SER421 Module 1 Lab

Learning Outcomes:

1. Gain proficiency in HTML5 semantic description tags and input validation
2. Gain proficiency in applying CSS to create responsive web designs
3. Identify patterns of interaction design
4. Gain proficiency in Javascript

Activity 1: HTML5 (15%)

As discussed in the videos, HTML5 has added a number of new input types. You can see a list of 22 input types at http://www.w3schools.com/html/html_form_input_types.asp in the middle of the page under "HTML5 Input Types".

PART A (8): Review the sample HTML5 file given to you named HTML5_InputForms.html. It uses 12 of the 22 input types listed on the W3Schools page. Ignoring the types "submit" and "reset" (since buttons already exist for those), add examples of the other 8 input types to the bottom of the page (at the end of the form but before the submit and reset buttons). One of the types, *range* should restrict inputs to be between 5 and 25.

NOTE: You will need to test your additions using Google Chrome as Firefox and Safari do not support all of these types.

For submission: Save this file as Module1Activity1A.html

PART B (7): Next, create a new input element of type "text" intended to hold the numbers you typically see at the bottom of a check (see http://www.routingnumbers.org/). Use the *pattern* attribute to create a custom validation for the combination of 9-digit routing number, 12-digit account number, and 4-digit check number (an example is shown on that web page). Constraints on the pattern:

1. All 3 numbers must be all digits
2. All 3 must be fixed length, with leading 0s if needed to meet the fixed length, and have a single space between them.
3. The textbox must be lengthened to accommodate the entire string.
4. Label the area "Routing number" on your HTML form and name the input element "<asurite>_RTN" (e.g. kgary_RTN)

NOTE: you are only checking for a valid routing number input, not trying to fix user input. This should be implemented by the pattern, not by adding any Javascript (no Javascript allowed!).

For submission: Save this file as Module1Activity1B.html

Activity 2: Interaction Design (15%)

Find a website that is an example of:

- A hub task flow
- A wizard or guide task flow
- Progressive disclosure

NOTE: You may not use any website that already appears in your notes, or is your own website or that of someone in the class, or is from the same domain more than once. You must use distinct websites for each of the four.

You do not have to perform screen captures for these, but you must describe each site with:

1. a URL to the start of the flow, and the number of screens in the flow
2. a description of the interaction pattern used
3. user goal(s) accomplished by the interaction pattern,
4. A critical analysis of whether you think the flow is appropriate for the user goal(s) (2-4 sentences)
5. Suggest improvements to the flow.

Each of the 3 examples is worth 5 points, 1 pt each for 1-5. You may not answer #4 and #5 with "the flow is appropriate and I would make no changes" more than once!

For submission: Put your answers in a Word/OpenOffice/Text document named Module1Activity2.[docx|odt|txt]

Activity 3: CSS/Responsive Web Design (25%)

Download the Module1Activity3Given.zip file from Canvas. Unzip it in a clean directory and use File → Open to open the rwdcss file in your browser. What you will find is an absolutely awful, unresponsive web page that was my homepage about a decade ago! Your job with this task is to clean this page up!

PART A (10): Review the source of the page. You will find that it uses old HTML tags and attributes for formatting. Specifically examine how fonts are sized, and how elements are centered. Replace these feeble attempts at positioning and styling with appropriate CSS elements – Note the page at this point should look roughly equivalent still, but just use CSS.

For submission: Put the CSS directly in the head of the page, and name the page Module1Activity3A.html.

PART B (15): Not only is the page poorly styled, but the layout uses tables and width percentages and in several places these don't even work properly. The page is certainly not responsive. Using the W3Schools RWD Introduction Tutorial linked off Canvas at https://www.w3schools.com/css/css_rwd_intro.asp, do the following:

1. Remove the table-based layout and replace it with a responsive web design. The design should have 3 columns and multiple rows, but fill in the ugly whitespace gap in the middle of the page (you have flexibility to move content around).
2. Create a breakpoint: at 800 pixels width you should reduce from 3 columns to 2.
3. Create 2nd breakpoint: at 550 pixels width you should reduce from 2 columns to 1.

For submission: Put the CSS directly in the head of the page (add on to Part A), and name the page Module1Activity3B.html.

