $(“*idName*”).value;
  + Radio & checkboxes
    - if($(“*idName*”).checked == true) {code to execute /\* like set a variable value \*/ ;}
    - Remember that these store *true/false*, so can store in a variable, then use if statements to execute code
  + Data can then be used for validation prior to being submitted to the server
* Form Methods
  + $(“*form\_id*”).submit(); // submits form/data to the server
  + $(“*form\_id*”).reset(); // resets the form control to their starting values
* Control Methods
  + .focus(); // moves the focus to the control
  + .blur(); // removes the focus from the control
* Control Events (should be attached in the window.onload event)
  + .onfocus // control receives the focus
  + .onblur // control loses the focus
  + .onclick // the control is clicked
  + .ondblclick // the control is double clicked
  + .onchange // the value changes
  + .onselect // the user selects text in a text box or text area
* JS Form General Design
  + “use strict” for most purposes
  + Contents
    - $ function to get elements // though if using jQuery this function needs # to get IDs
    - Process Entries function
    - isValid variable to provide final answer to “is data valid” before submitting
    - Get values for user entries
    - Validate entries
    - resetForm function
    - window.onload to attach event handlers
      * All onclick events as well as setting focus to first field
      * Could use .onchange to immediately validate some text boxes methinks
  + Tips
    - Use an isValid var initialized to true to track validation in function to submit data
      * After each failed attempt, when you set the span text, set isValid to false
      * Check isValid == true before submitting
    - Use asterisk in <span> elements <span>\*</span> as a sibling of each form control that is required (maybe a space would work for field that isn’t required but needs validating)
      * If you want to use the nodeValue statement below to code, then \* must be included, because JS cannot set property nodeValue of null (use space if want empty methinks)
    - CSS to make this text red ( #*form\_id* span { color: red; } )
    - During validation, remove asterisk and replace with necessary message or empty string
      * $(“*control\_id*”).nextElementSibing.firstChild.nodeValue = “*Text or empty string*”;
      * This code selects the sibling <span> of your control to validate and selects its text child before changing its value to the supplied string (<span> should be right after control element at the same level)
    - Any time the page reloads (window.onload or resetForm or submission etc.) set focus to first field $(“*first\_field\_id*”).focus();

**Images**

* Create an image object
  + var *image* = new Image(); // create a new image object that is empty
* Preloading images
  + *image*.src = “*filepath/image\_name.jpg*”; // set the image src value for the image object
  + Just the act of setting the image path this way will cause preloading of the image
  + Preloading will load the image while the page is loading, rather than loading the images afterwards (best practice to do this)
  + If you have an array of src values, a for loop can preload all of the images
    - This example assumes all <a> tags include links for images to be preloaded (like when using thumbnail images as links) (you could use document.getElementsByClassName(*“class”*) if you have other <a> elements you don’t want to include
    - You can also include thumbnails in a list and then getElementsByTagName from that list node
    - var links = document.getElementsByTagName(“a”); // create an array of image src’s

var i, link, image; // initialize variables for loop

for ( i = 0; i < links.length; i++) {

link = links[i]; // iterate through each link (or use imageLinks array ex. below)

image = new Image(); // create a new image object for each link

image.src = image.href; // set the src for each image using the array

}

Alternative code below to not get every <a> tag, but only those in an <ul id=”*image\_list\_id*”>

var listNode = $(“*image\_list\_id*”); // alternative code to not include all <a> tags

var imageLinks = listNode.getElementsByTagName(“a”); // only include <a> from your list

**Timers**

* One-Time Timers
  + Run only once
  + var *timerName* = setTimeout(*function, delayTime*); // store the reference to the timer in a var
    - Calls the *function* after the *delayTime* in milliseconds
  + clearTimeout (*timerName*); // cancels a timer
* Interval Timers
  + Calls its function repeatedly
  + var *timerName* = setInterval(*function, intervalTime*);
    - Calls the *function* after every *intervalTime* in milliseconds
  + clearInterval(*timerName*); // cancels an interval timer
* Basic timer function to display & count seconds on a page:
  + HTML
    - <span id=”counter”>0</span> // to count up, start at 0, to count down, start at max value
    - <input type=”button” id=”cancel” value=”Cancel Timer”>
  + JS
    - var timer; // create a global timer variable

var counter = 0; // create and initialize a global counter variable

var updateTimer = function () { // function to increment and update counter display

counter++; // increment counter

$(“counter”).firstChild.nodeValue = counter; // update counter display

};

var cancelTimer = function () {

clearInterval(timer);

};

window.onload = function () {

timer = setInterval (updateCounter, 1000);

