M2.F1: Lab 2: Modules and Error Checking

Due Sep 26, 2022 by 11:59pm **Points** 100 **Submitting** a file upload
File Types zip **Available** after Sep 19, 2022 at 12am

CS-546 Lab 2

The purpose of this lab is to familiarize yourself with Node.js modules and further your understanding of JavaScript syntax.

In addition, you must have error checking for the arguments of all your functions. If an argument fails error checking, you should throw a string describing which argument was wrong, and what went wrong. You can read more about error handling on the **MDN (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/throw)** .

You can download the starter template here: **lab2_stub.zip (https://sit.instructure.com/courses/62921/files/10237903?wrap=1)** **(https://sit.instructure.com/courses/62921/files/10237903/download?download_frd=1)** **PLEASE NOTE: THE STUB DOES NOT INCLUDE THE PACKAGE.JSON FILE. YOU WILL NEED TO CREATE IT! DO NOT FORGET TO ADD THE START COMMAND**

Initializing a Node.js Package

For all of the labs going forward, you will be creating Node.js packages, which have a `package.json`. To create a package, simply create a new folder and within that folder, run the command `npm init`. When it asks for a package name, name it **cs-546-lab-2**. You may leave the version as default and add a description if you wish. The entry file will be `app.js`.

All of the remaining fields are optional **except** author. For the author field, you **must** specify your first and last name, along with your CWID. **In addition,** You must also have a start script for your package, which will be invoked with `npm start`. You can set a start script within the `scripts` field of your `package.json`.

Here's an example of a valid package.json:

```
{
   "name": "cs-546-lab-2",
   "version": "1.0.0",
   "description": "My lab 2 module",
   "main": "app.js",
   "scripts": {
      "start": "node app.js"
   },
   "author": "John Smith 12345678",
```

```
  "license": "ISC"
}
```

arrayUtils.js

This file will export 3 functions, each of which will pertain to arrays.

arrayStats(array)

This function will return an `object` with the following stats of an array: **mean**, **median**, **mode**, **range**, **minimum**, **maximum**, **count** and **sum**. **You will first sort the array from lowest to highest numbers before performing your calculations.**

Note: If there is no mode, you will return 0 for that key. If there is more than one mode, you will return an array for the mode that has all the modes as elements (sorted by lowest to highest number).

Reminder: The order of the keys of an object do not matter. For example: `{a: 1, b:2. c:3}` is the same as/equal to `{c:3, a:1, b:2}`

You must check:

- That the array exists
- The array is of the proper type (meaning, it's an array)
- The array is not empty
- Each array element is a number (can be positive, negative, decimal, zero)

If any of those conditions fail, you will throw an error.

```
arrayStats([9,15,25.5, -5, 5, 7, 10, 5, 11, 30, 4,1,-20]); // Returns: { mean: 7.5, median: 7, mode: 5, rang
e: 50, minimum: -20, maximum: 30, count: 13, sum: 97.5 }

arrayStats([7, 9, 11, 15, 19, 20, 35, 0]); // Returns: { mean: 14.5, median: 13, mode: 0, range: 35, minimum:
0, maximum: 35, count: 8, sum: 116 }

arrayStats([11, 54, 79, 5, -25, 54, 19, 11, 56, 100]); // Returns: { mean: 36.4, median: 36.5, mode: [11,54],
range: 125, minimum: -25, maximum: 100, count: 10, sum: 364 }

arrayStats([]) // throws an error
arrayStats("banana"); // throws an error
arrayStats(["guitar", 1, 3, "apple"]); // throws an error
arrayStats(); // throws an error
```

makeObjects(array1, array2, array3, ...)

