CFGdegree

FOUNDATION FULL STACK

CODE FIRST GIRLS

CODE FIRST GIRLS

SESSION 18

TESTING WITH JAVASCRIPT: PART I

SESSION 18 : TESTING WITH JAVASCRIPT

SESSION OVERVIEW

From completing this lesson, you will be able to:

+ Understand the importance of testing in software development
+ Set up Jest for JavaScript projects
+ Write basic unit tests and understand TDD

TOPIC 1: INTRODUCTION TO TESTING

TOPIC OVERVIEW // INTRODUCTION TO TESTING

1. What is testing?
2. What are the different types of testing?
3. Why is testing useful?

WHAT IS TESTING?

INTRODUCTION TO TESTING

Testing in software development is the process of **evaluating a software application or system** to ensure it meets specified requirements, works as expected, and is free from defects.

- **Validation and Verification**: Testing verifies that the software does what it's supposed to do (validation) and that it was built according to specifications and design (verification). This includes checking against requirements, design, and user expectations.
- **Quality Assurance and Reliability**: Testing aims to improve the quality of software by detecting defects early in the development process. It helps ensure that the software is reliable, stable, and performs as expected under various conditions.
- **Risk Reduction and Confidence Building**: By identifying and fixing defects before software is released to users, testing reduces the risk of failures or errors in production. This process builds confidence among stakeholders that the software is safe and fit for use.

TYPES OF TESTING

INTRODUCTION TO TESTING

There are different types of **testing** in software development, each serving a specific purpose in the software quality assurance process. Here's an overview of **three** key types:

Unit Testing

- Focuses on individual components or units of code
- Tests the smallest pieces of code (functions, methods or classes)
- Typically automated

Integration Testing

- Involves testing the interaction between multiple units to ensure they work together as expected
- Helps identify issues with data flow, communication or dependencies between units

End-to-End Testing

- Tests the entire application from start to finish
- Ensures all parts of the system work together
- Includes UI interactions, communication between different services etc.

TYPES OF TESTING

INTRODUCTION TO TESTING

Let's consider the process of building a car to illustrate these concepts

Unit Testing

- **Engine** - generates power
- **Brakes** - can stop the car
- **Headlights** - they can turn on and off

Remember, all of the above should work in **isolation** without consider other car parts.

Integration Testing

- **Engine and Transmission** - power is transferred to the transmission
- **Brakes and Wheels** - can the brakes stop the car engaging the wheels?
- **Electrical System** - Do headlights, indicators, lights work well together?

End-to-End Testing

- **Full Car Drive** - Overall performance, braking, steering
- **Safety Tests** - Airbag, seatbelts work correctly
- **Environmental Conditions** - Drive the car in different weather to see how it performs in rain, heat etc

TYPES OF TESTING

INTRODUCTION TO TESTING

Can you think of any other real-life analogies where testing is **crucial**?

CODE FIRST GIRLS

TOPIC 2: JEST

TOPIC OVERVIEW // JEST

1. What is Jest?
2. How do you set it up?
3. Writing tests in JS using Jest

WHAT IS JEST?

JEST

Jest is a popular open-source testing framework for JavaScript applications, developed by Facebook.

Jest provides a comprehensive set of features that make it easy to write, run, and manage tests. Key features of Jest include:

- **Test Isolation:** Jest runs tests in parallel, ensuring test environments are isolated to avoid side effects.
- **Snapshot Testing:** Jest allows you to create snapshots of your application's output and automatically compare them during tests.
- **Mocking and Spies:** Jest offers built-in support for mocking functions, modules, and timers, helping simulate complex dependencies during tests.

SETTING UP JEST

JEST

Let’s install and configure Jest

To install Jest do the following:

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

By default, Jest is configured to work with most JavaScript projects without additional setup. However, you might want to add a test script to your **package.json** to make running tests easier

