npm install --save-dev jest

Let's get started by writing a test for a hypothetical function that adds two numbers. First, create a sum.js file:

function sum(a, b) {

return a + b;

}

module.exports = sum;

Then, create a file named sum.test.js. This will contain our actual test:

const sum = require('./sum');

test('adds 1 + 2 to equal 3', () => {

expect(sum(1, 2)).toBe(3);

});

Add the following section to your package.json:

{

"scripts": {

"test": "jest"

}

}

Finally, run yarn test or npm test and Jest will print this message:

PASS ./sum.test.js

✓ adds 1 + 2 to equal 3 (5ms)

You just successfully wrote your first test using Jest!

This test used expect and toBe to test that two values were exactly identical. To learn about the other things that Jest can test, see Using Matchers.

Running from command line

You can run Jest directly from the CLI (if it's globally available in your PATH, e.g. by yarn global add jest or npm install jest --global) with a variety of useful options.

Here's how to run Jest on files matching my-test, using config.json as a configuration file and display a native OS notification after the run:

jest my-test --notify --config=config.json

If you'd like to learn more about running jest through the command line, take a look at the Jest CLI Options page.

Additional Configuration

Generate a basic configuration file

Based on your project, Jest will ask you a few questions and will create a basic configuration file with a short description for each option:

• npm

• Yarn

• pnpm

npm init jest@latest

Using Babel

To use Babel, install required dependencies:

• npm

• Yarn

• pnpm

npm install --save-dev babel-jest @babel/core @babel/preset-env

Configure Babel to target your current version of Node by creating a babel.config.js file in the root of your project:

babel.config.js

module.exports = {

presets: [['@babel/preset-env', {targets: {node: 'current'}}]],

};

The ideal configuration for Babel will depend on your project. See Babel's docs for more details.

Making your Babel config jest-aware

Using webpack

Jest can be used in projects that use webpack to manage assets, styles, and compilation. webpack does offer some unique challenges over other tools. Refer to the webpack guide to get started.

Using Vite

Jest can be used in projects that use vite to serve source code over native ESM to provide some frontend tooling, vite is an opinionated tool and does offer some out-of-the box workflows. Jest is not fully supported by vite due to how the plugin system from vite works, but there are some working examples for first-class jest integration using vite-jest, since this is not fully supported, you might as well read the limitation of the vite-jest. Refer to the vite guide to get started.

Using Parcel

Jest can be used in projects that use parcel-bundler to manage assets, styles, and compilation similar to webpack. Parcel requires zero configuration. Refer to the official docs to get started.

Using TypeScript

Via babel

Jest supports TypeScript, via Babel. First, make sure you followed the instructions on using Babel above. Next, install the @babel/preset-typescript:

• npm

• Yarn

• pnpm

npm install --save-dev @babel/preset-typescript

Then add @babel/preset-typescript to the list of presets in your babel.config.js.

babel.config.js

module.exports = {

presets: [

['@babel/preset-env', {targets: {node: 'current'}}],

'@babel/preset-typescript',

],

};

However, there are some caveats to using TypeScript with Babel. Because TypeScript support in Babel is purely transpilation, Jest will not type-check your tests as they are run. If you want that, you can use ts-jest instead, or just run the TypeScript compiler tsc separately (or as part of your build process).

Via ts-jest

ts-jest is a TypeScript preprocessor with source map support for Jest that lets you use Jest to test projects written in TypeScript.

• npm

• Yarn

• pnpm

npm install --save-dev ts-jest

In order for Jest to transpile TypeScript with ts-jest, you may also need to create a configuration file.

Type definitions

There are two ways to have Jest global APIs typed for test files written in TypeScript.

You can use type definitions which ships with Jest and will update each time you update Jest. Install the @jest/globals package:

• npm

• Yarn

• pnpm

npm install --save-dev @jest/globals

And import the APIs from it:

sum.test.ts

import {describe, expect, test} from '@jest/globals';

import {sum} from './sum';

describe('sum module', () => {

test('adds 1 + 2 to equal 3', () => {

expect(sum(1, 2)).toBe(3);

});

});

TIP

See the additional usage documentation of describe.each/test.each and mock functions.

Or you may choose to install the @types/jest package. It provides types for Jest globals without a need to import them.

• npm

• Yarn

• pnpm

npm install --save-dev @types/jest

INFO

@types/jest is a third party library maintained at DefinitelyTyped, hence the latest Jest features or versions may not be covered yet. Try to match versions of Jest and @types/jest as closely as possible. For example, if you are using Jest 27.4.0 then installing 27.4.x of @types/jest is ideal.

# Using Matchers

Jest uses "matchers" to let you test values in different ways. This document will introduce some commonly used matchers. For the full list, see the [expect API doc](https://jestjs.io/docs/expect).

## Common Matchers[​](https://jestjs.io/docs/using-matchers#common-matchers)

The simplest way to test a value is with exact equality.

test('two plus two is four', () => {  
 expect(2 + 2).toBe(4);  
});

In this code, expect(2 + 2) returns an "expectation" object. You typically won't do much with these expectation objects except call matchers on them. In this code, .toBe(4) is the matcher. When Jest runs, it tracks all the failing matchers so that it can print out nice error messages for you.

toBe uses Object.is to test exact equality. If you want to check the value of an object, use toEqual:

test('object assignment', () => {  
 const data = {one: 1};  
 data['two'] = 2;  
 expect(data).toEqual({one: 1, two: 2});  
});

toEqual recursively checks every field of an object or array.

TIP

toEqual ignores object keys with undefined properties, undefined array items, array sparseness, or object type mismatch. To take these into account use toStrictEqual instead.

You can also test for the opposite of a matcher using not:

test('adding positive numbers is not zero', () => {  
 for (let a = 1; a < 10; a++) {  
 for (let b = 1; b < 10; b++) {  
 expect(a + b).not.toBe(0);  
 }  
 }  
});

## Truthiness[​](https://jestjs.io/docs/using-matchers#truthiness)

In tests, you sometimes need to distinguish between undefined, null, and false, but you sometimes do not want to treat these differently. Jest contains helpers that let you be explicit about what you want.

* toBeNull matches only null
* toBeUndefined matches only undefined
* toBeDefined is the opposite of toBeUndefined
* toBeTruthy matches anything that an if statement treats as true
* toBeFalsy matches anything that an if statement treats as false

For example:

test('null', () => {  
 const n = null;  
 expect(n).toBeNull();  
 expect(n).toBeDefined();  
 expect(n).not.toBeUndefined();  
 expect(n).not.toBeTruthy();  
 expect(n).toBeFalsy();  
});  
  
test('zero', () => {  
 const z = 0;  
 expect(z).not.toBeNull();  
 expect(z).toBeDefined();  
 expect(z).not.toBeUndefined();  
 expect(z).not.toBeTruthy();  
 expect(z).toBeFalsy();  
});

You should use the matcher that most precisely corresponds to what you want your code to be doing.

## Numbers[​](https://jestjs.io/docs/using-matchers#numbers)

Most ways of comparing numbers have matcher equivalents.

test('two plus two', () => {  
 const value = 2 + 2;  
 expect(value).toBeGreaterThan(3);  
 expect(value).toBeGreaterThanOrEqual(3.5);  
 expect(value).toBeLessThan(5);  
 expect(value).toBeLessThanOrEqual(4.5);  
  
 *// toBe and toEqual are equivalent for numbers*  
 expect(value).toBe(4);  
 expect(value).toEqual(4);  
});

For floating point equality, use toBeCloseTo instead of toEqual, because you don't want a test to depend on a tiny rounding error.

test('adding floating point numbers', () => {  
 const value = 0.1 + 0.2;  
 *//expect(value).toBe(0.3); This won't work because of rounding error*  
 expect(value).toBeCloseTo(0.3); *// This works.*  
});

## Strings[​](https://jestjs.io/docs/using-matchers#strings)

You can check strings against regular expressions with toMatch:

test('there is no I in team', () => {  
 expect('team').not.toMatch(/I/);  
});  
  
test('but there is a "stop" in Christoph', () => {  
 expect('Christoph').toMatch(/stop/);  
});

## Arrays and iterables[​](https://jestjs.io/docs/using-matchers#arrays-and-iterables)

You can check if an array or iterable contains a particular item using toContain:

const shoppingList = [  
 'diapers',  
 'kleenex',  
 'trash bags',  
 'paper towels',  
 'milk',  
];  
  
test('the shopping list has milk on it', () => {  
 expect(shoppingList).toContain('milk');  
 expect(new Set(shoppingList)).toContain('milk');  
});

## Exceptions[​](https://jestjs.io/docs/using-matchers#exceptions)

If you want to test whether a particular function throws an error when it's called, use toThrow.

function compileAndroidCode() {  
 throw new Error('you are using the wrong JDK!');  
}  
  
test('compiling android goes as expected', () => {  
 expect(() => compileAndroidCode()).toThrow();  
 expect(() => compileAndroidCode()).toThrow(Error);  
  
 *// You can also use a string that must be contained in the error message or a regexp*  
 expect(() => compileAndroidCode()).toThrow('you are using the wrong JDK');  
 expect(() => compileAndroidCode()).toThrow(/JDK/);  
  
 *// Or you can match an exact error message using a regexp like below*  
 expect(() => compileAndroidCode()).toThrow(/^you are using the wrong JDK$/); *// Test fails*  
 expect(() => compileAndroidCode()).toThrow(/^you are using the wrong JDK!$/); *// Test pass*  
});

TIP

The function that throws an exception needs to be invoked within a wrapping function otherwise the toThrow assertion will fail.

## And More[​](https://jestjs.io/docs/using-matchers#and-more)

This is just a taste. For a complete list of matchers, check out the [reference docs](https://jestjs.io/docs/expect).

Once you've learned about the matchers that are available, a good next step is to check out how Jest lets you [test asynchronous code](https://jestjs.io/docs/asynchronous).

# Testing Asynchronous Code

It's common in JavaScript for code to run asynchronously. When you have code that runs asynchronously, Jest needs to know when the code it is testing has completed, before it can move on to another test. Jest has several ways to handle this.

## Promises[​](https://jestjs.io/docs/asynchronous#promises)

Return a promise from your test, and Jest will wait for that promise to resolve. If the promise is rejected, the test will fail.

For example, let's say that fetchData returns a promise that is supposed to resolve to the string 'peanut butter'. We could test it with:

test('the data is peanut butter', () => {  
 return fetchData().then(data => {  
 expect(data).toBe('peanut butter');  
 });  
});

## Async/Await[​](https://jestjs.io/docs/asynchronous#asyncawait)

Alternatively, you can use async and await in your tests. To write an async test, use the async keyword in front of the function passed to test. For example, the same fetchData scenario can be tested with:

test('the data is peanut butter', async () => {  
 const data = await fetchData();  
 expect(data).toBe('peanut butter');  
});  
  
test('the fetch fails with an error', async () => {  
 expect.assertions(1);  
 try {  
 await fetchData();  
 } catch (e) {  
 expect(e).toMatch('error');  
 }  
});

You can combine async and await with .resolves or .rejects.

test('the data is peanut butter', async () => {  
 await expect(fetchData()).resolves.toBe('peanut butter');  
});  
  
test('the fetch fails with an error', async () => {  
 await expect(fetchData()).rejects.toMatch('error');  
});

In these cases, async and await are effectively syntactic sugar for the same logic as the promises example uses.

CAUTION

Be sure to return (or await) the promise - if you omit the return/await statement, your test will complete before the promise returned from fetchData resolves or rejects.

If you expect a promise to be rejected, use the .catch method. Make sure to add expect.assertions to verify that a certain number of assertions are called. Otherwise, a fulfilled promise would not fail the test.

test('the fetch fails with an error', () => {  
 expect.assertions(1);  
 return fetchData().catch(e => expect(e).toMatch('error'));  
});

## Callbacks[​](https://jestjs.io/docs/asynchronous#callbacks)

If you don't use promises, you can use callbacks. For example, let's say that fetchData, instead of returning a promise, expects a callback, i.e. fetches some data and calls callback(null, data) when it is complete. You want to test that this returned data is the string 'peanut butter'.

