# Exercise: Unit Testing and Modules

Problems for exercises and homework for the ["JavaScript Advanced" course @ SoftUni](https://softuni.bg/courses/js-advanced). Submit your solutions in the SoftUni judge system at <https://judge.softuni.bg/Contests/1532/Exercise-Unit-Testing-and-Modules>

# Error Handling

## Request Validator

Write a function that **validates** an HTTP request object. The object has the properties method, uri, version and message. Your function will receive **the object as a parameter** and has to **verify** that **each property** meets the following **requirements**:

* method - can be GET, POST, DELETE or CONNECT
* uri - must be a valid resource address or an asterisk (\*); a resource address is a combination of alphanumeric characters and periods; all letters are Latin; the URI **cannot** be an empty string
* version - can be HTTP/0.9, HTTP/1.0, HTTP/1.1 or HTTP/2.0 supplied as a string
* message - may contain **any number** or non-special characters;special characters are <, >, \, &, ', "

If a request is **valid**, return it **unchanged**.

If any part **fails** the check, throw an Error with message "Invalid request header: Invalid {Method/URI/Version/Message}".

Replace the part in curly braces with the relevant word. Note that some of the **properties may be missing**, in which case the request is **invalid**. Check the properties **in the order** in which they are listed above. If **more than** one property is **invalid**, throw an error for the **first** encountered.

### Input / Output

Your function will receive an object as a parameter. Return the same object or throw an Error as described above as an output.

### Examples

|  |  |
| --- | --- |
| Input | Output |
| {  method: 'GET',  uri: 'svn.public.catalog',  version: 'HTTP/1.1',  message: ''  } | {  method: 'GET',  uri: 'svn.public.catalog',  version: 'HTTP/1.1',  message: ''  } |
| {  method: 'OPTIONS',  uri: 'git.master',  version: 'HTTP/1.1',  message: '-recursive'  } | Invalid request header: Invalid Method |

|  |  |
| --- | --- |
| {  method: 'POST',  uri: 'home.bash',  message: 'rm -rf /\*'  } | Invalid request header: Invalid Version |

### Hints

Since validating some of the fields may require the use of RegExp, you can check your expressions using the following samples:

|  |  |
| --- | --- |
| URI | |
| Valid | Invalid |
| svn.public.catalog  git.master  version1.0  for..of  .babelrc  c | %appdata%  apt-get    home$  define apps  "documents" |

* Note that the URI **cannot** be an **empty string**.

|  |  |
| --- | --- |
| Message | |
| Valid | Invalid |
| -recursive  rm -rf /\*  hello world  https://svn.myservice.com/downloads/  %root% | <script>alert("xss vulnerable")</script>  \r\n  &copy;  "value"  '; DROP TABLE |

* Note that the message **may** be an **empty string**, but the property must still be present.

# Unit Testing

You are required to **submit only the unit tests** for the object/function you are testing.

## Even or Odd

You need to write unit tests for a function isOddOrEven() that checks whether the length of a passed in string is **even** or **odd**.

If the passed parameter is **NOT** a string return undefined. If the parameter is a string return either **"**even**"** or **"**odd**"** based on the length of the string.

### JS Code

You are provided with an implementation of the isOddOrEven() function:

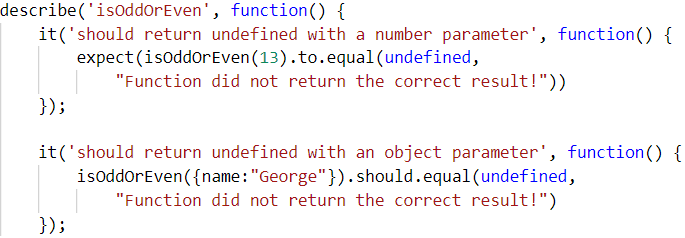
|  |
| --- |
| isOddOrEven.js |
| **function** *isOddOrEven*(string) {  **if** (**typeof**(string) !== **'string'**) {  **return undefined**;  }  **if** (string.**length** % 2 === 0) {  **return "even"**;  }   **return "odd"**; } |

### Hints

We can clearly see there are three outcomes for the function:

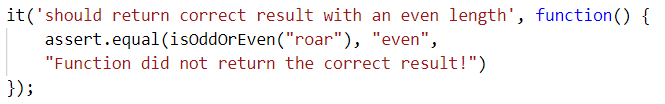
* Returning undefined
* Returning **"**even**"**
* Returning **"**odd**"**

Write one or two tests passing parameters that are **NOT** of type string to the function and expecting it to return undefined.

