

Lesson

Monday

Intermediate JavaScript (/intermediate-javascript)

/ Test-Driven Development and Environments with JavaScript

(/intermediate-javascript/test-driven-development-and-environments-with-javascript)

/ Improving Test Reports: Adding Test Coverage Information

Text

With our test-driven development approach, we should have 100% testing coverage of our business logic. What do we mean by testing coverage? Well, if we have ten lines of code and our tests only "hit" five lines of code, then we'd have 50% coverage. If there is a function or another piece of code that's untouched by our tests, it won't be included in the test coverage.

In this lesson, we'll configure Jest to add coverage information.

## Updating .gitignore

When we configure Jest to generate a coverage report for our tests, Jest will create new output in the terminal and create a folder called coverage/. We do not want to save this folder in our remote repo,

so the first thing we need to do is update our `.gitignore` to ignore the `coverage/` directory and commit this change to our Git history.

This directory should be added to our `.gitignore` file:

#### **.gitignore**

```
...  
coverage/
```

## Adding Test Coverage Information

Jest can give us information about test coverage if we add the `--coverage` flag to the `"test"` script in our `package.json` file:

#### **package.json**

```
...  
"scripts": {  
  ...  
  "test": "jest --coverage"  
},  
...
```

Go ahead and do this now!

With the `--coverage` flag added, let's see what happens when we run our tests:

```
> shape-tracker@1.0.0 test /Users/staff/Desktop/shape-tracker
> jest --coverage
```

```
PASS __tests__/triangle.test.js
```

```
Triangle
```

- ✓ should correctly create a triangle object with three lengths (2ms)
- ✓ should correctly determine whether three lengths are not a triangle
- ✓ should correctly determine whether three lengths make an isosceles triangle (1ms)
- ✓ should correctly determine whether three lengths make a scalene triangle
- ✓ should correctly determine whether three lengths make an equilateral triangle

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	100	100	100	100	
triangle.js	100	100	100	100	

```
Test Suites: 1 passed, 1 total
```

```
Tests: 5 passed, 5 total
```

```
Snapshots: 0 total
```

```
Time: 1.281s
```

```
Ran all test suites.
```

Our tests now include a chart that shows our total test coverage. It specifies the file `triangle.js` as well as the percentage of statements, branches, functions and lines that are covered. We have 100% test coverage, which is what we should be aiming for. The final column for Uncovered Line #s is blank. This column will show all lines that aren't covered in our tests.

Here's how the test coverage looks if we comment out our test for an equilateral triangle.

```
> shape-tracker@1.0.0 test /Users/staff/Desktop/shape-tracker
> jest --coverage
```

```
PASS __tests__/triangle.test.js
```

```
Triangle
```

- ✓ should correctly create a triangle object with three lengths (3ms)
- ✓ should correctly determine whether three lengths are not a triangle
- ✓ should correctly determine whether three lengths make an isosceles triangle (1ms)
- ✓ should correctly determine whether three lengths make a scalene triangle

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	90.91	92.86	100	90.91	
triangle.js	90.91	92.86	100	90.91	13

```
Test Suites: 1 passed, 1 total
```

```
Tests: 4 passed, 4 total
```

```
Snapshots: 0 total
```

```
Time: 1.178s
```

```
Ran all test suites.
```

Now we can see that a line has been added to the `Uncovered Lines` section. Our test coverage will show us exactly which lines aren't covered in our code so we can check them and then add tests if needed. Since we are using TDD, we should always have 100% coverage, and this is a great way to confirm that our coverage is complete.

Note that the coverage chart is mostly showing green — that's because our coverage is still considered good. Many enterprise companies won't have that level of coverage for their tests. Once again, though, because we are using TDD and our projects are small, we expect to hit 100% coverage for all business logic.

Ultimately, the most important columns to pay attention to are the last two. We want `% Lines` to be 100. If it's not, we should check the line numbers in `Uncovered Line #s`.

The other columns can be a bit confusing. For instance, we have 100 for `% Funcs` because our tests have hit both of our functions, but not because they are fully tested. This column can be misleading so it's not as helpful.

The `% Stmts` refers to the percentage of statements that are covered. We have 11 total statements in our code. Generally, it will be the same as the line number, though if there are multiple statements on a line, then it could be different. In this case, we'd need to make sure we have 100% coverage on this line as well.

The `% Branch` can also be confusing. We only have four branches so how are we getting 92.86% coverage? This coverage is based on the percentage of *paths* our tests are not hitting as they go through our code. There are 14 paths through our code, 13 which are covered. Generally, this percentage number is not as useful for us.

That's why it's best to keep things simple. It should be 100% across the board — if it's not, check the `Uncovered Line #s` to see which lines aren't covered.

