

Error Handling

Unit Testing with Mocha

Errors and Exceptions, Unit Testing, Tests Cases, Assertions, Hooks



SoftUni Team

Technical Trainers

Table of Contents

1. Error Handling
2. Exception Handling
3. Unit Testing – Concepts
4. Using Mocha and Chai for JS Unit Testing
5. Testing with Mocha – Examples



Have a Question?



sli.do

#JScore



Error Handling in JS

Error Handling – Concepts

- In programming, the fundamental principle of error handling says:

A function (method) should either do what its name suggests or indicate a problem.
Any other behavior is incorrect.

- In JavaScript, a function failed to do what its name suggests should:
 - Return a special value (e.g. undefined / false / -1)
 - Throw an exception / error

Error Handling – Examples

- Normal execution → a function does what its name says

```
let str = "Hello, SoftUni";
console.log(str.indexOf("Uni")); // 11
console.log(str.indexOf ""); // 0
```

- Special case → return a special value to indicate "*not found*"

```
let str = "Hello, SoftUni";
console.log(str.indexOf("Sofia")); // -1
console.log(str.indexOf(undefined)); // -1
```

Error Handling – Exceptions (Errors)

- Exception → a function is unable to do its work (fatal error)

```
let arr = new Array(-1); // Uncaught RangeError
```

```
let bigArr = new Array(9999999999); // RangeError
```

```
let index = undefined.indexOf("hi"); // TypeError
```

```
console.log(asfd); // Uncaught ReferenceError
```

```
console.print('hi'); // Uncaught TypeError
```

Error Handling – Special Values

```
let sqrt = Math.sqrt(-1); // NaN (special value)
```

```
let sub = "hello".substring(2, 1000); // llo
// Soft error → substring still does its job: takes all
available chars
```

```
let sub = "hello".substring(-100, 100); // hello
// Soft error → substring still does its job: takes all available chars
```

```
let inv = new Date("Christmas"); // Invalid Date
let dt = inv.getDate(); // NaN
```

Unexpected (Not Obvious) Behavior

```
let d = new Date(2016, 1, 20); // Feb 20 2016 (Jan = 0, Feb = 1)
```

```
let dn = new Date(2016, 1, 30); // Mar 01 2016 (next month)
```

```
let dp = new Date(2016, -1, 30); // Dec 30 2015 (prev month)
```

```
let d1 = new Date(1, 1, 1); // Feb 01 1901
```

```
let dMinus1 = new Date(-1, -1, -1); // Nov 29 -2
```

```
let pi = new Date(3.14, 3.14, 3.14); // Month / year truncated  
// Apr 03 1903 03:21:36 (fractial day turned to hours / minutes)
```

Problem: Sub-Sum

- Write a JS function to sum a range of elements from array

```
subsum([10, 20, 30, 40, 50, 60], 3, 4)
```



```
90
```

- This is a sample solution:

```
function subsum(arr, startIndex, endIndex) {  
    let sum = 0;  
    for (let i = startIndex; i <= endIndex; i++)  
        sum += arr[i];  
    return sum;  
}
```

Problem: Sub-Sum – Error Handling

- Think about the special cases:

```
subsum([10, 20, 30, 40, 50, 60], 3, 300)
```

150

```
subsum([1.1, 2.2, 3.3, 4.4, 5.5], -3, 1)
```

3.3

```
subsum([10, 'twenty', 30, 40], 0, 2)
```

NaN

```
subsum([], 0, 0)
```

```
subsum([], 1, 2)
```

0

```
subsum('text', 0, 2)
```

```
subsum({}, 0, 2)
```

NaN

Solution: Sub-Sum (with Error Handling)

```
function subsum(arr, startIndex, endIndex) {  
    if (!Array.isArray(arr))  
        return NaN;  
    if (startIndex < 0)  
        startIndex = 0;  
    if (endIndex > arr.length - 1)  
        endIndex = arr.length - 1;  
  
    let sum = 0;  
    for (let i = startIndex; i <= endIndex; i++)  
        sum += Number(arr[i]);  
    return sum;  
}
```

explicit error handling /
special case handling

Check your solution here: <https://judge.softuni.bg/Contests/307>



ERROR

Exception Handling in JS

Throwing Errors / Catching Errors

Exceptions vs. Special Values

- Return a special value for expected behavior
 - Return a value if no problem / a recoverable problem occurs
 - Valid input arguments → return valid value or special value
 - E.g. searching something → return the item found or **undefined**
- Thrown an exception for unexpected behavior
 - When unrecoverable problem occurs
 - Invalid input arguments / invalid object state / missing resource
 - E.g. create a card "-2 hearts" or create array of -5 elements

Throwing Errors (Exceptions)

```
throw new Error("Invalid state");
```

General error

```
> throw new Error("Invalid state");
```

```
✖ ➤ Uncaught Error: Invalid state(...)
```

```
throw new RangeError("Invalid index");
```

Value out of range

```
throw new TypeError("String expected");
```

Expected a different
type

```
throw new ReferenceError("Missing age");
```

Accessing non-existing
object



Problem: Playing Cards

- A function creates a **Card** to hold card face + suit
 - Valid card faces are: **2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, A**
 - Valid card suits are: **S (♠), H (♥), D (♦), C (♣)**
 - Throw an error in case of invalid face or suit



```
console.log('' + makeCard('A', 'S')); // A♠  
console.log('' + makeCard('10', 'H')); // 10♥  
console.log('' + makeCard('2', 'D')); // 2♦
```

```
console.log('' + makeCard('1', 'C')); // Error  
console.log('' + makeCard('4', 'W')); // Error
```

Solution: Playing Cards – Factory Function

```
function makeCard(face, suit) {  
    const validFaces = ['2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q',  
    'K', 'A'];  
    const validSuits = ['S', 'H', 'D', 'C'];  
    if (!validFaces.includes(face))  
        throw new Error("Invalid card face: " + face);  
    if (!validSuits.includes(suit))  
        throw new Error("Invalid card suit: " + suit);  
    let card = {  
        face: face,  
        suit: suit,  
        toString: () => { ... }  
    };  
    return card;  
}
```

✖ ▶ Uncaught Error: Invalid card face: 1(...)
set face @ VM1177:13
Card @ VM1177:3
(anonymous function) @ VM1177:41

Solution: Playing Cards – `toString()`

```
toString() {  
    let suitToChar = {  
        'S': "\u2660", // ♠  
        'H': "\u2665", // ♥  
        'D': "\u2666", // ♦  
        'C': "\u2663", // ♣  
    };  
    return card.face + suitToChar[card.suit];  
}
```

```
> console.log('' + new  
Card('A', 'S'));
```

A♠

```
> console.log('' + new  
Card('10', 'D'));
```

10♦

The try-catch block

- Use **try-catch** block to catch errors (exceptions)
 - If an error occurs, the control is passed to the **catch** section

```
try {  
    // code that can throw an exception  
    // some other code → not executed in case of error!  
}  
catch (ex) {  
    // this code is executed in case of exception  
    // ex holds the info about the exception  
}
```



Exceptions and Their Properties

```
try {
    throw new RangeError("Invalid range.");
    console.log("This will not be executed.");
} catch (ex) {
    console.log("-----");
    console.log("Exception object: " + ex);
    console.log("Type: " + ex.name);
    console.log("Message: " + ex.message);
    console.log("Stack: " + ex.stack);
    console.log("-----");
}
```

Problem: Print Deck of Cards

- Write a method to take a deck of cards (each given as string) and print them as sequences of cards (space separated)
 - Print "Invalid card: [card]" when invalid card is passed as input

```
printDeckOfCards(['AS', '10D', 'KH', '2C']);  
// A♠ 10♦ K♥ 2♣
```

```
printDeckOfCards(['5S', '3D', 'QD', '1C']);  
// Invalid card: 1C
```

```
printDeckOfCards(['3D', 'JC', '2S', '10H', '5X']);  
// Invalid card: 5X
```

Solution: Print Deck of Cards

```
function printDeckOfCards(cards) {  
    let deck = [];  
    for (let cardStr of cards) {  
        let face = cardStr.substring(0, cardStr.length-1);  
        let suit = cardStr.substr(cardStr.length-1, 1);  
        try {  
            deck.push(makeCard(face, suit));  
        }  
        catch (err) {  
            console.log("Invalid card: " + cardStr);  
            return;  
        }  
    }  
    console.log(deck.join(' '));  
}
```

Insert the factory
function here

This can throw an error
(when the card is invalid)



What is Unit Testing?

