
centaurS

General

CentaurJS is a modern JavaScript helper library. It offers helper functions for *objects, arrays* and *strings*. If you have any experience working with *lodash* this should sound familiar.

Template

The template includes two files: **centaur.js** and **index.html**. The JavaScript file should contain the implementation and it is also setup with optional tests which can be uncommented and run. **centaur.js** can either be tested by running it with Node (**node centaur.js**) or opening up **index.html** which references **centaur.js**

Functionality

CentaurJS uses _**c**, e.g. **_c.countProperties(myObj);**. There are three sections **objects, arrays** and **strings** which all have some functions which should be implemented. In all functions below for all sections this applies: *if not otherwise stated it is safe to assume the arguments sent to the function are valid and require no type checking.*

(20%) Objects

When applying the object functions it should always exclude the properties from the prototype. For example **countProperties** should not count properties which reside in the prototype such as **toString**.

1. **(6.66%)** Implement a function called **setReadonly** which should set all properties in the object to readonly. The function should accept a single argument: **obj**. Read-only means that the properties cannot be changed after the **setReadonly** function has been executed on the object. The function should change the object provided as the first argument and return nothing.
   Example of usage:

   ```
   var myObj = { x: 1 };
   _c.setReadonly(myObj);
   ```

2. **(6.66%)** Implement a function called **countProperties** which accepts a single argument: **obj**. The function should return a number indicating how many properties reside in the object. **Example of usage:**

   ```
   var myObj = { x: 1, y: 2 };
   _c.countProperties(myObj); // 2
   ```

3. **(6.66%)** Implement a function called **toArray** which accepts a single argument: **obj**. It should return all property values from the object as an array.
   Example of usage:

   ```
   var myObj = {
       x: 1,
       student: { name: 'Miyagi' },
       list: [1, 2]
   }
   // [1, { name: 'Miyagi' }, [ 1, 2 ] ]
   _c.toArray(myObj);
   ```

(40%) Arrays

1. **(8%)** Implement a function called **removeAtIndex** which should remove an element at a specific index. The function should accept two arguments: **array**, **index**. The index is zero-based. If the **index** is out of bounds it should throw an error using the **throw** keyword. This function should remove from the array provided as an argument and return nothing.
   Example of usage:

```
var arr = [1, 2, 3];
_c.removeAtIndex(arr, 0);
console.log(arr); // [2, 3]
```

2. **(8%)** Implement a function called **removeByCondition** which should accept two arguments: **array**, **filterFn**. The filter function is an anonymous function which should be used to filter out data to remove. The filter function should accept a single argument: **element**. If an element is found, it should be removed from the array and the function should stop, therefore only removing one element. If nothing is found by the provided condition, nothing should be done. This function should remove from the array provided as an argument and return nothing.
   Example of usage:

```
var arr = [1, 2, 3];
_c.removeByCondition(arr, elem => elem > 1);
console.log(arr); // [1, 3]
```

3. **(8%)** Implement a function called **findIndexByCondition** which should accept two arguments: **array**, **filterFn**. The filter function is an anonymous function which should be used to filter out data to find. The filter function should accept a single argument: **element**. If an element is found it should return the index of that element and stop looking. Otherwise if nothing is found it should return **-1.** The function should start searching at the beginning of the array.
   Example of usage:

```
var arr = [1, 2, 3];
var idx = _c.findIndexByCondition(arr, elem => elem % 2 === 0);
var notFound = _c.findIndexByCondition(arr, elem => elem > 5);
console.log(idx); // 1
console.log(notFound); // -1
```

4. **(8%)** Implement a function called **sortByDescending** which should sort an array in a descending order (highest to lowest). The function should accept a single argument: **sortArray**. The function should return a new array which is sorted in a descending order, leaving the original array intact.
   Example of usage:

```
var arr = [2, 4, 1, 3];
var sortedArr = _c.sortByDescending(arr);
console.log(arr); // [2, 4, 1, 3];
console.log(sortedArr); // [4, 3, 2, 1]
```

5. **(8%)** Implement a function called **union** which should combine two arrays in a set-wise manner. With a set-wise manner I mean that there should be no duplicates in the new united array. The function should accept two arguments: **firstArray, secondArray** and return a new array which combines the two and removes all duplicates.
   Example of usage:

```
var unitedArray = _c.union([1, 2, 3], [1, 4, 5, 6]);
console.log(unitedArray); // [1, 2, 3, 4, 5, 6];
```

(40%) Strings

1. **(8%)** Implement a function called **capitalize** which accepts a single argument: a string. It should change the first letter in every word to uppercase, the string can contain multiple words which should all have an uppercase first letter. The function should return the new string. **Example of usage:**

```
// 1. capitalize
var capitalizedString = _c.capitalize('hi ho silver away!');
console.log(capitalizedString); // Hi Ho Silver Away!
```

2. **(8%)** Implement a function called **removeAllInstanceOf** which should accept two arguments: **originalString, removePattern.** The function should remove all occurrences of the **removePattern** from the **originalString**. If none are found the original string is returned. The function should return nothing but remove from the **originalString**.
   Example of usage:

```
var str = 'Morning!';
_c.removeAllInstanceOf(str, 'n');
console.log(str); // Morig!

var otherStr = 'NaNNaNNaNNaN Batman!';
_c.removeAllInstanceOf(otherStr, 'NaN');
console.log(otherStr); //  Batman!
```

3. **(8%)** Implement a function called **shiftRight** which should accept two arguments: **originalString, shiftNum**. The number indicates how many places the string should be shifted to the right. Shifting a string means that the letter in index 0 should move to index 1, when at the end of the string it should start again at the beginning. It should return the new shifted string, leaving the original string intact.
   Example of usage:

```
console.log(_c.shiftRight('number', 2)); // ernumb
console.log(_c.shiftRight('timber', 6)); // timber
```

4. **(8%)** Implement a function called **padLeft** which should add a padding to the string on the left side. The function accepts three arguments: **originalString**, **length, pattern**. The length indicates how long the string may be with the additional padding. Padding a string means that it should make the string as long as the second argument says and filling it up with the pattern provided by the third parameter. If the pattern does not fit within the length restriction, it should take only a part of the pattern (*starting from left*) as padding. The function should return a new string which has been padded from the left, leaving the original intact.
   Example of usage:

```
console.log(_c.padLeft('batman!', 20, 'NaN')); // NNaNNaNNaNNaNbatman!
console.log(_c.padLeft('12', 5, '0')); // 00012
```

5. **(8%)** Implement a function called **isValidSsn** which should check if the string is a valid SSN (kennitala). It accepts as first argument a string which should be validated. The function should return a boolean determining if it is valid or not. The format of an SSN is the following: *Length of 10, contains only numbers between 0 - 9*. It is advised to use RegExp in this implementation.
   Example of usage:

```
console.log(_c.isValidSsn('1111992319')); // true
console.log(_c.isValidSsn('111199231a')); // false
```

Other

All implementations should use **JavaScript** and no external libraries are allowed.