**Introduction**

Testing is an essential part of development. When used properly, testing can catch and identify issues with your implementation code before you deploy it to users. Instead of testing every function manually, developers automate their tests with a *test framework*.

Developers use test frameworks to organize and automate tests that provide useful feedback when errors occur. In this lesson we will use the [Mocha test framework](https://mochajs.org/) to write tests against JavaScript methods.

In this lesson you will:

* Learn to write a basic Mocha test suite
* Use Node’s assert.ok method to verify the expected output of your code
* Understand and apply the four phases of a test to create an expressive testing suite
* Evaluate the quality of your tests against the [characteristics of a good test](https://www.codecademy.com/articles/tdd-u2-good-test)

**Instructions**

**1.**

A complete test suite is provided for you on the right. Run the suite and view its output by typing npm test in the terminal and hitting Enter.

Can you match the test output in your terminal to strings in **index\_test.js**?

const assert = require('assert');

describe('Math', () => {

  describe('.max', () => {

    it('returns the argument with the highest value', () => {

      const minimum = 1;

      const median = 2;

      const maximum = 3;

      const result = Math.max(median, minimum, maximum);

      assert.ok(result === maximum);

    });

    it('returns -Infinity when no arguments are provided', () => {

      const negInfinity = -Infinity;

      const result = Math.max();

      assert.ok(result === negInfinity);

    });

  });

});

**Install Mocha I**

Before writing any tests you’ll need to use [Node.js and npm](https://www.codecademy.com/articles/what-is-node) to set up a JavaScript project and install Mocha.

* *Node* allows you to run JavaScript in the terminal
* *npm* is a Node tool that allows you to download packages from the web, and manage them in a JavaScript project
* *Mocha* is one of those packages and is used to test other JavaScript code

A JavaScript project is a directory of files. The following command creates a file **package.json** that can be used to manage packages for the project.

$ npm init

After running this command you will be prompted to enter information about your project. It’s okay to skip some fields if you’re not ready to enter that information.

With your project setup, you can install packages.

$ npm install mocha -D

Here’s what this command means:

* npm install tells npm to install a package from the internet and any other packages it depends on
* mocha is the package you want to download
* -D notes that this package is a dependency for your project, which makes it easier for other developers to use

Once you npm install packages, you can find the packages and all their dependencies in the **node\_modules** folder. The new directory structure contains the following:

project

|\_ node\_modules

|\_\_\_ .bin

|\_\_\_ mocha

|\_\_\_ ...

|\_ package.json

The ... in the file structure represents other packages that are a dependency for Mocha.

**Instructions**

**1.**

Initialize the project. In the terminal window type:

$ npm init

Hit the Enter key to skip each prompt. This creates a **package.json** file in your project directory.

Check your work after each instruction.

Checkpoint 2 Passed

**2.**

Install Mocha as a package and save it as a dependency. In the terminal window type:

$ npm install mocha -D

You can view **package.json** in the text editor. You can now see mocha as a dependency!

# Install Mocha II

After installing Mocha as a dependency we can run it in two ways.

The first (and more tedious) method is to call it directly from **node\_modules**:

$ ./node\_modules/mocha/bin/mocha

The second (and recommended) method is to add a script to **package.json**. In the scripts object in **package.json**, set the value of "test" to mocha. It should look like this:

"scripts": {

"test": "mocha"

}

Now you can call Mocha with the following command:

$ npm test

Instead of manually running each test in the **test** directory, you can use this command to run the full test suite automatically.

**Instructions**

**1.**

Using the guide above, set the test script to "mocha" in **package.json**. This connects the command line command npm test with the Mocha package.

Checkpoint 2 Passed

**2.**

Run the test suite with Mocha. In the terminal window type:

$ npm test

The command will be executed but you will see a warning in the terminal: there aren’t any tests to run yet!

**describe and it blocks**

In Mocha we group tests using the describe function and define tests using the it function. These two functions can be used to make your test suite *complete*, *maintainable*, and *expressive* in the following ways:

* Structure your test suite: you can organize tests into nested groups that reflect the structure of your implementation code.
* Provide informative messages: you can define your tests using human-readable strings.