Unit Testing – Definition

A **unit test** is a piece of code, written by a developer that checks whether a piece of functionality (e.g. function or method), is working as expected.

Source code

```
function sortNums(arr) {  
    arr.sort(  
        (a,b) => a-b  
    );  
}
```

Test code (unit tests)

```
let nums = [2, 15, -2, 4];  
sortNums(nums);  
if (JSON.stringify(nums)  
    != "[-2,2,4,15]")  
    console.error("test failed!");
```

- The AAA Pattern: Arrange, Act, Assert

```
// Arrange all necessary preconditions and inputs
let nums = [2, 15, -2, 4];
// Act on the object or method under test
sortNums(nums);
// Assert that the obtained results are what we expect
if (JSON.stringify(nums) != "[-2,2,4,15]")
    console.error("test failed!");
```



Unit Test – Sum Numbers – Example

```
function sum(arr) {  
    let sum = 0;  
    for (num of arr)  
        sum += Number(num);  
    return sum;  
}  
  
function testSum() {  
    if (sum([1, 2]) != 3) throw new Error("1+2 != 3");  
    if (sum([-2]) != -2) throw Error("-2 != -2");  
    if (sum([]) != 0) throw new Error("0 != 0");  
}
```



Unit Testing Framework

- Unit testing frameworks make writing unit tests simple
 - Provides test cases, assertions, hooks, test runners, etc.
 - [**JUnit**](#), [**NUnit**](#), [**Visual Studio Team Test \(VSTT\)**](#), [**PHPUnit**](#)
 - JS unit testing: [**Mocha**](#), [**QUnit**](#), [**Unit.js**](#), [**Jasmine**](#), ...
- Assertion frameworks (perform checks)
 - [**Chai**](#), [**Assert.js**](#), [**Should.js**](#)
- Mocking frameworks (mocks and stubs)
 - [**Sinon**](#), [**JMock**](#), [**Mockito**](#), [**Moq**](#)





Mocha and Chai

Unit Testing in JavaScript

Mocha and Chai

- Mocha is JS testing framework
 - Runs in Node.js and in Web browsers
 - Usually used together with Chai (assertion library)

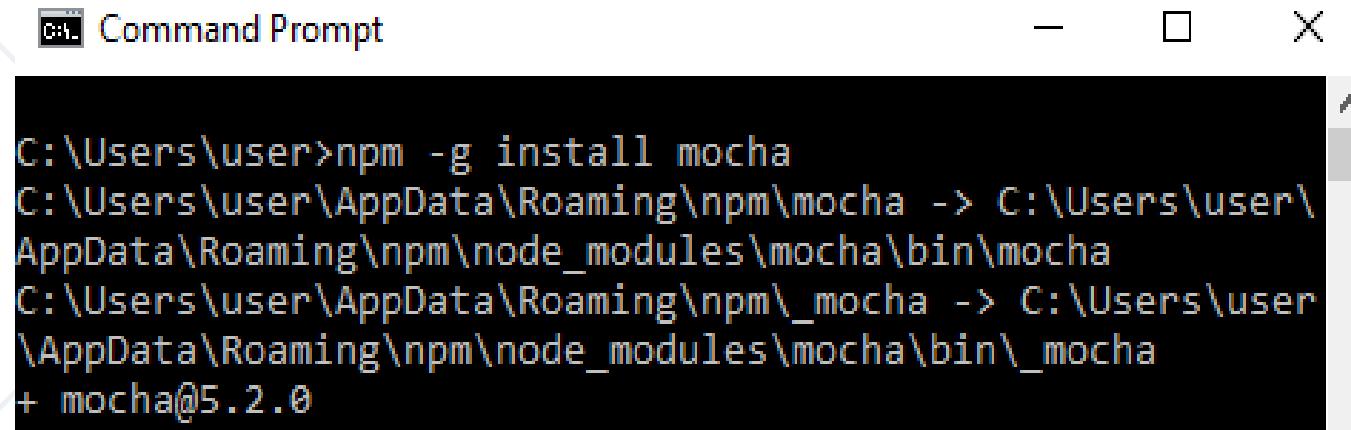


```
let expect = require("chai").expect;  
  
describe("sum(arr)", function() {  
  it("should return 3 for [1, 2]", function() {  
    expect(sum([1, 2])).to.be.equal(3);  
  });  
});
```

Installing Mocha and Chai

- Install Mocha through **npm**

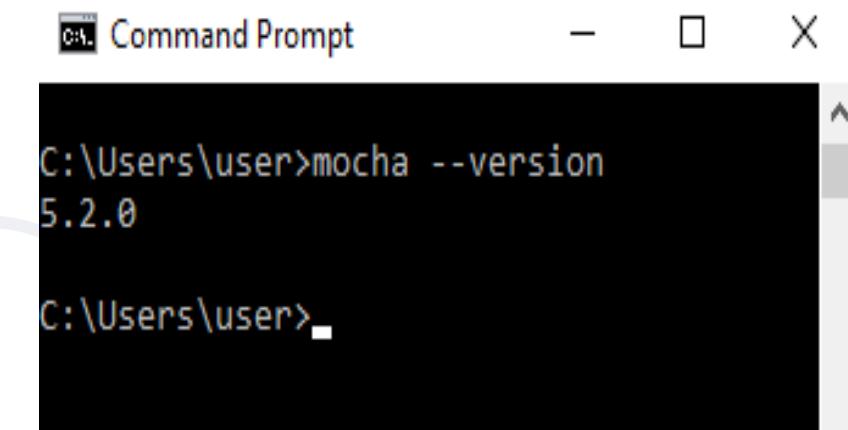
```
npm -g install mocha
```



```
C:\Users\user>npm -g install mocha
C:\Users\user\AppData\Roaming\npm\mocha -> C:\Users\user\
AppData\Roaming\npm\node_modules\mocha\bin\mocha
C:\Users\user\AppData\Roaming\npm\_mocha -> C:\Users\user\
\AppData\Roaming\npm\node_modules\mocha\bin\_mocha
+ mocha@5.2.0
```

- Check if Mocha is installed

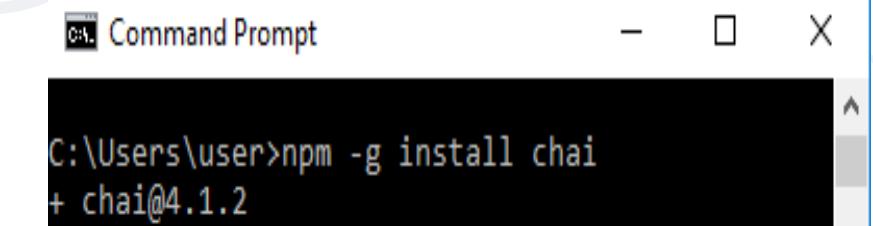
```
mocha --version
```



```
C:\Users\user>mocha --version
5.2.0
C:\Users\user>
```

- Install Chai through **npm**

```
npm -g install chai
```



```
C:\Users\user>npm -g install chai
+ chai@4.1.2
```

