Lab 2

Due Sep 23, 2021 by 11:59pm **Points** 100 **Submitting** a file upload
File Types zip **Available** after Sep 16, 2021 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)** .

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 `index.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": "index.js",
   "scripts": {
      "start": "node index.js"
   },
   "author": "John Smith 12345678",
   "license": "ISC"
}
```

arrayUtils.js

This file will export 4 functions, each of which will pertain to arrays.

average([arrays])

Given an **array of arrays**, you will return the rounded average value of all elements in the array(i.e use the `Math.round` method).

You must check:

- That the `array` parameter exists (meaning the function has input at all)
- The `array` parameter is of the proper type (meaning, it's an array)
- The `array` is not empty.
- Each `array` element is also an array and that there is at least one element within that array that is a number (no other datatypes can be used).

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

```
average([[1], [2], [3]]); // Returns: 2
average([[1,3], [2,4,5]]); // Returns: 3
average([[1,3], ["hi",4,5]]); // throws an error
average([[1,3], []]); // throws an error
average([[1,3], [1],[]]); // throws an error
average([]) // throws an error
average("banana"); // throws an error
average(["guitar", 1, 3, "apple"]); // throws an error
average(); // throws an error
```

modeSquared(array)

Returns the mode value of the elements of an array squared. **As the function name states, it's the mode squared, so you will first find the mode and then square it! If there is no mode, you will return 0. If there are multiple modes, you will sum the square of them.**

You must check:

- That the `array` parameter exists
- The `array` parameter is of the proper type (meaning, it's an array)
- The `array` is not empty
- Each `array` element is a `number`

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

```
modeSquared([1, 2, 3, 3, 4]); // Returns: 9
modeSquared([]) // throws an error
modeSquared("banana"); // throws an error
modeSquared(1,2,3); // throws an error
modeSquared(["guitar", 1, 3, "apple"]); // throws an error
modeSquared(); // throws an error
```

medianElement(array)

Scan the array from one end to the other to find the median element. Return both the median element of the array and the index (original position) of this element as a new object with the value as the key and the index as the value. If the array has an even length, you will take the average of the two elements as the key and the higher index as the value.

note: If there are multiple elements that are the same as the median, take the first index of it.

You must check:

- That the `array` parameter exists
- The `array` parameter is of the proper type (meaning, it's an array)
- The `array` is not empty
- Each `array` element is a `number`

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

```
medianElement([5, 6, 7]); // Returns: {'6': 1}
medianElement(5, 6, 7); // throws error
medianElement([]); // throws error
medianElement(); // throws error
medianElement("test"); // throws error
medianElement([1,2,"nope"]); // throws error
```

merge(arrayOne, arrayTwo)

Given two arrays, you will return one sorted array. Each element can either be a number or a character. You will first sort alphabetically (lowercase to uppercase) and then numerically.

You must check:

- That the `array` parameter exists
- Each `array` parameter is of the proper type (meaning it's an array)
- Each `array` parameter is not an empty array
- Each element is either a `number` or `char` string (can be either upper or lower case)
- If string element is not a char, then you will throw

You will not have to worry about if the elements are objects or functions. We will not test with objects or functions as elements for this function, however, you should test with all the other data types, boolean, number, string, undefined, null etc..

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

```
merge([1, 2, 3], [3, 1, 2]); // Returns: [1,1,2,2,3,3]
merge([1, 2, 3, 'g'], ['d','a', 's']); // Returns:['a', 'd', 'g', 's', 1, 2, 3]
merge(['A', 'B', 'a'], [1, 2, 'Z']); // Returns ['a', 'A', 'B', 'Z', 1, 2]
merge([null, null, null], [null, null, null]); // throws
merge([], ['ab, 'ts'] // throws
```

stringUtils.js

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

sortString(string)

Given a `string`, you will return the sorted string. You will sort the string from uppercase to lowercase, any special characters, numbers, and then spaces.

You must check:

- That the `string` parameter exists
- The length of the `string` parameter is greater than `0`
- The `string` parameter is of the proper type
- That the `string` parameter is not just empty spaces

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

```
sortString('123 FOO BAR!'); // Returns: "ABFOOR!123  "
sortString(); // Throws Error
sortString(''); // Throws Error
sortString(123); // Throws Error
sortString(["Hello", "World"]); // 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
```

replaceChar(string, idx)

Given `string` and index, you will find the value from the string index, then you will replace any characters in the string that are the same as that value except for that string index value. You will grab the value before and after the index and those are the values that you will be alternating between when replacing the characters.

- That the `string` parameter exists
- The length of the `string` parameter is greater than `0`
- The `string` parameter is of the proper type
- That the `string` parameter is not just empty spaces
- The idx parameter is valid within the string

- The idx parameter is greater than 0 and less than the length of the string - 2 (since we are grabbing the before and after values from the index)
- The idx parameter is of the proper type

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

