Exercises: Unit Testing and Error Handling

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

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	%appdata%
git.master	apt-get
version1.0	
for..of	home$
.babelrc	define apps
c	"documents"

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

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

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

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.

Hints

We can clearly see there are 3 outcomes of the function:

- Returning **undefined**.
- Returning **"even"**.
- Returning **"odd"**.

We can write one or two tests passing things other than string to the function and expecting it to return **undefined**.

```
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!");
    });
```

After we've checked the validation it's time to check whether the function works correctly with proper arguments. We can write a test for each of the cases:

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

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

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

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

Finally we can make an extra test passing multiple different strings in a row to ensure the function is consistent:

```
    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!");
    });
});
```

Our code is now ready to be submitted to the Judge System.

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.

Hints

Writing tests is all about thinking, a good first step in testing a method is usually to determine all exit conditions (all ways in which the method can end its execution - **return** statements**, throw** statements or if none of the previous are present **simply running to the end**). Reading through the specification or taking a look at the implementation we can easily determine **3 main exit conditions** - returning **undefined,** returning an **empty string** or **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. We'll start with returning **undefined**. Reading the specification we can see that if any of the parameters are of the incorrect type we need to return **undefined**. Having two input parameters we easily have our first 2 tests.

```
describe('lookupChar', function () {
    it('with a non-string first parameter, should return undefined', function () {
        expect(lookupChar(13, 0)).to.equal(undefined,
            "The function did not return the correct result!");
    });

    it('with a non-number second parameter, should return undefined', function () {
        expect(lookupChar("pesho", "gosho")).to.equal(undefined,
            "The function did not return the correct result!");
    });
```

It may look like these two tests are enough to cover the first exit condition, however taking 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** (so it could be used for getting the char at the index), since floating point numbers won't work as indexes, we need to make sure that the passed in index is not a floating point number.

```
it('with a floating point number second parameter, should return undefined', function () {
    expect(lookupChar("pesho", 3.12)).to.equal(undefined,
        "The function did not return the correct message!");
});
```

Having added the extra test we have now covered the first exit condition, moving on we go to the next one - returning **empty string**. Checking the specification again we can see two distinct cases that we should check for - getting passed an index that is a negative number or getting passed an index which is outside of the bounds of the string.

```
it('with an incorrect index value, should return incorrect index', function () {
    expect(lookupChar("gosho", 13)).to.equal("Incorrect index",
        "The function did not return the correct value!");
});

it('with a negative index value, should return incorrect index', function () {
    expect(lookupChar("stamat", -1)).to.equal("Incorrect index",
        "The function did not return the correct value!");
});
```

Normally this would be enough to cover this condition, however the situation where the **index** passed is **equal to the string length** (known as an edge case) can be easily missed when writing the code, so it's a good idea to test for it too.

```
it('with an index value equal to string length, should return incorrect index', function () {
    expect(lookupChar("pesho", 5)).to.equal("Incorrect index",
        "The function did not return the correct value!");
});
```

Having cleared all the validation checks it's time for the last exit condition - **returning a correct result**, when checking results there are usually a number of things to check depending on the returned value and specification of the function. In our situation we are returning a **char** so a simple check for the returned value will be enough, however a single test for the correct return value is akin to guessing.

In a situation where there are limited correct results (for example a method which returns **`true`** or **`false`**) getting the correct value in one test does not guarantee the correctness of the method, even if the method did not function correctly there would still be a 50-50 chance of us receiving the correct result. To counteract this problem we usually create more than one test to check for the correct result. More tests is always better, but in most situations a few tests with different input parameters and different expected return values would be enough to cover the function.

```
it('with correct parameters, should return correct value', function () {
    expect(lookupChar("pesho", 0)).to.equal("p",
        "The function did not return the correct result!");
});

it('with correct parameters, should return correct value', function () {
    expect(lookupChar("stamat", 3)).to.equal("m",
        "The function did not return the correct result!");
});
```

With these last two tests we have covered the **`lookupChar`** function. Our code is now ready to be submitted to the Judge System.

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.

Screenshots

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

```
describe('mathEnforcer',function(){
    describe('addFive',function(){
        it('with a non-number parameter, should return correct result', function () {
            //TODO
        });
    });

    describe('subtractTen',function(){
        it('with a non-number parameter, should return correct result', function () {
            //TODO
        });
    });

    describe('sum',function(){
        it('with a non-number parameter, should return correct result', function () {
            //TODO
        });
```

Your tests will be supplied a variable named **"mathEnforcer"** 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 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).