If you are testing a Math object with the method .max, you could use the following test code.

describe('Math', () => {

describe('.max', () => {

it('returns the argument with the highest value', () => {

// Your test goes here

});

it('returns -Infinity when no arguments are provided', () => {

// Your test goes here

});

});

});

Both the describe and it functions accept two parameters: a descriptive string and a callback function. Though the functions are flexible, they are commonly used in the structure above: nest describe blocks to resemble the structure of your implementation code and write individual tests in it blocks. This makes your test suite *isolated*, *maintainable*, and *expressive*.

**Instructions**

**1.**

Using the previous example as a guide, you will write a describe - describe - it block for the function Math.min().

In **test/index\_test.js**, create your first describe block using 'Math' as the descriptive string.

Checkpoint 2 Passed

**2.**

Within the block, create another describe block using '.min' as the descriptive string.

Checkpoint 3 Passed

**3.**

Within the block, create an it block using 'returns the argument with the lowest value' as the descriptive string.

# assert

In the last exercise, you learned how to organize and automate tests using the Mocha test framework. To write the tests themselves, we can use the assert.ok method provided by Node.js.

In programming, a test compares an expected outcome to an actual outcome. For example, we expect the outcome of the following code…

const a = 1 + 2;

…to be: a has a value of 3. To test the value saved to a with plain JavaScript, you would need to write a conditional statement comparing a to the expected result. Inside the statement, you would construct an error when the actual outcome does not match the expected.

assert.ok() allows you to compare values and throw errors as needed using one function call. The small, human-readable format of the functions will help you make a more expressive test suite.

As a Node module, assert can be imported at the top of your files with

const assert = require('assert');

You call assert functions like this:

assert.ok(a === 3);

In this case a === 3 evaluates to true, so no error is thrown.

If an argument passed to assert.ok() evaluates to false, an AssertionError is thrown. The error communicates to Mocha that a test has failed, and Mocha logs the error message to the console.

**Instructions**

**1.**

At the top of **index\_test.js**, import assert with the require function.

Checkpoint 2 Passed

**2.**

Within the provided it block, enter the following expression:

assert.ok(3 + 4 === 8);

Checkpoint 3 Passed

**3.**

Run the test suite.

Checkpoint 4 Passed

Hint

Run npm test in the terminal.

**4.**

The argument passed to assert.ok() throws an error! You should see some descriptive feedback in the console.

Make the expression evaluate to true by replacing 8 with 7.

Checkpoint 5 Passed

**5.**

Run the suite again. Your test should pass now!

# Setup, Exercise, and Verify

In this exercise you will be separating a test into setup, exercise, and verify phases. This distinct and well-defined separation of steps makes your test more reliable, maintainable, and expressive.

The phases are defined as follows:

* Setup - create objects, variables, and set conditions that your test depends on
* Exercise - execute the functionality you are testing
* Verify - check your expectations against the result of the exercise phase. You can use the assert library here

Clearly separating each phase makes a test easier to read, change, and validate.

**Instructions**

**1.**

The same test is written twice – once in a naive approach, once using setup, exercise, and verify phases.

Run the test suite.

Checkpoint 2 Passed

**2.**

Both tests have errors! The test output will tell you which line threw an error for each test.

Following the examples below, fix both tests using the correct call to .pop().

const arr = ['entry1', 'entry2'];

['entry1', 'entry2'].pop(); // for naive test

arr.pop(); // for 3phase test

Checkpoint 3 Passed

Hint

In the naive approach edit the following expression:

assert.ok(pop(['padawan', 'knight']) === 'knight');

In the 3 phase approach replace the following expression:

const popped = pop(jediPath);

**3.**

Which test approach was easier to read and edit? Answer this for yourself then move on to the next exercise.

# Teardown

Some tests require a fourth phase called teardown. This step makes your test isolated.

* Teardown - reset any conditions that were changed during the test

A test, like the example in this exercise, can make changes to its environment that could affect other tests. The teardown phase is used to reset the environment before the next test runs.

Some common changes to an environment include

* altering files and directory structure
* changing read and write permissions on a file
* editing records in a database

In some cases — including the previous exercise — the teardown phase isn’t needed because there are no conditions to reset.