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

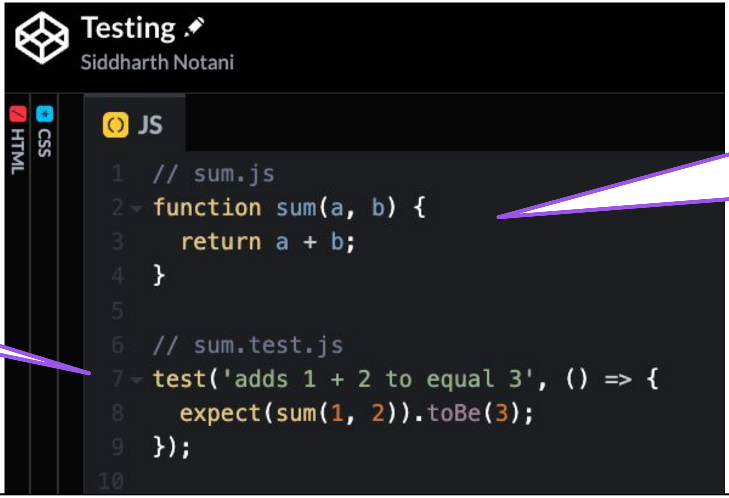
```
> npm run test
```

SETTING UP JEST

JEST

A simple test that you can write in Jest

You would usually create a directory called **tests** and inside it you would create a test file like **sum.test.js** for the example below - in there you would put **only** the test and **import the function** from where your function is



```
// sum.js
function sum(a, b) {
  return a + b;
}

// sum.test.js
test('adds 1 + 2 to equal 3', () => {
  expect(sum(1, 2)).toBe(3);
});
```

This test will check if 1 + 2 = 3

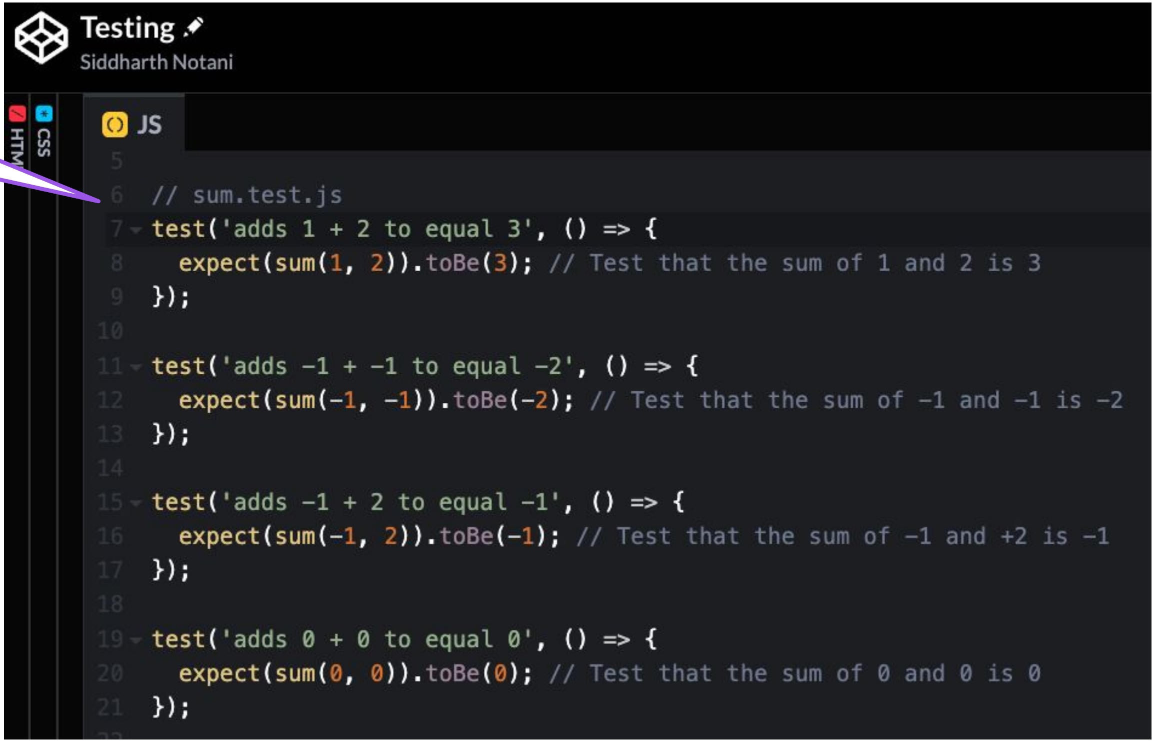
What else could you test for the sum function? Think outside the box!

WRITING TESTS IN JEST

JEST

Tip: Think of the edge cases when writing tests for a function!

Will all of these tests pass?