By default, Jest tests complete once they reach the end of their execution. That means this test will not work as intended:

*// Don't do this!*  
test('the data is peanut butter', () => {  
 function callback(error, data) {  
 if (error) {  
 throw error;  
 }  
 expect(data).toBe('peanut butter');  
 }  
  
 fetchData(callback);  
});

The problem is that the test will complete as soon as fetchData completes, before ever calling the callback.

There is an alternate form of test that fixes this. Instead of putting the test in a function with an empty argument, use a single argument called done. Jest will wait until the done callback is called before finishing the test.

test('the data is peanut butter', done => {  
 function callback(error, data) {  
 if (error) {  
 done(error);  
 return;  
 }  
 try {  
 expect(data).toBe('peanut butter');  
 done();  
 } catch (error) {  
 done(error);  
 }  
 }  
  
 fetchData(callback);  
});

If done() is never called, the test will fail (with timeout error), which is what you want to happen.

If the expect statement fails, it throws an error and done() is not called. If we want to see in the test log why it failed, we have to wrap expect in a try block and pass the error in the catch block to done. Otherwise, we end up with an opaque timeout error that doesn't show what value was received by expect(data).

CAUTION

Jest will throw an error, if the same test function is passed a done() callback and returns a promise. This is done as a precaution to avoid memory leaks in your tests.

## .resolves / .rejects[​](https://jestjs.io/docs/asynchronous#resolves--rejects)

You can also use the .resolves matcher in your expect statement, and Jest will wait for that promise to resolve. If the promise is rejected, the test will automatically fail.

test('the data is peanut butter', () => {  
 return expect(fetchData()).resolves.toBe('peanut butter');  
});

Be sure to return the assertion—if you omit this return statement, your test will complete before the promise returned from fetchData is resolved and then() has a chance to execute the callback.

If you expect a promise to be rejected, use the .rejects matcher. It works analogically to the .resolves matcher. If the promise is fulfilled, the test will automatically fail.

test('the fetch fails with an error', () => {  
 return expect(fetchData()).rejects.toMatch('error');  
});

None of these forms is particularly superior to the others, and you can mix and match them across a codebase or even in a single file. It just depends on which style you feel makes your tests simpler.

# Setup and Teardown

Often while writing tests you have some setup work that needs to happen before tests run, and you have some finishing work that needs to happen after tests run. Jest provides helper functions to handle this.

## Repeating Setup[​](https://jestjs.io/docs/setup-teardown#repeating-setup)

If you have some work you need to do repeatedly for many tests, you can use beforeEach and afterEach hooks.

For example, let's say that several tests interact with a database of cities. You have a method initializeCityDatabase() that must be called before each of these tests, and a method clearCityDatabase() that must be called after each of these tests. You can do this with:

beforeEach(() => {  
 initializeCityDatabase();  
});  
  
afterEach(() => {  
 clearCityDatabase();  
});  
  
test('city database has Vienna', () => {  
 expect(isCity('Vienna')).toBeTruthy();  
});  
  
test('city database has San Juan', () => {  
 expect(isCity('San Juan')).toBeTruthy();  
});

beforeEach and afterEach can handle asynchronous code in the same ways that [tests can handle asynchronous code](https://jestjs.io/docs/asynchronous) - they can either take a done parameter or return a promise. For example, if initializeCityDatabase() returned a promise that resolved when the database was initialized, we would want to return that promise:

beforeEach(() => {  
 return initializeCityDatabase();  
});

## One-Time Setup[​](https://jestjs.io/docs/setup-teardown#one-time-setup)

In some cases, you only need to do setup once, at the beginning of a file. This can be especially bothersome when the setup is asynchronous, so you can't do it inline. Jest provides beforeAll and afterAll hooks to handle this situation.

For example, if both initializeCityDatabase() and clearCityDatabase() returned promises, and the city database could be reused between tests, we could change our test code to:

beforeAll(() => {  
 return initializeCityDatabase();  
});  
  
afterAll(() => {  
 return clearCityDatabase();  
});  
  
test('city database has Vienna', () => {  
 expect(isCity('Vienna')).toBeTruthy();  
});  
  
test('city database has San Juan', () => {  
 expect(isCity('San Juan')).toBeTruthy();  
});

## Scoping[​](https://jestjs.io/docs/setup-teardown#scoping)

The top level before\* and after\* hooks apply to every test in a file. The hooks declared inside a describe block apply only to the tests within that describe block.

For example, let's say we had not just a city database, but also a food database. We could do different setup for different tests:

*// Applies to all tests in this file*  
beforeEach(() => {  
 return initializeCityDatabase();  
});  
  
test('city database has Vienna', () => {  
 expect(isCity('Vienna')).toBeTruthy();  
});  
  
test('city database has San Juan', () => {  
 expect(isCity('San Juan')).toBeTruthy();  
});  
  
describe('matching cities to foods', () => {  
 *// Applies only to tests in this describe block*  
 beforeEach(() => {  
 return initializeFoodDatabase();  
 });  
  
 test('Vienna <3 veal', () => {  
 expect(isValidCityFoodPair('Vienna', 'Wiener Schnitzel')).toBe(true);  
 });  
  
 test('San Juan <3 plantains', () => {  
 expect(isValidCityFoodPair('San Juan', 'Mofongo')).toBe(true);  
 });  
});

Note that the top-level beforeEach is executed before the beforeEach inside the describe block. It may help to illustrate the order of execution of all hooks.

beforeAll(() => console.log('1 - beforeAll'));  
afterAll(() => console.log('1 - afterAll'));  
beforeEach(() => console.log('1 - beforeEach'));  
afterEach(() => console.log('1 - afterEach'));  
  
test('', () => console.log('1 - test'));  
  
describe('Scoped / Nested block', () => {  
 beforeAll(() => console.log('2 - beforeAll'));  
 afterAll(() => console.log('2 - afterAll'));  
 beforeEach(() => console.log('2 - beforeEach'));  
 afterEach(() => console.log('2 - afterEach'));  
  
 test('', () => console.log('2 - test'));  
});  
  
*// 1 - beforeAll*  
*// 1 - beforeEach*  
*// 1 - test*  
*// 1 - afterEach*  
*// 2 - beforeAll*  
*// 1 - beforeEach*  
*// 2 - beforeEach*  
*// 2 - test*  
*// 2 - afterEach*  
*// 1 - afterEach*  
*// 2 - afterAll*  
*// 1 - afterAll*

## Order of Execution[​](https://jestjs.io/docs/setup-teardown#order-of-execution)

Jest executes all describe handlers in a test file before it executes any of the actual tests. This is another reason to do setup and teardown inside before\* and after\* handlers rather than inside the describe blocks. Once the describe blocks are complete, by default Jest runs all the tests serially in the order they were encountered in the collection phase, waiting for each to finish and be tidied up before moving on.

Consider the following illustrative test file and output:

describe('describe outer', () => {  
 console.log('describe outer-a');  
  
 describe('describe inner 1', () => {  
 console.log('describe inner 1');  
  
 test('test 1', () => console.log('test 1'));  
 });  
  
 console.log('describe outer-b');  
  
 test('test 2', () => console.log('test 2'));  
  
 describe('describe inner 2', () => {  
 console.log('describe inner 2');  
  
 test('test 3', () => console.log('test 3'));  
 });  
  
 console.log('describe outer-c');  
});  
  
*// describe outer-a*  
*// describe inner 1*  
*// describe outer-b*  
*// describe inner 2*  
*// describe outer-c*  
*// test 1*  
*// test 2*  
*// test 3*

Just like the describe and test blocks Jest calls the before\* and after\* hooks in the order of declaration. Note that the after\* hooks of the enclosing scope are called first. For example, here is how you can set up and tear down resources which depend on each other:

beforeEach(() => console.log('connection setup'));  
beforeEach(() => console.log('database setup'));  
  
afterEach(() => console.log('database teardown'));  
afterEach(() => console.log('connection teardown'));  
  
test('test 1', () => console.log('test 1'));  
  
describe('extra', () => {  
 beforeEach(() => console.log('extra database setup'));  
 afterEach(() => console.log('extra database teardown'));  
  
 test('test 2', () => console.log('test 2'));  
});  
  
*// connection setup*  
*// database setup*  
*// test 1*  
*// database teardown*  
*// connection teardown*  
  
*// connection setup*  
*// database setup*  
*// extra database setup*  
*// test 2*  
*// extra database teardown*  
*// database teardown*  
*// connection teardown*

NOTE

If you are using jasmine2 test runner, take into account that it calls the after\* hooks in the reverse order of declaration. To have identical output, the above example should be altered like this:

beforeEach(() => console.log('connection setup'));  
+ afterEach(() => console.log('connection teardown'));  
  
 beforeEach(() => console.log('database setup'));  
+ afterEach(() => console.log('database teardown'));  
  
- afterEach(() => console.log('database teardown'));  
- afterEach(() => console.log('connection teardown'));  
  
 // ...

## General Advice[​](https://jestjs.io/docs/setup-teardown#general-advice)

If a test is failing, one of the first things to check should be whether the test is failing when it's the only test that runs. To run only one test with Jest, temporarily change that test command to a test.only:

test.only('this will be the only test that runs', () => {  
 expect(true).toBe(false);  
});  
  
test('this test will not run', () => {  
 expect('A').toBe('A');  
});

If you have a test that often fails when it's run as part of a larger suite, but doesn't fail when you run it alone, it's a good bet that something from a different test is interfering with this one. You can often fix this by clearing some shared state with beforeEach. If you're not sure whether some shared state is being modified, you can also try a beforeEach that logs data.

# Mock Functions

Mock functions allow you to test the links between code by erasing the actual implementation of a function, capturing calls to the function (and the parameters passed in those calls), capturing instances of constructor functions when instantiated with new, and allowing test-time configuration of return values.

There are two ways to mock functions: Either by creating a mock function to use in test code, or writing a [manual mock](https://jestjs.io/docs/manual-mocks) to override a module dependency.

## Using a mock function[​](https://jestjs.io/docs/mock-functions#using-a-mock-function)

Let's imagine we're testing an implementation of a function forEach, which invokes a callback for each item in a supplied array.

forEach.js

export function forEach(items, callback) {  
 for (let index = 0; index < items.length; index++) {  
 callback(items[index]);  
 }  
}

To test this function, we can use a mock function, and inspect the mock's state to ensure the callback is invoked as expected.

forEach.test.js

const forEach = require('./forEach');  
  
const mockCallback = jest.fn(x => 42 + x);  
  
test('forEach mock function', () => {  
 forEach([0, 1], mockCallback);  
  
 *// The mock function was called twice*  
 expect(mockCallback.mock.calls).toHaveLength(2);  
  
 *// The first argument of the first call to the function was 0*  
 expect(mockCallback.mock.calls[0][0]).toBe(0);  
  
 *// The first argument of the second call to the function was 1*  
 expect(mockCallback.mock.calls[1][0]).toBe(1);  
  
 *// The return value of the first call to the function was 42*  
 expect(mockCallback.mock.results[0].value).toBe(42);  
});

## .mock property[​](https://jestjs.io/docs/mock-functions#mock-property)

All mock functions have this special .mock property, which is where data about how the function has been called and what the function returned is kept. The .mock property also tracks the value of this for each call, so it is possible to inspect this as well:

const myMock1 = jest.fn();  
const a = new myMock1();  
console.log(myMock1.mock.instances);  
*// > [ <a> ]*  
  
const myMock2 = jest.fn();  
const b = {};  
const bound = myMock2.bind(b);  
bound();  
console.log(myMock2.mock.contexts);  
*// > [ <b> ]*

These mock members are very useful in tests to assert how these functions get called, instantiated, or what they returned:

*// The function was called exactly once*  
expect(someMockFunction.mock.calls).toHaveLength(1);  
  
