Exercises: Unit Testing and Error Handling

Problems for exercises and homework for the "JavaScript Advanced" course @ SoftUni. Submit your solutions in the SoftUni judge system at https://judge.softuni.bg/Contests/335/.

Error Handling

1. Request Validator

Write a JS function that validates an HTTP request object. The object has the properties `method`, `uri`, `version` and `message`. Your function must receive the object as a parameter and 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** of 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 here. 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. As output, **return** the same object or throw an **Error** as described above.

Examples

Sample Input	Output

```
validateRequest({
  method: 'GET',
  uri: 'svn.public.catalog',
  version: 'HTTP/1.1',
  message: ''
});
```
	```
	{
	  method: 'GET',
	  uri: 'svn.public.catalog',
	  version: 'HTTP/1.1',
	  message: ''
	}
	```

Sample Input	**Output**

```
validateRequest({
  method: 'OPTIONS',
  uri: 'git.master',
  version: 'HTTP/1.1',
  message: '-recursive'
});
```
	Invalid request header: Invalid Method

Sample Input	Output

`validateRequest({`
`  method: 'POST',`
`  uri: 'home.bash',`
`  message: 'rm -rf /*'`
`});`
	`Invalid request header: Invalid Version`

Unit Testing

The Unit Tests with Sinon and Mocha strategy gives you access to the following libraries to help you test your code - Mocha, Sinon, Chai, Sinon-Chai and jQuery.

You are required to **only submit the unit tests** for the object/function you are testing. The strategy provides access to Chai's **expect**, **assert** and **should** methods and jQuery.

Example Submission

```
describe('isOddOrEven',function(){
    it('with a number parameter, should return undefined',function () {
        expect(isOddOrEven(13)).to.equal(undefined,
            "Function did not return the correct result!");
    });

    it('with a object parameter, should return undefined',function () {
        isOddOrEven({name: "pesho"}).should.equal(undefined,
            "Function did not return the correct result!");
    });

    it('with an even length string, should return correct result',function () {
        assert.equal(isOddOrEven("roar"),"even",
            "Function did not return the correct result!");
    });

    it('with an odd length string, should return correct result',function () {
        expect(isOddOrEven("pesho")).to.equal("odd",
            "Function did not return the correct result!");
    });

    it('with multiple consecutive checks, should return correct values',function () {
        expect(isOddOrEven("cat")).to.equal("odd",
            "Function did not return the correct result!");
        expect(isOddOrEven("alabala")).to.equal("odd",
            "Function did not return the correct result!");
        expect(isOddOrEven("is it even")).to.equal("even",
            "Function did not return the correct result!");
    });
});
```

2. Even or Odd

You need to write **unit tests** for a function **`isOddOrEven`** that checks whether a passed in **string** has **even** or **odd length**. The function has the following functionality:

- **`isOddOrEven(string)`** - A function that accepts a string and determines if the string has **even** or **odd length**.
  - If the passed parameter is **not a string** return **undefined**.
  - If the parameter is a **string** - return either **"even"** or **"odd"** based on the string's length.

JS Code

To ease you in the process, you are provided with an implementation which meets all of the specification requirements for the **`isOddOrEven`** function:

isOdd.js

```
function isOddOrEven(string) {
    if (typeof(string) !== 'string') {
        return undefined;
    }
    if (string.length % 2 === 0) {
        return "even";
    }

    return "odd";
}
```

Submit in the judge your code containing Mocha tests testing the above functionality.

Your tests will be supplied a function named "**`isOddOrEven`**" which contains the above mentioned logic, all test cases you write should reference this function. You can check the example at the beginning of this document to grasp the syntax.

3. Char Lookup

You are tasked with writing **unit tests** for a simplistic function that **retrieves a character** (a string containing only 1 symbol in JS) at a given **index** from a passed in **string**.

You are supplied a function named **lookupChar**, which should have the following functionality:

- **`lookupChar(string, index)`** - A function that accepts a string - the **string** in which we'll search and a number - the **index** of the char we want to lookup:
  - If the first parameter is not a string or the second parameter is not an integer - 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 the text **"Incorrect index"**.
  - If both parameters have correct types and values - return the **character at the specified index** in the string.