Overall, for Activity 3 submission you should have a zipfile named Module1Activity3.zip. It should have Module1Activity3A.html and Module1Activity3B.html at the root directory, and copy in the images directory from the given (you can just add your HTML files to where you unzipped the given file, and the zip it back up!).

Activity 4: Javascript Programming (45%)
For this activity you will create a Javascript (or Typescript)-based prefix calculator. A prefix calculator is one where the operator appears at the beginning of the expression. Your expressions will be written as JSON strings.

PART A (18%): For the initial calculator, process add/subtract against a single stored calculator value, initially 0. Examples:

1. `'{"operation" : "add", "operand" : 5}'` returns 5 (assumes a starting init value of 0)
2. `'{"operation" : "subtract", "operand" : 2}'` returns 3 (5-2)
3. `'{"operation" : "add", "operand" : 19}'` returns 22 (19+3)

You should implement a function *calc(string)* to compute your responses. Note you should have your function *return* the value, not console.log it. Assume we will test your solution by cutting and pasting your code into our browser console. Also note that each of the expressions above is a JSON string, not a literal Javascript object, so it will need to be parsed.

For submission: Save your solution as Module1Activity4A.js [.ts]

Part B (27%): For this part we will extend our calculator in 3 ways:

1. Make a Calculator class. You may choose the way to create Calculator objects based on the techniques presented in the notes and code walkthrough on objects in Javascript.
    a. The constructor should take a single parameter, a number value, to which the calculator is initialized
    b. The *calc* function should now be a method on Calculator
2. Add a function on the Calculator class named *undo* that takes no parameters and undoes the previous operation. The value it returns is the new current value of the Calculator. If there is no previous operation, set the calculator to zero and return zero.
3. Add the capability to store and access the history of calculator results.
    a. Add a property that is a *stack* that stores the sequence of results from the calculator. For example, suppose the 3 expressions in part A were given in sequence, then the contents of the stack would be `[22, 3, 5]` where 22 in the top of the stack (and the Calculator's *current value*).
    b. Add the following functions to the Calculator class:
        i. `peek([optional] n)`: with no arguments, `peek()` returns the value at the top of the stack without popping the stack. With a single numeric argument n, peek returns the nth stack element (e.g., `peek(1) == 3`). Again note that the parameter is optional. If the stack is empty or `n > stack.length`, return null.
        ii. `pop()`: returns the top of the stack and removes that element from the stack. If the stack is empty return null.
        iii. `printMe()`: prints the contents of the stack to the console and does not return a value.
        iv. `clear()`: resets the Calculator to a stack == [0]. Resets any previous operations (no undo may be applied).

For submission: Save your solution as Module1Activity4B.js [.ts]

Note: You may include your own test cases for Activity 4 (parts A or B). These may be direct function calls or calls to calc with JSON string expressions. Put these in a separate file named Module1Activity4[A|B].js. We'd expect to run these by copy-paste after loading your code. Test cases may be shared freely on Slack with the class.

Constraints for Activity 4:

1. You should practice defensive programming by validating input types on your functions.
2. You do not have to check for well-formedness or errors in the JSON.
3. We are not doing file I/O, so assume test cases we will do to be in 2 forms:
    a. Consecutive calls of the calc function with JSON string expressions
    b. Direct calls of the methods on the Calculator class (for Part B).
4. Keep in mind that "return" and "console.log" are not the same thing. Pay attention to which is asked for.

Submission:

- Your submission should be a single zipfile named <asurite>_lab1.zip that will be submitted to Canvas

- In the zipfile should be the following files:
    1. Module1Activity1A.html
    2. Module1Activity1B.html
    3. Module1Activity2.[docx|odt|txt]
    4. Module1Activity3.zip
    5. Module1Activity4A.js
    6. Module1Activity4B.js
    7. (Optional) Any test cases you may want to add for Activity 4
    8. (Optional) Readme
- You may include a README.txt in the root directory of the zipfile that explains to us anything you want us to know about your submission. We will read this before we start grading. This will be standard practice for this class.
- You may include additional test case files if you develop them (you should!). Indicate in your README.txt. You are also free to share test cases for Activity 4 on the class Slack.
- We reserve, at our discretion, the right to deduct up to 10% of the grade for poor coding practices unbecoming upperclassmen in software engineering. This can include (but is not limited to) deductions for a) not following instructions and b) not following proper coding practices (documentation/comments, indentation, program structure, variable names, etc.).