$(“cancel”).onclick = cancelTimer;

};

* + Tips
    - You can add additional code in the timer/cancel functions to do various things like update image displays (slideshows) or hide elements by changing their classes to “hidden” & hiding with css
    - It’s common to use anonymous functions with timers (i.e. instead of storing timer function in a variable, you can use it as the first argument passed to the setInterval function
      * This will keep you from being able to cancel the timer, however

**Built-In Properties/Methods**

* Math (Math object, which is global)
  + Math.PI; // returns pi
  + Math.round(*value*); // rounds value to the nearest integer
  + Math.sqrt(*value*); // returns the square root of a positive number
  + Math.ceil(*value*); // rounds up to nearest int
  + Math.floor(*value*); // rounds down to nearest int
  + Math.random(); // generates random number 0 <= n < 1
  + Math.sin(); // these functions return radians (multiply by (180 / pi) to get degrees)
  + Math.cos();
  + Math.tan();

**Debugging**

* Use Chrome’s developer tools (F12), and look at the Console tab, find the error, and use the Sources panel
* Step through code
  + Set/remove breakpoints
    - Get to Sources tab of Google Chrome
    - Click on the line number to add/remove a stop point
  + Step into (F11) step through one line at a time
  + Step Over (F10) run any called functions without stepping through them
  + Step Out (SHIFT+F11) Execute the rest of a function without stepping through it
  + Resume (F8)
* The “Sources” tab will show only the code that is written for the HTML, not any JS changes
  + The “Elements” tab will show changes to the DOM made by your JS code
* The “Styles” tab will show CSS code that is applied
  + A strikethrough will show code that is overridden by another style
* Use console.log(“”); to see what is happening with your code
* Use W3C HTML validator to validate your HTML code (<http://validator.w3.org/>)
* Using Aptana: Commands->HTML->Validate Syntax (W3C) command

**Browser Objects, Cookies, and Web Storage**

* Chapter 15 of Murach

**jQuery**

**Including jQuery in a Web Page**

* Include the script file just like any other JS file
  + <script src=”jquery-3.1.1.min.js”></script> // if the file name is jquery-3.1.1.min.js that is
  + I have version 3.3.1 saved as D:\Chuck Tucker\Documents\Comp Sci\jQuery\jquery-3.3.1.min.js
  + **Include the jQuery file first before your .js files** (especially if your JS file uses jQuery!)
* Use a content delivery network (CDN) (**best practice**)
  + <script src=<https://code.jquery.com/jquery-3.1.1.min.js>></script> // again with the file name
  + Use the “slim” version with jquery-3.1.1.slim.min.js
  + jQuery recommends using **Subresource Integrity (SRI)** checking to be sure the files you’re referencing haven’t been tampered with p. 231 Murach (<https://www.srihash.org/>)
    - **USE THIS METHOD FOR WEB RELEASES:**
      * <script src="https://code.jquery.com/jquery-3.3.1.min.js" integrity="sha384-tsQFqpEReu7ZLhBV2VZlAu7zcOV+rXbYlF2cqB8txI/8aZajjp4Bqd+V6D5IgvKT" crossorigin="anonymous"></script>
* Older versions
  + <script src=”<https://code.jquery-3.1.1.min.js>”

integrity=”sha256-hVVnYaiADRTO2PzUGmuLJr8BLUSjGIZsDYGmIJLv2b8=”

crossorigin=”anonymous”></script> // from Murach

* + May need to include migrate plugins to upgrade the app if using older code
    - versions 1.9 & 3.0 need these plugins
    - Visit jquery.com/upgrade-guide for help