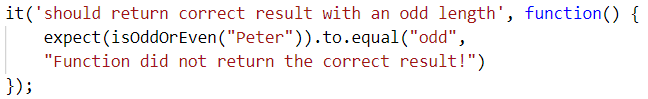


After we have checked the validation it's time to check whether the function works correctly with valid arguments. Write a test for each of the cases:

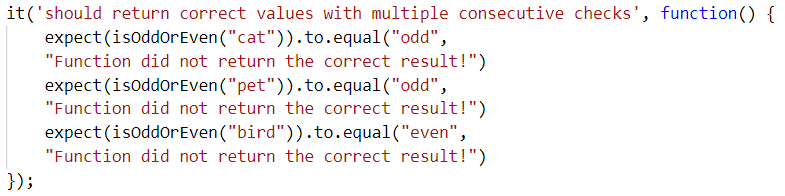
One where we pass a string with **even** length:



And one where we pass a string with an **odd** length:



Finally make an extra test passing **multiple different strings** in a row to ensure the function works correctly:



## Char Lookup

Write unit tests for a function that **retrieves a character** at a given **index** from a passed in **string**.

You are given a function named lookupChar(), which has the following functionality:

* lookupChar(string, index)- accepts a string and an integer (the **index** of the char we want to lookup) :
  + If the **first parameter** is **NOT a string** or the **second parameter** is **NOT a number** - return undefined.
  + If **both parameters** are of the **correct type** but the value of the **index is incorrect** (bigger than or equal to the string length or a negative number) - return **"**Incorrect index**"**.
  + If **both parameters** have **correct types** and **values** - return the character at the specified index in the string.

### JS Code

You are provided with an implementation of the lookupChar() function:

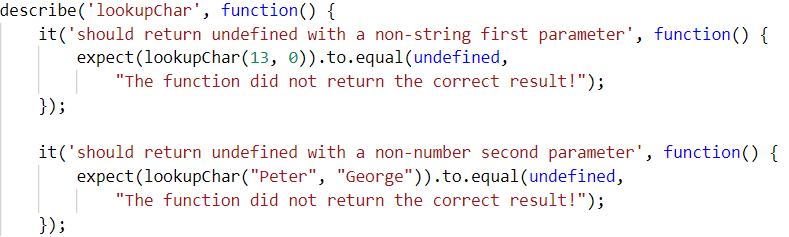
|  |
| --- |
| charLookUp.js |
| **function** *lookupChar*(string, index) {  **if** (**typeof**(string) !== **'string'** || !Number.isInteger(index)) {  **return undefined**;  }  **if** (string.**length** <= index || index < 0) {  **return "Incorrect index"**;  }   **return** string.charAt(index); } |

### Hints

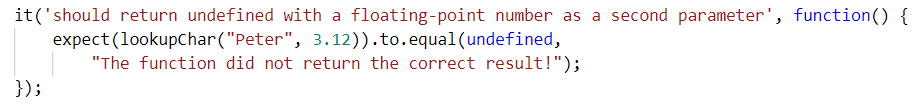
А good first step in testing a method is usually to determine all exit conditions. Reading through the specification or taking a look at the implementation we can easily determine **3 main exit conditions**:

* Returning undefined
* Returning anempty string
* Returning the char at the specified index

