Test-Driven Development in JavaScript – How to Use Jest

Test-driven development is a coding practice where you write the result you want your program to produce before creating the program.

In other words, TDD requires you to pre-specify the output your intended program must produce to pass the test of functioning the way you envisioned.

So, in an effective test-driven development practice, you would first write tests that express the result you expect from your intended program.

Afterward, you would develop the program to pass the prewritten test.

1. Write a test specifying the result you expect the method to produce to pass the test of being the program you had in mind.
2. Develop the method to pass the prewritten test.
3. Run the test to check whether the method passes or fails the test.
4. Refactor your test code (if necessary).
5. Refactor your program (if necessary).
6. Continue the cycle until the method matches your vision.

Let's now see a JavaScript example of a TDD workflow.

JavaScript Example of a Test-Driven Development Workflow:

Suppose we want to build a calculator. The steps below will use a simple JavaScript program to show us how to approach TDD.

1. Write your test

Write a test that specifies the result we expect our calculator program to produce:

```
function additionCalculatorTester() {
  if (additionCalculator(4, 6) === 10) {
    console.log("✅ Test Passed");
  } else {
    console.error("❌ Test Failed");
  }
}
```

2. Develop our program

Develop the calculator program to pass the prewritten test:

```
function additionCalculator(a, b) {
  return a + b;
}
```

3. Run the test

Run the test to check whether the calculator passes or fails the test:

```
additionCalculatorTester();
```

4. Refactor the test

After you've confirmed that your program passed the prewritten test, it's time to check if there's any need to refactor the test.

For instance, you could refactor additionCalculatorTester() to use a conditional operator like so:

```
function additionCalculatorTester() {
  additionCalculator(4, 6) === 10
    ? console.log("✅ Test Passed")
    : console.error("❌ Test Failed");
}
```

5. Refactor the program

Let's also refactor the production code to use an arrow function.

```
const additionCalculator = (a, b) => a + b;
```

6. Run the test

Rerun the test to ensure your program still works as intended.

```
additionCalculatorTester();
```

Notice that in the examples above, we implemented TDD without using any libraries.

But you can also use powerful test-running tools, like Jasmine, Mocha, Tape, and Jest, to make your test implementation faster, simpler, and more fun.

Let's see how to use Jest, for example.

How to Use Jest as a Test Implementation Tool

Here are the steps you'll need to follow to get started using Jest as your test implementation tool:

Step 1: Get the right Node and NPM version

Make sure you have Node 10.16 (or greater) and NPM 5.6 (or greater) installed on your system.

Step 2: Create a project directory

Create a new folder for your project.

```
mkdir addition-calculator-jest-project
```

Step 3: Navigate to your project folder

Using the command line, navigate to your project directory.

```
cd path/to/addition-calculator-jest-project
```

Step 4: Create a package.json file

Initialize a package.json file for your project.

```
npm init -y
```

Step 5: Install Jest

Install Jest as a development dependency package like so:

```
npm install jest --save-dev
```

Step 6: Make Jest your project's test runner tool

Open your package.json file and add Jest to the test field.

```
{
  "scripts": {
    "test": "jest"
  }
}
```

Step 7: Create your project file

Create a file that you will use to develop your program.

```
touch additionCalculator.js
```

Step 8: Create your test file

Create a file that you will use to write your test cases.

```
touch additionCalculator.test.js
```

Your test file's name must end with .test.js—so that Jest can recognize it as the file containing your test code.

Step 9: Write your test case

Open your test file and write a test code that specifies the result you expect your program to produce.

Here's an example:

```
// additionCalculator.test.js

const additionCalculator = require("./additionCalculator");

test("addition of 4 and 6 to equal 10", () => {
  expect(additionCalculator(4, 6)).toBe(10); });
```

Here's what we did in the snippet above:

1. We imported the additionCalculator.js project file into the additionCalculator.test.js test file.
2. We wrote a test case specifying that we expect the additionCalculator() program to output 10 whenever users provide 4 and 6 as its argument.

Step 10: Develop your program

Open your project file and develop a program to pass the prewritten test.

Here's an example:

```
// additionCalculator.js

function additionCalculator(a, b) {
  return a + b;
}

module.exports = additionCalculator;
```

The snippet above created an additionCalculator() program and exported it with the module.exports statement.

Step 11: Run the test

Run the prewritten test to check if your program passed or failed.

```
npm run test
```

Suppose your project contains multiple test files and you wish to run a specific one. In such a case, specify the test file like so:

```
npm run test additionCalculator.test.js
```

Once you've initiated the test, Jest will print a pass or fail message on your editor's console. The message will look similar to this:

```
$ jest
 PASS  ./additionCalculator.test.js
  √ addition of 4 and 6 to equal 10 (2 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        2.002 s
Ran all test suites.
Done in 7.80s.
```

If you prefer Jest to run your test automatically, add the --watchAll option to your package.json's test field.

Here's an example:

```
{
  "scripts": {
    "test": "jest --watchAll"
  }
}
```

After adding --watchAll, re-execute the npm run test (or yarn test) command to make Jest automatically begin rerunning your test whenever you save changes.

For instance, suppose you realized that the additionalCalculator should allow users to add any number of digits. In that case, you can refactor your test code like so:

```
// additionCalculator.test.js
const additionCalculator = require("./additionCalculator");
describe("additionCalculator's test cases", () => {
  test("addition of 4 and 6 to equal 10", () => {
    expect(additionCalculator(4, 6)).toBe(10);
  });

  test("addition of 100, 50, 20, 45 and 30 to equal 245", () => {
    expect(additionCalculator(100, 50, 20, 45, 30)).toBe(245);
  });

  test("addition of 7 to equal 7", () => {
    expect(additionCalculator(7)).toBe(7);
  });

  test("addition of no argument provided to equal 0", () => {
```

```
  expect(additionCalculator()).toBe(0);
 });
});
```

Note that the describe() method we used in the snippet above is an optional code—it helps organize related test cases into groups.

describe() accepts two arguments:

1. A name you wish to call the test case group—for instance, "additionCalculator's test cases".
2. A function containing your test cases.

Step 13: Refactor the program

So, now that you've refactored your test code, let's do the same for the additionalCalculator program.

```
// additionCalculator.js

function additionCalculator(...numbers) {
 return numbers.reduce((sum, item) => sum + item, 0);
}

module.exports = additionCalculator;
```

Here's what we did in the snippet above:

1. The ...numbers code used JavaScript's rest operator (...) to put the function's arguments into an array.
2. The numbers.reduce((sum, item) => sum + item, 0) code used JavaScript's reduce() method to sum up all the items in the numbers array.

Step 14: Rerun the test

Once you've finished refactoring your code, rerun the test to confirm that your program still works as expected.

And that's it!