

Exercise: Unit Testing with JS

Exercise problems for the ["Back-End Technologies Basics"](#) Course @ SoftUni.

You can check your solutions in [Judge](#).

You are required to **submit only the unit tests** for the **object / function** you are testing.

1. 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 |
|---|
| <pre>function isOddOrEven(string) { if (typeof(string) !== 'string') { return undefined; } if (string.length % 2 === 0) { return "even"; } return "odd"; }</pre> |

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.

2. 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 |
|---|
| <pre>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); }</pre> |

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 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.

3. 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 |
|--|
| <pre>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; } };</pre> |

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).

4. 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 |
|--|
| <pre>function analyzeArray(arr) { if (!Array.isArray(arr) arr.length === 0) {</pre> |

```

    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 };
}

```

5. Art Gallery

Using **Mocha** and **Chai** write **JS Unit Tests** to test a variable named **artGallery**, which represents an object. You may use the following code as a template:

```

describe("Tests ...", function() {
    describe("TODO ...", function() {
        it("TODO ...", function() {
            // TODO: ...
        });
    });
});
// TODO: ...
});

```

The object should have the following functionality:

- **addArtwork (title, dimensions, artist)** - A function that accepts **three** parameters, all of them must be **strings**.
 - There is a **need for validation** for the input.
 - **title** and **artist** should be **non-empty** strings.
 - If any parameter is invalid, **throw an error**:

"Invalid Information!"
 - **dimensions** should be a **string** in the format "**width x height**", where both width and height are positive numbers (e.g., "30 x 40").
 - If dimensions does not match the required format, **throw an error**:

"Invalid Dimensions!"
 - If the **artist** is not one of ["Van Gogh", "Picasso", "Monet"], **throw an error**:

"This artist is not allowed in the gallery!"

- If the input is valid, **return** the message:
"Artwork added successfully: '{title}' by {artist} with dimensions {dimensions}."
- **calculateCosts (exhibitionCosts, insuranceCosts, sponsor)** - A function that accepts three parameters: **number**, **number**, and **boolean**.
 - There is a **need for validation** for the input.
 - **exhibitionCosts** and **insuranceCosts** should be positive **numbers**.
 - **sponsor** should be a **boolean**.
 - If any parameter is invalid, **throw an error**:
"Invalid Information!".
 - Calculate the **total cost** by summing **exhibitionCosts** and **insuranceCosts**.
 - If sponsor is **true**, apply a **10%** discount to the total cost. **Return**:
"Exhibition and insurance costs are {totalPrice}\$, reduced by 10% with the help of a donation from your sponsor."
 - If sponsor is **false**, **return**:
"Exhibition and insurance costs are {totalPrice}\$."
- **organizeExhibits (artworksCount, displaySpacesCount)** - A function that accepts two parameters: **number**, **number**.
 - There is a **need for validation** for the input.
 - Both **artworksCount** and **displaySpacesCount** should be positive **numbers**.
 - If any parameter is **not** a number or is **negative**, **throw an error**:
"Invalid Information!"
 - Calculate the **artworksPerSpace** by divide **artworksCount** by **displaySpacesCount** and rounded down.
 - If the number of artworks per display space is **less** than 5, **return**:
"There are only {artworksPerSpace} artworks in each display space, you can add more artworks."
 - Otherwise, **return**:
"You have {displaySpacesCount} display spaces with {artworksPerSpace} artworks in each space."

JS Code

To ease you in the process, you are provided with an implementation that meets all of the specification requirements for the **artGallery** object:

artGallery.js

```

const artGallery = {
  addArtwork(title, dimensions, artist) {
    if (typeof title !== "string" || typeof artist !== "string") {
      throw new Error("Invalid Information!");
    }

    if (!/^\d+ x \d+$/ .test(dimensions)) {
      throw new Error("Invalid Dimensions!");
    }

    const validArtists = ["Van Gogh", "Picasso", "Monet"];
    if (!validArtists.includes(artist)) {
      throw new Error("This artist is not allowed in the gallery!");
    }

    return `Artwork added successfully: '${title}' by ${artist} with dimensions
    ${dimensions}.`;
  },

  calculateCosts(exhibitionCosts, insuranceCosts, sponsor) {
    if (
      typeof exhibitionCosts !== "number" ||
      typeof insuranceCosts !== "number" ||
      typeof sponsor !== "boolean" ||
      exhibitionCosts < 0 ||
      insuranceCosts < 0
    ) {
      throw new Error("Invalid Information!");
    }

    let totalPrice = exhibitionCosts + insuranceCosts;
  }
}