For this function, **you will have to take into account a variable number of input parameters**. You will take in arrays as input. **Each array should have two and only two elements.** You will construct an `object` that will have the first element of each array as the key and the second element of each array as the value. If more than one of your arrays has the same first element, you will take the value from the

last one that was passed in. So for example: `makeObjects (["foo", "bar"], ["name", "Patrick Hill"], ["foo", "not bar"])` would return `{foo: "not bar", name: "Patrick Hill"}` and `makeObjects (["foo", "bar"], ["name", "Patrick Hill"], ["foo", "not bar"], ["class", "CS-546"], ["name", "John Smith"], ["foo", "not bar and not 'not bar'"])` would return `{foo: "not bar and not 'not bar'", name: "John Smith", class: "CS-546"}`

You only have to worry about primitive datatypes being used for the elements (string, boolean, number, null, undefined)

You must check:

- That each input is an array
- Each array is of the proper type (meaning, it's an array)
- Each array is not empty
- Each array has two and only two elements

If any of those conditions fail, you will throw an error.

```
makeObjects([4, 1], [1, 2]); // returns {'4':1, '1': 2}
makeObjects(["foo", "bar"], [5, "John"]); // returns {foo:'bar', '5': "John"}
makeObjects (["foo", "bar"], ["name", "Patrick Hill"], ["foo", "not bar"]) //returns {foo: "not bar", name:
"Patrick Hill"}
makeObjects([true, undefined], [null, null]); // returns {true: undefined, null : null}
makeObjects([undefined, true], ["date", "9/11/2022"]); // returns {undefined: true, date : "9/11/2022"}
makeObjects([4, 1, 2], [1,2]); // throws error
makeObjects([]) // throws an error
makeObjects("banana"); // throws an error
makeObjects(1,2,3); // throws an error
makeObjects(["guitar", 1, 3, "apple"]); // throws an error
makeObjects(); // throws an error
makeObjects([1],[1,2]); // throws an error
```

commonElements(array1, array2, array3, ...)

For this function, **you will have to take into account a variable number of input parameters**. This function will return an array of elements that appear in every array passed as input parameters. If there are no element values that appear in more than one array, just return an empty array.

You only have to worry about primitive datatypes being used for the elements (string, boolean, number, null, undefined) and two dimensional arrays.

You must check:

- That each input is an array and there are at LEAST two arrays passed in as input parameters.
- Each array is of the proper type (meaning, it's an array)
- Each array is not empty

```
If any of those conditions fail, you will throw an error.
const arr1 = [5, 7];
const arr2 = [20, 5];
const arr3 = [true, 5, 'Patrick'];
const arr4 = ["CS-546", 'Patrick'];
const arr5 = [67.7, 'Patrick', true];
```

```
const arr6 = [true, 5, 'Patrick'];
const arr7 = [undefined, 5, 'Patrick'];
const arr8 = [null, undefined, true];
const arr9 = ["2D case", ["foo", "bar"], "bye bye"]
const arr10= [["foo", "bar"], true, "String", 10]

commonElements(arr1, arr2); // Returns [5]
commonElements(arr3,arr4,arr5); // returns ['Patrick']
commonElements(arr5,arr6); // returns ['Patrick', true]
commonElements(arr9,arr6); // returns []
commonElements(arr7,arr8); // returns [undefined]
commonElements(arr3, arr4, arr5, arr7); // returns ['Patrick']
commonElements(arr9, arr10); // returns [["foo", "bar"]]
commonElements(); // throws error
commonElements("test"); // throws error
commonElements([1,2,"nope"]); // throws error
```

stringUtils.js

This file will export 3 functions, each are useful functions when dealing with strings in JavaScript.

palindromes(string)

Given a string, you will return an array with any palindromes that are contained in the string in the order in which they appear in the string. If there are no palindromes in the string, just return an empty array. (you only have to do it for single words that are in the string, not sentences) so for example "mom" is a palindrome which you'd have to handle, but a full sentence that is a palindrome like "A Santa dog lived as a devil god at NASA." for example, you don't have to worry about.