*// The first arg of the first call to the function was 'first arg'*  
expect(someMockFunction.mock.calls[0][0]).toBe('first arg');  
  
*// The second arg of the first call to the function was 'second arg'*  
expect(someMockFunction.mock.calls[0][1]).toBe('second arg');  
  
*// The return value of the first call to the function was 'return value'*  
expect(someMockFunction.mock.results[0].value).toBe('return value');  
  
*// The function was called with a certain `this` context: the `element` object.*  
expect(someMockFunction.mock.contexts[0]).toBe(element);  
  
*// This function was instantiated exactly twice*  
expect(someMockFunction.mock.instances.length).toBe(2);  
  
*// The object returned by the first instantiation of this function*  
*// had a `name` property whose value was set to 'test'*  
expect(someMockFunction.mock.instances[0].name).toBe('test');  
  
*// The first argument of the last call to the function was 'test'*  
expect(someMockFunction.mock.lastCall[0]).toBe('test');

## Mock Return Values[​](https://jestjs.io/docs/mock-functions#mock-return-values)

Mock functions can also be used to inject test values into your code during a test:

const myMock = jest.fn();  
console.log(myMock());  
*// > undefined*  
  
myMock.mockReturnValueOnce(10).mockReturnValueOnce('x').mockReturnValue(true);  
  
console.log(myMock(), myMock(), myMock(), myMock());  
*// > 10, 'x', true, true*

Mock functions are also very effective in code that uses a functional continuation-passing style. Code written in this style helps avoid the need for complicated stubs that recreate the behavior of the real component they're standing in for, in favor of injecting values directly into the test right before they're used.

const filterTestFn = jest.fn();  
  
*// Make the mock return `true` for the first call,*  
*// and `false` for the second call*  
filterTestFn.mockReturnValueOnce(true).mockReturnValueOnce(false);  
  
const result = [11, 12].filter(num => filterTestFn(num));  
  
console.log(result);  
*// > [11]*  
console.log(filterTestFn.mock.calls[0][0]); *// 11*  
console.log(filterTestFn.mock.calls[1][0]); *// 12*

Most real-world examples actually involve getting ahold of a mock function on a dependent component and configuring that, but the technique is the same. In these cases, try to avoid the temptation to implement logic inside of any function that's not directly being tested.

## Mocking Modules[​](https://jestjs.io/docs/mock-functions#mocking-modules)

Suppose we have a class that fetches users from our API. The class uses [axios](https://github.com/axios/axios) to call the API then returns the data attribute which contains all the users:

users.js

import axios from 'axios';  
  
class Users {  
 static all() {  
 return axios.get('/users.json').then(resp => resp.data);  
 }  
}  
  
export default Users;

Now, in order to test this method without actually hitting the API (and thus creating slow and fragile tests), we can use the jest.mock(...) function to automatically mock the axios module.

Once we mock the module we can provide a mockResolvedValue for .get that returns the data we want our test to assert against. In effect, we are saying that we want axios.get('/users.json') to return a fake response.

users.test.js

import axios from 'axios';  
import Users from './users';  
  
jest.mock('axios');  
  
test('should fetch users', () => {  
 const users = [{name: 'Bob'}];  
 const resp = {data: users};  
 axios.get.mockResolvedValue(resp);  
  
 *// or you could use the following depending on your use case:*  
 *// axios.get.mockImplementation(() => Promise.resolve(resp))*  
  
 return Users.all().then(data => expect(data).toEqual(users));  
});

## Mocking Partials[​](https://jestjs.io/docs/mock-functions#mocking-partials)

Subsets of a module can be mocked and the rest of the module can keep their actual implementation:

foo-bar-baz.js

export const foo = 'foo';  
export const bar = () => 'bar';  
export default () => 'baz';

*//test.js*  
import defaultExport, {bar, foo} from '../foo-bar-baz';  
  
jest.mock('../foo-bar-baz', () => {  
 const originalModule = jest.requireActual('../foo-bar-baz');  
  
 *//Mock the default export and named export 'foo'*  
 return {  
 \_\_esModule: true,  
 ...originalModule,  
 default: jest.fn(() => 'mocked baz'),  
 foo: 'mocked foo',  
 };  
});  
  
test('should do a partial mock', () => {  
 const defaultExportResult = defaultExport();  
 expect(defaultExportResult).toBe('mocked baz');  
 expect(defaultExport).toHaveBeenCalled();  
  
 expect(foo).toBe('mocked foo');  
 expect(bar()).toBe('bar');  
});

## Mock Implementations[​](https://jestjs.io/docs/mock-functions#mock-implementations)

Still, there are cases where it's useful to go beyond the ability to specify return values and full-on replace the implementation of a mock function. This can be done with jest.fn or the mockImplementationOnce method on mock functions.

const myMockFn = jest.fn(cb => cb(null, true));  
  
myMockFn((err, val) => console.log(val));  
*// > true*

The mockImplementation method is useful when you need to define the default implementation of a mock function that is created from another module:

foo.js

module.exports = function () {  
 *// some implementation;*  
};

test.js

jest.mock('../foo'); *// this happens automatically with automocking*  
const foo = require('../foo');  
  
*// foo is a mock function*  
foo.mockImplementation(() => 42);  
foo();  
*// > 42*

When you need to recreate a complex behavior of a mock function such that multiple function calls produce different results, use the mockImplementationOnce method:

const myMockFn = jest  
 .fn()  
 .mockImplementationOnce(cb => cb(null, true))  
 .mockImplementationOnce(cb => cb(null, false));  
  
myMockFn((err, val) => console.log(val));  
*// > true*  
  
myMockFn((err, val) => console.log(val));  
*// > false*

When the mocked function runs out of implementations defined with mockImplementationOnce, it will execute the default implementation set with jest.fn (if it is defined):

const myMockFn = jest  
 .fn(() => 'default')  
 .mockImplementationOnce(() => 'first call')  
 .mockImplementationOnce(() => 'second call');  
  
console.log(myMockFn(), myMockFn(), myMockFn(), myMockFn());  
*// > 'first call', 'second call', 'default', 'default'*

For cases where we have methods that are typically chained (and thus always need to return this), we have a sugary API to simplify this in the form of a .mockReturnThis() function that also sits on all mocks:

const myObj = {  
 myMethod: jest.fn().mockReturnThis(),  
};  
  
*// is the same as*  
  
const otherObj = {  
 myMethod: jest.fn(function () {  
 return this;  
 }),  
};

## Mock Names[​](https://jestjs.io/docs/mock-functions#mock-names)

You can optionally provide a name for your mock functions, which will be displayed instead of 'jest.fn()' in the test error output. Use [.mockName()](https://jestjs.io/docs/mock-function-api#mockfnmocknamename) if you want to be able to quickly identify the mock function reporting an error in your test output.

const myMockFn = jest  
 .fn()  
 .mockReturnValue('default')  
 .mockImplementation(scalar => 42 + scalar)  
 .mockName('add42');

## Custom Matchers[​](https://jestjs.io/docs/mock-functions#custom-matchers)

Finally, in order to make it less demanding to assert how mock functions have been called, we've added some custom matcher functions for you:

*// The mock function was called at least once*  
expect(mockFunc).toHaveBeenCalled();  
  
*// The mock function was called at least once with the specified args*  
expect(mockFunc).toHaveBeenCalledWith(arg1, arg2);  
  
*// The last call to the mock function was called with the specified args*  
expect(mockFunc).toHaveBeenLastCalledWith(arg1, arg2);  
  
*// All calls and the name of the mock is written as a snapshot*  
expect(mockFunc).toMatchSnapshot();

These matchers are sugar for common forms of inspecting the .mock property. You can always do this manually yourself if that's more to your taste or if you need to do something more specific:

*// The mock function was called at least once*  
expect(mockFunc.mock.calls.length).toBeGreaterThan(0);  
  
*// The mock function was called at least once with the specified args*  
expect(mockFunc.mock.calls).toContainEqual([arg1, arg2]);  
  
*// The last call to the mock function was called with the specified args*  
expect(mockFunc.mock.calls[mockFunc.mock.calls.length - 1]).toEqual([  
 arg1,  
 arg2,  
]);  
  
*// The first arg of the last call to the mock function was `42`*  
*// (note that there is no sugar helper for this specific of an assertion)*  
expect(mockFunc.mock.calls[mockFunc.mock.calls.length - 1][0]).toBe(42);  
  
*// A snapshot will check that a mock was invoked the same number of times,*  
*// in the same order, with the same arguments. It will also assert on the name.*  
expect(mockFunc.mock.calls).toEqual([[arg1, arg2]]);  
expect(mockFunc.getMockName()).toBe('a mock name');

# Expect

When you're writing tests, you often need to check that values meet certain conditions. expect gives you access to a number of "matchers" that let you validate different things.

TIP

For additional Jest matchers maintained by the Jest Community check out [jest-extended](https://github.com/jest-community/jest-extended).

INFO

The TypeScript examples from this page will only work as documented if you explicitly import Jest APIs:

import {expect, jest, test} from '@jest/globals';

Consult the [Getting Started](https://jestjs.io/docs/getting-started#using-typescript) guide for details on how to setup Jest with TypeScript.