**jQuery Selectors**

* Syntax
  + $(“*cssSelector*”)
  + Separate multiple selectors with commas
* Recall the common CSS selectors
  + #*idName*
  + .*className*
* Examples:
  + $(“p”) // selects every <p> element
  + $(“#pause”) // selects the element with the id of “pause”
  + $(“.minus”) // selects every element with the class of “minus”
  + $(“#faqs p”) // selects every <p> element within the element with id=”faqs” (descendants)
  + $(“h2 + div”) // selects the <div> elements that are adjacent siblings to <h2> elements
  + $(“ul ~ p”) // selects any <p> element that is a sibling of a <ul> element (not just adjacent)
  + $(“div > ul”) // selects any <ul> elements that are direct children of a <div> element
  + $(“#faqs li, div p”) // selects every <li> within id=”faqs” and every <p> within a <div>
* Summary of jQuery Selectors (all of these are appended to a selector: see first two examples)
  + $(‘*selector*[*attribute*]’) // chooses any of *selector* with *attribute* (all <p> that have a class)
  + $(‘*selector*[*attribute=value*]’) // same as above but now class=”*className*” for example
  + :contains(*text*) // all elements that contain the specified text
  + :empty // all elements with no children including text nodes
  + :eq(*n*) // element at index *n* within the specified set
    - $(‘#faqs p:eq(2)’) // selects the third <p> element in id=”faqs” (0 index)
  + :even // all elements with an even index within the selected set
    - $(‘table > tr:even’) // selects all even rows (including first row where index is 0)
    - table > tr ensures that each <tr> is direct child of <table>
  + :first // the first element within the set
  + :first-child // all elements that are first children of their parent elements
    - $(‘li:first-child’) // selects the first list item from each list
  + :gt(*n*) // all elements within the selected set with index greater than *n*
  + :has(*selector*) // all elements that contain the element specified by the *selector*
  + :header // all elements that are headers (h1, h2 …)
  + :hidden // all elements that are hidden
  + :last // the last element within the selected set
  + :last-child // all elements that are the last children of their parent elements
  + :lt(*n*) // all elements within the selected set that have an index less than *n*
  + :not(*selector*) // all elements that aren’t selected by the selector
  + :nth-child // all elements that are the nth children of their parent elements
  + :odd // all elements with an odd index
  + :only-child // all elements that are the only children of their parent element
  + :parent // all elements that are parents of other elements, including text nodes
  + :text // all input elements with the type attribute set to “text”
    - $(‘:text’) // will select every single element with type=”text”
    - $(‘input[type=text]’) // will do the same thing (every input with type=”text”)
  + :visible // all elements that are visible
* This
  + As a handler for an event, use $(this) to add actions
  + Example:
    - $(“:text”).dblclick(function () {

$(this).val(“”); // clears any text input when you double click it, but only “this” box

});

**Methods**

* Syntax for calling a jQuery method
  + $(“*selector*”).*methodName*(*parameters*);
* Common jQuery methods
  + .val(); // returns the value of a text box or other form control
  + .val(*value*); // sets the value of a text box or other form control
  + .text(); // get the text of an element
  + .text(*value*); // set the text of an element
  + .attr(*attributeName*); // get the value of the specified attribute from the first selected element
    - $(‘#image’).attr(‘src’) // will return the src value for id=”image”
  + .attr(*attributeName, value*); // set the value of the specified attribute for each selected element
    - $(‘#image’).attr(‘src’, imageSource); // sets the src value for id=”image” using a variable
  + .css(*propertyName*); // get the value of the specified property from the first selected element
  + .css(*propertyName, value*); // set the value of the specified property for each selected element
    - $(“h2”).css(“color”, “blue”); // will set all <h2> text color to blue
  + .addClass(*className*); // add one or more (separated by spaces) classes to the selected elements/creating if doesn’t exist
    - $(‘#faqs h2’).addClass(‘minus’); // adds the class=”minus” to all <h2> in the id=”faqs” section
  + .removeClass([*className*]); // remove one or more classes (separated by spaces)
  + .toggleClass(*className*); // if the class is present, remove it, otherwise add it
  + .html(*htmlString*); // sets the HTML contents of each selected element to the specified string
    - $(“aside”).html(“<h2>Table of Contents</h2>”); // put an <h2> into an aside element
    - like innerHTML
  + .next([*type*]); // get the next sibling of each selected element or the next sibling of type (optional)
  + .prev([*type*]); // get the previous sibling of each selected element or the previous of type (optional)
  + .find(*selector*); // search the selected element and return descendant elements
  + .each(*function*); // run the function for each element in an array
    - $(“#image\_list a”).each(function() { /\* statements \*/ } ); // run function on each <a> in that id
  + .submit(); // submit a form
  + .focus(); // move the focus to the selected form control or link
* Effect Methods
  + General syntax
    - methodName([*duration*] [, *callback*]);
  + .hide([*duration*]); // hide the selected elements
    - duration can be “slow”, “fast” or a number in milliseconds
    - default is 400 ms, slow is 600 ms, and fast is 200 ms // if omitted, appears to be immediate
  + .show([*duration*]); // show the selected element, duration the same as .hide
  + toggle(); // display or hide
  + slideDown(); // display the element with a sliding motion
  + slideUp(); // hide the element with a sliding motion
  + slideToggle(); // **display/hide using a sliding motion (good for FAQs type thing) or menu dropdowns**
  + fadeIn(); // display by fading in
  + fadeOut(); // hide by fading out
  + fadeToggle(); // display or hide using fading
  + .fadeTo(*duration, opacity*[, *callback*]); // fade to the specified opacity level
    - opacity ranges from 0 (transparent) to 1 (fully opaque)
  + callback functions
    - coded as last parameter (nested function within the method)
    - usually other animations like below
      * function() { $(this).fadeTo(1000, 1); }
* Animate Method
  + Use for custom animation effects
    - Changes target(s) to the specified properties map over the optional duration before executing optional callback function
  + Syntax
    - animate({*properties*}[, duration][, *callback*]);
  + Properties map
    - name/value pairs (i.e. fontSize: 75%;) // very css-esq
    - Syntax for names in properties map
      * Use camel casing (i.e. fontSize)
      * Can use css naming but enclose name in quotes (“font-size”: )
    - Syntax for values in properties map
      * non-numeric (if any characters other than numbers) use “” around the value
      * units of px are assumed for numbers
      * values of “show” “hide” and “toggle” can be used
    - Color transitions are not handled properly by jQuery but are by jQuery UI
  + delay(*duration*)
    - Delay the start of the net animation in the queue
  + stop([*clearQueue*][, *jumpToEnd*])
    - Stop the current animation for the selected element
    - Two optional parameters have “false” as default
    - clearQueue = true will cancel any further animations
    - jumpToEnd = true will cause the current animation to be completed immediately
  + finish(*queue*)
    - Stop the current animation for the selected element
    - clear the queue
    - complete all animations for the selected element
  + Easings
    - You can add varying speeds and bounces to animations using easings
    - Need to add easing plugin from a CDN
      * add script after jQuery script
      * p. 279 Murach for more info
  + Advanced syntax
    - Can add certain options to animations
    - see p. 281 Murach for more info
  + Carousel app info p. 283-284 Murach (ch 9 apps/exercises)