You must check:

- That `string` exists.
- The length of `string` is greater than 0 (a string with just spaces is not valid)
- That `string` is of the proper type (string)

If any of those conditions fails, the function will throw.

```
palindromes('Hi mom, At noon, I'm going to take my kayak to the lake'); // Returns: ["mom", "noon", "kayak"]
palindromes('Wow! Did you see that racecar go?'); // Returns: ["Wow", "Did", "racecar"]
palindromes('Hello World'); // Returns: []
palindromes(); // throws error
palindromes("    "); //  throws error
palindromes(1);  //throws error
palindromes(["hello there"]) //throws error
```

replaceChar(string)

Given `string` you will replace every other character with alternating `*` and `$` characters. Spaces, punctuation special characters all count as characters!

- That the string exists

- The length of the string is greater than 0 (a string with just spaces is not valid)
- The string is of the proper type

If any of those conditions fails, the function will throw.

```
replaceChar("Daddy"); // Returns: "D*d$y"
replaceChar("Mommy"); // Returns: "M*m$y"
replaceChar("Hello, How are you? I hope you are well"); // Returns: "H*l$o* $o* $r* $o*?$I*h$p* $o* $r* $e*l"
replaceChar(""); // Throws Error
replaceChar(123); // Throws Error
```

NOTE: In the above example of the output, you should NOT return it with quotes. The quotes are there to denote that you are returning a string. In your function, you just return the string. If you add quotes to your output, points will be deducted.

```
let returnValue = "myFunctionRocks"
return returnValue //This is the correct way to return it
return `"${returnValue}"` //This is NOT correct
return '"' + returnValue + '"' //This is NOT correct
```

charSwap(string1, string2)

Given `string1` and `string2` return the concatenation of the two strings, separated by a space and swapping the first 4 characters of each.

You must check:

- That both strings exist
- The strings are of the proper type
- The length of each string is at least 4 characters. (a string with just spaces is not valid)

If any of those conditions fail, the function will throw.

```
charSwap("Patrick", "Hill"); //Returns "Hillick Patr"
charSwap("hello", "world"); //Returns "worlo helld"
charSwap("Patrick", ""); //Throws error
charSwap(); // Throws Error
charSwap("John") // Throws error
charSwap ("h", "Hello") // Throws Error
charSwap ("h","e") // Throws Error
```

NOTE: In the above example of the output, you should NOT return it with quotes. The quotes are there to denote that you are returning a string. In your function, you just return the string. If you add quotes to your output, points will be deducted.

```
let returnValue = "myFunctionRocks"
return returnValue //This is the correct way to return it
return `"${returnValue}"` //This is NOT correct
return '"' + returnValue + '"' //This is NOT correct
```

objUtils.js

This file will export 3 functions that are useful when dealing with objects in JavaScript.

deepEquality(obj1, obj2)

This method checks each field (**at every level deep**) in `obj1` and `obj2` for equality. It will return `true` if each field is equal, and `false` if not. **Note: Empty objects can be passed into this function**.

For example, if given the following:

```
const first = {a: 2, b: 3};
const second = {a: 2, b: 4};
const third = {a: 2, b: 3};
const forth = {a: {sA: "Hello", sB: "There", sC: "Class"}, b: 7, c: true, d: "Test"}
const fifth  = {c: true, b: 7, d: "Test", a: {sB: "There", sC: "Class", sA: "Hello"}}
console.log(deepEquality(first, second)); // false
console.log(deepEquality(forth, fifth)); // true
console.log(deepEquality(forth, third)); // false
console.log(deepEquality({}, {})); // true
console.log(deepEquality([1,2,3], [1,2,3])); // throws error
console.log(deepEquality("foo", "bar")); // throws error
```

You must check:

- That `obj1` and `obj2` exists and is of proper type (an Object). If not, throw and error.

Hint: Using recursion is the best way to solve this one.