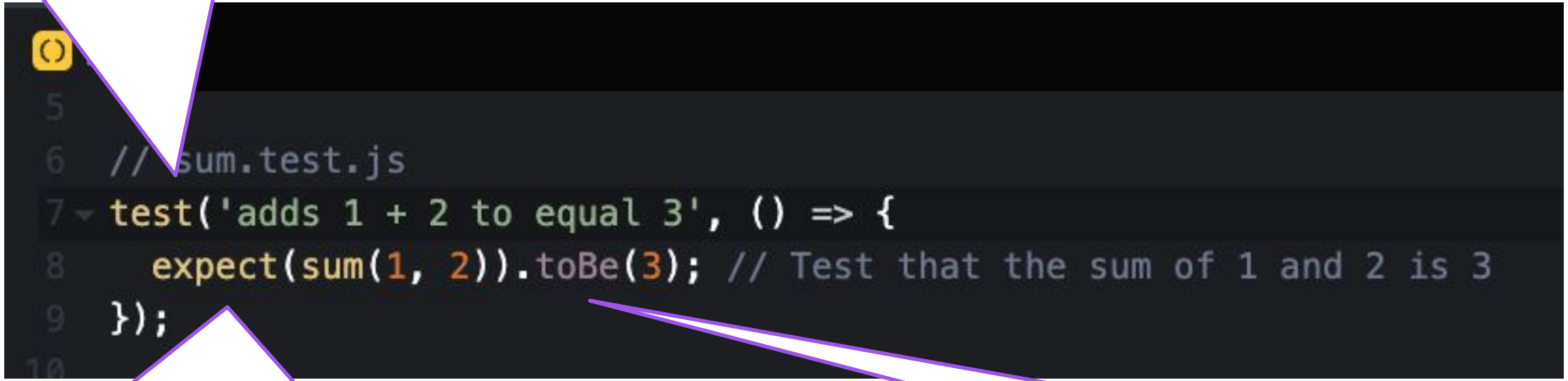
```
// sum.test.js
test('adds 1 + 2 to equal 3', () => {
  expect(sum(1, 2)).toBe(3); // Test that the sum of 1 and 2 is 3
});

test('adds -1 + -1 to equal -2', () => {
  expect(sum(-1, -1)).toBe(-2); // Test that the sum of -1 and -1 is -2
});

test('adds -1 + 2 to equal -1', () => {
  expect(sum(-1, 2)).toBe(-1); // Test that the sum of -1 and +2 is -1
});

test('adds 0 + 0 to equal 0', () => {
  expect(sum(0, 0)).toBe(0); // Test that the sum of 0 and 0 is 0
});
```

WRITING TESTS IN JEST

JEST

Let's look at what each thing in the test means

Test: defines a single test case. The first argument is the name of the test, and the second is a function containing the test logic.



```
// sum.test.js
test('adds 1 + 2 to equal 3', () => {
  expect(sum(1, 2)).toBe(3); // Test that the sum of 1 and 2 is 3
});
```

Expect: Specifies what you expect to happen in a test. It's used with matchers to define expected outcomes.

toBe: Is a **Matcher**. Checks for exact equality (like strict equality ===)

WRITING TESTS IN JEST

JEST

In Jest, matchers are used to specify expected outcomes in tests. They allow you to compare actual values to expected values and determine if a test passes or fails.

Common list of matchers (we have already seen toBe):

- **toBeTruthy** - Asserts that a value is truthy (**not** false, 0, null, undefined or empty string)
- **toBeFalsy** - Asserts that a value is falsy (false, 0, null, undefined or empty string)
- **toContain** - Checks if an array or iterable contains a specific item
- **toHaveLength**: Asserts that an array or string has a specific length.
- **toBeGreaterThan**: Asserts that a value is greater than a given number.
- toBeGreaterThanOrEqual, toBeLessThan, toBeLessThanOrEqualTo etc

WRITING TESTS IN JEST

JEST

Over to you!

Now let's write tests for a new function, multiply(a, b), which multiplies two numbers

Write at least three tests for the multiply function

Tip: Think of the edge cases and think outside the box!

CODE FIRST GIRLS

TOPIC 3: Test-Driven Development

TOPIC OVERVIEW // Test-Driven Development

1. What is Test-Driven Development?
2. What are the benefits of TDD?
3. Trying it out!

WHAT IS TDD?

TDD

Test-Driven Development (TDD) is a software development process in which you write tests for a feature or function before you implement the code.

TDD aims to create a robust, maintainable codebase by ensuring that all functionality is thoroughly tested from the start. It follows a specific cycle known as "Red-Green-Refactor."

- **Red**: Write a test that defines the desired behavior or output for a given function or feature. Since the code hasn't been implemented yet, the test should fail (i.e., turn "red").
- **Green**: Implement the minimum amount of code needed to make the test pass. This involves writing the function or feature to meet the test's expectations. When the test passes, it "turns green."
- **Refactor**: Once the test passes, refactor the code for better readability, optimization, or reusability, while ensuring that the test still passes. This step allows you to clean up the code and make improvements without changing the behavior.