**Event Methods**

* Syntax for an event method
  + $(“*selector*”).*eventMethodName*(function() {

// statements of the event handler

});

* Common jQuery event methods
  + ready(*handler*) // event handler runs when the DOM is ready (like onload but faster)
    - $(document).ready(function() { /\* statements \*/ } );
    - $(function() { /\* statements \*/ } ); // works the same as above
  + click(*handler*) // event handler runs when the sleected element is clicked
    - $(“a”).click(function() { /\* statements \*/ } ); // to select all <a> and set onclick events
  + dblclick(*handler*) // runs on double click
  + mouseenter(*handler*) // runs when mouse enters the selected element
  + mouseover(*handler*) // runs when mouse moves over the selected element
  + mouseout(*handler*) // runs when mouse moves out of the selected element
  + hover(*handlerIn, handlerOut*) // two events that run when the mouse enters, then when exits
  + event.preventDefault() // stops the defaults action from happening
    - When you need to use this:
      * Clicking on an <a> tag that doesn’t want to load the href value into the page
      * code evt as function parameter:

$(“#image\_list a”).click(function(evt) { … evt.preventDefault(); } ) ;

* + Nesting click events within DOM ready
    - $(function() {

$(“a + div”).click(function() {

// click statements that select all <div> that are adj sibs to <a>

}); // end click

}); // end ready (end of event comments usually included for debugging)

* Other event methods
  + on(*event[s]*, *handler*)
    - $(“#clear”).on(“click, function () { … } ); // works the same as below, but accepts multiple events on(“click mouseover”, function() { … } );
    - $(“#clear”).click(function () { … } );
  + off(*event[s], handler*) // remove an event handler from one or more events
  + one(*event, handler*) // the event will run the handler only once, then it will be removed
  + trigger(*event*) // trigger the event for the selected element
    - Can use to attach multiple events to the same handler (dblclick, click etc.)
    - Shortcut method
      * $(“#clear”).trigger(“click”) vs. $(“#clear”).click().
    - Example
      * $(“#clear”).dblclick(function () {

$(“#clear”).click(); // calls the click event method previously defined

});