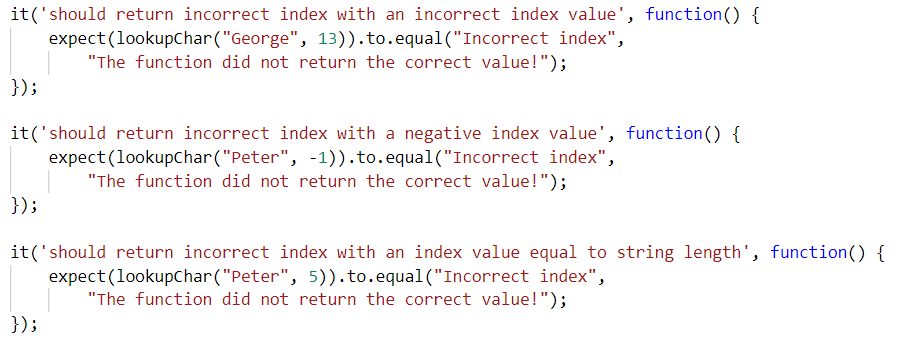
Now that we have our exit conditions we should start checking in what situations we can reach them. If any of the parameters are of **incorrect type**, undefinedshould be returned.



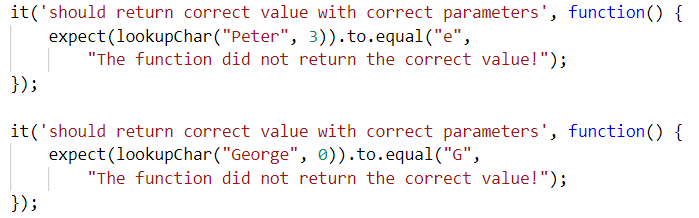
If we take a closer look at the implementation, we see that the check uses Number.isInteger() instead of typeof(index === number) to check the index. While typeof would protect us from getting passed an index that is a non-number, it won’t protect us from being passed a floating-point number. The specification says that **index** needs to be an **integer**, since floating point numbers won’t work as indexes.



Moving on to the next **exit condition** - returning an **empty string** if we get passed an index that is a **negative number** or an index which is **outside of the bounds** of the string.



For the last exit condition - **returning a correct result**. A simple check for the returned value will be enough.

  
With these last two tests we have covered the lookupChar() function.

## Math Enforcer

Your task is to test an object named mathEnforcer, which should have the following functionality:

* addFive(num) - A function that accepts a **single** parameter:
  + If the **parameter** is **NOT a number**, the funtion should return undefined.
  + If the **parameter** is a **number**, **add 5** to it, and return the result.
* subtractTen(num) - A function that accepts a **single** parameter:
  + If the **parameter** is **NOT a number**, the function should return undefined.
  + If the **parameter** is a **number**, **subtract 10** from it, and **return the result**.
* sum(num1, num2) - A function that accepts **two** parameters:
  + If **any** of the 2 parameters is **NOT a number**, the function should return undefined.
  + If **both** parameters are **numbers**, the function should **return their** **sum**.

### JS Code

You are provided with an implementation of the mathEnforcer object:

|  |
| --- |
| mathEnforcer.js |
| **let** mathEnforcer = {  addFive: **function** (num) {  **if** (**typeof**(num) !== **'number'**) {  **return undefined**;  }  **return** num + 5;  },  subtractTen: **function** (num) {  **if** (**typeof**(num) !== **'number'**) {  **return undefined**;  }  **return** num - 10;  },  sum: **function** (num1, num2) {  **if** (**typeof**(num1) !== **'number'** || **typeof**(num2) !== **'number'**) {  **return undefined**;  }  **return** num1 + num2;  } }; |

The methods should function correctly for **positive**, **negative** and **floating-point** numbers. In case of **floating-point** numbers the result should be considered correct if it is **within 0.01** of the correct value.

### Screenshots

When testing a **more complex** object write a nested describe for each function:



Your tests will be supplied with a variable named **"**mathEnforcer**"** which contains the mentioned above logic. All test cases you write should reference this variable.