```
replaceChar("Daddy", 2); // Returns: "Daday"
replaceChar("foobar", 0); // Throws Error
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 denot
e that you are returning a string. If you add quotes to your output, points will be deducted.
```

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

mashUp(string1, string2, char)

Given `string1` and `string2` return the concatenation of the two strings, with alternating characters of both strings. Note: If the strings are not the same length, you will pad the one that has less characters with the `char` parameter (you only use the 3rd parameter if `string1` and `string2` are not the same length: For example: "Patrick" and "Hill", "Hill" has 3 characters less than "Patrick" so you would pad "Hill" with the character supplied as the 3rd `char` parameter

You must check:

- That `string1,` `string2` and `char` parameters exist
- That `string1,` `string2 and char` parameters are of the proper type (they should all be strings)
- That the `string1` and `string2` and `char` parameters are not just empty spaces

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

```
mashUp("Patrick", "Hill", "$"); //Returns "PHaitlrli$c$k$"
mashUp("hello", "world", "#"); //Returns "hweolrllod"  notice that since both string are the same length, we
 can ignore the 3rd char parameter
mashUp("Hi", "There", "@"); //Returns "HTih@e@r@e"  notice since string1 is shorter than string2, we pad "Hi"
with 3 @ to make string1 and string2 equal lengths
mashUp("Patrick", ""); //Throws error
mashUp(); // Throws Error
mashUp("John") // Throws error
mashUp ("h", "Hello", 4) // Throws Error
mashUp ("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. DO NOT ADD EXTRA QUOTES TO IT!!

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

objUtils.js

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

computeObjects([objects], func)

This method will take in an array of objects and a function, and will return one object. You must check that each value is a number type. You will evaluate the function on the values of the object, if multiple objects have the same key, you will add that value to the current value. The return object will not have any duplicated keys.

Example:

```
const first = { x: 2, y: 3};
const second = { a: 70, x: 4, z: 5 };
const third = { x: 0, y: 9, q: 10 };
```

```
const firstSecondThird = computeObjects([first, second], x => x * 2);
//{ x: 12, y: 6, a: 140, z: 10 }
// x = (2 * 2) + (4 * 2)
```

- You must check to make sure an array is supplied as an input parameter
- You must check that an array of objects is supplied, if one or more elements in the array is NOT an object, you will throw an error.
- You must check that each object is not an empty object ; if any are empty, you must throw an error.
- You must also check that there are at least 1 elements (objects) in the array.
- You must check that each value is a number type
- You must make sure that the second parameter is a function

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

commonKeys(obj1, obj2)

This method will return a new `object` with key-value pairs that exist in both objects. Objects as values are valid. If no common keys are found, return an empty object.

For example, if given the following:

```
const first = {a: 2, b: 4};
const second = {a: 5, b: 4};
const third = {a: 2, b: {x: 7}};
```

```
const fourth = {a: 3, b: {x: 7, y: 10}};
console.log(commonKeys(first, second)); // {b: 4}
console.log(commonKeys(first, third)); // {a: 2}
console.log(commonKeys(second, third)); // {}
console.log(commonKeys(third, fourth)); // {b: { x: 7}}
console.log(commonKeys({}, {})); // {}
```

- You must check that each argument is provided
- You must check that each argument is an object.

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

Empty objects are valid as shown in the last example above.

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

flipObject(object)

Given an object, you will return a new object where the values are now the keys and the keys are now the value. If a value has a type of array, for each element, you will have the element as the key and the value will be the original key. If a value has an object, you will flip those keys and values as well, but keep the key as the same.

```
flipObject({ a: 3, b: 7, c: 5 });
/* Returns:
{
  3: a,
  7: b,
  5: c
}
*/
```

```
flipObject({ a: 3, b: 7, c: { x: 1 } });
/* Returns:
{
  3: a,
  7: b,
  c: {1: x}
}
*/
```

```
```

- You must check that the input parameter is an object
- You must check that the object has at least one key/value

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

Testing

In your `index.js` file, you must import all the modules you created above and create one passing and one failing test case for each function in each module. So you will have a total of 24 function calls (there are 10 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 or negative numbers for different operations.
3. Import ALL module functions and write 2 test cases in `index.js`.
4. Submit all files (including `package.json`) in a zip with your name in the following format: `LastName_FirstName.zip`.
5. You are not allowed to use any npm dependencies for this lab.