**Instructions**

**1.**

This exercise uses Node’s filesystem library fs in addition to assert. It’s okay if you’re not familiar with fs: each function is described in comments.

The call to fs.appendFileSync(path, str) will do one of two things:

1. If a file at path does not exist, the method creates the file and adds the string str to that file.
2. If the file does exist, the method adds the string str to the end of the file.

The current test does not have a teardown phase. Run it using the terminal.

Checkpoint 2 Passed

**2.**

Run the test suite again. Compare this output to the output of the previous test run.

Note the new file message.txt created in your directory.

project

|\_ node\_modules

|\_ test

|\_\_\_ index\_test.js

|\_ message.txt

|\_ package.json

Checkpoint 3 Passed

**3.**

You got different output because the test was not isolated.

* In the first run the file didn’t exist, so fs.appendFileSync created a message.txt file containing the string.
* In the second run, the method added the string to the end of message.txt because it already existed.

In the terminal enter rm message.txt to delete the file.

Checkpoint 4 Passed

**4.**

Add this code below // Teardown. This method will delete the file at path before the test finishes.

fs.unlinkSync(path);

Checkpoint 5 Passed

**5.**

Verify that you fixed the issue: run the test suite twice and compare the outputs.

# Hooks

Using teardown in the it block made your test isolated, but not reliable.

If the system encounters an error before it reaches the teardown, it will not execute that phase. In the previous example, an error may occur after the file is created but before it is deleted. The file would persist and may cause false negatives in future test runs.

Mocha provides hooks to solve that problem.

A hook is a piece of code that is executed when a certain event happens. Hooks can be used to set and reset conditions like the setup and teardown phases do. In Mocha, a hook is written within a describe block.

describe('example', () => {

afterEach(() => {

// teardown goes here

});

it('.sample', () => {

// test goes here

});

});

In this example the function passed to afterEach is called after each it block is executed.

The other hooks in the Mocha library are before(), beforeEach(), and after(). You can read more about them on the [Mocha website](https://mochajs.org/#hooks).

**Instructions**

**1.**

Run the test suite.

Checkpoint 2 Passed

**2.**

A misspelling on line 16 caused the TypeError that you see. It prevented the execution of the teardown phase! This means the message.txt file wasn’t deleted (see below).

project

|\_ node\_modules

|\_ test

|\_\_\_ index\_test.js

|\_ message.txt

|\_ package.json

In the terminal enter rm message.txt to delete the file.

Checkpoint 3 Passed

**3.**

Within the describe block, create an afterEach hook with an empty function as its argument.

Checkpoint 4 Passed

Hint

describe('...', () => {

afterEach(() => {

});

it('...', () => {

});

**4.**

Move the teardown phase into the afterEach hook.

Checkpoint 5 Passed

Hint

describe('...', () => {

afterEach( () => {

// paste teardown phase here

});

it('...', () => {

// cut teardown phase here

});

**5.**

Confirm the hook works!

Run test suite and find that message.txt is deleted despite the error in it block.

# Review

You can now write tests with Mocha and Node’s assert.ok() ! In this lesson you learned to:

* Install Mocha with npm
* Organize tests with describe() and it()
* Ensure your tests are isolated and expressive with the four phases of a test
* Ensure your tests are reliable with hooks
* Write assertions with assert.ok()

As you continue to write tests, remember to always evaluate them against [the characteristics of a good test](https://www.codecademy.com/articles/tdd-u2-good-test): fast, complete, reliable, isolated, maintainable, and expressive. If you are meeting these six criteria, you are creating high quality test frameworks!

# Introduction

A good test framework is [fast, complete, reliable, isolated, maintainable, and expressive](https://www.codecademy.com/articles/tdd-u2-good-test). In this lesson you will learn how to use Node’s assert library to write more expressive tests.

An expressive test is easy to read and descriptive, making it useful as a form of documentation for your implementation code. One way to make a test more expressive is clarifying its verify phase — the step where expected outcome is compared to actual outcome.

[Node.js](https://www.codecademy.com/articles/what-is-node) provides a library called assert with methods that help you write more expressive verification code. You can use the methods in this library in place of conditional statements to write less code and use human-readable language. It can be used within the Mocha testing framework, and you will be using both throughout this lesson.

