# Exercise: Unit Testing with JS

Exercise problems for the ["Back-End Technologies Basics"](https://softuni.bg/trainings/4398/back-end-technologies-basics-january-2024) Course @ SoftUni.   
You can check your solutions in [Judge](https://judge.softuni.org/Contests/4679/Unit-Testing-with-JS-Exercise).

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 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.

## 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 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**, undefinedshould be returned.

If we take a closer look at the implementation, we can 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.

## 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:



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).

## Array Analyzer

Write unit tests for a function that **takes an array as an input and returns an object with the following properties based on the array's elements**:

* **min** – the smallest number in the array
* **max** – the largest number in the array
* **length** – the number of elements in the array

If the input is not an array or the array is empty, the function should return **undefined.**

You can test the following cases:

* The input is an array of numbers
* The input is an empty array
* The input is a non-array input
* The input is a single element array
* The input is an array with equal elements

### JS Code

You are provided with an implementation of the **arrayAnalyzer** object:

|  |
| --- |
| arrayAnalyzer.js |
| function *analyzeArray*(arr) {  if (!Array.isArray(arr) || arr.length === 0) {  return undefined;  }  let min = arr[0];  let max = arr[0];  for (let i = 0; i < arr.length; i++) {  if (typeof arr[i] !== 'number') {  return undefined;  }  if (arr[i] < min) {  min = arr[i];  }  if (arr[i] > max) {  max = arr[i];  }  }  return { min, max, length: arr.length };  } |