BENEFITS OF TDD?

TDD

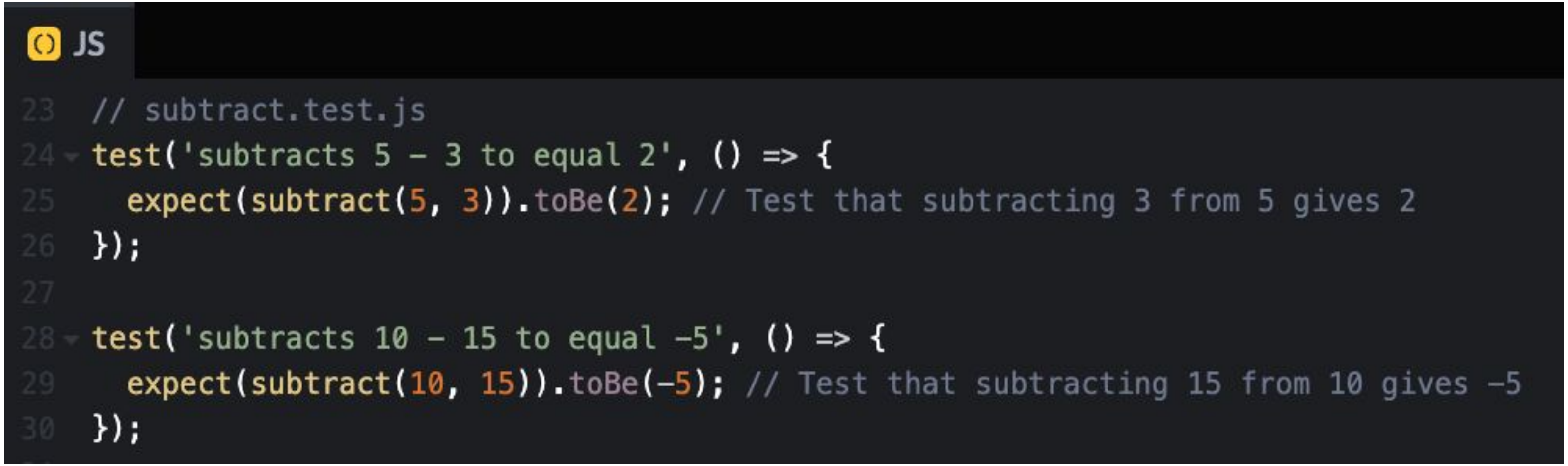
There are several benefits to using TDD in software development:

- **Improved Code Design**: Writing tests before implementing code forces you to think about the code's design and behavior from the user's perspective. This leads to cleaner, more modular code, as you're focusing on fulfilling specific requirements.
- **Fewer Bugs and Faster Debugging**: Since tests are written before the code, you catch errors early in the development process. This reduces the likelihood of bugs making it into production, leading to higher-quality software. If bugs do occur, the tests help isolate them more quickly.
- **Confidence in Refactoring**: With a comprehensive test suite, you can refactor code with confidence, knowing that the tests will catch any unintended changes in behavior. This promotes code maintainability and flexibility for future changes.
- **Documentation of Expected Behavior**: Tests act as living documentation, showing how the code is expected to behave. This can be useful for onboarding new developers or revisiting code after a long time.

EXERCISE

TDD

Now let's apply **TDD** by writing a test for a function before implementing it.
In this case, the function **subtract(a, b)** should return the result of subtracting b from a.



```js
// subtract.test.js
test('subtracts 5 - 3 to equal 2', () => {
  expect(subtract(5, 3)).toBe(2); // Test that subtracting 3 from 5 gives 2
});

test('subtracts 10 - 15 to equal -5', () => {
  expect(subtract(10, 15)).toBe(-5); // Test that subtracting 15 from 10 gives -5
});
```

Notice that the subtract function **hasn't** been implemented yet. At this stage, running the test will fail (turn "red"), indicating that the function does not exist or does not behave as expected.

EXERCISE

TDD

With the tests in place, your next step would be to **implement** the subtract function to ensure the tests pass (turn "green").

This is the essence of TDD—write tests, implement code, then refactor as needed to maintain the green state.

CODE FIRST GIRLS

Q&A

CODE FIRST GIRLS

THANK YOU