

Lab 2

Due Oct 3 by 11:59pm **Points** 100 **Submitting** a file upload

Available Sep 14 at 12am - Oct 3 at 11:59pm 20 days

This assignment was locked Oct 3 at 11:59pm.

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 \(https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/throw\)](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 6 functions, each of which will pertain to arrays.

mean(array)

Returns the [mean](https://www.mathsisfun.com/mean.html) value of the elements of an array

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

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

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

medianSquared(array)

Returns the [median](https://www.mathsisfun.com/median.html) value of the elements of an array squared

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

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

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

maxElement(array)

Scan the array from one end to the other to find the largest element. Return both the maximum element of the array and the index (position) of this element as a new object with the value as the key and the index as the value.

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

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

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

fill(end, value)

Creates a new numbered array starting at 0 increasing by one up to, but not including the **end** argument. The **value** argument is **optional**, but when specified each element will be set to that value.

You must check that the end number exists and is of proper type, and is a positive integer greater than **0**.

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

```
fill(6); // Returns: [0, 1, 2, 3, 4, 5]
fill(3, 'Welcome'); // Returns: ['Welcome', 'Welcome', 'Welcome']
fill(); // Throws error
fill("test"); // Throws error
fill(0); // Throws Error
fill(-4); // Throws Error
```

countRepeating(array)

Will return an object with the count of each element that is repeating in the array.

Note: Order does not matter in a JavaScript object, so your answer may have a different ordering.

Note: in JavaScript, all object keys are coerced to strings. For example:

```
const foo = { };
foo[1] = "bar";
```

```
foo["1"] = "foobar";

console.log(foo); // { "1": "foobar"}
```

You must check:

- That the array exists
- The array is of the proper type (meaning, it's an array)

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

This function allows empty arrays.

If an empty array is passed in, just return an empty object.

if there are no repeating elements, just return an empty object.

If the element's value is a number and there is a string value of that same number in the array, you can count that as a repeating element.

If it's a string, it's case sensitive.

Notice 7 and '7' are counted as 2 and Hello, Hello, hello is only counted as 2

```
countRepeating([7, '7', 13, true, true, true, "Hello","Hello", "hello"]);
/* Returns:
{
  "7": 2,
  true: 3,
  "Hello": 2,
}
*/
```

isEqual(arrayOne, arrayTwo)

Given two arrays, check if they are equal in terms of size. Next you will sort them in **ascending** order and then check the elements to see if they are equal. and return a boolean.

You must check:

- That the arrays exist
- Each array is of the proper type (meaning it's an array)

This function allows empty arrays. You must also take into account if it's an array of arrays!!!!

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

```
isEqual([1, 2, 3], [3, 1, 2]); // Returns: true
isEqual([ 'Z', 'R', 'B', 'C', 'A' ], [ 'R', 'B', 'C', 'A', 'Z' ]); // Returns: true
isEqual([1, 2, 3], [4, 5, 6]); // Returns: false
isEqual([1, 3, 2], [1, 2, 3, 4]); // Returns: false
isEqual([1, 2], [1, 2, 3]); // Returns: false
isEqual([[ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ]], [[ 3, 1, 2 ], [ 5, 4, 6 ], [ 9, 7, 8 ]]); // Returns: true
isEqual([[ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ]], [[ 3, 1, 2 ], [ 5, 4, 11 ], [ 9, 7, 8 ]]); // Returns: false
```

stringUtils.js

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

camelCase(string)

Given a string, you will construct a camel case version of the string, based on the spaces in words in the string

You must check:

- That the string exist
- The length of the string is greater than 0
- The string is of the proper type

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

```
camelCase('my function rocks'); // Returns: "myFunctionRocks"
camelCase('FOO BAR'); // Returns: "fooBar"
camelCase("How now brown cow"); // Returns: "howNowBrownCow"
camelCase(); // Throws Error
camelCase(''); // Throws Error
camelCase(123); // Throws Error
camelCase(["Hello", "World"]); // Throws Error
```

replaceChar(string)

Given `string` you will replace any characters in the string that are the same as the starting character but appear after the starting character (so not including the starting character) with alternating `*` and `$` characters.

Also, it is not case sensitive as shown in the examples below.

- That the string exist
- The length of the string is greater than 0
- The string is of the proper type

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

```
replaceChar("Daddy"); // Returns: "Da*$y"
replaceChar("Mommy"); // Returns: "Mo*$y"
replaceChar("Hello, How are you? I hope you are well"); // Returns: "Hello, *ow are you? I $ope you are well"
replaceChar("babbbbbb"); // Returns: "ba*$*$*le"
replaceChar(""); // Throws Error
replaceChar(123); // Throws Error
```

mashUp(string1, string2)

Given `string1` and `string2` return the concatenation of the two strings, separated by a space and swapping the first 2 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 2 characters.

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

```
mashUp("Patrick", "Hill"); //Returns "Hitrick Pall"
mashUp("hello", "world"); //Returns "wollo herld"
mashUp("Patrick", ""); //Throws error
mashUp(); // Throws Error
mashUp("John") // Throws error
mashUp ("h", "Hello") // Throws Error
mashUp ("h","e") // Throws Error
```

objUtils.js

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

makeArrays([objects])

This method will take in an array of objects , and will return an array of arrays where an array of each key and value is an element in the array.

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 = makeArrays([first, second, third]);
// [ ['x',2],['y',3], ['a',70], ['x', 4], ['z', 5], ['x',0], ['y',9], ['q',10] ]
```

```
const secondThird = makeArrays([second, third]);
// [ ['a',70], ['x', 4], ['z', 5], ['x',0], ['y',9], ['q',10] ]

const thirdFirstSecond = makeArrays([third, first, second]);
// [ ['x',0], ['y',9], ['q',10], ['x',2], ['y',3], ['a',70], ['x', 4], ['z', 5] ]
```

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 2 elements (objects) in the array.

isDeepEqual(obj1, obj2)

This method check 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.

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(isDeepEqual(first, second)); // false
console.log(isDeepEqual(forth, fifth)); // true
console.log(isDeepEqual(forth, third)); // false
console.log(isDeepEqual({}, {})); // true
```

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

Hint: You will need to use recursion.

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

computeObject(object, func)

Given an object and a function, evaluate the function on the values of the object and return a new object.

You must check that the object and function exist and they have proper types. If not, this function must throw.

```
computeObject({ a: 3, b: 7, c: 5 }, n => n * 2);
/* Returns:
{
  a: 6,
```

```
b: 14,  
c: 10  
}  
*/
```

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 12 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.