Remember: The order of the keys is not important so: `{a: 2, b: 4}` is equal to `{b: 4, a: 2}`

commonKeysValues(obj1, obj2)

This method checks each field (**at every level deep**) in `obj1` and `obj2` for and finds the common key/value pairs that appear in both `obj1` and `obj2`. You will return an object with the common keys/value pairs. if two empty objects are passed in or there are no common key/value pairs, just return an empty object.

For example, if given the following:

```
const first = {name: {first: "Patrick", last: "Hill"}, age: 46};
const second = {school: "Stevens", name: {first: "Patrick", last: "Hill"}};
const third = {a: 2, b: {c: true, d: false}};
const forth = {b: {c: true, d: false}, foo: "bar"};

console.log(commonKeysValues(first, second)); // returns  {name: {first: "Patrick", last: "Hill"}, first: "Pa
trick", last: "Hill"}
console.log(commonKeysValues(third, forth)); // returns {b: {c: true, d: false}, c: true, d: false }
console.log(commonKeysValues({}, {})); // {}
console.log(commonKeysValues({a: 1}, {b: 2})); // {}
console.log(commonKeysValues([1,2,3], [1,2,3])); // throws error
console.log(commonKeysValues("foo", "bar")); // throws error
```

You must check:

- That `obj1` and `obj2` exists and is of proper type (an Object). If not, throw and error.

Remember: The order of the keys is not important so: `{a: 2, b: 4}` is equal to `{b: 4, a: 2}`

calculateObject(object, func)

Given an object and a function, evaluate the function on the values of the object and then calculate the square root after you evaluate the function and return a new object with the results. Note, on the result, please use the toFixed(2) function to only display 2 decimal places rounded.

You must check:

- That the `object` exists and is of proper type (an Object). If not, throw and error.
- That the `func` exists and is of proper type (a function) If not, throw and error.
- That the `object` values are all numbers (positive, negative, decimal). If not, throw an error

You can assume that the correct types will be passed into the `func` parameter since you are checking the types of the values of the object beforehand.

```
calculateObject({ a: 3, b: 7, c: 5 }, n => n * 2);
/* Returns:
{
  a: 2.45,
  b: 3.74,
  c: 3.16
}
*/
```

Testing

In your `app.js` file, you must import all the functions the modules you created above export and create one passing and one failing test case for each function in each module. So you will have a total of 18 function calls (there are 9 total functions)

For example:

```
// Mean Tests
try {
   // Should Pass
   const meanOne = mean([2, 3, 4]);
   console.log('mean passed successfully');
} catch (e) {
   console.error('mean failed test case');
}
try {
   // Should Fail
   const meanTwo = mean(1234);
   console.error('mean did not error');
} catch (e) {
```

```
    console.log('mean failed successfully');
}
```

Requirements

1. Write each function in the specified file and export the function so that it may be used in other files.
2. Ensure to properly error check for different cases such as arguments existing and of the proper type as well as throw if anything is out of bounds such as invalid array index.
3. Import ALL exported module functions and write 2 test cases for each in `app.js`.
4. Submit all files (including `package.json`) in a zip with your name in the following format: `LastName_FirstName.zip`.
5. do NOT have the files in any folders, they should be in the root of the zip file
6. **You are not allowed to use any npm dependencies for this lab.**

CS 546 Labs Rubric				
Criteria	**Ratings**			**Pts**
Demonstrates critical thinking that considers all the edge cases to ensure a function returns intended results.	**100 to >90.0 pts** **Exemplary Competence** Demonstrates high competence in critical thinking that considers all the edge cases to ensure a function returns intended results.	**90 to >75.0 pts** **Developing Competence** Demonstrates developing competence in critical thinking that considers all the edge cases to ensure a function returns intended results.	**75 to >0 pts** **Insufficient Competence** Demonstrates insufficient competence in critical thinking that considers all the edge cases to ensure a function returns intended results.	100 pts
				Total Points: 100