## The Coverage Report Can Provide False Positives and Negatives

Keep in mind that the coverage isn't foolproof. For instance, if we comment out our test for constructor properties, we'll still get 100% coverage across the board. That is because all of our other tests rely on the constructor, so our tests are hitting every line of code. We should consider this a false positive. We should always test our constructors separately, and we should always endeavor to test every *behavior* in our code.

A false negative for Jest coverage is when Jest reports partial coverage for a function that has an `if` statement (and a test for it), but intentionally no `else` statement. Well, this feels like a false negative. Let's look at an example to understand how this works, as well as three solutions. As we'll learn this "false negative" is actually by design.

Say we want to write another `Triangle` method called `Triangle.prototype.isBigTriangle()` that lets us know if we have a big triangle. Determining how big a triangle is is an arbitrary process, but we've decided that we'll know we have a big triangle if the sum of all of its sides is greater than 10. This is what our test looks like:

### **`__tests__/triangle.test.js`**

```
describe('Triangle', () => {  
  ...  
  test('should say if a triangle is big', () => {  
    const triangle = new Triangle(3,4,5);  
    expect(triangle.isBigTriangle()).toEqual('big');  
  });  
});
```

Then, we declare just the name of the method in `triangle.js`, without any body, and run `$ npm run test` to get a good fail.

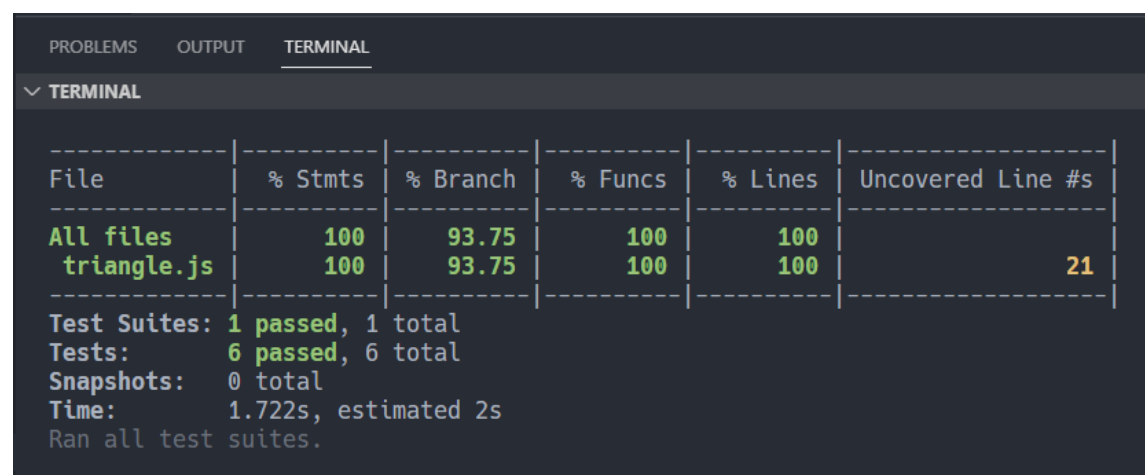
Now we're ready to write the code for

`Triangle.prototype.isBigTriangle()` to get our test to pass. Note that in the following code snippet we've added line numbers in order to better understand the Jest coverage report that we'll look at below.

#### src/triangle.js

```
17 // other Triangle code
18
19 Triangle.prototype.isBigTriangle = function(){
20   const sum = this.side1 + this.side2 + this.side3;
21   if(sum > 10){
22     return 'big';
23   }
24 };
```

With this code, I would expect my test to pass and to be done with my function. Let's run `$ npm run test` and see what we get!



The screenshot shows a terminal window with the 'TERMINAL' tab selected. It displays a table of Jest coverage results and a summary of test outcomes.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	100	93.75	100	100	
triangle.js	100	93.75	100	100	21

Test Suites: 1 passed, 1 total  
Tests: 6 passed, 6 total  
Snapshots: 0 total  
Time: 1.722s, estimated 2s  
Ran all test suites.

The last thing I would expect is for Jest to tell me that I have an Uncovered Line #. But this is also confusing. Jest is telling me that line 21 is not covered, but that's the if statement that my test is

testing for. What's going on? This seems like a false negative.

This happens because Jest is letting us know that we have not tested for the `else` path, or, what happens when the triangle is not big. The problem isn't that we're not including an `else` statement specifically, but rather what happens if the `if` statement evaluates to false. In the case of `Triangle.prototype.isBigTriangle()`, nothing happens! Well, for Jest nothing is still something, and we need to test for it. In fact, this functionality is by design, and in fact is not a false negative even though it seems like one.

One distinction to note is that the `Uncovered Line #` in the last image of the Jest coverage report was red, and this time it's yellow, which gives us a warning. It doesn't mean that we have code that is untested. It means that we have a path that's not tested for which, again, is what happens when the triangle is not big.

There's a few ways to solve this issue. If this happens to you, the first thing you should do is ask yourself, do I need an `else` statement? Perhaps you could use an `else` statement to return an error message to improve your code?