**Instructions**

**1.**

An assertion has been provided within a Mocha describe - it block. Run the Mocha test suite by typing npm test in the terminal and hitting Enter.

Review the output and move on to the next exercise.

const assert = require('assert');

describe('+', () => {

  it('returns the sum of its arguments', () => {

    assert.ok(3 + 4 === 7);

  });

});

# assert.ok

As a Node module, assert can be imported at the top of your files with

const assert = require('assert');

The [functions in the assert library](https://nodejs.org/api/assert.html) compare values and throw errors as needed using one function call. The small, human-readable format of the functions will help you make a more expressive test suite.

assert.ok(6 - 1 === 5);

In this case 6 - 1 === 5 evaluates to true, so no error is thrown.

If an argument passed to assert.ok() evaluates to false, an AssertionError is thrown. The error communicates to Mocha that a test has failed, and Mocha logs the error message to the console.

**Instructions**

**1.**

At the top of **index\_test.js**, import assert with the require function.

Checkpoint 2 Passed

Hint

const someVar = require('moduleName')

**2.**

Within the provided it block, enter the following expression:

assert.ok(result === 91);

Checkpoint 3 Passed

**3.**

Run the test suite.

Checkpoint 4 Passed

**4.**

The argument passed to assert.ok() throws an error! You should see some descriptive feedback in the terminal.

Make the expression evaluate to true by replacing 91 with expected.

Checkpoint 5 Passed

**5.**

Run the suite again. Your test should pass now!

# assert.equal

You can use assert.ok() for most verifications, but sometimes it can be difficult to determine the condition you are evaluating.

Read the example code below. Will this assertion throw an error?

const landAnimals = ['giraffe', 'squirrel'];

const waterAnimals = ['shark', 'stingray'];

landAnimals.push('frog');

waterAnimals.push('frog');

assert.ok(landAnimals[2] == waterAnimals[2]);

The above assertion is checking for equality. In order to understand this you must evaluate the entire expression within the parentheses of .ok().

You can instead use assert.equal() which does the == comparison for us.

In the example below, the two methods achieve the same outcome.

assert.ok(landAnimals[2] == waterAnimals[2]);

assert.equal(landAnimals[2], waterAnimals[2]);

The second line is more expressive: instead of parsing the entire statement, a reader only needs to read the first two words to know the test is verifying equality!

**Instructions**

**1.**

Replace the call to assert.ok() with the equivalent call to assert.equal().

Though your test will work regardless of the order you pass the actual and expected results, it is common practice to pass actual first and expected second.

Checkpoint 2 Passed

Hint

Don’t forget to use a comma: assert.equal(result, expected).

**2.**

Run the test suite.

// Import assert here

const assert = require('assert');

describe('-', () => {

  it('returns the difference of two values', () => {

    const bigNum = 100;

    const smallNum = 4;

    const expected = 96;

    const result =  bigNum - smallNum;

    // Write assertion here

    assert.equal(result, expected);

  });

});

# assert.strictEqual

Take a look at the code below. Will these assertions throw errors?

const a = 3;

const b = '3';

assert.ok(a == b);

assert.ok(a === b);

* The first assertion will not throw an error because it uses loose (==) equality. It performs a type conversion when comparing two things.
* The second will throw an error because it uses strict (===) equality. It returns false if the types differ.

If you need to be strict in evaluating equality, you can use assert.strictEqual().

* assert.equal() performs a == comparison
* assert.strictEqual() performs a === comparison

Compare the following code to the previous example. This code performs the same verifications, but it is more expressive. Without parsing any logic, a reader would know the intention of your tests by reading the method names.

const a = 3;

const b = '3';

assert.equal(a, b);

assert.strictEqual(a, b);

**Instructions**

**1.**

Set expected to the string '96' and run the test suite. The test still passes!

Checkpoint 2 Passed

**2.**

Replace assert.equal() with assert.strictEqual()

Checkpoint 3 Passed

**3.**

Run the test suite in the command line. The test fails!

Checkpoint 4 Passed

**4.**

Reset expected to 96 so that assert.strictEqual() does not return an error.