Configuring NODE_PATH

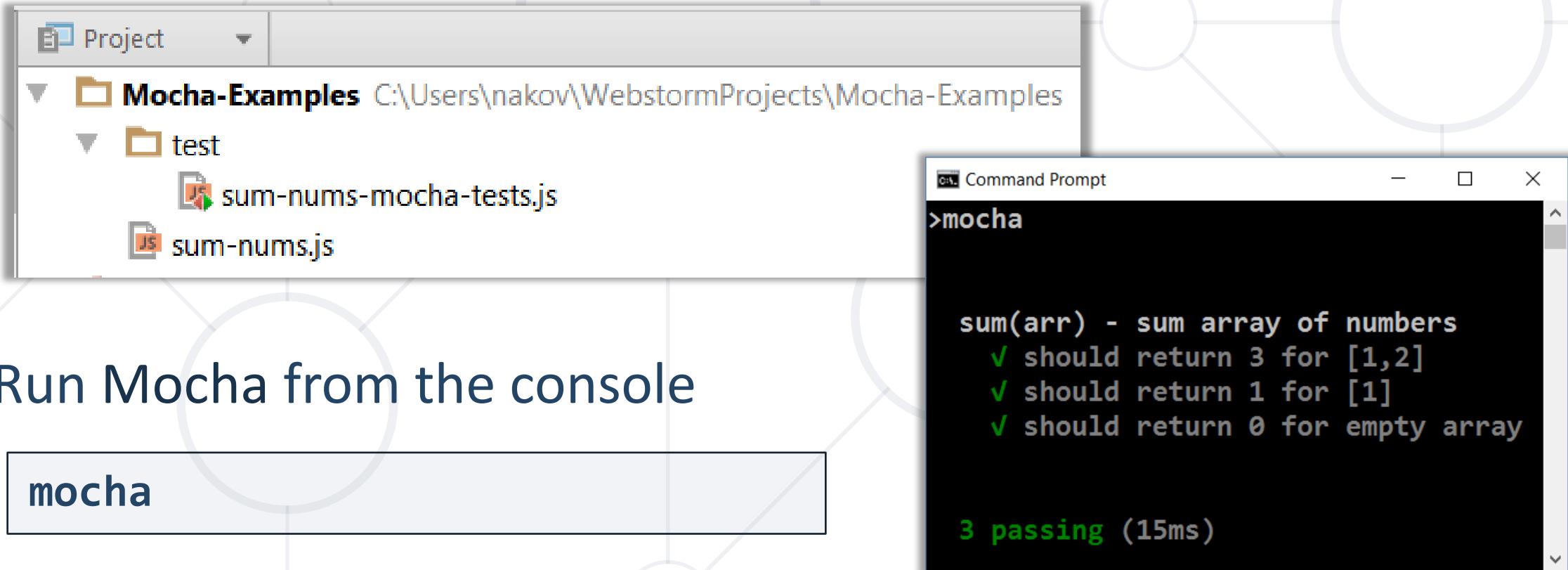
- By default Node.js does not find its globally-installed modules
- You need to set the **NODE_PATH** environment variable

```
rem for any future sessions
setx NODE_PATH %AppData%\npm\node_modules
rem for current session
set NODE_PATH=%AppData%\npm\node_modules
```

- You may need to restart your IDE after changing **NODE_PATH**

Running Mocha

- Create folder "test" in your JS project
 - Put your test code in **test/{test-group-name}.js**



The screenshot shows a WebStorm IDE interface. On the left, the Project tool window displays a file structure under the 'Mocha-Examples' project. It includes a 'test' folder containing two files: 'sum-nums-mocha-tests.js' and 'sum-nums.js'. To the right, a 'Command Prompt' window is open, showing the output of running the Mocha test suite. The command '>mocha' is entered at the prompt. The test results are displayed below, indicating three passing tests: 'should return 3 for [1,2]', 'should return 1 for [1]', and 'should return 0 for empty array'. The total time for the tests is 15ms.

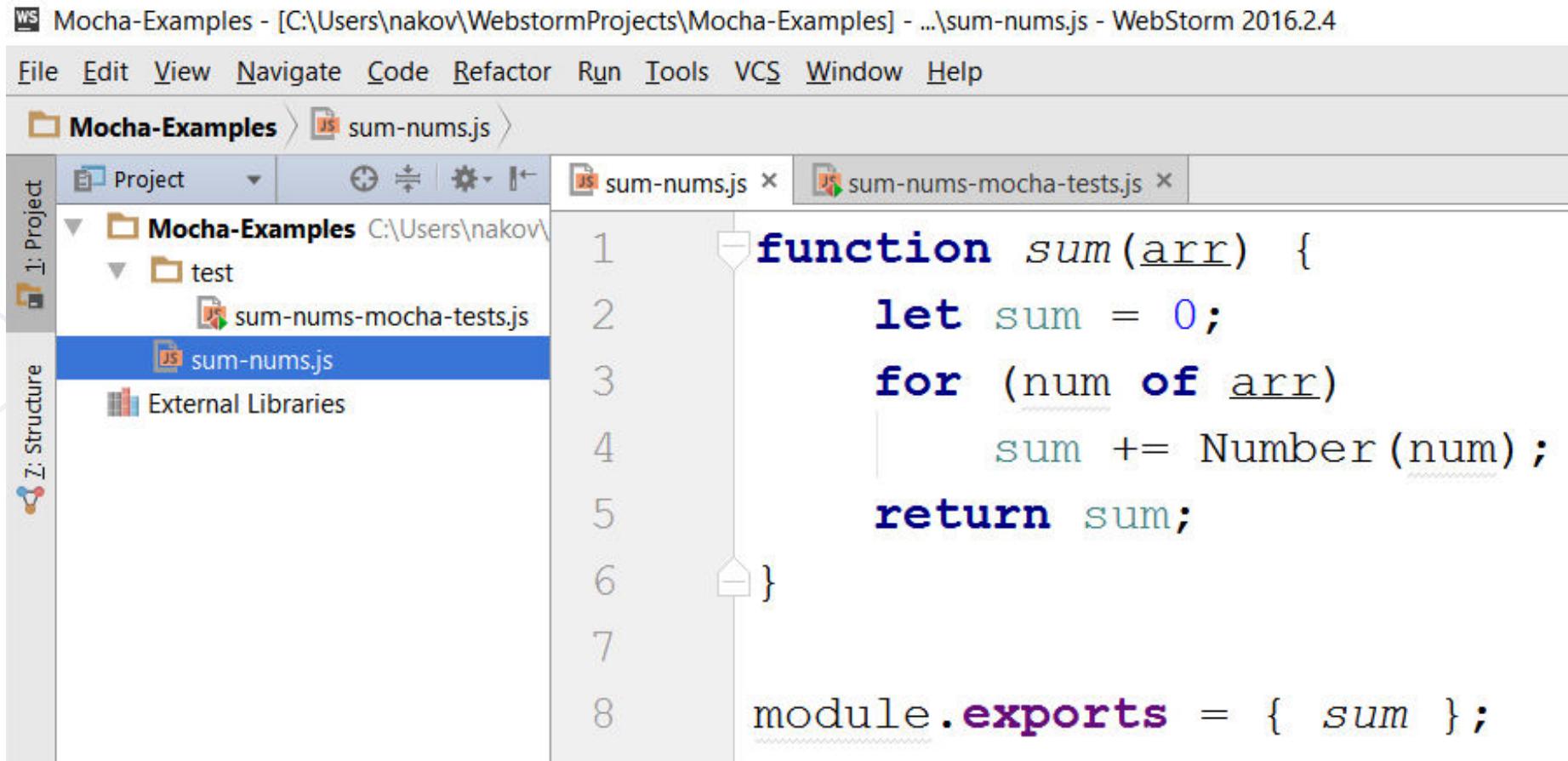
```
mocha

sum(arr) - sum array of numbers
  ✓ should return 3 for [1,2]
  ✓ should return 1 for [1]
  ✓ should return 0 for empty array

3 passing (15ms)
```

