

Exercise: Unit Testing and Error Handling

Problems for exercises and homework for the ["JavaScript Advanced" course @ SoftUni](https://judge.softuni.org/Contests/2767/Unit-Testing-Exercise). Submit your solutions in the SoftUni judge system at <https://judge.softuni.org/Contests/2767/Unit-Testing-Exercise>

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** 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 the 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
<pre>{ method: 'GET', uri: 'svn.public.catalog', version: 'HTTP/1.1', message: '' }</pre>	<pre>{ method: 'GET', uri: 'svn.public.catalog', version: 'HTTP/1.1', message: '' }</pre>
<pre>{ method: 'OPTIONS', uri: 'git.master', version: 'HTTP/1.1', message: '-recursive' }</pre>	Invalid request header: Invalid Method

<pre>{ method: 'POST', uri: 'home.bash', message: 'rm -rf /*' }</pre>	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 © "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.

2. Even or Odd

You need to write **unit tests** for a function **isOddOrEven()** that checks whether the **length** of a passed **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 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.

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 **"Incorrect index"**
- 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.

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 past an index that is a non-number, it won't protect us from being passed a **floating-point number**. The specification says that the **index** needs to be an **integer**, since floating-point numbers won't work as indexes.

Moving on to the next **exit condition** - returning an **"Incorrect index"** if we get past an index that is a **negative number** or an index that 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.

4. 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 function 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 the 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 description** for each function:

```

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

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

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

```

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).

DOM Error Handling

The following problems must be solved using DOM manipulation techniques.

Environment Specifics

Please, be aware that every JS environment may **behave differently** when executing code. Certain things that work in the browser are not supported in **Node.js**, which is the environment used by **Judge**.

The following actions are **NOT** supported:

- `.forEach()` with **NodeList** (returned by `querySelector()` and `querySelectorAll()`)
- `.forEach()` with **HTMLCollection** (returned by `getElementsByClassName()` and `element.children`)
- Using the **spread-operator** (`...`) to convert a **NodeList** into an array
- `append()` in Judge (use only `appendChild()`)
- `prepend()`
- `replaceWith()`
- `replaceAll()`
- `closest()`
- `replaceChildren()`
- Always turn the collection into a **JS array** (`forEach`, `forOf`, et.)

If you want to perform these operations, you may use `Array.from()` to first convert the collection into an array.

5. Notification

Write a JS function that receives a string **message** and **displays** it inside a div with id "**notification**". The div starts **hidden** and when the function is called, **reveal** it. After the user clicks on it, **hide** the div. In the example document, a notification is shown when you click on the button ["**Get notified**"].

Example



6. Dynamic Validation

Write a JS function that dynamically validates an email input field when it is **changed**. If the input is invalid, apply to it the class "**error**". Do not validate on every keystroke, as it is annoying for the user, consider only **change** events.

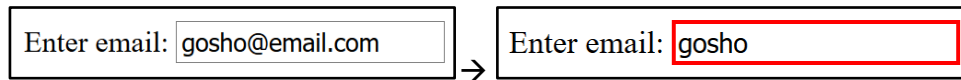
A valid email will be in format: <name>@<domain>.<extension>

Only lowercase Latin characters are allowed for any of the parts of the email. If the input is valid, clear the style.

Input/Output

There will be no input/output, your program should instead **modify** the DOM of the given HTML document.

Example



7. Form Validation

You are given the task to write **validation** for the fields of a simple form.

HTML and JavaScript Code

You are provided a **skeleton** containing the necessary files for your program.

The validations should be as follows:

- The **username** needs to be between **3** and **20** symbols **inclusively** and only **letters** and **numbers** are allowed.
- The **password** and **confirm-password** must be between **5** and **15** **inclusively** symbols and only **word characters** are allowed (**letters**, **numbers**, and **_**).
- The **inputs** of the **password** and **confirm-password** field **must match**.
- The **email** field must contain the **"@"** symbol and **at least one** **"."**(dot) after it.

If the **"Is company?"** checkbox is **checked**, the **CompanyInfo** fieldset should become **visible** and the **Company Number** field must also be **validated**, if it isn't checked the **Company** fieldset should have the style **"display: none;"** and the **value** of the **Company Number** field shouldn't matter.

- The **Company Number** field must be a number between **1000** and **9999**.
- Use **addEventListener()** function to **attach** an **event listener** for the **"change"** event to the **checkbox**.

Every field with an **incorrect** value when the **[Submit]** button is **pressed** should have the following style applied **border-color: red;**, alternatively, if it's correct it should have style **border: none;**. If there are **required** fields with an incorrect value when the **[Submit]** button is pressed, the **div** with **id="valid"** should become **hidden** (**"display: none;"**), **alternatively** if all fields are correct the **div** should become **visible**.

Constraints

- You are **NOT** allowed to change the HTML or CSS files provided.

Screenshots

User Information:

Username:

Email:

Password:

Confirm Password:

Is Company? ☐

Submit

User Information:

Username:

Email:

Password:

Confirm Password:

Is Company? ☒

Company Informaion:

Company Number

Submit

User Information:

Username:

Email:

Password:

Confirm Password:

Is Company? ☐

Submit

Valid

User Information:

Username:

Email:

Password:

Confirm Password:

Is Company? ☒

Company Informaion:

Company Number

Submit

Valid

Hints

- All buttons within an `<form>` automatically work as **submit** buttons, unless their type is **manually assigned** to something else, to avoid **reloading the page** upon **clicking** the **[Submit]** button you can use `event.preventDefault()`
- The validation for the separate fields can be done using **regex**.