## Reference[​](https://jestjs.io/docs/expect#reference)

* [Expect](https://jestjs.io/docs/expect#expect)
  + [expect(value)](https://jestjs.io/docs/expect#expectvalue)
* [Modifiers](https://jestjs.io/docs/expect#modifiers)
  + [.not](https://jestjs.io/docs/expect#not)
  + [.resolves](https://jestjs.io/docs/expect#resolves)
  + [.rejects](https://jestjs.io/docs/expect#rejects)
* [Matchers](https://jestjs.io/docs/expect#matchers)
  + [.toBe(value)](https://jestjs.io/docs/expect#tobevalue)
  + [.toHaveBeenCalled()](https://jestjs.io/docs/expect#tohavebeencalled)
  + [.toHaveBeenCalledTimes(number)](https://jestjs.io/docs/expect#tohavebeencalledtimesnumber)
  + [.toHaveBeenCalledWith(arg1, arg2, ...)](https://jestjs.io/docs/expect#tohavebeencalledwitharg1-arg2-)
  + [.toHaveBeenLastCalledWith(arg1, arg2, ...)](https://jestjs.io/docs/expect#tohavebeenlastcalledwitharg1-arg2-)
  + [.toHaveBeenNthCalledWith(nthCall, arg1, arg2, ....)](https://jestjs.io/docs/expect#tohavebeennthcalledwithnthcall-arg1-arg2-)
  + [.toHaveReturned()](https://jestjs.io/docs/expect#tohavereturned)
  + [.toHaveReturnedTimes(number)](https://jestjs.io/docs/expect#tohavereturnedtimesnumber)
  + [.toHaveReturnedWith(value)](https://jestjs.io/docs/expect#tohavereturnedwithvalue)
  + [.toHaveLastReturnedWith(value)](https://jestjs.io/docs/expect#tohavelastreturnedwithvalue)
  + [.toHaveNthReturnedWith(nthCall, value)](https://jestjs.io/docs/expect#tohaventhreturnedwithnthcall-value)
  + [.toHaveLength(number)](https://jestjs.io/docs/expect#tohavelengthnumber)
  + [.toHaveProperty(keyPath, value?)](https://jestjs.io/docs/expect#tohavepropertykeypath-value)
  + [.toBeCloseTo(number, numDigits?)](https://jestjs.io/docs/expect#tobeclosetonumber-numdigits)
  + [.toBeDefined()](https://jestjs.io/docs/expect#tobedefined)
  + [.toBeFalsy()](https://jestjs.io/docs/expect#tobefalsy)
  + [.toBeGreaterThan(number | bigint)](https://jestjs.io/docs/expect#tobegreaterthannumber--bigint)
  + [.toBeGreaterThanOrEqual(number | bigint)](https://jestjs.io/docs/expect#tobegreaterthanorequalnumber--bigint)
  + [.toBeLessThan(number | bigint)](https://jestjs.io/docs/expect#tobelessthannumber--bigint)
  + [.toBeLessThanOrEqual(number | bigint)](https://jestjs.io/docs/expect#tobelessthanorequalnumber--bigint)
  + [.toBeInstanceOf(Class)](https://jestjs.io/docs/expect#tobeinstanceofclass)
  + [.toBeNull()](https://jestjs.io/docs/expect#tobenull)
  + [.toBeTruthy()](https://jestjs.io/docs/expect#tobetruthy)
  + [.toBeUndefined()](https://jestjs.io/docs/expect#tobeundefined)
  + [.toBeNaN()](https://jestjs.io/docs/expect#tobenan)
  + [.toContain(item)](https://jestjs.io/docs/expect#tocontainitem)
  + [.toContainEqual(item)](https://jestjs.io/docs/expect#tocontainequalitem)
  + [.toEqual(value)](https://jestjs.io/docs/expect#toequalvalue)
  + [.toMatch(regexp | string)](https://jestjs.io/docs/expect#tomatchregexp--string)
  + [.toMatchObject(object)](https://jestjs.io/docs/expect#tomatchobjectobject)
  + [.toMatchSnapshot(propertyMatchers?, hint?)](https://jestjs.io/docs/expect#tomatchsnapshotpropertymatchers-hint)
  + [.toMatchInlineSnapshot(propertyMatchers?, inlineSnapshot)](https://jestjs.io/docs/expect#tomatchinlinesnapshotpropertymatchers-inlinesnapshot)
  + [.toStrictEqual(value)](https://jestjs.io/docs/expect#tostrictequalvalue)
  + [.toThrow(error?)](https://jestjs.io/docs/expect#tothrowerror)
  + [.toThrowErrorMatchingSnapshot(hint?)](https://jestjs.io/docs/expect#tothrowerrormatchingsnapshothint)
  + [.toThrowErrorMatchingInlineSnapshot(inlineSnapshot)](https://jestjs.io/docs/expect#tothrowerrormatchinginlinesnapshotinlinesnapshot)
* [Asymmetric Matchers](https://jestjs.io/docs/expect#asymmetric-matchers)
  + [expect.anything()](https://jestjs.io/docs/expect#expectanything)
  + [expect.any(constructor)](https://jestjs.io/docs/expect#expectanyconstructor)
  + [expect.arrayContaining(array)](https://jestjs.io/docs/expect#expectarraycontainingarray)
  + [expect.not.arrayContaining(array)](https://jestjs.io/docs/expect#expectnotarraycontainingarray)
  + [expect.closeTo(number, numDigits?)](https://jestjs.io/docs/expect#expectclosetonumber-numdigits)
  + [expect.objectContaining(object)](https://jestjs.io/docs/expect#expectobjectcontainingobject)
  + [expect.not.objectContaining(object)](https://jestjs.io/docs/expect#expectnotobjectcontainingobject)
  + [expect.stringContaining(string)](https://jestjs.io/docs/expect#expectstringcontainingstring)
  + [expect.not.stringContaining(string)](https://jestjs.io/docs/expect#expectnotstringcontainingstring)
  + [expect.stringMatching(string | regexp)](https://jestjs.io/docs/expect#expectstringmatchingstring--regexp)
  + [expect.not.stringMatching(string | regexp)](https://jestjs.io/docs/expect#expectnotstringmatchingstring--regexp)
* [Assertion Count](https://jestjs.io/docs/expect#assertion-count)
  + [expect.assertions(number)](https://jestjs.io/docs/expect#expectassertionsnumber)
  + [expect.hasAssertions()](https://jestjs.io/docs/expect#expecthasassertions)
* [Extend Utilities](https://jestjs.io/docs/expect#extend-utilities)
  + [expect.addEqualityTesters(testers)](https://jestjs.io/docs/expect#expectaddequalitytesterstesters)
  + [expect.addSnapshotSerializer(serializer)](https://jestjs.io/docs/expect#expectaddsnapshotserializerserializer)
  + [expect.extend(matchers)](https://jestjs.io/docs/expect#expectextendmatchers)

## Expect[​](https://jestjs.io/docs/expect#expect)

### expect(value)[​](https://jestjs.io/docs/expect#expectvalue)

The expect function is used every time you want to test a value. You will rarely call expect by itself. Instead, you will use expect along with a "matcher" function to assert something about a value.

It's easier to understand this with an example. Let's say you have a method bestLaCroixFlavor() which is supposed to return the string 'grapefruit'. Here's how you would test that:

test('the best flavor is grapefruit', () => {  
 expect(bestLaCroixFlavor()).toBe('grapefruit');  
});

In this case, toBe is the matcher function. There are a lot of different matcher functions, documented below, to help you test different things.

The argument to expect should be the value that your code produces, and any argument to the matcher should be the correct value. If you mix them up, your tests will still work, but the error messages on failing tests will look strange.

## Modifiers[​](https://jestjs.io/docs/expect#modifiers)

### .not[​](https://jestjs.io/docs/expect#not)

If you know how to test something, .not lets you test its opposite. For example, this code tests that the best La Croix flavor is not coconut:

test('the best flavor is not coconut', () => {  
 expect(bestLaCroixFlavor()).not.toBe('coconut');  
});

### .resolves[​](https://jestjs.io/docs/expect#resolves)

Use resolves to unwrap the value of a fulfilled promise so any other matcher can be chained. If the promise is rejected the assertion fails.

For example, this code tests that the promise resolves and that the resulting value is 'lemon':

test('resolves to lemon', () => {  
 *// make sure to add a return statement*  
 return expect(Promise.resolve('lemon')).resolves.toBe('lemon');  
});

NOTE

Since you are still testing promises, the test is still asynchronous. Hence, you will need to [tell Jest to wait](https://jestjs.io/docs/asynchronous#promises) by returning the unwrapped assertion.

Alternatively, you can use async/await in combination with .resolves:

test('resolves to lemon', async () => {  
 await expect(Promise.resolve('lemon')).resolves.toBe('lemon');  
 await expect(Promise.resolve('lemon')).resolves.not.toBe('octopus');  
});

### .rejects[​](https://jestjs.io/docs/expect#rejects)

Use .rejects to unwrap the reason of a rejected promise so any other matcher can be chained. If the promise is fulfilled the assertion fails.

For example, this code tests that the promise rejects with reason 'octopus':

test('rejects to octopus', () => {  
 *// make sure to add a return statement*  
 return expect(Promise.reject(new Error('octopus'))).rejects.toThrow(  
 'octopus',  
 );  
});

NOTE

Since you are still testing promises, the test is still asynchronous. Hence, you will need to [tell Jest to wait](https://jestjs.io/docs/asynchronous#promises) by returning the unwrapped assertion.

Alternatively, you can use async/await in combination with .rejects.

test('rejects to octopus', async () => {  
 await expect(Promise.reject(new Error('octopus'))).rejects.toThrow('octopus');  
});

## Matchers[​](https://jestjs.io/docs/expect#matchers)

### .toBe(value)[​](https://jestjs.io/docs/expect#tobevalue)

Use .toBe to compare primitive values or to check referential identity of object instances. It calls Object.is to compare values, which is even better for testing than === strict equality operator.

For example, this code will validate some properties of the can object:

const can = {  
 name: 'pamplemousse',  
 ounces: 12,  
};  
  
describe('the can', () => {  
 test('has 12 ounces', () => {  
 expect(can.ounces).toBe(12);  
 });  
  
 test('has a sophisticated name', () => {  
 expect(can.name).toBe('pamplemousse');  
 });  
});

Don't use .toBe with floating-point numbers. For example, due to rounding, in JavaScript 0.2 + 0.1 is not strictly equal to 0.3. If you have floating point numbers, try .toBeCloseTo instead.

Although the .toBe matcher **checks** referential identity, it **reports** a deep comparison of values if the assertion fails. If differences between properties do not help you to understand why a test fails, especially if the report is large, then you might move the comparison into the expect function. For example, to assert whether or not elements are the same instance:

* rewrite expect(received).toBe(expected) as expect(Object.is(received, expected)).toBe(true)
* rewrite expect(received).not.toBe(expected) as expect(Object.is(received, expected)).toBe(false)

### .toHaveBeenCalled()[​](https://jestjs.io/docs/expect#tohavebeencalled)

Also under the alias: .toBeCalled()

Use .toHaveBeenCalledWith to ensure that a mock function was called with specific arguments. The arguments are checked with the same algorithm that .toEqual uses.

For example, let's say you have a drinkAll(drink, flavour) function that takes a drink function and applies it to all available beverages. You might want to check that drink gets called for 'lemon', but not for 'octopus', because 'octopus' flavour is really weird and why would anything be octopus-flavoured? You can do that with this test suite:

function drinkAll(callback, flavour) {  
 if (flavour !== 'octopus') {  
 callback(flavour);  
 }  
}  
  
describe('drinkAll', () => {  
 test('drinks something lemon-flavoured', () => {  
 const drink = jest.fn();  
 drinkAll(drink, 'lemon');  
 expect(drink).toHaveBeenCalled();  
 });  
  
 test('does not drink something octopus-flavoured', () => {  
 const drink = jest.fn();  
 drinkAll(drink, 'octopus');  
 expect(drink).not.toHaveBeenCalled();  
 });  
});

### .toHaveBeenCalledTimes(number)[​](https://jestjs.io/docs/expect#tohavebeencalledtimesnumber)

Also under the alias: .toBeCalledTimes(number)

Use .toHaveBeenCalledTimes to ensure that a mock function got called exact number of times.

For example, let's say you have a drinkEach(drink, Array<flavor>) function that takes a drink function and applies it to array of passed beverages. You might want to check that drink function was called exact number of times. You can do that with this test suite:

test('drinkEach drinks each drink', () => {  
 const drink = jest.fn();  
 drinkEach(drink, ['lemon', 'octopus']);  
 expect(drink).toHaveBeenCalledTimes(2);  
});

### .toHaveBeenCalledWith(arg1, arg2, ...)[​](https://jestjs.io/docs/expect#tohavebeencalledwitharg1-arg2-)

Also under the alias: .toBeCalledWith()

Use .toHaveBeenCalledWith to ensure that a mock function was called with specific arguments. The arguments are checked with the same algorithm that .toEqual uses.

For example, let's say that you can register a beverage with a register function, and applyToAll(f) should apply the function f to all registered beverages. To make sure this works, you could write:

test('registration applies correctly to orange La Croix', () => {  
 const beverage = new LaCroix('orange');  
 register(beverage);  
 const f = jest.fn();  
 applyToAll(f);  
 expect(f).toHaveBeenCalledWith(beverage);  
});

### .toHaveBeenLastCalledWith(arg1, arg2, ...)[​](https://jestjs.io/docs/expect#tohavebeenlastcalledwitharg1-arg2-)

Also under the alias: .lastCalledWith(arg1, arg2, ...)

If you have a mock function, you can use .toHaveBeenLastCalledWith to test what arguments it was last called with. For example, let's say you have a applyToAllFlavors(f) function that applies f to a bunch of flavors, and you want to ensure that when you call it, the last flavor it operates on is 'mango'. You can write:

test('applying to all flavors does mango last', () => {  
 const drink = jest.fn();  
 applyToAllFlavors(drink);  
 expect(drink).toHaveBeenLastCalledWith('mango');  
});

### .toHaveBeenNthCalledWith(nthCall, arg1, arg2, ....)[​](https://jestjs.io/docs/expect#tohavebeennthcalledwithnthcall-arg1-arg2-)

Also under the alias: .nthCalledWith(nthCall, arg1, arg2, ...)