Mocha in Action – Sample Project

- Source code to be tested



The screenshot shows the WebStorm IDE interface with the following details:

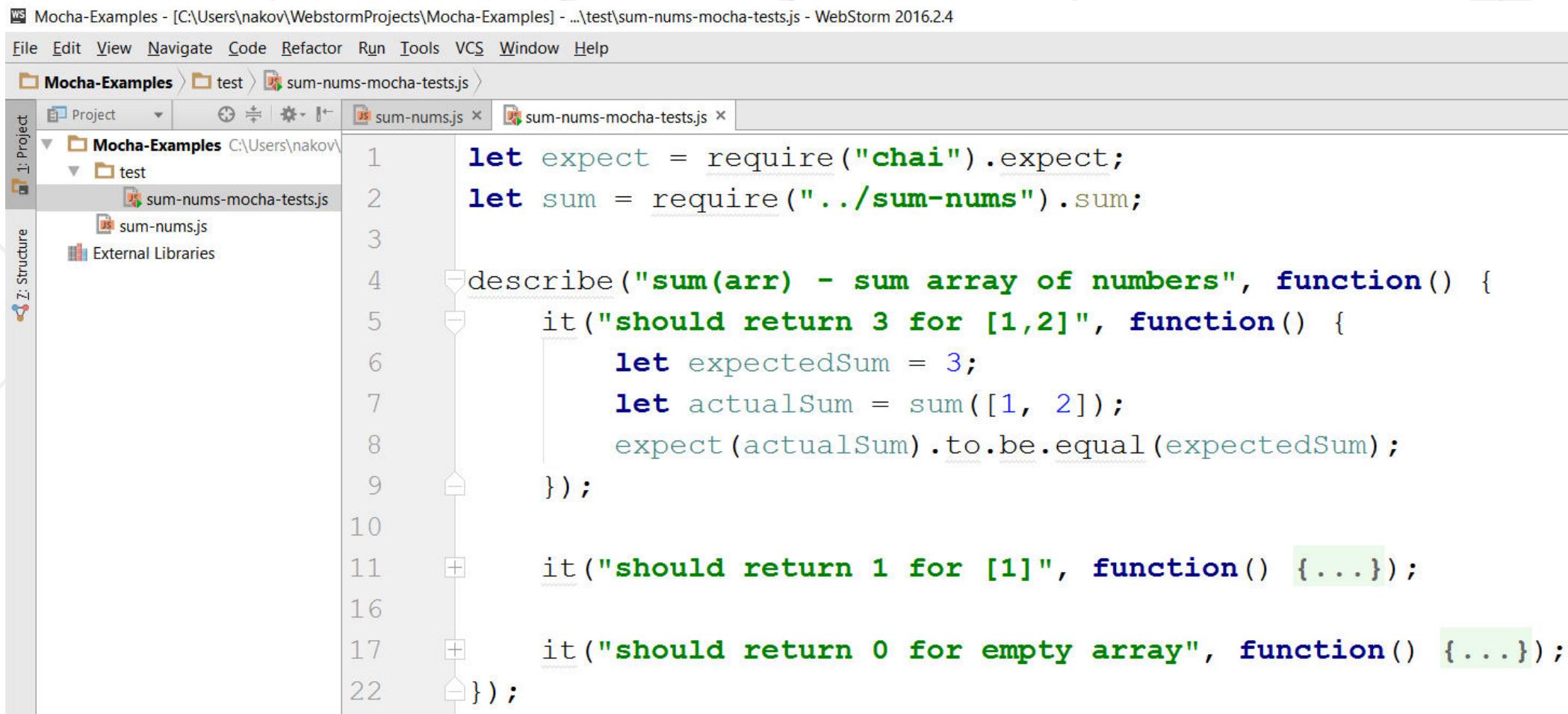
- Title Bar:** Mocha-Examples - [C:\Users\nakov\WebstormProjects\Mocha-Examples] - ...sum-nums.js - WebStorm 2016.2.4
- Menu Bar:** File Edit View Navigate Code Refactor Run Tools VCS Window Help
- Project Structure:** The project is named "Mocha-Examples". It contains a "test" folder which includes "sum-nums-mocha-tests.js" and "sum-nums.js". "sum-nums.js" is currently selected.
- Code Editor:** Two files are open:
 - sum-nums.js:** Contains the following JavaScript code:

```
function sum(arr) {
    let sum = 0;
    for (num of arr)
        sum += Number(num);
    return sum;
}

module.exports = { sum };
```
 - sum-nums-mocha-tests.js:** A file icon is shown, but no code is visible.

Mocha in Action – Sample Unit Tests

■ Tests for the **sum(arr)** function



The screenshot shows the WebStorm IDE interface with the following details:

- Project Structure:** The project is named "Mocha-Examples". It contains a "test" folder which includes "sum-nums-mocha-tests.js" and "sum-nums.js".
- Code Editor:** The file "sum-nums-mocha-tests.js" is open. The code uses Chai assertions to test the "sum" function from "sum-nums.js".
- Code Content:**

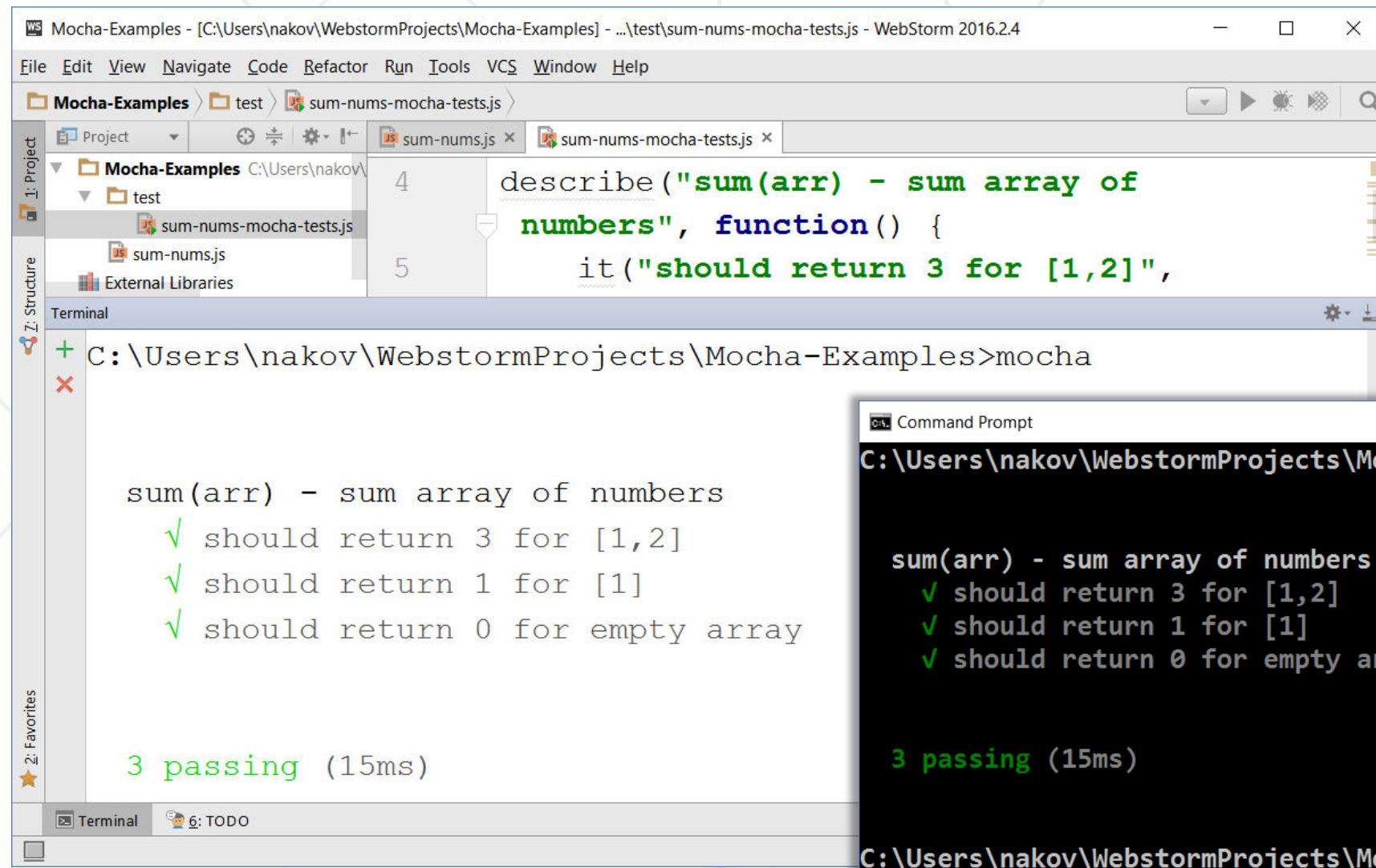
```
let expect = require("chai").expect;
let sum = require("../sum-nums").sum;

describe("sum(arr) - sum array of numbers", function() {
    it("should return 3 for [1,2]", function() {
        let expectedSum = 3;
        let actualSum = sum([1, 2]);
        expect(actualSum).to.be.equal(expectedSum);
    });

    it("should return 1 for [1]", function() { ... });

    it("should return 0 for empty array", function() { ... });
});
```