**Forms and Data Validation with jQuery**

* HTML5 tricks (in <input> tag)
  + **Usually don’t do this though, and just use JS to validate (use type=”text”)**
    - This prevents unexpected browser messages
  + Input types that use browser validation by default using html
    - email
    - url
    - number
    - range
  + date will show a date picker (still need to validate)
  + Can add the key word *required* to an html form to require the field with browser validation
  + Can add regular expressions (pattern=””) to html form if using html5
  + *novalidate* in the <input> will tell the browser not to validate that field
  + Can set *autocomplete=”off”* to disable autocomplete for a form or control
* jQuery Selectors for from controls
  + :input // all input, select, textarea, and button elements
    - var *varName* = $(“input[name=’*field\_name’*]”).val(); // get value from typed input
    - var *varName* = $(“input[name=’*field\_name’*]:checked”).val(); // get value from check/radio
  + :text // all text boxes: input elements with type=”text”
  + :radio // all input type=”radio”
    - really only use this if selecting all radio, otherwise use the :checked example below
    - $(“:radio”).change() can be used to check the value of a a radio button to choose to display/hide other options depending on what is checked (function with if statement) p. 303 Murach
    - inputs require to ask for val() but :radio doesn’t (see checkbox example below)
  + :checkbox // all input type=”checkbox”
    - same use as :radio vs. :checked
    - get the values of all checkboxes
      * var *varName* = [];
      * *varName* = $(“:checkbox:checked”);
      * inputs require you to ask for val() but :checkbox doesn’t
  + :file // all file upload fields input type=”file”
  + :password // all type=”password”
  + :submit // all input type=”submit” and button elements
  + :reset // all input type=”reset”
  + :image // all input type=”image”
  + :button // all button elements and input type=”button”
  + :disabled // all elements with disabled attribute
  + :enabled // all elements that don’t have disabled attribute
  + :checked // all check boxes and radio buttons that are checked
    - inputs require you to ask for val(), but :radio & :checkbox don’t
    - If getting an array, initialize var as array
    - var *inputArray* = [];

*inputArray* = $(“input[name=’*flounder\_fields*’]:checked”).val(); // “input” requires val()

*inputArray =* $(“:checkbox:checked”);

* + :selected // all options in select elements that are selected
    - Note not using val() here
    - If allowing multiple selections and storing in an array:
    - *var inputArray* = [];

*inputArray* = $(“#*select\_list* :selected”); // use space because “options” are selected (children of id=”select\_list”

or

*inputArray* = $(“#*select\_list* option:selected”); // no space if specify option first

* jQuery methods for form controls
  + val() // gets the value
  + val(*value*) // sets the value
  + trim() // removes spaces at the start and end of a string
* **\* Examples \***
  + Get numeric entry from a text box
    - var *age* = parseInt($(“#*age*”).val()); // stores input from id=”age” as an integer in var age
  + **Trim an entry, then put it back in the text box it came from (common to do this during validation)**
    - var *firstName* = $(“#first\_name”).val().trim(); // get and trim the entry and store in var
    - $(“#first\_name).val(firstName); // replace contents using the var
* jQuery Event Methods for Forms
  + focus(*handler*) // handler runs when the focus moves to the selected element
  + blur(*handler*) // handler runs when the focus leaves the selected element
  + change(*handler*) // handler runs when the value in the selected element is changed
    - Example to show new options based on user entries
    - This example will show radio boxes for how the user wants to be contacted, but only if the user checks the id=”contact\_me” checkbox
    - $(“#contact\_me”).change(

function() {

if ($(“#contact\_me”).attr(“checked”)) { // if contact\_me is checked

$(“:radio”)attr(“disabled”, false) } // enable radio boxes (all)

else {

$(“:radio”)attr(“disabled”, true) } // else disable

});

* + select(*handler*) // handler runs when the user selects text in a text or textarea box
  + submit(*handler*) // handler runs when a submit button is clicked
* jQuery methods for triggering events
  + focus() // moves the focus to the selected element and triggers the focus event
  + blur() // removes the focus from the selected element and triggers the blur event
  + change() // triggers the change event
  + select() // triggers the select event
  + submit() // triggers the submit event
    - Using a button rather than a submit button for data validation and submission
    - $(function() { // $(document).ready(function() { // statement

$(“#*button\_id*”).click( // button, not submit

function() {

// data validation statements, only go to next statement if passes all

$(“#*form\_id*”).submit(); // submits form to server

} // end function

); // end click

}); // end ready

* Setting up messages for data validation
  + Use <span>\*</span> elements next to required fields
  + Use <span></span> elements next to fields that aren’t required but need validation if entered
  + Update the <span> if errors are detected & set isValid variable to false to prevent default action
    - $(“#*control\_id*”).next().val(“*Validation error message*.”);
    - Code above selects the <span> after the field that failed validation
* Executing Data Validation
  + Either make your “submit” button a type=”button” and attach a handler that validates then calls the submit() method if passes
  + Or make your type=”submit” and attach validation handler that will preventDefault() if any of the validation fails // sometimes using isValid variable set to false if anything fails
    - if (isValid == false) { event.preventDefault(); } // last statement in the event handler below
    - need to setup this event to include the “event” object
    - $(“#*form\_id*”).submit(

function(event) { … data validation … } ) ;