If you have a mock function, you can use .toHaveBeenNthCalledWith to test what arguments it was nth called with. For example, let's say you have a drinkEach(drink, Array<flavor>) function that applies f to a bunch of flavors, and you want to ensure that when you call it, the first flavor it operates on is 'lemon' and the second one is 'octopus'. You can write:

test('drinkEach drinks each drink', () => {  
 const drink = jest.fn();  
 drinkEach(drink, ['lemon', 'octopus']);  
 expect(drink).toHaveBeenNthCalledWith(1, 'lemon');  
 expect(drink).toHaveBeenNthCalledWith(2, 'octopus');  
});

NOTE

The nth argument must be positive integer starting from 1.

### .toHaveReturned()[​](https://jestjs.io/docs/expect#tohavereturned)

Also under the alias: .toReturn()

If you have a mock function, you can use .toHaveReturned to test that the mock function successfully returned (i.e., did not throw an error) at least one time. For example, let's say you have a mock drink that returns true. You can write:

test('drinks returns', () => {  
 const drink = jest.fn(() => true);  
  
 drink();  
  
 expect(drink).toHaveReturned();  
});

### .toHaveReturnedTimes(number)[​](https://jestjs.io/docs/expect#tohavereturnedtimesnumber)

Also under the alias: .toReturnTimes(number)

Use .toHaveReturnedTimes to ensure that a mock function returned successfully (i.e., did not throw an error) an exact number of times. Any calls to the mock function that throw an error are not counted toward the number of times the function returned.

For example, let's say you have a mock drink that returns true. You can write:

test('drink returns twice', () => {  
 const drink = jest.fn(() => true);  
  
 drink();  
 drink();  
  
 expect(drink).toHaveReturnedTimes(2);  
});

### .toHaveReturnedWith(value)[​](https://jestjs.io/docs/expect#tohavereturnedwithvalue)

Also under the alias: .toReturnWith(value)

Use .toHaveReturnedWith to ensure that a mock function returned a specific value.

For example, let's say you have a mock drink that returns the name of the beverage that was consumed. You can write:

test('drink returns La Croix', () => {  
 const beverage = {name: 'La Croix'};  
 const drink = jest.fn(beverage => beverage.name);  
  
 drink(beverage);  
  
 expect(drink).toHaveReturnedWith('La Croix');  
});

### .toHaveLastReturnedWith(value)[​](https://jestjs.io/docs/expect#tohavelastreturnedwithvalue)

Also under the alias: .lastReturnedWith(value)

Use .toHaveLastReturnedWith to test the specific value that a mock function last returned. If the last call to the mock function threw an error, then this matcher will fail no matter what value you provided as the expected return value.

For example, let's say you have a mock drink that returns the name of the beverage that was consumed. You can write:

test('drink returns La Croix (Orange) last', () => {  
 const beverage1 = {name: 'La Croix (Lemon)'};  
 const beverage2 = {name: 'La Croix (Orange)'};  
 const drink = jest.fn(beverage => beverage.name);  
  
 drink(beverage1);  
 drink(beverage2);  
  
 expect(drink).toHaveLastReturnedWith('La Croix (Orange)');  
});

### .toHaveNthReturnedWith(nthCall, value)[​](https://jestjs.io/docs/expect#tohaventhreturnedwithnthcall-value)

Also under the alias: .nthReturnedWith(nthCall, value)

Use .toHaveNthReturnedWith to test the specific value that a mock function returned for the nth call. If the nth call to the mock function threw an error, then this matcher will fail no matter what value you provided as the expected return value.

For example, let's say you have a mock drink that returns the name of the beverage that was consumed. You can write:

test('drink returns expected nth calls', () => {  
 const beverage1 = {name: 'La Croix (Lemon)'};  
 const beverage2 = {name: 'La Croix (Orange)'};  
 const drink = jest.fn(beverage => beverage.name);  
  
 drink(beverage1);  
 drink(beverage2);  
  
 expect(drink).toHaveNthReturnedWith(1, 'La Croix (Lemon)');  
 expect(drink).toHaveNthReturnedWith(2, 'La Croix (Orange)');  
});

NOTE

The nth argument must be positive integer starting from 1.

### .toHaveLength(number)[​](https://jestjs.io/docs/expect#tohavelengthnumber)

Use .toHaveLength to check that an object has a .length property and it is set to a certain numeric value.

This is especially useful for checking arrays or strings size.

expect([1, 2, 3]).toHaveLength(3);  
expect('abc').toHaveLength(3);  
expect('').not.toHaveLength(5);

### .toHaveProperty(keyPath, value?)[​](https://jestjs.io/docs/expect#tohavepropertykeypath-value)

Use .toHaveProperty to check if property at provided reference keyPath exists for an object. For checking deeply nested properties in an object you may use [dot notation](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Property_accessors) or an array containing the keyPath for deep references.

You can provide an optional value argument to compare the received property value (recursively for all properties of object instances, also known as deep equality, like the toEqual matcher).

The following example contains a houseForSale object with nested properties. We are using toHaveProperty to check for the existence and values of various properties in the object.

*// Object containing house features to be tested*  
const houseForSale = {  
 bath: true,  
 bedrooms: 4,  
 kitchen: {  
 amenities: ['oven', 'stove', 'washer'],  
 area: 20,  
 wallColor: 'white',  
 'nice.oven': true,  
 },  
 livingroom: {  
 amenities: [  
 {  
 couch: [  
 ['large', {dimensions: [20, 20]}],  
 ['small', {dimensions: [10, 10]}],  
 ],  
 },  
 ],  
 },  
 'ceiling.height': 2,  
};  
  
test('this house has my desired features', () => {  
 *// Example Referencing*  
 expect(houseForSale).toHaveProperty('bath');  
 expect(houseForSale).toHaveProperty('bedrooms', 4);  
  
 expect(houseForSale).not.toHaveProperty('pool');  
  
 *// Deep referencing using dot notation*  
 expect(houseForSale).toHaveProperty('kitchen.area', 20);  
 expect(houseForSale).toHaveProperty('kitchen.amenities', [  
 'oven',  
 'stove',  
 'washer',  
 ]);  
  
 expect(houseForSale).not.toHaveProperty('kitchen.open');  
  
 *// Deep referencing using an array containing the keyPath*  
 expect(houseForSale).toHaveProperty(['kitchen', 'area'], 20);  
 expect(houseForSale).toHaveProperty(  
 ['kitchen', 'amenities'],  
 ['oven', 'stove', 'washer'],  
 );  
 expect(houseForSale).toHaveProperty(['kitchen', 'amenities', 0], 'oven');  
 expect(houseForSale).toHaveProperty(  
 'livingroom.amenities[0].couch[0][1].dimensions[0]',  
 20,  
 );  
 expect(houseForSale).toHaveProperty(['kitchen', 'nice.oven']);  
 expect(houseForSale).not.toHaveProperty(['kitchen', 'open']);  
  
 *// Referencing keys with dot in the key itself*  
 expect(houseForSale).toHaveProperty(['ceiling.height'], 'tall');  
});

### .toBeCloseTo(number, numDigits?)[​](https://jestjs.io/docs/expect#tobeclosetonumber-numdigits)

Use toBeCloseTo to compare floating point numbers for approximate equality.

The optional numDigits argument limits the number of digits to check **after** the decimal point. For the default value 2, the test criterion is Math.abs(expected - received) < 0.005 (that is, 10 \*\* -2 / 2).

Intuitive equality comparisons often fail, because arithmetic on decimal (base 10) values often have rounding errors in limited precision binary (base 2) representation. For example, this test fails:

test('adding works sanely with decimals', () => {  
 expect(0.2 + 0.1).toBe(0.3); *// Fails!*  
});

It fails because in JavaScript, 0.2 + 0.1 is actually 0.30000000000000004.

For example, this test passes with a precision of 5 digits:

test('adding works sanely with decimals', () => {  
 expect(0.2 + 0.1).toBeCloseTo(0.3, 5);  
});

Because floating point errors are the problem that toBeCloseTo solves, it does not support big integer values.

### .toBeDefined()[​](https://jestjs.io/docs/expect#tobedefined)

Use .toBeDefined to check that a variable is not undefined. For example, if you want to check that a function fetchNewFlavorIdea() returns something, you can write:

test('there is a new flavor idea', () => {  
 expect(fetchNewFlavorIdea()).toBeDefined();  
});

You could write expect(fetchNewFlavorIdea()).not.toBe(undefined), but it's better practice to avoid referring to undefined directly in your code.

### .toBeFalsy()[​](https://jestjs.io/docs/expect#tobefalsy)

Use .toBeFalsy when you don't care what a value is and you want to ensure a value is false in a boolean context. For example, let's say you have some application code that looks like:

drinkSomeLaCroix();  
if (!getErrors()) {  
 drinkMoreLaCroix();  
}

You may not care what getErrors returns, specifically - it might return false, null, or 0, and your code would still work. So if you want to test there are no errors after drinking some La Croix, you could write:

test('drinking La Croix does not lead to errors', () => {  
 drinkSomeLaCroix();  
 expect(getErrors()).toBeFalsy();  
});

In JavaScript, there are six falsy values: false, 0, '', null, undefined, and NaN. Everything else is truthy.

### .toBeGreaterThan(number | bigint)[​](https://jestjs.io/docs/expect#tobegreaterthannumber--bigint)

Use toBeGreaterThan to compare received > expected for number or big integer values. For example, test that ouncesPerCan() returns a value of more than 10 ounces:

test('ounces per can is more than 10', () => {  
 expect(ouncesPerCan()).toBeGreaterThan(10);  
});

### .toBeGreaterThanOrEqual(number | bigint)[​](https://jestjs.io/docs/expect#tobegreaterthanorequalnumber--bigint)

Use toBeGreaterThanOrEqual to compare received >= expected for number or big integer values. For example, test that ouncesPerCan() returns a value of at least 12 ounces:

test('ounces per can is at least 12', () => {  
 expect(ouncesPerCan()).toBeGreaterThanOrEqual(12);  
});

### .toBeLessThan(number | bigint)[​](https://jestjs.io/docs/expect#tobelessthannumber--bigint)

Use toBeLessThan to compare received < expected for number or big integer values. For example, test that ouncesPerCan() returns a value of less than 20 ounces:

test('ounces per can is less than 20', () => {  
 expect(ouncesPerCan()).toBeLessThan(20);  
});

### .toBeLessThanOrEqual(number | bigint)[​](https://jestjs.io/docs/expect#tobelessthanorequalnumber--bigint)

Use toBeLessThanOrEqual to compare received <= expected for number or big integer values. For example, test that ouncesPerCan() returns a value of at most 12 ounces:

test('ounces per can is at most 12', () => {  
 expect(ouncesPerCan()).toBeLessThanOrEqual(12);  
});

### .toBeInstanceOf(Class)[​](https://jestjs.io/docs/expect#tobeinstanceofclass)

Use .toBeInstanceOf(Class) to check that an object is an instance of a class. This matcher uses instanceof underneath.

class A {}  
  
expect(new A()).toBeInstanceOf(A);  
expect(() => {}).toBeInstanceOf(Function);  
expect(new A()).toBeInstanceOf(Function); *// throws*

### .toBeNull()[​](https://jestjs.io/docs/expect#tobenull)

.toBeNull() is the same as .toBe(null) but the error messages are a bit nicer. So use .toBeNull() when you want to check that something is null.

function bloop() {  
 return null;  
}  
  
test('bloop returns null', () => {  
 expect(bloop()).toBeNull();  
});

### .toBeTruthy()[​](https://jestjs.io/docs/expect#tobetruthy)

Use .toBeTruthy when you don't care what a value is and you want to ensure a value is true in a boolean context. For example, let's say you have some application code that looks like:

drinkSomeLaCroix();  
if (thirstInfo()) {  
 drinkMoreLaCroix();  
}

You may not care what thirstInfo returns, specifically - it might return true or a complex object, and your code would still work. So if you want to test that thirstInfo will be truthy after drinking some La Croix, you could write:

test('drinking La Croix leads to having thirst info', () => {  
 drinkSomeLaCroix();  
 expect(thirstInfo()).toBeTruthy();  
});

In JavaScript, there are six falsy values: false, 0, '', null, undefined, and NaN. Everything else is truthy.