```

```

    if (sponsor) {
        totalPrice *= 0.9;

        return `Exhibition and insurance costs are ${totalPrice}$, reduced by 10% with the
help of a donation from your sponsor.`;
    } else {
        return `Exhibition and insurance costs are ${totalPrice}$.`;
    }
},

organizeExhibits(artworksCount, displaySpacesCount) {
    if (
        typeof artworksCount !== "number" ||
        typeof displaySpacesCount !== "number" ||
        artworksCount <= 0 ||
        displaySpacesCount <= 0
    ) {
        throw new Error("Invalid Information!");
    }

    let artworksPerSpace = Math.floor(artworksCount / displaySpacesCount);

    if (artworksPerSpace < 5) {
        return `There are only ${artworksPerSpace} artworks in each display space, you can
add more artworks.`;
    } else {
        return `You have ${displaySpacesCount} display spaces with ${artworksPerSpace}
artworks in each space.`;
    }
}

```



```
},  
};
```

Submission

Submit your tests inside a **describe()** statement, as shown above.

6. Food Delivery

Using **Mocha** and **Chai** write **JS Unit Tests** to test a variable named **foodDelivery**, which represents an object. You may use the following code as a template:

```
describe("Tests ...", function() {  
  describe("TODO ...", function() {  
    it("TODO ...", function() {  
      // TODO: ...  
    });  
  });  
});  
// TODO: ...  
});
```

The object should have the following functionality:

- **getCategory(category)** - A function that accepts one parameter: **string**.
 - If the **category** is "Vegan" return the string:
"Dishes that contain no animal products."
 - If the **category** is "Vegetarian" return the string:
"Dishes that contain no meat or fish."
 - If the **category** is "Gluten-Free" return the string:
"Dishes that contain no gluten."
 - If the **category** is "All" return the string:
"All available dishes."
 - If the value of the string type is **different** from "Vegan", "Vegetarian", "Gluten-Free", "All", throw an error:
"Invalid Category!"
- **addItem(menuItem, maxPrice)** - A function that accepts an **array** of objects({**name**: Item name, **price**: item price}) and **number**.
 - You must **add** an element (**menuItem**) if the price is **less** or equal to **maxPrice** from the array to **availableItems** array.
 - Finally, **return** the array **length** in the following string:
 - **"There are {availableItems.length} available menu items matching your criteria!"**
 - There is a need for validation for the input, as an **array** and

number may not always be valid. In case of submitted invalid parameters, **throw an error**

"Invalid Information!"

- If passed **menuItem** or **maxPrice** parameters are not an **array** and **number**.
- If the **menuItem** array has fewer than 1 item, and if **maxPrice** is **less** than 5.

- **calculateOrderCost(shipping, addons, discount)** - A function that accepts three parameters: **array**, **array**, and **boolean**.
 - Calculate the total **price** you are going to **pay** depending on the chosen **shipping** options and **addons**.
 - The result must be formatted to the second digit after the decimal point.
 - The available options for **shipping** are:
 - **standard**, which costs \$3
 - **express**, which costs \$5
 - The available options for **addons** are:
 - **sauce**, which costs \$1
 - **beverage**, which costs \$3.5
 - If the **discount** is **true**, a 15% discount should be applied. Then **return** the following message:
"You spend \${totalPrice} for shipping and addons with a 15% discount!"
 - Else, **return** the following message:
"You spend \${totalPrice} for shipping and addons!"
 - You need to **validate** the input. If the **shipping**, **addons**, and **discount** are not an **array**, **array**, and **boolean**, **throw** an error:
"Invalid Information!"
 - **Note:** The **totalPrice** must be rounded to the **second** decimal

JS Code

To ease you in the process, you are provided with an implementation that meets all of the specification requirements for the **foodDelivery** object:

foodDelivery.js

```

const foodDelivery = {
  getCategory(category) {
    if (category === "Vegan") {
      return "Dishes that contain no animal products.";
    } else if (category === "Vegetarian") {
      return "Dishes that contain no meat or fish.";
    } else if (category === "Gluten-Free") {
      return "Dishes that contain no gluten.";
    } else if (category === "All") {
      return "All available dishes.";
    } else {
      throw new Error("Invalid Category!");
    }
  },
  addMenuItem(menuItem, maxPrice) {
    if (
      !Array.isArray(menuItem) ||
      typeof maxPrice !== "number" ||
      menuItem.length < 1 ||
      maxPrice < 5
    ) {
      throw new Error("Invalid Information!");
    }
    let availableItems = [];
    menuItem.forEach((item) => {
      if (item.price <= maxPrice) {
        availableItems.push(item);
      }
    });
    return `There are ${availableItems.length} available menu items matching your criteria!`;
  },
  calculateOrderCost(shipping, addons, discount) {
    if (
      !Array.isArray(shipping) ||
      !Array.isArray(addons) ||