* + Usually need to trim entries and replace back in the box before testing for validation and submitting form ex. var password = $(“#password”).val().trim(); $(“#password”).val(password);

**jQuery Plugins**

* Plugins can simplify code and help design apps faster, and they use the standard jQuery library
* Finding jQuery plugins
  + You can google search, just make sure to use “jQuery” in the search
  + jQuery repository websites
    - <http://plugins.jquery.com>
    - <http://jquery-plugins.net>
  + General code reposityory websites
    - <http://code.google.com>
    - <https://github.com>
    - <http://sourceforge.net>
* Displaying Images
  + Popular plugins
    - <http://lokeshdhakar./projects/lightbox2/> // Lightbox (more info below)
    - <http://fancybox.net> // Fancybox
    - <http://codylindley.com/thickbox/> // Thickbox
    - <http://www.jacklmoore.com/colorbox> // ColorBox
* Slide Shows and Carousels
  + Popular plugins
    - <http://bxslider.com> // bxSlider (more info below)
    - <http://jquery.malsup.com/cycle2> // Malsup jQuery Cycle 2 (more info below)
    - <http://sorgalla.com/jcarousel> // jCarousel
* Using Plugins
  + Read documentation for…
    - HTML requirements (for design, and selectors)
    - CSS requirements (for interactivity)
    - Methods/options provided
    - Evaluate if the plugin does what you want
    - Version(s) of jQuery necessary for the plugin to work
  + Download plugin (to the server)
    - Will always have js files, but may include css or image files too
    - May have two js files, one that is written in standard view, and one with whitespace removed (compressed) that is usually the min.js file
    - Some may be available from a CDN much like the jQuery links
  + Code the css ref links in the HTML doc as needed
  + Code the js file(s) in the HTML doc as needed (remember order is important, plugins after jQuery library)
  + Make sure any image folders are in the correct location to be referenced by the css/js files
  + Code HTML and CSS as needed for the plugin to work
  + Code the jQuery as needed for the plugin to work (calling methods, etc.)
* Lightbox (view images)
  + <http://lokeshdhakar./projects/lightbox2/>
  + Enlarges thumbnails in a popup dialogue box/window, and can do sets this way
  + Contains js and css files
  + Contains no extra jQuery methods to be called, it just works! p. 312-313 Murach
* bxSlider (carousel)
  + <http://bxslider.com>
  + Many options to use p. 314-315 Murach
  + Contains CSS and image files
  + Options are set when calling the method
  + bxSlider does not work when run from Aptana (not sure why)
  + Can modify the CSS to move left/right arrows, captions, and controls
* Cycle 2 (slideshow)
  + <http://jquery.malsup.com/cycle2>
  + Uses a CDN to reference the plugin (src=”http://malsup.github.com/jquery.cycle2.js”)
  + Does not require any CSS or images
  + p. 316-317 Murach
  + Options set by coding attributes in the HTML file

**jQuery UI (User Interface)**

* Sort of like an official plugin for the jQuery library (uses jQuery library and provided by jQuery)
* <http://jqueryui.com> // to download the plugin (use the “stable” button for quick downloads)
* Contains widgets, themes, and interactions (less used)
  + The documentation is very useful for options etc.
  + Click on the widget or example you want
  + Explore it in the window
  + View Source link will show the source code for the example for review
  + API documentation link to review methods, options, and events
* Download
  + Go to the website and click “stable” for the quick way
  + Create a custom download for production websites to only include what you need (selectable options)
  + Widgets only need the -ui.min.css, ui.min.js file, and the images folder
  + Downloads include:
    - jQuery library in external folder // for use if not already using a different version (compatibility?)
    - index.html file that displays a page to demo the features
    - Compressess and uncompressed css/js files to use/review code
    - Structure and theme css files // not needed unless using only the structure or theme rule sets (because all of the rule sets are included in the core css file)
* Include the css file(s), the ui plugin (after the jQuery script), and my external file last
* Implementing and using widgets
  + Code link/script elements for the jQuery UI in the head of the HTML
  + Code the required html for the widget
  + Code the jQuery for running the widget
  + Ensure the files have the proper relationships
    - images folder at same level as the jQuery CSS file, etc.
* **Accordion Widget**
  + Hides/expands content with a click (like FAQ app from Murach)
  + <div id=”accordion”> that contains <h3></h3> titles for each section followed by the <div></div> that contains content for the panel
  + jQuery for the accordion
    - in the “doc ready” event
    - $(“#accordion”).accordion({