### Hints

* Test how the program behaves when passing in **negative** values.
* Test the program with floating-point numbers (use Chai’s closeTo() method to compare floating-point numbers).

# Unit Testing On Classes

## String Builder

You are given the following **JavaScript class**:

|  |
| --- |
| string-builder.js |
| **class** StringBuilder {  constructor(string) {  **if** (string !== ***undefined***) {  StringBuilder.*\_vrfyParam*(string);  **this**.**\_stringArray** = Array.from(string);  } **else** {  **this**.**\_stringArray** = [];  }  }   append(string) {  StringBuilder.*\_vrfyParam*(string);  **for**(**let** i = 0; i < string.**length**; i++) {  **this**.**\_stringArray**.push(string[i]);  }  }   prepend(string) {  StringBuilder.*\_vrfyParam*(string);  **for**(**let** i = string.**length** - 1; i >= 0; i--) {  **this**.**\_stringArray**.unshift(string[i]);  }  }   insertAt(string, startIndex) {  StringBuilder.*\_vrfyParam*(string);  **this**.**\_stringArray**.splice(startIndex, 0, ...string);  }   remove(startIndex, length) {  **this**.**\_stringArray**.splice(startIndex, length);  }   **static** *\_vrfyParam*(param) {  **if** (**typeof** param !== **'string'**) **throw new TypeError**(**'Argument must be а string'**);  }   toString() {  **return this**.**\_stringArray**.join(**''**);  } } |

### Functionality

The above code defines a **class** that holds **characters** (strings with length 1) in an array. An **instance** of the class should support the following operations:

* Can be **instantiated** with a passed in **string** argument or **without** anything
* Functionappend(string) - **converts** the passed in **string** argument to an **array** and adds it to the **end** of the storage
* Function **prepend**(**string**) - **converts** the passed in **string** argument to an **array** and adds it to the **beginning** of the storage
* FunctioninsertAt(string, index) - **converts** the passed in **string** argument to an **array** and adds it at the **given** index (there is **no** need to check if the index is in range)
* Functionremove(startIndex, length) - **removes** elements from the storage, starting at the given index (**inclusive**), **length** number of characters (there is **no** need to check if the index is in range)
* FunctiontoString() - **returns** a string with **all** elements joined by an **empty** string
* All passed in **arguments** should be **strings.** If any of them are **not**, **throws** a type **error** with the following message: "**Argument must be a string**"

### Example

This is an example how this code is **intended to be used**:

|  |  |  |
| --- | --- | --- |
| Sample code usage |  | Corresponding output |
| **let** str = **new** StringBuilder(**'hello'**); str.append(**', there'**); str.prepend(**'User, '**); str.insertAt(**'woop'**,5 ); **console**.log(str.toString()); str.remove(6, 3); **console**.log(str.toString()); | User,woop hello, there  User,w hello, there |

### Your Task

Using **Mocha** and **Chai** write **JS unit tests** to test the entire functionality of the StringBuilder class. Make sure it is **correctly defined as a class** and instances of it have all the required functionality. You may use the following code as a template:

|  |
| --- |
| describe(**"*TODO* …"**, **function**() {  ***it***(**"*TODO …*"**, **function**() {  *//* ***TODO:*** …  });  *//* ***TODO:*** …  }); |

## Payment Package

You are given the following **JavaScript class**:

|  |
| --- |
| PaymentPackage.js |
| **class** PaymentPackage {  constructor(name, value) {  **this**.name = name;  **this**.value = value;  **this**.VAT = 20; *// Default value* **this**.active = **true**; *// Default value* }   **get** name() {  **return this**.**\_name**;  }   **set** name(newValue) {  **if** (**typeof** newValue !== **'string'**) {  **throw new** Error(**'Name must be a non-empty string'**);  }  **if** (newValue.length === 0) {  **throw new** Error(**'Name must be a non-empty string'**);  }  **this**.**\_name** = newValue;  }   **get** value() {  **return this**.**\_value**;  }   **set** value(newValue) {  **if** (**typeof** newValue !== **'number'**) {  **throw new** Error(**'Value must be a non-negative number'**);  }  **if** (newValue < 0) {  **throw new** Error(**'Value must be a non-negative number'**);  }  **this**.**\_value** = newValue;  }   **get** VAT() {  **return this**.**\_VAT**;  }   **set** VAT(newValue) {  **if** (**typeof** newValue !== **'number'**) {  **throw new** Error(**'VAT must be a non-negative number'**);  }  **if** (newValue < 0) {  **throw new** Error(**'VAT must be a non-negative number'**);  }  **this**.**\_VAT** = newValue;  }   **get** active() {  **return this**.**\_active**;  }   **set** active(newValue) {  **if** (**typeof** newValue !== **'boolean'**) {  **throw new** Error(**'Active status must be a boolean'**);  }  **this**.**\_active** = newValue;  }   toString() {  **const** output = [  **`Package:** ${**this**.name}**`** + (**this**.active === **false** ? **' (inactive)'** : **''**),  **`- Value (excl. VAT):** ${**this**.value}**`**,  **`- Value (VAT** ${**this**.VAT}**%):** ${**this**.value \* (1 + **this**.VAT / 100)}**`** ];  **return** output.join(**'\n'**);  } } |

### Functionality

The above code defines a **class** that contains information about a **payment package**. An **instance** of the class should support the following operations:

* Can be **instantiated** with two parameters - a string name and number value
* Accessor name - used to get and set the value of name
* Accessor value - used to get and set the value of value
* Accessor VAT - used to get and set the value of VAT
* Accessor active - used to get and set the value of active
* Function toString() - return a string, containing an overview of the instance; if the package is **not active**, append the label "**(inactive)**" to the printed **name**

When creating an instance, or changing any of the property values, the parameters are validated. They must follow these rules:

* name - non-empty string
* value - non-negative number
* VAT - non-negative number
* active - Boolean

If any of the requirements aren’t met, the operation must throw an error.

***Scroll down for examples and details about submitting to Judge.***

### Example

This is an example how this code is **intended to be used**:

|  |
| --- |
| Sample code usage |
| *// Should throw an error* **try** {  **const *hrPack*** = **new** PaymentPackage(**'HR Services'**); } **catch**(err) {  **console**.log(**'Error: '** + err.**message**); } **const *packages*** = [  **new** PaymentPackage(**'HR Services'**, 1500),  **new** PaymentPackage(**'Consultation'**, 800),  **new** PaymentPackage(**'Partnership Fee'**, 7000), ]; **console**.log(***packages***.join(**'\n'**));  **const *wrongPack*** = **new** PaymentPackage(**'Transfer Fee'**, 100); *// Should throw an error* **try** {  ***wrongPack***.active = **null**; } **catch**(err) {  **console**.log(**'Error: '** + err.**message**); } |
| Corresponding output |
| Error: Value must be a non-negative number  Package: HR Services  - Value (excl. VAT): 1500  - Value (VAT 20%): 1800  Package: Consultation  - Value (excl. VAT): 800  - Value (VAT 20%): 960  Package: Partnership Fee  - Value (excl. VAT): 7000  - Value (VAT 20%): 8400  Error: Active status must be a boolean |

### Your Task

Using **Mocha** and **Chai** write **unit tests** to test the entire functionality of the PaymentPackage class. Make sure instances of it have all the required functionality and validation. You may use the following code as a template:

|  |
| --- |
| describe(**"*TODO* …"**, **function**() {  ***it***(**"*TODO …*"**, **function**() {  *//* ***TODO:*** …  });  *//* ***TODO:*** …  }); |