```

```

        typeof discount !== "boolean"
    ) {
        throw new Error("Invalid Information!");
    }
    let totalPrice = 0;

    shipping.forEach((item) => {
        if (item === "standard") {
            totalPrice += 3;
        } else if (item === "express") {
            totalPrice += 5;
        }
    });
    addons.forEach((item) => {
        if (item === "sauce") {
            totalPrice += 1;
        } else if (item === "beverage") {
            totalPrice += 3.5;
        }
    });
    if (discount) {
        totalPrice = totalPrice * 0.85;
        return `You spend ${totalPrice.toFixed(
            2
        )} for shipping and addons with a 15% discount!`;
    } else {
        return `You spend ${totalPrice.toFixed(2)} for shipping and addons!`;
    }
},
};

```

Submission

Submit your tests inside a **describe()** statement, as shown above.

7. Workforce Management

Using **Mocha** and **Chai** write **JS Unit Tests** to test a variable named **workforceManagement**, which represents an object. You may use the following code as a template:

```
describe("Tests ...", function() {
  describe("TODO ...", function() {
    it("TODO ...", function() {
      // TODO: ...
    });
  });
  // TODO: ...
});
```

The object should have the following functionality:

recruitStaff (name, role, experience) - A function that accepts three parameters: **string**, **string**, and **number**.

- If the value of the string **role** is different from **"Developer"**, **throw** an error: **`We are not currently hiring for this role.`**
- To be hired, the **employee** must meet the **following requirements**:
 - If the **experience** is **greater** than or **equal** to **4**, **return** the string:
`{name} has been successfully recruited for the role of {role}.`
- Otherwise, if the above conditions are not met, **return** the following message:
`{name} is not suitable for this role.`
- There is **no** need for **validation** for the **input**, you will always be given a string, string, and number.
- **computeWages (hoursWorked)** - A function that accepts one parameter: **number**.
 - Workers in this company receive **equal** pay per **hour** and this is **BGN 18**.
 - You need to **calculate** the salary by **multiplying** the pay for **one hour** by the number of **hours**.
 - **Also**, if the employee has been working for **more than 160 hours**, he must receive an additional **BGN 1500 bonus**.
 - Finally, **return** the employee's salary.
 - You need to validate the input, if the **hoursWorked** are not a **number**, or are a **negative** number, **throw** an error: **"Invalid hours"**.
- **dismissEmployee (workforce, employeeIndex)** - A function that accepts an array and number.
 - The **workforce** array will store the names of its employees (**["Petar", "Ivan", "George"]**).
 - You must **remove** an **element** (employee) from the **array** that is located on the **employeeIndex** specified as a parameter.
 - Finally, **return** the changed array of workforce as a string, **joined** by a **comma** and a **space**.

- There is a need for validation for the input, an **array** and index may not always be valid. In case of submitted **invalid** parameters, **throw** an error **"Invalid input"**:
 - If passed **workforce** parameter is not an array.
 - If the **employeeIndex** is not a number and is outside the limits of the array.

JS Code

To ease you in the process, you are provided with an implementation that meets all of the specification requirements for the **workforceManagement** object:

workforceManagement.js

```
const workforceManagement = {

  recruitStaff(name, role, experience) {
    if (role === "Developer") {
      if (experience >= 4) {
        return `${name} has been successfully recruited for the role of ${role}.`;
      } else {
        return `${name} is not suitable for this role.`;
      }
    }
    throw new Error(`We are not currently hiring for this role.`);
  },

  computeWages(hoursWorked) {
    let hourlyRate = 18;
    let totalPayment = hourlyRate * hoursWorked;

    if (typeof hoursWorked !== "number" || hoursWorked < 0) {
      throw new Error("Invalid hours");
    } else if (hoursWorked > 160) {
      totalPayment += 1500;
    }
    return totalPayment;
  },

  dismissEmployee(workforce, employeeIndex) {
```

```
let updatedStaff = [];  
  
if (!Array.isArray(workforce) || !Number.isInteger(employeeIndex) ||  
employeeIndex < 0 || employeeIndex >= workforce.length) {  
    throw new Error("Invalid input");  
}  
for (let i = 0; i < workforce.length; i++) {  
    if (i !== employeeIndex) {  
        updatedStaff.push(workforce[i]);  
    }  
}  
return updatedStaff.join(", ");  
};
```

Submission

Submit your tests inside a **describe()** statement, as shown above.