Exercise: Unit Testing and Modules

Error Handling

1. 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',`
	`Invalid request header: Invalid Method`

`  message: '-recursive'` `}`	

`{`
`  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`	`&copy;`
`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.

What to submit?

Export the function in **requestValidator.js** and import it in your test file to test it. Submit a **zip** file containing the **requestValidator.js** and **tests folder** containing the **requestValidator.test.js**. **Do not** include the **node_modules** folder.

File Name: REQUEST-VALIDATOR.zip

2. 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`**.

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

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

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:

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

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

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

What to submit?

Export the function in **isOddOrEven.js** and import it in your test file to test it. Submit a **zip** file containing the **isOddOrEven.js** and **tests folder** containing the **isOddOrEven.test.js**. **Do not** include the **node_modules** folder.

File Name: EVEN-OR-ODD.zip

3. 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

A 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 an **`empty 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**, **`undefined`** should be returned.

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

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

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.

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

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.

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

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

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

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

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

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

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

What to submit?

Export the function in **charLookup.js** and import it in your test file to test it. Submit a **zip** file containing the **charLookup.js** and **tests folder** containing the **charLookup.test.js**. **Do not** include the **node_modules** folder.

File Name: CHAR-LOOKUP.zip

Unit Testing On Classes

4. String Builder

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

stringBuilder.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 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
- Function **`append(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
- Function **`insertAt(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)
- Function **`remove(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)
- Function **`toString()`** - **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: ...
});
```

What to submit?

Export the class in **stringBuilder.js** and import it in your test file to test it. Submit a **zip** file containing the **stringBuilder.js** and **tests folder** containing the **stringBuilder.test.js**. **Do not** include the **node_modules** folder.

File Name: STRING-BUILDER.zip

5. 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
- If any of the requirements aren't met, the operation must throw an error.
- **`active`** - Boolean

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: ...
});
```

What to submit?

Export the class in **paymentPackage.js** and import it in your test file to test it. Submit a **zip** file containing the **paymentPackage.js** and **tests folder** containing the **paymentPackage.test.js**. **Do not** include the **node_modules** folder.

File Name: PAYMENT-PACKAGE.zip