Mocha in Action – Running the Tests



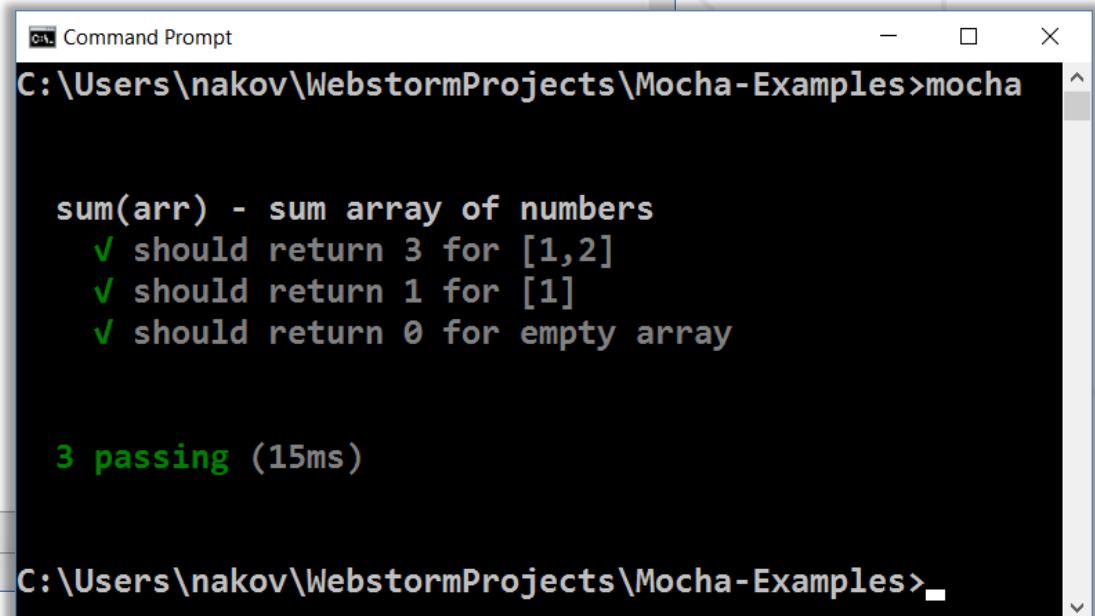
The screenshot shows the WebStorm IDE interface. The top navigation bar includes File, Edit, View, Navigate, Code, Refactor, Run, Tools, VCS, Window, and Help. The left sidebar has sections for Project, Structure, and Favorites. The main editor area displays a file named sum-nums-mocha-tests.js with the following code:

```
describe("sum(arr) - sum array of numbers", function() {
    it("should return 3 for [1,2]",
```

Below the editor is a Terminal window showing the command C:\Users\nakov\WebstormProjects\Mocha-Examples>mocha. The output of the test execution is displayed in green text:

```
sum(arr) - sum array of numbers
  ✓ should return 3 for [1,2]
  ✓ should return 1 for [1]
  ✓ should return 0 for empty array

3 passing (15ms)
```



The screenshot shows a Command Prompt window with the title "Command Prompt". It displays the same Mocha test results as the WebStorm terminal:

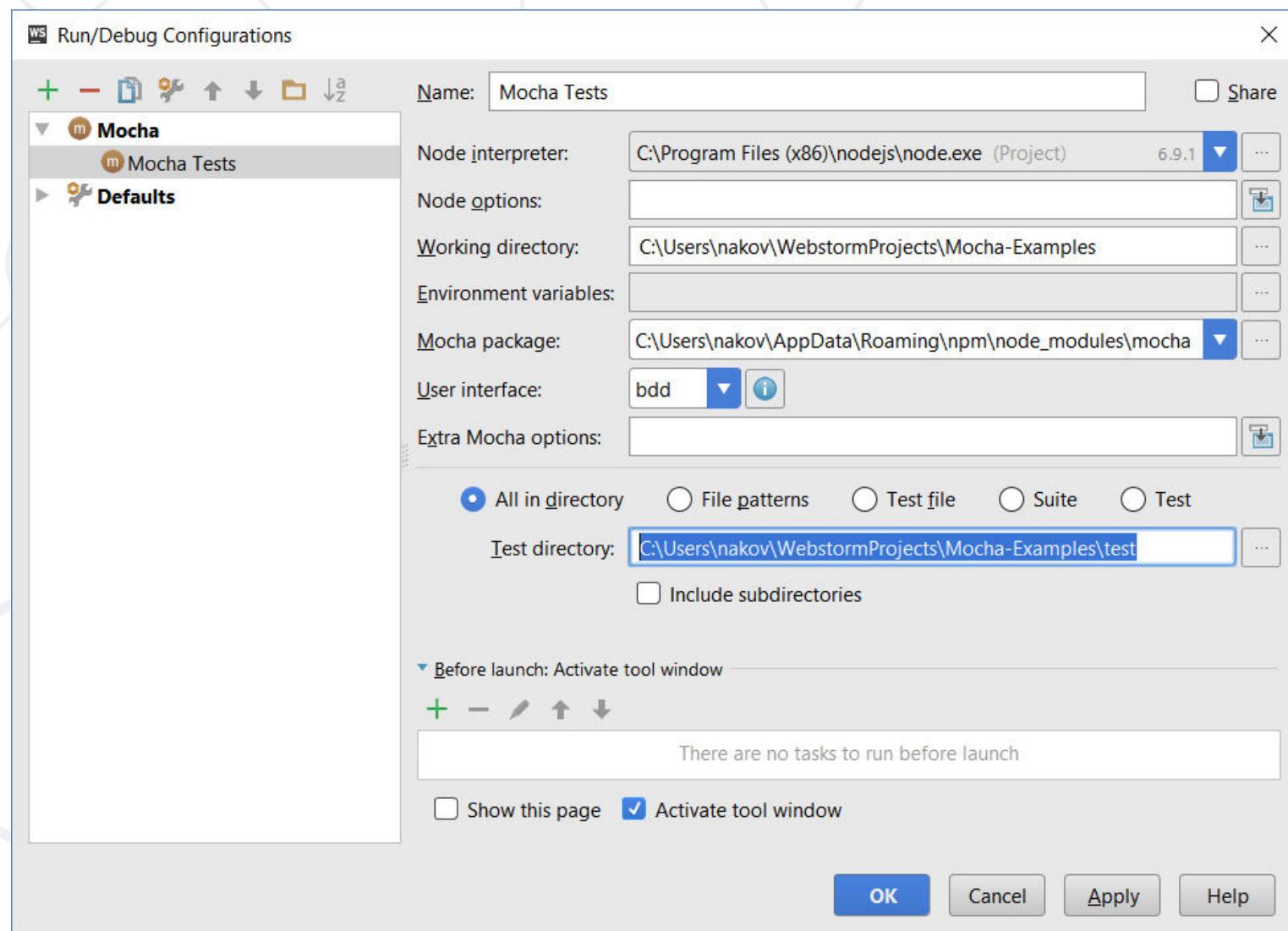
```
C:\Users\nakov\WebstormProjects\Mocha-Examples>mocha

sum(arr) - sum array of numbers
  ✓ should return 3 for [1,2]
  ✓ should return 1 for [1]
  ✓ should return 0 for empty array

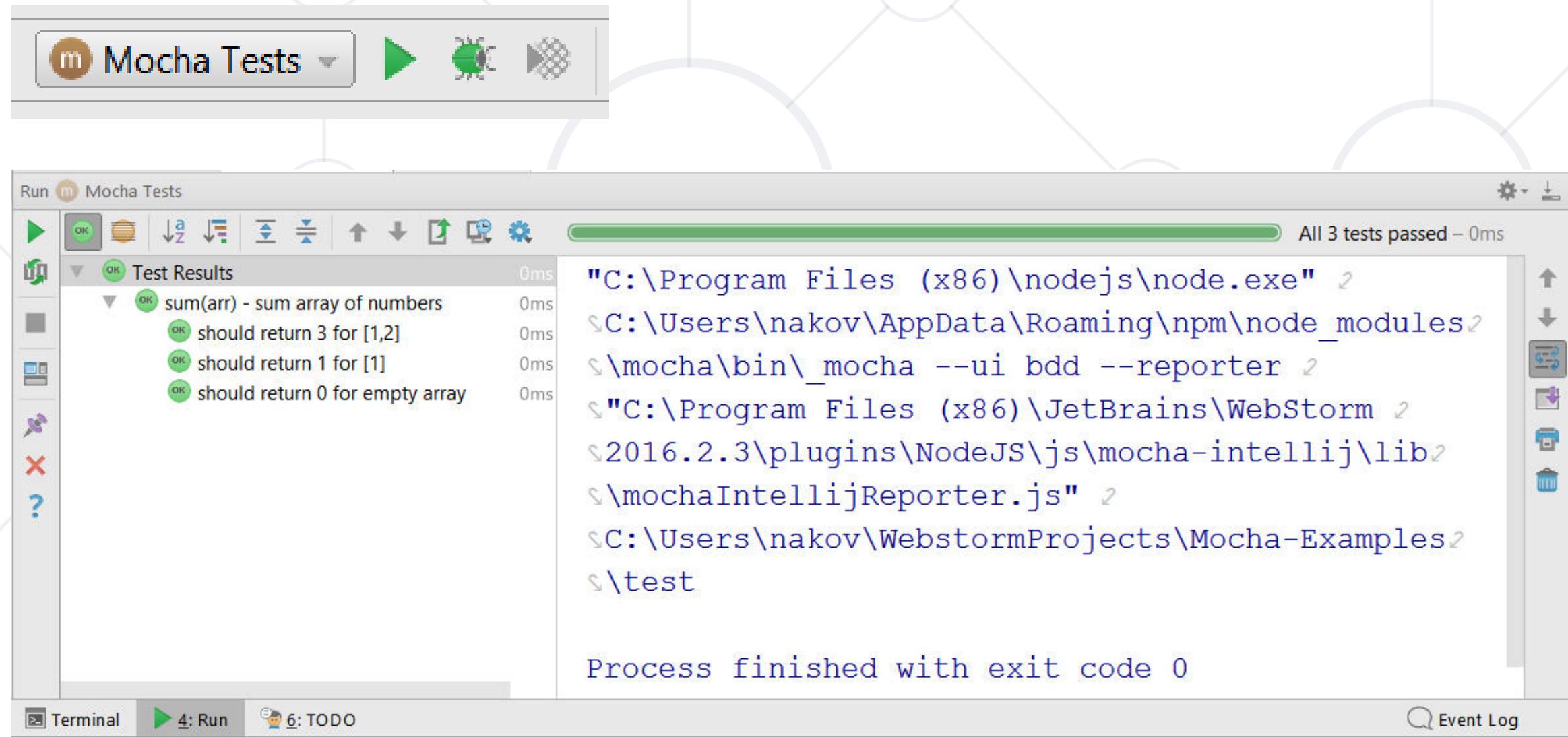
3 passing (15ms)
```

At the bottom of the Command Prompt window, the text "C:\Users\nakov\WebstormProjects\Mocha-Examples>" is visible.

Running Mocha Tests from WebStorm



Running Mocha Tests from WebStorm (2)



The screenshot shows the WebStorm IDE interface with the "Mocha Tests" tool window open. The title bar of the tool window displays "Mocha Tests". Below the title bar are three icons: a green play button, a green circular progress bar with a dot, and a circular grid icon.

The main area of the tool window shows the "Test Results" section. It lists a single test suite named "sum(arr) - sum array of numbers" with four test cases: "should return 3 for [1,2]", "should return 1 for [1]", "should return 0 for empty array", and "should return 0 for null". All test cases are marked as "OK" with green status indicators and 0ms execution time.

A large green progress bar at the top of the results table indicates "All 3 tests passed – 0ms".

On the right side of the results table, there is a vertical toolbar with icons for navigating between tests, copying the current path, and deleting the current test.

Below the results table, a terminal window displays the command used to run the tests:

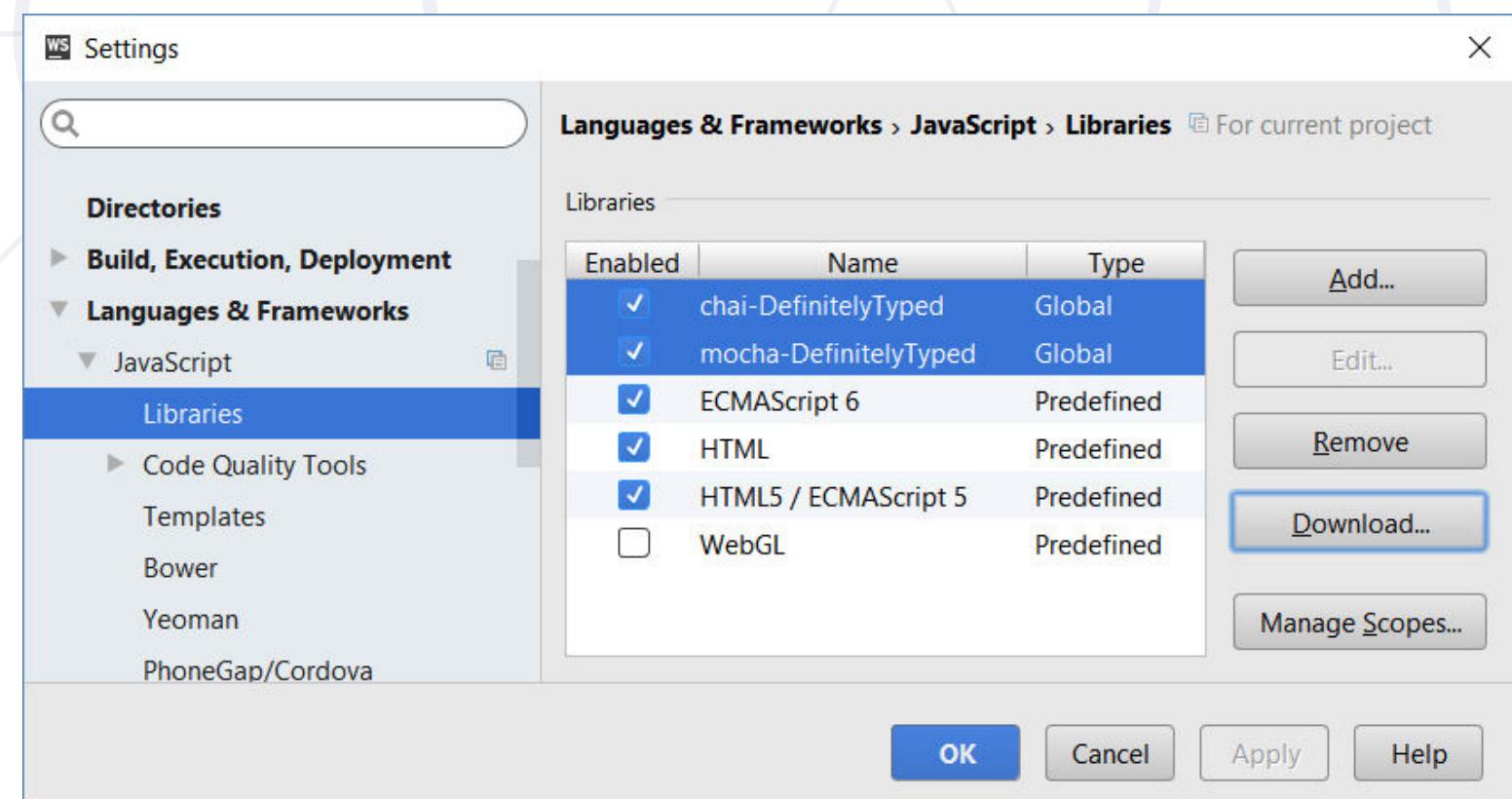
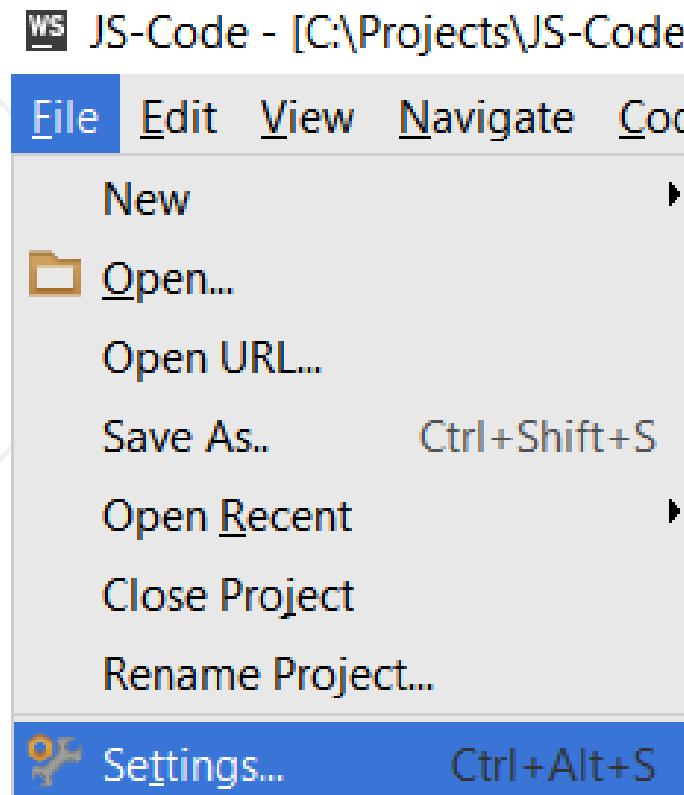
```
"C:\Program Files (x86)\nodejs\node.exe" "C:\Users\nakov\AppData\Roaming\npm\node_modules\mocha\bin\_mocha --ui bdd --reporter "C:\Program Files (x86)\JetBrains\WebStorm 2016.2.3\plugins\NodeJS\js\mocha-intelliJ\lib\mochaIntelliJReporter.js" "C:\Users\nakov\WebstormProjects\Mocha-Examples\test"
```

At the bottom of the terminal window, the message "Process finished with exit code 0" is displayed.

The bottom navigation bar of the IDE includes tabs for "Terminal", "Run" (with a value of "4: Run"), "TODO" (with a value of "6: TODO"), and "Event Log".

Configuring Libraries in WebStorm

- To get the "auto complete" and "syntax checks" working for Mocha and Chai, add them as libraries in WebStorm



Test Groups and Test Cases

```
describe("Test group #1", function() {  
    it("should... when...", function() {  
        expect(actual).to.be.equal(expected);  
    });  
    it("should... when...", function() { ... });  
});  
describe("Test group #2", function() {  
    it("should... when...", function() {  
        expect(actual).to.be.equal(expected);  
    });  
    ...  
});
```

Problem – Sum of Numbers

- You are given a JS code intended to sum an array of numbers:

```
function sum(arr) {  
    let sum = 0;  
    for (num of arr)  
        sum += Number(num);  
    return sum;  
}
```

- Write unit tests to test the **sum(arr)** function with Mocha

Check your solution here: <https://judge.softuni.bg/Contests/307>

Solution – Sum of Numbers – Test Cases

- Design the cases
 - `sum([1, 2]) → 3`
 - `sum([1]) → 1`
 - `sum([]) → 0`
 - `sum([1.5, 2.5, -1]) → 3`
 - `sum("invalid data") → NaN`



Solution – Sum of Numbers – Test Code

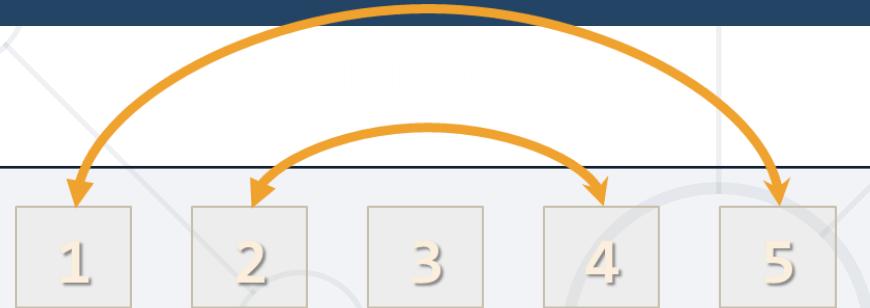
```
describe("sum(arr) - sum array of numbers", function() {  
    it("should return 3 for [1, 2]", function() {  
        expect(sum([1, 2])).to.be.equal(3);  
    });  
    it("should return 1 for [1]", function() {  
        expect(sum([1])).to.be.equal(1);  
    });  
    it("should return 0 for empty array", function() { ... });  
    it("should return 3 for [1.5, 2.5, -1]", function() { ... });  
    it("should return NaN for invalid data", function() { ... });  
});
```

Check your solution here: <https://judge.softuni.bg/Contests/307>

Problem – Check for Symmetry

- You are given the following JS code:

```
function isSymmetric(arr) {  
    if (!Array.isArray(arr))  
        return false; // Non-arrays are non-symmetric  
    let reversed = arr.slice(0).reverse(); // Clone + reverse  
    let equal =  
        (JSON.stringify(arr) == JSON.stringify(reversed));  
    return equal;  
}
```



- Write unit tests to test the above function with Mocha

Check your solution here: <https://judge.softuni.bg/Contests/307>

Solution – Check for Symmetry – Test Cases

- Design the cases
 - [1,2,3,3,2,1] → true
 - [-1,2,-1] → true
 - [1,2] → false
 - [5,'hi',{a:5},new Date(),{a:5),'hi',5] → true
 - [5,'hi',{a:5},new Date(),{X:5),'hi',5] → false
 - 1,2,2,1 → false (not an array)

Solution – Check for Symmetry – Test Code

```
describe("isSymmetric(arr)", function() {
  it("should return true for [1,2,3,3,2,1]", function() {
    let symmetric = isSymmetric([1,2,3,3,2,1]);
    expect(symmetric).to.be.equal(true);
  });
  it("should return false for [1,2,3,4,2,1]", function() { ... });
  it("should return true for [-1,2,-1]", function() { ... });
  it("should return false for [-1,2,1]", function() { ... });
  ...
});
```