In this case, if the triangle isn't big, it could actually be helpful to include an `else` statement that returns `'little'`. Generally, it's harder to manage a function that returns either something or nothing wherever we call it. In fact, we might have to include a conditional where we call this function to determine whether there's a result, and if so, do something with it. So, for `Triangle.prototype.isBigTriangle()`, I think the best solution is adding an `else` statement and a test for that other path.

```
__tests__/triangle.test.js
```

```
describe('Triangle', () => {  
  ...  
  test('should say if a triangle is big', () => {  
    const triangle = new Triangle(3,4,5);  
    expect(triangle.isBigTriangle()).toEqual('big');  
  });  
});
```

### src/triangle.js

```
Triangle.prototype.isBigTriangle = function(){  
  const sum = this.side1 + this.side2 + this.side3;  
  if(sum > 10){  
    return 'big';  
  } else {  
    return 'little';  
  }  
};
```

After this change, we may even prefer to update the method name itself to account for this update functionality, something like:

`Triangle.prototype.isBigOrLittle()`.

However, we just don't always need an `else` statement, and we certainly shouldn't add one just to get 100% test coverage. If this is the case, you have other options. The first is to write a test for the other path. For our original `Triangle.prototype.isBigTriangle()` method, nothing happens, and we can in fact write a test for nothing with the Jest matcher `toBeUndefined()`:

```
test('should say if a triangle is little', () => {  
  const triangle = new Triangle(2,4,1);  
  expect(triangle.bigOrLittle()).toBeUndefined();  
});
```



The other solution is to disable Jest to ignore the intentionally not included `else` statement. We'd do so like this:

#### src/triangle.js

```
Triangle.prototype.isBigTriangle = function(){
  const sum = this.side1 + this.side2 + this.side3;
  /* istanbul ignore else */
  if(sum > 10){
    return 'big';
  }
};
```

We add the disable message `/* istanbul ignore else */` right before the uncovered line number that the coverage report lists (in this example, it was line 21).

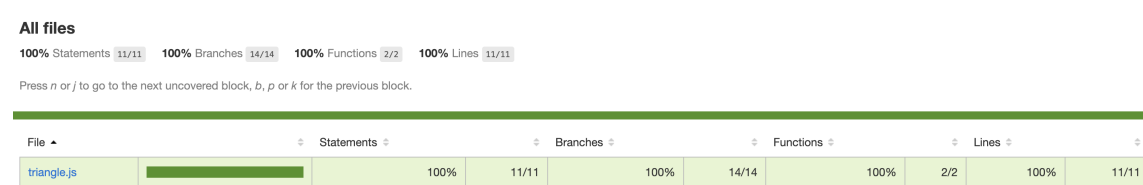
Note that Jest uses another software called Istanbul (<https://istanbul.js.org/>) to generate coverage reports, and this is why we reference `istanbul` in the disable message `/* istanbul ignore else */`.

We've covered an example of a false positive and a confusing situation of unexpected partial coverage, all to remind you that we can't assume that everything is perfect just because our coverage shows 100%. We still need to have good TDD practices in place.

However, we can and should assume that we have more testing to do if we *don't* have 100% — so the coverage report is an excellent tool for ensuring that we have at least some coverage for every line of code. If you do run into unexpected results that you can't seem to resolve or understand, reach out to your instructor for help!

## The coverage/ Directory

Finally, when we run our tests with the `--coverage` flag, Jest will generate an HTML report in a directory called `coverage`. We can go to `coverage/lcov-report/` and then open its `index.html` file in the browser if we want a nicer looking version of our tests.



As we can see here, we get a little more information as well, such as the exact number of statements and paths we have in our code.

**Note:** If you're using a version of Jest where the `coverage` directory isn't automatically generated, you'll need to update your `package.json` file:

### package.json

```
...
"scripts": {
  ...
  "test": "jest --coverage"
},
"jest": {
  "collectCoverage": true,
  "coverageReporters": ["html"]
},
...
```

We just have to pass in a configuration for Jest with `"collectCoverage"` set to `true`. Then we have to specify that the `"coverageReporters"` should be `["html"]`. (We could also specify other coverage reporters as well.)

This shows us yet another way we can update our `package.json` file to configure our environment further. Pretty neat!

In this lesson, we've looked at how Jest's coverage tools can give us more information about how thoroughly we are testing our code. While this tool can be very helpful for ensuring our tests cover every line in our business logic, they aren't a replacement for good TDD practices.

[Previous \(/intermediate-javascript/test-driven-development-and-environments-with-javascript/expanding-our-testing-tools-adding-setup-and-teardown\)](#)

[Next \(/intermediate-javascript/test-driven-development-and-environments-with-javascript/asking-and-listening\)](#)

Lesson 32 of 49

Last updated more than 3 months ago.

[disable dark mode](#)



© 2023 Epicodus (<http://www.epicodus.com/>), Inc.