### .toBeUndefined()[​](https://jestjs.io/docs/expect#tobeundefined)

Use .toBeUndefined to check that a variable is undefined. For example, if you want to check that a function bestDrinkForFlavor(flavor) returns undefined for the 'octopus' flavor, because there is no good octopus-flavored drink:

test('the best drink for octopus flavor is undefined', () => {  
 expect(bestDrinkForFlavor('octopus')).toBeUndefined();  
});

You could write expect(bestDrinkForFlavor('octopus')).toBe(undefined), but it's better practice to avoid referring to undefined directly in your code.

### .toBeNaN()[​](https://jestjs.io/docs/expect#tobenan)

Use .toBeNaN when checking a value is NaN.

test('passes when value is NaN', () => {  
 expect(NaN).toBeNaN();  
 expect(1).not.toBeNaN();  
});

### .toContain(item)[​](https://jestjs.io/docs/expect#tocontainitem)

Use .toContain when you want to check that an item is in an array. For testing the items in the array, this uses ===, a strict equality check. .toContain can also check whether a string is a substring of another string.

For example, if getAllFlavors() returns an array of flavors and you want to be sure that lime is in there, you can write:

test('the flavor list contains lime', () => {  
 expect(getAllFlavors()).toContain('lime');  
});

This matcher also accepts others iterables such as strings, sets, node lists and HTML collections.

### .toContainEqual(item)[​](https://jestjs.io/docs/expect#tocontainequalitem)

Use .toContainEqual when you want to check that an item with a specific structure and values is contained in an array. For testing the items in the array, this matcher recursively checks the equality of all fields, rather than checking for object identity.

describe('my beverage', () => {  
 test('is delicious and not sour', () => {  
 const myBeverage = {delicious: true, sour: false};  
 expect(myBeverages()).toContainEqual(myBeverage);  
 });  
});

### .toEqual(value)[​](https://jestjs.io/docs/expect#toequalvalue)

Use .toEqual to compare recursively all properties of object instances (also known as "deep" equality). It calls Object.is to compare primitive values, which is even better for testing than === strict equality operator.

For example, .toEqual and .toBe behave differently in this test suite, so all the tests pass:

const can1 = {  
 flavor: 'grapefruit',  
 ounces: 12,  
};  
const can2 = {  
 flavor: 'grapefruit',  
 ounces: 12,  
};  
  
describe('the La Croix cans on my desk', () => {  
 test('have all the same properties', () => {  
 expect(can1).toEqual(can2);  
 });  
 test('are not the exact same can', () => {  
 expect(can1).not.toBe(can2);  
 });  
});

TIP

toEqual ignores object keys with undefined properties, undefined array items, array sparseness, or object type mismatch. To take these into account use [.toStrictEqual](https://jestjs.io/docs/expect#tostrictequalvalue) instead.

INFO

.toEqual won't perform a deep equality check for two errors. Only the message property of an Error is considered for equality. It is recommended to use the .toThrow matcher for testing against errors.

If differences between properties do not help you to understand why a test fails, especially if the report is large, then you might move the comparison into the expect function. For example, use equals method of Buffer class to assert whether or not buffers contain the same content:

* rewrite expect(received).toEqual(expected) as expect(received.equals(expected)).toBe(true)
* rewrite expect(received).not.toEqual(expected) as expect(received.equals(expected)).toBe(false)

### .toMatch(regexp | string)[​](https://jestjs.io/docs/expect#tomatchregexp--string)

Use .toMatch to check that a string matches a regular expression.

For example, you might not know what exactly essayOnTheBestFlavor() returns, but you know it's a really long string, and the substring grapefruit should be in there somewhere. You can test this with:

describe('an essay on the best flavor', () => {  
 test('mentions grapefruit', () => {  
 expect(essayOnTheBestFlavor()).toMatch(/grapefruit/);  
 expect(essayOnTheBestFlavor()).toMatch(new RegExp('grapefruit'));  
 });  
});

This matcher also accepts a string, which it will try to match:

describe('grapefruits are healthy', () => {  
 test('grapefruits are a fruit', () => {  
 expect('grapefruits').toMatch('fruit');  
 });  
});

### .toMatchObject(object)[​](https://jestjs.io/docs/expect#tomatchobjectobject)

Use .toMatchObject to check that a JavaScript object matches a subset of the properties of an object. It will match received objects with properties that are **not** in the expected object.

You can also pass an array of objects, in which case the method will return true only if each object in the received array matches (in the toMatchObject sense described above) the corresponding object in the expected array. This is useful if you want to check that two arrays match in their number of elements, as opposed to arrayContaining, which allows for extra elements in the received array.

You can match properties against values or against matchers.

const houseForSale = {  
 bath: true,  
 bedrooms: 4,  
 kitchen: {  
 amenities: ['oven', 'stove', 'washer'],  
 area: 20,  
 wallColor: 'white',  
 },  
};  
const desiredHouse = {  
 bath: true,  
 kitchen: {  
 amenities: ['oven', 'stove', 'washer'],  
 wallColor: expect.stringMatching(/white|yellow/),  
 },  
};  
  
test('the house has my desired features', () => {  
 expect(houseForSale).toMatchObject(desiredHouse);  
});

describe('toMatchObject applied to arrays', () => {  
 test('the number of elements must match exactly', () => {  
 expect([{foo: 'bar'}, {baz: 1}]).toMatchObject([{foo: 'bar'}, {baz: 1}]);  
 });  
  
 test('.toMatchObject is called for each elements, so extra object properties are okay', () => {  
 expect([{foo: 'bar'}, {baz: 1, extra: 'quux'}]).toMatchObject([  
 {foo: 'bar'},  
 {baz: 1},  
 ]);  
 });  
});

### .toMatchSnapshot(propertyMatchers?, hint?)[​](https://jestjs.io/docs/expect#tomatchsnapshotpropertymatchers-hint)

This ensures that a value matches the most recent snapshot. Check out [the Snapshot Testing guide](https://jestjs.io/docs/snapshot-testing) for more information.

You can provide an optional propertyMatchers object argument, which has asymmetric matchers as values of a subset of expected properties, **if** the received value will be an **object** instance. It is like toMatchObject with flexible criteria for a subset of properties, followed by a snapshot test as exact criteria for the rest of the properties.

You can provide an optional hint string argument that is appended to the test name. Although Jest always appends a number at the end of a snapshot name, short descriptive hints might be more useful than numbers to differentiate **multiple** snapshots in a **single** it or test block. Jest sorts snapshots by name in the corresponding .snap file.

### .toMatchInlineSnapshot(propertyMatchers?, inlineSnapshot)[​](https://jestjs.io/docs/expect#tomatchinlinesnapshotpropertymatchers-inlinesnapshot)

Ensures that a value matches the most recent snapshot.

You can provide an optional propertyMatchers object argument, which has asymmetric matchers as values of a subset of expected properties, **if** the received value will be an **object** instance. It is like toMatchObject with flexible criteria for a subset of properties, followed by a snapshot test as exact criteria for the rest of the properties.

Jest adds the inlineSnapshot string argument to the matcher in the test file (instead of an external .snap file) the first time that the test runs.

Check out the section on [Inline Snapshots](https://jestjs.io/docs/snapshot-testing#inline-snapshots) for more info.

### .toStrictEqual(value)[​](https://jestjs.io/docs/expect#tostrictequalvalue)

Use .toStrictEqual to test that objects have the same structure and type.

Differences from .toEqual:

* keys with undefined properties are checked, e.g. {a: undefined, b: 2} will not equal {b: 2};
* undefined items are taken into account, e.g. [2] will not equal [2, undefined];
* array sparseness is checked, e.g. [, 1] will not equal [undefined, 1];
* object types are checked, e.g. a class instance with fields a and b will not equal a literal object with fields a and b.

class LaCroix {  
 constructor(flavor) {  
 this.flavor = flavor;  
 }  
}  
  
describe('the La Croix cans on my desk', () => {  
 test('are not semantically the same', () => {  
 expect(new LaCroix('lemon')).toEqual({flavor: 'lemon'});  
 expect(new LaCroix('lemon')).not.toStrictEqual({flavor: 'lemon'});  
 });  
});

### .toThrow(error?)[​](https://jestjs.io/docs/expect#tothrowerror)

Also under the alias: .toThrowError(error?)

Use .toThrow to test that a function throws when it is called. For example, if we want to test that drinkFlavor('octopus') throws, because octopus flavor is too disgusting to drink, we could write:

test('throws on octopus', () => {  
 expect(() => {  
 drinkFlavor('octopus');  
 }).toThrow();  
});

TIP

You must wrap the code in a function, otherwise the error will not be caught and the assertion will fail.

You can provide an optional argument to test that a specific error is thrown:

* regular expression: error message **matches** the pattern
* string: error message **includes** the substring
* error object: error message is **equal to** the message property of the object
* error class: error object is **instance of** class

For example, let's say that drinkFlavor is coded like this:

function drinkFlavor(flavor) {  
 if (flavor == 'octopus') {  
 throw new DisgustingFlavorError('yuck, octopus flavor');  
 }  
 *// Do some other stuff*  
}

We could test this error gets thrown in several ways:

test('throws on octopus', () => {  
 function drinkOctopus() {  
 drinkFlavor('octopus');  
 }  
  
 *// Test that the error message says "yuck" somewhere: these are equivalent*  
 expect(drinkOctopus).toThrow(/yuck/);  
 expect(drinkOctopus).toThrow('yuck');  
  
 *// Test the exact error message*  
 expect(drinkOctopus).toThrow(/^yuck, octopus flavor$/);  
 expect(drinkOctopus).toThrow(new Error('yuck, octopus flavor'));  
  
 *// Test that we get a DisgustingFlavorError*  
 expect(drinkOctopus).toThrow(DisgustingFlavorError);  
});

### .toThrowErrorMatchingSnapshot(hint?)[​](https://jestjs.io/docs/expect#tothrowerrormatchingsnapshothint)

Use .toThrowErrorMatchingSnapshot to test that a function throws an error matching the most recent snapshot when it is called.

You can provide an optional hint string argument that is appended to the test name. Although Jest always appends a number at the end of a snapshot name, short descriptive hints might be more useful than numbers to differentiate **multiple** snapshots in a **single** it or test block. Jest sorts snapshots by name in the corresponding .snap file.

For example, let's say you have a drinkFlavor function that throws whenever the flavor is 'octopus', and is coded like this:

function drinkFlavor(flavor) {  
 if (flavor == 'octopus') {  
 throw new DisgustingFlavorError('yuck, octopus flavor');  
 }  
 *// Do some other stuff*  
}

The test for this function will look this way:

test('throws on octopus', () => {  
 function drinkOctopus() {  
 drinkFlavor('octopus');  
 }  
  
 expect(drinkOctopus).toThrowErrorMatchingSnapshot();  
});

And it will generate the following snapshot:

exports[`drinking flavors throws on octopus 1`] = `"yuck, octopus flavor"`;

Check out [React Tree Snapshot Testing](https://jestjs.io/blog/2016/07/27/jest-14) for more information on snapshot testing.

### .toThrowErrorMatchingInlineSnapshot(inlineSnapshot)[​](https://jestjs.io/docs/expect#tothrowerrormatchinginlinesnapshotinlinesnapshot)

Use .toThrowErrorMatchingInlineSnapshot to test that a function throws an error matching the most recent snapshot when it is called.

Jest adds the inlineSnapshot string argument to the matcher in the test file (instead of an external .snap file) the first time that the test runs.

Check out the section on [Inline Snapshots](https://jestjs.io/docs/snapshot-testing#inline-snapshots) for more info.