Check your solution here: <https://judge.softuni.bg/Contests/307>

Problem – RGB to Hex Color

- Using Mocha write unit tests for the following function:

```
function rgbToHexColor(red, green, blue) {  
    if (!Number.isInteger(red) || (red < 0) || (red > 255))  
        return undefined; // Red value is invalid  
    if (!Number.isInteger(green) || (green < 0) || (green > 255))  
        return undefined; // Green value is invalid  
    if (!Number.isInteger(blue) || (blue < 0) || (blue > 255))  
        return undefined; // Blue value is invalid  
    return "#" +  
        ("0" + red.toString(16).toUpperCase()).slice(-2) +  
        ("0" + green.toString(16).toUpperCase()).slice(-2) +  
        ("0" + blue.toString(16).toUpperCase()).slice(-2);  
}
```

Solution – RGB to Hex Color – Test Cases

- Design the cases
 - (255, 158, 170) → #FF9EAA
 - (0, 0, 0) → #000000
- Start from the typical (normal) cases
 - (12, 13, 14) → #0C0D0E
 - (255, 255, 255) → #FFFFFF
- Then, test the special cases, e.g. invalid input
 - (-1, 0, 0), (0, -1, 0), (0, -1, 0) → undefined
 - (256, 0, 0), (0, 256, 0), (0, 0, 256) → undefined
 - (3.14, 0, 0), (0, 3.14, 0), (0, 0, 3.14) → undefined
 - ("5", [3], {8:9}), (empty input) → undefined

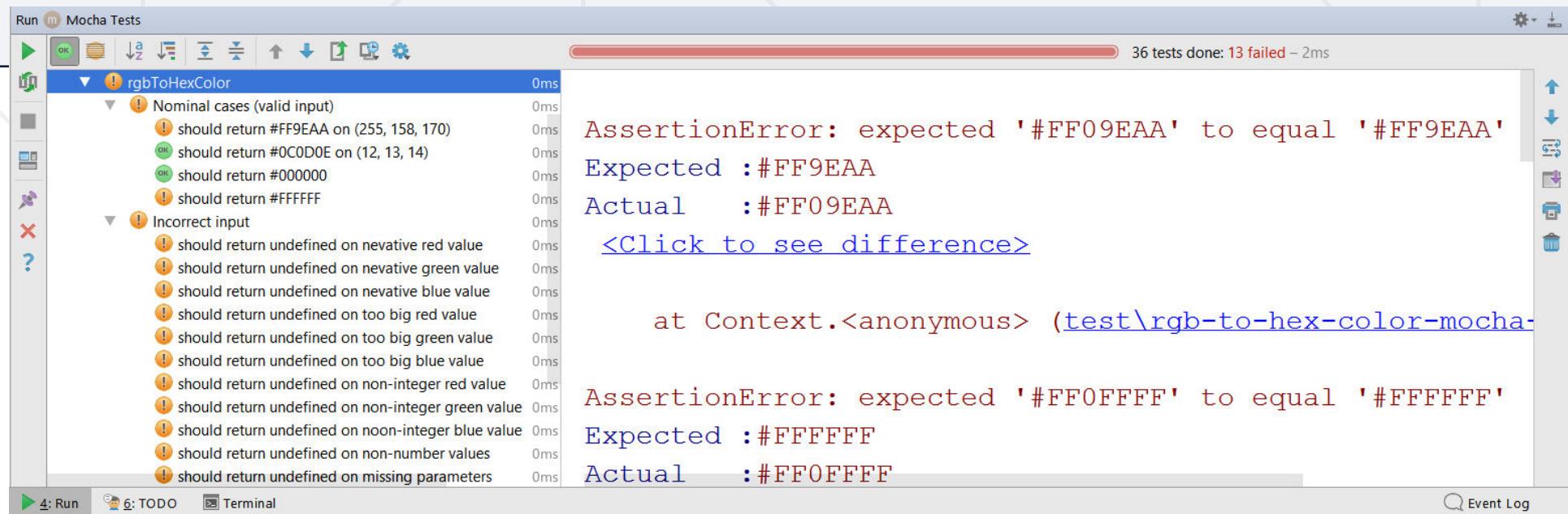
Solution – RGB to Hex Color – Test Code

```
describe("rgbToHexColor(red, green, blue)", function() {
  describe("Nominal cases (valid input)", function() {
    it("should return #FF9EAA on (255, 158, 170)", function() {
      let hex = rgbToHexColor(255, 158, 170);
      expect(hex).to.be.equal('#FF9EAA');
    });
    ...
  });
  describe("Special cases (invalid input)", function() {
    ...
  });
});
```

Check your solution here: <https://judge.softuni.bg/Contests/307>

Testing Incorrect Code

```
function rgbToHexColor(red, green, blue) {  
    return "#" +  
        ("0" + red.toString(16).toUpperCase().slice(-2)) +  
        ("0" + green.toString(16).toUpperCase().slice(-2)) +  
        ("0" + blue.toString(16).toUpperCase().slice(-2));  
}
```



Problem – Add / Subtract Module

- You are given the following JS code:

```
function createCalculator() {  
    let value = 0;  
    return {  
        add: function(num) { value += Number(num); },  
        subtract: function(num) { value -= Number(num); },  
        get: function() { return value; }  
    }  
}
```

- Write unit tests to test the above code with Mocha

Check your solution here: <https://judge.softuni.bg/Contests/307>

Solution – Add / Subtract – Test Cases

- Design the test cases
 - create; get → 0
 - create; add(2); add(3); get → 5
 - create; subtract(3); subtract(2); get → -5
 - create; add(5.3); subtract(1.1); get → 4.2
 - create; add(10); subtract('7'); add('-2'); subtract(-1); get → 2
 - create; add('hello'); get → NaN
 - create; subtract('hello'); get → NaN

Solution – Add / Subtract – Test Code

```
describe("createCalculator()", function() {
  let calc;
  beforeEach(function() {
    calc = createCalculator();
  });
  it("should return 5 after {add 3; add 2}", function() {
    calc.add(3); calc.add(2); let value = calc.get();
    expect(value).to.be.equal(5);
  });
  ...
});
```

Mocha **hook**: called
before each test

Check your solution here: <https://judge.softuni.bg/Contests/307>



Live Exercises in Class (Lab)
Practice: Unit Testing in JS

Summary

- JS functions should do what their name says or indicate a problem
- Unit tests are programs that check whether a piece of code works as expected

```
let expect = require("chai").expect;  
  
describe("sum(arr)", function() {  
  it("shoud sum 1+2 = 3", function() {  
    expect(sum([1, 2])).to.be.equal(3);  
  });  
});
```



Questions?



SoftUni



Software
University



SoftUni
Svetlina



SoftUni
Creative



SoftUni
Digital



SoftUni
Foundation



SoftUni
Kids

SoftUni Diamond Partners



xssoftware



SBTech



telenor



SoftwareGroup
doing it right

NETPEAK



SmartIT



Postbank

Решения за твое упре

**SUPER
HOSTING
.BG**

INDEAVR
Serving the high achievers

INFRASTICS®

LIEBHERR

æternity

codexio

SoftUni Organizational Partners



ИНФОРМАЦИОННО
ОБСЛУЖВАНЕ

One
SOFTV

WORLD
OF
MYTHS



Lukanet.com



codexio

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education and Employment Opportunities
 - softuni.bg
- Software University Foundation
 - <http://softuni.foundation/>
- Software University @ Facebook
 - facebook.com/SoftwareUniversity
- Software University Forums
 - forum.softuni.bg



- This course (slides, examples, demos, videos, homework, etc.) is licensed under the "Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International" license