## \*Warehouse

You are given the following JavaScriptclass:

|  |
| --- |
| **Warehouse.js** |
| **class** Warehouse {   **get** capacity() {  **return this**.**\_capacity**;  }   **set** capacity(givenSpace) {   **if** (**typeof** givenSpace === **'number'** && givenSpace > 0) {  **return this**.**\_capacity** = givenSpace;  } **else** {  **throw `Invalid given warehouse space`**;  }  }   constructor(capacity) {  **this**.capacity = capacity;  **this**.**availableProducts** = {**'Food'**: {}, **'Drink'**: {}};  }   addProduct(type, product, quantity) {   **let** addedQuantity = ((**this**.capacity - **this**.occupiedCapacity()) - quantity);  **let** output;   **if** (addedQuantity >= 0) {   **if** (**this**.**availableProducts**[type].hasOwnProperty(product) === **false**) {  **this**.**availableProducts**[type][product] = 0;  }   **this**.**availableProducts**[type][product] += quantity;  output = **this**.**availableProducts**[type];   } **else** {  **throw `There is not enough space or the warehouse is already full`**;  }   **return** output;  }   orderProducts(type) {   **let** output;  **let** sortedKeys = ***Object***.keys(**this**.**availableProducts**[type])  .sort((a, b) => **this**.**availableProducts**[type][b] - **this**.**availableProducts**[type][a]);   **let** newObj = {};   **for** (**let** product **of** sortedKeys) {   **if** (newObj.hasOwnProperty(product) === **false**) {  newObj[product] = 0;  }   newObj[product] += **this**.**availableProducts**[type][product];  }   **this**.**availableProducts**[type] = newObj;  output = **this**.**availableProducts**[type];   **return** output;  }   occupiedCapacity() {   **let** output = 0;  **let** productsCount = ***Object***.keys(**this**.**availableProducts**[**'Food'**]).length +  ***Object***.keys(**this**.**availableProducts**[**'Drink'**]).length;   **if** (productsCount > 0) {   **let** quantityInStock = 0;   **for** (**let** type **of *Object***.keys(**this**.**availableProducts**)) {   **for** (**let** product **of *Object***.keys(**this**.**availableProducts**[type])) {   quantityInStock += **this**.**availableProducts**[type][product];  }  }   output = quantityInStock;  }   **return** output;  }   revision() {   **let** output = **""**;   **if** (**this**.occupiedCapacity() > 0) {   **for** (**let** type **of *Object***.keys(**this**.**availableProducts**)) {  output += **`Product type - [**${type}**]\n`**;  **for** (**let** product **of *Object***.keys(**this**.**availableProducts**[type])) {  output += **`-** ${product} ${**this**.**availableProducts**[type][product]}**\n`**;  }  }  } **else** {  output = **'The warehouse is empty'**;  }   **return** output.trim();  }   scrapeAProduct(product, quantity) {   **let** type = ***Object***.keys(**this**.**availableProducts**).find(t => ***Object***.keys(**this**.**availableProducts**[t]).includes(product));  **let** output;   **if** (type !== **undefined**) {   **if** (quantity <= **this**.**availableProducts**[type][product]) {  **this**.**availableProducts**[type][product] -= quantity;  } **else** {  **this**.**availableProducts**[type][product] = 0;  }   output = **this**.**availableProducts**[type];   } **else** {  **throw `**${product} **do not exists`**;  }   **return** output;  } } |

### Functionality

An **instance** of the **Vacation** class should support the following operations:

If the **constructor** gets a **negative number** or **0 should throw a string:  
"Invalid given warehouse space"**

**AddProduct(type, Product, Quantity)**

**Adds** the given product if there is space in the warehouse and **return the object with the given type with already added products.** In these cases when the product is added more than 1 time, the quantity should be **sum**. When there is **no place** for the current product, you should **throw** a string that says:

**"There is not enough space or the warehouse is already full"**

**OrderProducts(type)**

**Sorts all products** of a given **type in descending order** by the **quantity.**

**OccupiedCapacity()**

**Returns** a number, which represents the **already occupied** place in the warehouse.

**Revision()**

**Returns** a string in which we print **all products** of **each type**, into the following **format**:

**'Product type - [Food]'**

**- {product} {quantity}**

**- {product} quantity  
...  
...**

**'Product type - [Drink]'**

**- {product} {quantity}**

**- {product} quantity  
...  
...  
If there is not at least 1 product in the warehouse** we **return** the string:   
**'The Warehouse is empty'**