## Asymmetric Matchers[​](https://jestjs.io/docs/expect#asymmetric-matchers)

### expect.anything()[​](https://jestjs.io/docs/expect#expectanything)

expect.anything() matches anything but null or undefined. You can use it inside toEqual or toBeCalledWith instead of a literal value. For example, if you want to check that a mock function is called with a non-null argument:

test('map calls its argument with a non-null argument', () => {  
 const mock = jest.fn();  
 [1].map(x => mock(x));  
 expect(mock).toHaveBeenCalledWith(expect.anything());  
});

### expect.any(constructor)[​](https://jestjs.io/docs/expect#expectanyconstructor)

expect.any(constructor) matches anything that was created with the given constructor or if it's a primitive that is of the passed type. You can use it inside toEqual or toBeCalledWith instead of a literal value. For example, if you want to check that a mock function is called with a number:

class Cat {}  
function getCat(fn) {  
 return fn(new Cat());  
}  
  
test('randocall calls its callback with a class instance', () => {  
 const mock = jest.fn();  
 getCat(mock);  
 expect(mock).toHaveBeenCalledWith(expect.any(Cat));  
});  
  
function randocall(fn) {  
 return fn(Math.floor(Math.random() \* 6 + 1));  
}  
  
test('randocall calls its callback with a number', () => {  
 const mock = jest.fn();  
 randocall(mock);  
 expect(mock).toHaveBeenCalledWith(expect.any(Number));  
});

### expect.arrayContaining(array)[​](https://jestjs.io/docs/expect#expectarraycontainingarray)

expect.arrayContaining(array) matches a received array which contains all of the elements in the expected array. That is, the expected array is a **subset** of the received array. Therefore, it matches a received array which contains elements that are **not** in the expected array.

You can use it instead of a literal value:

* in toEqual or toBeCalledWith
* to match a property in objectContaining or toMatchObject

describe('arrayContaining', () => {  
 const expected = ['Alice', 'Bob'];  
 it('matches even if received contains additional elements', () => {  
 expect(['Alice', 'Bob', 'Eve']).toEqual(expect.arrayContaining(expected));  
 });  
 it('does not match if received does not contain expected elements', () => {  
 expect(['Bob', 'Eve']).not.toEqual(expect.arrayContaining(expected));  
 });  
});

describe('Beware of a misunderstanding! A sequence of dice rolls', () => {  
 const expected = [1, 2, 3, 4, 5, 6];  
 it('matches even with an unexpected number 7', () => {  
 expect([4, 1, 6, 7, 3, 5, 2, 5, 4, 6]).toEqual(  
 expect.arrayContaining(expected),  
 );  
 });  
 it('does not match without an expected number 2', () => {  
 expect([4, 1, 6, 7, 3, 5, 7, 5, 4, 6]).not.toEqual(  
 expect.arrayContaining(expected),  
 );  
 });  
});

### expect.not.arrayContaining(array)[​](https://jestjs.io/docs/expect#expectnotarraycontainingarray)

expect.not.arrayContaining(array) matches a received array which does not contain all of the elements in the expected array. That is, the expected array **is not a subset** of the received array.

It is the inverse of expect.arrayContaining.

describe('not.arrayContaining', () => {  
 const expected = ['Samantha'];  
  
 it('matches if the actual array does not contain the expected elements', () => {  
 expect(['Alice', 'Bob', 'Eve']).toEqual(  
 expect.not.arrayContaining(expected),  
 );  
 });  
});

### expect.closeTo(number, numDigits?)[​](https://jestjs.io/docs/expect#expectclosetonumber-numdigits)

expect.closeTo(number, numDigits?) is useful when comparing floating point numbers in object properties or array item. If you need to compare a number, please use .toBeCloseTo instead.

The optional numDigits argument limits the number of digits to check **after** the decimal point. For the default value 2, the test criterion is Math.abs(expected - received) < 0.005 (that is, 10 \*\* -2 / 2).

For example, this test passes with a precision of 5 digits:

test('compare float in object properties', () => {  
 expect({  
 title: '0.1 + 0.2',  
 sum: 0.1 + 0.2,  
 }).toEqual({  
 title: '0.1 + 0.2',  
 sum: expect.closeTo(0.3, 5),  
 });  
});

### expect.objectContaining(object)[​](https://jestjs.io/docs/expect#expectobjectcontainingobject)

expect.objectContaining(object) matches any received object that recursively matches the expected properties. That is, the expected object is a **subset** of the received object. Therefore, it matches a received object which contains properties that **are present** in the expected object.

Instead of literal property values in the expected object, you can use matchers, expect.anything(), and so on.

For example, let's say that we expect an onPress function to be called with an Event object, and all we need to verify is that the event has event.x and event.y properties. We can do that with:

test('onPress gets called with the right thing', () => {  
 const onPress = jest.fn();  
 simulatePresses(onPress);  
 expect(onPress).toHaveBeenCalledWith(  
 expect.objectContaining({  
 x: expect.any(Number),  
 y: expect.any(Number),  
 }),  
 );  
});

### expect.not.objectContaining(object)[​](https://jestjs.io/docs/expect#expectnotobjectcontainingobject)

expect.not.objectContaining(object) matches any received object that does not recursively match the expected properties. That is, the expected object **is not a subset** of the received object. Therefore, it matches a received object which contains properties that are **not** in the expected object.

It is the inverse of expect.objectContaining.

describe('not.objectContaining', () => {  
 const expected = {foo: 'bar'};  
  
 it('matches if the actual object does not contain expected key: value pairs', () => {  
 expect({bar: 'baz'}).toEqual(expect.not.objectContaining(expected));  
 });  
});

### expect.stringContaining(string)[​](https://jestjs.io/docs/expect#expectstringcontainingstring)

expect.stringContaining(string) matches the received value if it is a string that contains the exact expected string.

### expect.not.stringContaining(string)[​](https://jestjs.io/docs/expect#expectnotstringcontainingstring)

expect.not.stringContaining(string) matches the received value if it is not a string or if it is a string that does not contain the exact expected string.

It is the inverse of expect.stringContaining.

describe('not.stringContaining', () => {  
 const expected = 'Hello world!';  
  
 it('matches if the received value does not contain the expected substring', () => {  
 expect('How are you?').toEqual(expect.not.stringContaining(expected));  
 });  
});

### expect.stringMatching(string | regexp)[​](https://jestjs.io/docs/expect#expectstringmatchingstring--regexp)

expect.stringMatching(string | regexp) matches the received value if it is a string that matches the expected string or regular expression.

You can use it instead of a literal value:

* in toEqual or toBeCalledWith
* to match an element in arrayContaining
* to match a property in objectContaining or toMatchObject

This example also shows how you can nest multiple asymmetric matchers, with expect.stringMatching inside the expect.arrayContaining.

describe('stringMatching in arrayContaining', () => {  
 const expected = [  
 expect.stringMatching(/^Alic/),  
 expect.stringMatching(/^[BR]ob/),  
 ];  
 it('matches even if received contains additional elements', () => {  
 expect(['Alicia', 'Roberto', 'Evelina']).toEqual(  
 expect.arrayContaining(expected),  
 );  
 });  
 it('does not match if received does not contain expected elements', () => {  
 expect(['Roberto', 'Evelina']).not.toEqual(  
 expect.arrayContaining(expected),  
 );  
 });  
});

### expect.not.stringMatching(string | regexp)[​](https://jestjs.io/docs/expect#expectnotstringmatchingstring--regexp)

expect.not.stringMatching(string | regexp) matches the received value if it is not a string or if it is a string that does not match the expected string or regular expression.

It is the inverse of expect.stringMatching.

describe('not.stringMatching', () => {  
 const expected = /Hello world!/;  
  
 it('matches if the received value does not match the expected regex', () => {  
 expect('How are you?').toEqual(expect.not.stringMatching(expected));  
 });  
});

## Assertion Count[​](https://jestjs.io/docs/expect#assertion-count)

### expect.assertions(number)[​](https://jestjs.io/docs/expect#expectassertionsnumber)

expect.assertions(number) verifies that a certain number of assertions are called during a test. This is often useful when testing asynchronous code, in order to make sure that assertions in a callback actually got called.

For example, let's say that we have a function doAsync that receives two callbacks callback1 and callback2, it will asynchronously call both of them in an unknown order. We can test this with:

test('doAsync calls both callbacks', () => {  
 expect.assertions(2);  
 function callback1(data) {  
 expect(data).toBeTruthy();  
 }  
 function callback2(data) {  
 expect(data).toBeTruthy();  
 }  
  
 doAsync(callback1, callback2);  
});

The expect.assertions(2) call ensures that both callbacks actually get called.

### expect.hasAssertions()[​](https://jestjs.io/docs/expect#expecthasassertions)

expect.hasAssertions() verifies that at least one assertion is called during a test. This is often useful when testing asynchronous code, in order to make sure that assertions in a callback actually got called.

For example, let's say that we have a few functions that all deal with state. prepareState calls a callback with a state object, validateState runs on that state object, and waitOnState returns a promise that waits until all prepareState callbacks complete. We can test this with:

test('prepareState prepares a valid state', () => {  
 expect.hasAssertions();  
 prepareState(state => {  
 expect(validateState(state)).toBeTruthy();  
 });  
 return waitOnState();  
});

The expect.hasAssertions() call ensures that the prepareState callback actually gets called.

## Extend Utilities[​](https://jestjs.io/docs/expect#extend-utilities)

### expect.addEqualityTesters(testers)[​](https://jestjs.io/docs/expect#expectaddequalitytesterstesters)

You can use expect.addEqualityTesters to add your own methods to test if two objects are equal. For example, let's say you have a class in your code that represents volume and can determine if two volumes using different units are equal. You may want toEqual (and other equality matchers) to use this custom equality method when comparing to Volume classes. You can add a custom equality tester to have toEqual detect and apply custom logic when comparing Volume classes:

* JavaScript
* TypeScript

Volume.js

*// For simplicity in this example, we'll just support the units 'L' and 'mL'*  
export class Volume {  
 constructor(amount, unit) {  
 this.amount = amount;  
 this.unit = unit;  
 }  
  
 toString() {  
 return `[Volume ${this.amount}${this.unit}]`;  
 }  
  
 equals(other) {  
 if (this.unit === other.unit) {  
 return this.amount === other.amount;  
 } else if (this.unit === 'L' && other.unit === 'mL') {  
 return this.amount \* 1000 === other.unit;  
 } else {  
 return this.amount === other.unit \* 1000;  
 }  
 }  
}

areVolumesEqual.js

import {expect} from '@jest/globals';  
import {Volume} from './Volume.js';  
  
function areVolumesEqual(a, b) {  
 const isAVolume = a instanceof Volume;  
 const isBVolume = b instanceof Volume;  
  
 if (isAVolume && isBVolume) {  
 return a.equals(b);  
 } else if (isAVolume === isBVolume) {  
 return undefined;  
 } else {  
 return false;  
 }  
}  
  
expect.addEqualityTesters([areVolumesEqual]);

\_\_tests\_\_/Volume.test.js

import {expect, test} from '@jest/globals';  
import {Volume} from '../Volume.js';  
import '../areVolumesEqual.js';  
  
test('are equal with different units', () => {  
 expect(new Volume(1, 'L')).toEqual(new Volume(1000, 'mL'));  
});

#### Custom equality testers API[​](https://jestjs.io/docs/expect#custom-equality-testers-api)

Custom testers are functions that return either the result (true or false) of comparing the equality of the two given arguments or undefined if the tester does not handle the given objects and wants to delegate equality to other testers (for example, the builtin equality testers).

Custom testers are called with 3 arguments: the two objects to compare and the array of custom testers (used for recursive testers, see the section below).

These helper functions and properties can be found on this inside a custom tester:

#### this.equals(a, b, customTesters?)[​](https://jestjs.io/docs/expect#thisequalsa-b-customtesters)

This is a deep-equality function that will return true if two objects have the same values (recursively). It optionally takes a list of custom equality testers to apply to the deep equality checks. If you use this function, pass through the custom testers your tester is given so further equality checks equals applies can also use custom testers the test author may have configured. See the example in the [Recursive custom equality testers](https://jestjs.io/docs/expect#recursive-custom-equality-testers) section for more details.