// options

});

* **Tabs Widget**
  + Has tabs you can click on to display the content of each tab
  + Similar to accordion, but displays content of a tab in panel
  + <div id=”tabs”> that contains everything
  + <ul> that contains <li><a href=”#tabs-1” (etc.)>Tab Text</a></li> attributes for the tabs (p. 327 Murach)
  + <div id=”tabs-1” (etc.)> for the content after the <ul>
  + jQuery for the tabs widget
    - $(“#tabs”).tabs(); // can set options, but often don’t need to
* **Button and Diaglog Widgets**
  + Turn something into a button (like a picture, say of a book or item) and display a dialog box (which might contain info about the item)
  + Dialogue box is resizable, draggable, and you can make the user close it before continuing
  + Set object to click as (image used as example) <a id=”*id*”><img src=”*file.png*” alt=”*alt text*”/></a>
  + <div id=”*dialog*” title=”*Title of the dialog box*” style=”display:none”;>*Dialog box text.*</div>
  + jQuery for the widget
    - $(“#*id*”).button(); // id of the <a> element

$(“#*id*”).click(function() { // id of the <a> element

$(“#*dialog*”).dialog({ // select the diaglog <div>

modal: true // must close the box to continue

});

});

* **Datepicker Widget**
  + Use a text input rather than a date input, set options including max/min dates
  + <label>*Input Name*:</label>

<label><input type=”text” id=”datepicker”></label>

* + $(“#datepicker”).datepicker({

minDate: new Date(), // sets min date to today

maxDate: +45, // sets max date to 45 days from now

showButtonPanel: true // will show “today” and “done” buttons on the picker

});

**Ajax**

**Ajax**

* Asynchronous Javacscript and XML
* Updates part of a page without refreshing
  + Google’s autofill in the search bar is an example (partial page refresh)
* How it works
  + Sends an XMLHttpRequest object (XHR object) to the server (may contain data)
  + The object receives the returned data and updates the DOM accordingly
  + js is used to issue the request, parse the returned data, and modify the DOM
  + Unlike an HTTP request, only data is returned from the server, not a new web page
  + Scripts written in PHP or ASP.NET are commonly used to return the data from the server
  + Ajax requests and server-side scripts must be compatible (either work together or design the server scripts with Ajax in mind)
  + Some sites have Application Programming Interfaces (APIs) that show how to use Ajax to get data from their sites (like Youtube, Flickr, Twitter, etc.)
* Data Formats with Ajax
  + HTML
    - Doesn’t have to be parsed, because already includes HTML tags
  + XML
    - Ajax was originally designed to use this language
    - Difficult to use js and jQuery to parse data in XML files
  + JSON
    - Most popular format for working with Ajax
    - Easily interfaces with PHP and ASP.NET
      * json\_encode() in PHP
    - [www.json.org](http://www.json.org) has APIs built in to languages (PHP/ASP.NET)
  + Other formats
    - plain text, YAML, CSV
* XMLHttpRequest Object
  + Methods
    - abort()
      * cancels the current request
    - getAllResponseHeaders()
      * returns a string that contains the names and values of all response headers
    - getResponseHeader(*name*)
      * returns the value of a specific response header
    - open(*method, url[, async][, user][, pass]*)
      * Opens a connection for a request
      * method = GET or POST
      * url // for the request
      * async = true or false // if true, app continues while request is processed
      * username and password if authentication is required
    - send([*data*])
      * starts the request, can include data, must come after a connection is opened
    - setRequestHeader(*name, value*)
      * specifies a name and value for a request header
  + Properties
    - readyState
      * numeric value (state of current request)
        + 0 is USNEST
        + 1 is OPENED
        + 2 is HEADERS\_RECEIVED
        + 3 is LOADING
        + 4 is DONE
    - responseText
      * content that’s returned from the server in plain text format
    - responseXml
      * content that’s returned from the server in XML format
    - status
      * status code from server in numeric format
        + 200 for success
        + 404 for not found
        + others
    - statusText
      * status message returned from the server in text format
  + Event
    - onreadystatechange
      * event that occurs when the state of the request changes
* Using the XMLHttpRequest object
  + p. 342-343 Murach for using js (no jQuery) an the HXR object to load XML data
* jQuery shorthand for working with Ajax
  + Methods
    - load(*url[, data][, success]*)
      * load html data
      * will only load content from files onteh same server as the page making the call
      * to test this method, must use Firefox or Safari (Chrome, IE, Edge, Opera won’t load it without a web server)
      * Can use Aptana to test, because is provides its own internal server
      * p. 346-347 for using with HTML data
    - $.get(*url[, data][, success][, datatype]*)
      * load data with a GET request
      * p. 348-349 Murach for using GET/POST and parsing XML data
    - $.post(*url[, data][, success][, datatype]*)
      * load data with a POST request
    - $.getJSON(*url[, data][, success]*)
      * load JSON data with a GET request
      * In general, the url for the file must be on a web server, not a file server
        + also must be in the some domain as the page making the request
        + this is when the JSON file processes and returning JSON
        + if it’s a simple JSON file that only contains the object, it may work on local computer/file server
      * p. 350-351 Murach
      * Example below
    - Parameters for all of the above methods
      * url // string for the URL where the request is sent
      * data // map or string that is sent to the server, usually to filter returned data
      * success // callback function executed if request is successful
      * datatype // string specifying type of data (html, xml, json, script, or text) (default XML)
    - $.ajax( { *options* } )
      * See more info below
      * Provides more control over the way the request works over the above options
    - $.each(*collection, callback*)
      * collection is an object or array
      * callback is function that’s done for each item in the collection
* Loading JSON Data Example
  + The JSON file (team.json)