**ScrapeAProduct(product, Quantity)**

If the given **product exists** we reduce his quantity, otherwise we **reset it.** If we cannot find the given product we **return** **the** **string**:

**'{product} do not exists'**

### TODO

Using **Mocha** and **Chai** write **JS unit tests** to test the entire functionality of the **Vacation** class. You may use the following code as a template:

**Submit** only your **describe()** statements..

|  |
| --- |
| describe(**"*TODO* …"**, **function**() {  ***it***(**"*TODO …*"**, **function**() {  *//* ***TODO:*** …  });  *//* ***TODO:*** …  });  describe(**"*TODO* …"**, **function**() {  ***it***(**"*TODO …*"**, **function**() {  *//* ***TODO:*** …  });  *//* ***TODO:*** …  });  ...  ... |

Don't forget to require the chai library!

## \*C# Console

Write **Mocha Unit tests** to verify the functionality of a JavaScript implementation of the C# Console **class**. If you've written some code in C#, you would know that you can format text using placeholders, an example would look like this:

Console.WriteLine("The sum of {0} and {1} is {2}", 3, 4, 7);

Here the first placeholder {0} is exchanged for the first parameter passed after the text template - 3. The second placeholder {1} for the second parameter - 4 and so on.

You will be provied with a class Console which has similar functionality to the C# one. The Console should have a static method writeLine which supports the following:

* writeLine(string) - if only a single argument is passed and it is a string, the function should simply return it.
* writeLine(object) - if only a single parameter is passed and it is an object - return the **JSON** representation of the object.
* writeLine(templateString, parameters) - It should support the following:
  + If multiple arguments are passed, but the first is not a string - throw a TypeError.
  + If the number of parameters does not correspond to the number of placeholders in the template string - throw a RangeError.
  + If the placeholders have indexes not withing the parameters range(for instance we have a placeholder {13} and only 5 params) throw a RangeError.
  + If multiple arguments are passed and the first is a string, find all placeholders from the string and **exchange** them with the supplied parameters.

Any cases not mentioned above, do not need to be checked.

### Constraints

* All arguments in the writeLine(templateString, parameters) will be **strings**.
* There will never be two placeholders with the same number.

### JS Code

To ease you in the process, you are provided with an implementation which meets all of the specification requirements for the Console object:

|  |
| --- |
| specialConsole.js |
| **class *Console*** {   **static get** *placeholder*() {  **return** /{\d+}/g;  }   **static** *writeLine*() {  **let** message = arguments[0];  **if** (arguments.**length** === 1) {  **if** (**typeof** (message) === **'object'**) {  message = **JSON**.stringify(message);  **return** message;  }  **else if** (**typeof**(message) === **'string'**) {  **return** message;  }  }  **else** {  **if** (**typeof** (message) !== **'string'**) {  **throw new TypeError**(**"No string format given!"**);  }  **else** {  **let** tokens = message.match(**this**.*placeholder*).sort(**function** (a, b) {  a = ***Number***(a.substring(1, a.**length** - 1));  b = ***Number***(b.substring(1, b.**length** - 1));  **return** a - b;  });  **if** (tokens.**length** !== (arguments.**length** - 1)) {  **throw new RangeError**(**"Incorrect amount of parameters given!"**);  }  **else** {  **for** (**let** i = 0; i < tokens.**length**; i++) {  **let** number = ***Number***(tokens[i].substring(1, tokens[i].**length** - 1));  **if** (number !== i) {  **throw new RangeError**(**"Incorrect placeholders given!"**);  }  **else** {  message = message.replace(tokens[i], arguments[i + 1])  }  }  **return** message;  }  }  }  } }; |

Your tests will be supplied a class named **"Console"** which contains the above-mentioned logic, all test cases you write should reference this variable. Submit in the Judge your code containing Mocha tests testing the above functionality.