Checkpoint 5 Passed

**5.**

Run the test suite.

// Import assert here

const assert = require('assert');

describe('-', () => {

  it('returns the difference of two values', () => {

    const bigNum = 100;

    const smallNum = 4;

    const expected = 96;

    const result =  bigNum - smallNum;

    // Write assertion here

    assert.strictEqual(result, expected);

  });

});

**assert.deepEqual I**

Do you think these assertions will throw errors?

const a = {relation: 'twin', age: '17'};

const b = {relation: 'twin', age: '17'};

assert.equal(a, b);

assert.strictEqual(a, b);

Both assertions will throw an error because distinct objects are not considered equal when using either loose or strict equality in JavaScript.

If you need to compare the values within two objects, you can use assert.deepEqual(). This method compares the values of each object using loose (==) equality.

The following code will not throw an error…

assert.deepEqual(a, b);

…and you can confirm by manually comparing the relation and age properties of each object.

a.relation == b.relation;

a.age == b.age;

**Instructions**

**1.**

Run the test suite. You should see an error thrown.

Checkpoint 2 Passed

**2.**

Replace assert.equal() with assert.deepEqual().

Checkpoint 3 Passed

**3.**

Run the test suite again.

const assert = require('assert');

describe('+', () => {

  it('returns the sum of two values', () => {

    // Setup

    let expected = {a: 3, b: 4, result: 7};

    let sum = {a: 3, b: 4};

    // Exercise

    sum.result = sum.a + sum.b;

    // Verify

    assert.deepEqual(sum, expected);

  });

});

# assert.deepEqual II

In the last exercise you used deepEqual() to compare the values of two objects with loose equality. Arrays are also objects, so deepEqual() also compares their values with loose equality.

const arr1 = [1, 2, 3];

const arr2 = [1, 2, 3];

const arr3 = [1, 2, '3'];

assert.deepEqual(arr1, arr2); // No error

assert.deepEqual(arr1, arr3); // No error

If you’d like to learn more about deepEqual(), the documentation is available [here](https://nodejs.org/api/assert.html#assert_assert_deepequal_actual_expected_message).

**Instructions**

**1.**

Run the test suite. You should see an error thrown.

Checkpoint 2 Passed

**2.**

Replace assert.equal() with assert.deepEqual().

Checkpoint 3 Passed

**3.**

Run the test suite again.

const assert = require('assert');

describe('+', () => {

  it('returns the sum of two values', () => {

    // Setup

    let expected = [3, 4, 7];

    let sum = [3, 4];

    // Exercise

    sum.push(3 + 4);

    // Verify

    assert.deepEqual(sum, expected);

  });

});

# Other assert methods

Node’s assert library includes more than the four methods covered so far. You can find them all in the [Node.js documentation](https://nodejs.org/api/assert.html).

Many of the methods can be implemented using other methods, like using assert.ok(1 == 1) to implement assert.equal(1,1). Although this can be used for many types of tests, more descriptive functions like assert.equal, assert.strictEqual, and assert.deepEqual allow us to be more expressive: the expected functionality of the implementation code is clear to anyone reading your tests.

**Instructions**

**1.**

Using the link above, go to the assert documentation and find a more expressive way to implement assert.ok(1 !== 2).

Hint

Read the documentation for [assert.notEqual()](https://nodejs.org/api/assert.html#assert_assert_notequal_actual_expected_message).

const assert = require('assert');

describe('Numbers', () => {

  it('1 does not equal 2', () => {

    // Verify

    assert.notEqual(1, 2);

  });

});

# Review

You can now write tests with Node’s assert library! In this lesson you learned to:

* Check for loose (==) equality with assert.equal()
* Check for strict (===) equality with assert.strictEqual()
* Check the equality of two object’s values with assert.deepEqual()
* Make your tests expressive by using different assert methods found in the [Node.js documentation](https://nodejs.org/api/assert.html).

As you continue to write tests, remember to always evaluate them against [the characteristics of a good test](https://www.codecademy.com/articles/tdd-u2-good-test): fast, complete, reliable, isolated, maintainable, and expressive. If you are meeting these six criteria, you are creating high quality test frameworks!