{“teammembers”:[

{

“name”:”Agnes”,

“title”:”Vice President of Accounting”,

“bio”:”Over 14 years…”

},

{

“name”:”Damon”,

“title”:”Director of Development”,

“bio”:”Director of Development for…”

}

]}

* + HTML div element that receives the data
    - <div id=”team”></div>
  + The jQuery to get and process the data

$(function(){ // $(document).ready(function(){

$.getJSON(“team.json”, function(data) { // “data” object that receives data

$.each(data, function() { // process each collection (teammembers)

$.each(this, function(key, value) { // process each member for “this” member

$(#team).append( // add to <div id=”team”>

“Name: “ + value.name + “<br>” + // value of the name key

“Title: “ + value.title + “<br>” + // value of the title key

“Bio: “ + value.bio + “<br><br>”

); // end append

}); // end $.each for this

}); // end $.each for teammembers

}); // end $.getJSON

}); // end ready

* + Strategy
    - Sort through each collection (teammembers is the only collection in this example)
    - Sort through each item (a team member is an item)
    - Use object notation to obtain data and parse using html tags
    - The callback should accept the “data” parameter that is the object returned from the request
* Sending Data with an Ajax Request
  + In this case, the *varName* is a parameter in the PHP file that we are passing
  + The *callback* function should be provided after the $.get request or simply nested in the request to process the returned data (this is the function(data) from above)
  + Use a string
    - $.get(“*filename.php*”, “*varName*=*value*”, *callback*);

function *callback*(data) { /\* process data \*/ }

});

* + Use a map
    - $.get(“*filename.php*”, {*varName*:*value*}, *callback*);

function *callback*(data) { /\* process data \*/ }

});

* + Helper methods for sending data in an Ajax request
    - serialize()
      * Encodes a set of form elements as a string that can be used for the data parameter
      * Example
        + var *formData* = $(“#*formId”*).serialize();

$.get(“*filename.php”*, *formData*, *callback*);

* + - serializeArray()
      * Econdes a set of form elements as an array of name/value pairs
      * Similar to serialize
* $.ajax({ options }
  + Most control of all jQuery methods
  + Example on p. 357 Murach
  + Options
    - url
      * string for where the URL request is sent
    - beforeSend(*jqXHR, settings*)
      * function executed before the request is sent
      * jqXHR obect and a map of the settings for this object
    - cache
      * Boolean that determines if the browser can cache the response
    - complete(*jqXHR, status*)
      * function executed when the request finishes
      * jqXHR object and a string that represents the status of the request
    - data
      * map or string sent with the request, usually to filter the data returned
    - dataType
      * string that specifies the type of data (html, xml, json, script, or text) (xml default)
    - error(*jqXHR, status, error*)
      * function executed if the request fails
      * jqXHR object
      * status of the request
      * object to receive the text portion of the HTTP status
    - jsonp
      * string containing the name of the JSONP parameter to be passed to the server (default is “callback”)
      * JSON with padding
      * lets you request data from a server in a different domain
      * often used by APIs for websites
    - password
      * string containing a password that will be used to respond to an authentication challenge
    - success(*data, status, jqXHR*)
      * function executed if the request is successful
      * data returned
      * status string
      * and the jqXHR object
    - timeout
      * number of milliseconds after which the request will time out in failure
    - type
      * string specifying GET or POST
    - username
      * string with username to respond to authentication challenge
* Using Ajax to access Flickr
  + See p. 358-367 Murach