JS Code

To ease you in the process, you are provided with an implementation which meets all of the specification requirements for the **`lookupChar`** function:

lookupChar.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);
}
```

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

4. Math Enforcer

Your task is to test a variable named **mathEnforcer**, which represents an object that 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 should 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

To ease you in the process, you are provided with an implementation which meets all of the specification requirements for 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. Submit in the judge your code containing Mocha tests testing the above functionality.

5. Shared Object

You are tasked to test a **sharedObject** responsible for keeping a valid state between a JS object and two HTML textboxes. The two textboxes will always have **ids name** for the **name textbox** and **income** for the **income textbox**, and will always start with **empty strings** as values. The sharedObject should have the following functionality:

- **name** - a property holding a string, starts with value **null** by default.
- **income** - a property holding a number, starts with value **null** by default.
- **changeName(name)** - In case the passed in parameter is an **empty string** - **no changes should be made**, alternatively the sharedObject's **name** property and the **name textbox's value** should be set to the passed in parameter.
- **changeIncome(income)** - In case the passed in parameter is **not a positive integer** - **no changes should be made**, alternatively the sharedObject's **income** property and the **income textbox's value** should be set to the passed in parameter.
- **updateName()** - In case the name textbox's value is an **empty string** - **no changes should be made**, alternatively the **name** property of the sharedObject should be set to the **value** of the **name textbox**.
- **updateIncome()** - In case the income textbox's value **cannot be parsed** to a **positive integer** - **no changes should be made**, alternatively the **income** property of the sharedObject should be set to the **value** of the **income textbox**.

HTML and JS Code

To ease you in the process, you are provided with an HTML template for testing:

shared-object.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Shared Object</title>
</head>
<body>
    <div id="wrapper">
        <input type="text" id="name">
        <input type="text" id="income">
    </div>
</body>
</html>
```

And an implementation which meets all of the specification requirements for the **sharedObject**:

shared-object.js

```javascript
let sharedObject = {
    name: null,
    income: null,
    changeName: function (name) {
        if (name.length === 0) {
            return;
        }
        this.name = name;
        let newName = $('#name');
        newName.val(this.name);
    },
    changeIncome: function (income) {
        if (!Number.isInteger(income) || income <= 0) {
            return;
        }
        this.income = income;
        let newIncome = $('#income');
        newIncome.val(this.income);
    },
    updateName: function () {
        let newName = $('#name').val();
        if (newName.length === 0) {
            return;
        }
        this.name = newName;
    },
    updateIncome: function () {
        let newIncome = $('#income').val();
        if (isNaN(newIncome) || !Number.isInteger(Number(newIncome)) ||
Number(newIncome) <= 0) {
            return;
        }
        this.income = Number(newIncome);
    }
};
```

Your tests will be supplied a variable named **"sharedObject"** 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.

Hints

- Test that the initial state of the **sharedObject** meets the specification (i.e. **name** and **income** start as **null**).
- Test the functions with preexisting values to ensure that the old values are kept.

6. ArmageDOM

Write Mocha tests to check the functionality of the following JS code:

armagedom.js

```
function nuke(selector1, selector2) {
    if (selector1 === selector2) return;
    $(selector1).filter(selector2).remove();
}
```

Your tests will be supplied a function named **'nuke'**. It needs to meet the following:

- Operates inside an HTML document
- Takes two **string** arguments, each argument is a jQuery selector

- Upon execution, deletes **all** nodes that match **both** selectors
- Does nothing if **either** selector is invalid or omitted
- Does nothing if the two selectors are the same

When testing, note that if the selector is an ID, jQuery will always return only the first element that is a match, as opposed to a collection of all elements with that ID. You may use the following HTML for testing:

armagedom.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>ArmageDOM</title>
</head>
<body>
<div id="target">
    <div class="nested target">
        <p>This is some text</p>
    </div>
    <div class="target">
        <p>Empty div</p>
    </div>
    <div class="inside">
        <span class="nested">Some more text</span>
        <span class="target">Some more text</span>
    </div>
</div>
</body>
</html>
```

Hints

You need to manually include the HTML you want to test with in a **`beforeEach()`** Mocha statement.