#### Matchers vs Testers[​](https://jestjs.io/docs/expect#matchers-vs-testers)

Matchers are methods available on expect, for example expect().toEqual(). toEqual is a matcher. A tester is a method used by matchers that do equality checks to determine if objects are the same.

Custom matchers are good to use when you want to provide a custom assertion that test authors can use in their tests. For example, the toBeWithinRange example in the [expect.extend](https://jestjs.io/docs/expect#expectextendmatchers) section is a good example of a custom matcher. Sometimes a test author may want to assert two numbers are exactly equal and should use toBe. Other times, however, a test author may want to allow for some flexibility in their test, and toBeWithinRange may be a more appropriate assertion.

Custom equality testers are good for globally extending Jest matchers to apply custom equality logic for all equality comparisons. Test authors can't turn on custom testers for certain assertions and turn them off for others (a custom matcher should be used instead if that behavior is desired). For example, defining how to check if two Volume objects are equal for all matchers would be a good custom equality tester.

#### Recursive custom equality testers[​](https://jestjs.io/docs/expect#recursive-custom-equality-testers)

If your custom equality testers are testing objects with properties you'd like to do deep equality with, you should use the this.equals helper available to equality testers. This equals method is the same deep equals method Jest uses internally for all of its deep equality comparisons. It's the method that invokes your custom equality tester. It accepts an array of custom equality testers as a third argument. Custom equality testers are also given an array of custom testers as their third argument. Pass this argument into the third argument of equals so that any further equality checks deeper into your object can also take advantage of custom equality testers.

For example, let's say you have a Book class that contains an array of Author classes and both of these classes have custom testers. The Book custom tester would want to do a deep equality check on the array of Authors and pass in the custom testers given to it, so the Authors custom equality tester is applied:

customEqualityTesters.js

function areAuthorEqual(a, b) {  
 const isAAuthor = a instanceof Author;  
 const isBAuthor = b instanceof Author;  
  
 if (isAAuthor && isBAuthor) {  
 *// Authors are equal if they have the same name*  
 return a.name === b.name;  
 } else if (isAAuthor === isBAuthor) {  
 return undefined;  
 } else {  
 return false;  
 }  
}  
  
function areBooksEqual(a, b, customTesters) {  
 const isABook = a instanceof Book;  
 const isBBook = b instanceof Book;  
  
 if (isABook && isBBook) {  
 *// Books are the same if they have the same name and author array. We need*  
 *// to pass customTesters to equals here so the Author custom tester will be*  
 *// used when comparing Authors*  
 return (  
 a.name === b.name && this.equals(a.authors, b.authors, customTesters)  
 );  
 } else if (isABook === isBBook) {  
 return undefined;  
 } else {  
 return false;  
 }  
}  
  
expect.addEqualityTesters([areAuthorsEqual, areBooksEqual]);

NOTE

Remember to define your equality testers as regular functions and **not** arrow functions in order to access the tester context helpers (e.g. this.equals).

### expect.addSnapshotSerializer(serializer)[​](https://jestjs.io/docs/expect#expectaddsnapshotserializerserializer)

You can call expect.addSnapshotSerializer to add a module that formats application-specific data structures.

For an individual test file, an added module precedes any modules from snapshotSerializers configuration, which precede the default snapshot serializers for built-in JavaScript types and for React elements. The last module added is the first module tested.

import serializer from 'my-serializer-module';  
expect.addSnapshotSerializer(serializer);  
  
*// affects expect(value).toMatchSnapshot() assertions in the test file*

If you add a snapshot serializer in individual test files instead of adding it to snapshotSerializers configuration:

* You make the dependency explicit instead of implicit.
* You avoid limits to configuration that might cause you to eject from [create-react-app](https://github.com/facebookincubator/create-react-app).

See [configuring Jest](https://jestjs.io/docs/configuration#snapshotserializers-arraystring) for more information.

### expect.extend(matchers)[​](https://jestjs.io/docs/expect#expectextendmatchers)

You can use expect.extend to add your own matchers to Jest. For example, let's say that you're testing a number utility library and you're frequently asserting that numbers appear within particular ranges of other numbers. You could abstract that into a toBeWithinRange matcher:

* JavaScript
* TypeScript

toBeWithinRange.js

import {expect} from '@jest/globals';  
  
function toBeWithinRange(actual, floor, ceiling) {  
 if (  
 typeof actual !== 'number' ||  
 typeof floor !== 'number' ||  
 typeof ceiling !== 'number'  
 ) {  
 throw new Error('These must be of type number!');  
 }  
  
 const pass = actual >= floor && actual <= ceiling;  
 if (pass) {  
 return {  
 message: () =>  
 `expected ${this.utils.printReceived(  
 actual,  
 )} not to be within range ${this.utils.printExpected(  
 `${floor} - ${ceiling}`,  
 )}`,  
 pass: true,  
 };  
 } else {  
 return {  
 message: () =>  
 `expected ${this.utils.printReceived(  
 actual,  
 )} to be within range ${this.utils.printExpected(  
 `${floor} - ${ceiling}`,  
 )}`,  
 pass: false,  
 };  
 }  
}  
  
expect.extend({  
 toBeWithinRange,  
});

\_\_tests\_\_/ranges.test.js

import {expect, test} from '@jest/globals';  
import '../toBeWithinRange';  
  
test('is within range', () => expect(100).toBeWithinRange(90, 110));  
  
test('is NOT within range', () => expect(101).not.toBeWithinRange(0, 100));  
  
test('asymmetric ranges', () => {  
 expect({apples: 6, bananas: 3}).toEqual({  
 apples: expect.toBeWithinRange(1, 10),  
 bananas: expect.not.toBeWithinRange(11, 20),  
 });  
});

toBeWithinRange.d.ts

*// optionally add a type declaration, e.g. it enables autocompletion in IDEs*  
declare module 'expect' {  
 interface AsymmetricMatchers {  
 toBeWithinRange(floor: number, ceiling: number): void;  
 }  
 interface Matchers<R> {  
 toBeWithinRange(floor: number, ceiling: number): R;  
 }  
}  
  
export {};

TIP

The type declaration of the matcher can live in a .d.ts file or in an imported .ts module (see JS and TS examples above respectively). If you keep the declaration in a .d.ts file, make sure that it is included in the program and that it is a valid module, i.e. it has at least an empty export {}.

TIP

Instead of importing toBeWithinRange module to the test file, you can enable the matcher for all tests by moving the expect.extend call to a [setupFilesAfterEnv](https://jestjs.io/docs/configuration#setupfilesafterenv-array) script:

import {expect} from '@jest/globals';  
*// remember to export `toBeWithinRange` as well*  
import {toBeWithinRange} from './toBeWithinRange';  
  
expect.extend({  
 toBeWithinRange,  
});

#### Async Matchers[​](https://jestjs.io/docs/expect#async-matchers)

expect.extend also supports async matchers. Async matchers return a Promise so you will need to await the returned value. Let's use an example matcher to illustrate the usage of them. We are going to implement a matcher called toBeDivisibleByExternalValue, where the divisible number is going to be pulled from an external source.

expect.extend({  
 async toBeDivisibleByExternalValue(received) {  
 const externalValue = await getExternalValueFromRemoteSource();  
 const pass = received % externalValue == 0;  
 if (pass) {  
 return {  
 message: () =>  
 `expected ${received} not to be divisible by ${externalValue}`,  
 pass: true,  
 };  
 } else {  
 return {  
 message: () =>  
 `expected ${received} to be divisible by ${externalValue}`,  
 pass: false,  
 };  
 }  
 },  
});  
  
test('is divisible by external value', async () => {  
 await expect(100).toBeDivisibleByExternalValue();  
 await expect(101).not.toBeDivisibleByExternalValue();  
});

#### Custom Matchers API[​](https://jestjs.io/docs/expect#custom-matchers-api)

Matchers should return an object (or a Promise of an object) with two keys. pass indicates whether there was a match or not, and message provides a function with no arguments that returns an error message in case of failure. Thus, when pass is false, message should return the error message for when expect(x).yourMatcher() fails. And when pass is true, message should return the error message for when expect(x).not.yourMatcher() fails.

Matchers are called with the argument passed to expect(x) followed by the arguments passed to .yourMatcher(y, z):

expect.extend({  
 yourMatcher(x, y, z) {  
 return {  
 pass: true,  
 message: () => '',  
 };  
 },  
});

These helper functions and properties can be found on this inside a custom matcher:

#### this.isNot[​](https://jestjs.io/docs/expect#thisisnot)

A boolean to let you know this matcher was called with the negated .not modifier allowing you to display a clear and correct matcher hint (see example code).

#### this.promise[​](https://jestjs.io/docs/expect#thispromise)

A string allowing you to display a clear and correct matcher hint:

* 'rejects' if matcher was called with the promise .rejects modifier
* 'resolves' if matcher was called with the promise .resolves modifier
* '' if matcher was not called with a promise modifier

#### this.equals(a, b, customTesters?)[​](https://jestjs.io/docs/expect#thisequalsa-b-customtesters-1)

This is a deep-equality function that will return true if two objects have the same values (recursively). It optionally takes a list of custom equality testers to apply to the deep equality checks (see this.customTesters below).

#### this.expand[​](https://jestjs.io/docs/expect#thisexpand)

A boolean to let you know this matcher was called with an expand option. When Jest is called with the --expand flag, this.expand can be used to determine if Jest is expected to show full diffs and errors.

#### this.utils[​](https://jestjs.io/docs/expect#thisutils)

There are a number of helpful tools exposed on this.utils primarily consisting of the exports from [jest-matcher-utils](https://github.com/jestjs/jest/tree/main/packages/jest-matcher-utils).

The most useful ones are matcherHint, printExpected and printReceived to format the error messages nicely. For example, take a look at the implementation for the toBe matcher:

const {diff} = require('jest-diff');  
expect.extend({  
 toBe(received, expected) {  
 const options = {  
 comment: 'Object.is equality',  
 isNot: this.isNot,  
 promise: this.promise,  
 };  
  
 const pass = Object.is(received, expected);  
  
 const message = pass  
 ? () =>  
 *// eslint-disable-next-line prefer-template*  
 this.utils.matcherHint('toBe', undefined, undefined, options) +  
 '\n\n' +  
 `Expected: not ${this.utils.printExpected(expected)}\n` +  
 `Received: ${this.utils.printReceived(received)}`  
 : () => {  
 const diffString = diff(expected, received, {  
 expand: this.expand,  
 });  
 return (  
 *// eslint-disable-next-line prefer-template*  
 this.utils.matcherHint('toBe', undefined, undefined, options) +  
 '\n\n' +  
 (diffString && diffString.includes('- Expect')  
 ? `Difference:\n\n${diffString}`  
 : `Expected: ${this.utils.printExpected(expected)}\n` +  
 `Received: ${this.utils.printReceived(received)}`)  
 );  
 };  
  
 return {actual: received, message, pass};  
 },  
});

This will print something like this:

expect(received).toBe(expected)  
  
 Expected value to be (using Object.is):  
 "banana"  
 Received:  
 "apple"

When an assertion fails, the error message should give as much signal as necessary to the user so they can resolve their issue quickly. You should craft a precise failure message to make sure users of your custom assertions have a good developer experience.

#### this.customTesters[​](https://jestjs.io/docs/expect#thiscustomtesters)

If your matcher does a deep equality check using this.equals, you may want to pass user-provided custom testers to this.equals. The custom equality testers the user has provided using the addEqualityTesters API are available on this property. The built-in Jest matchers pass this.customTesters (along with other built-in testers) to this.equals to do deep equality, and your custom matchers may want to do the same.

#### Custom snapshot matchers[​](https://jestjs.io/docs/expect#custom-snapshot-matchers)

To use snapshot testing inside of your custom matcher you can import jest-snapshot and use it from within your matcher.

Here's a snapshot matcher that trims a string to store for a given length, .toMatchTrimmedSnapshot(length):

const {toMatchSnapshot} = require('jest-snapshot');  
  
expect.extend({  
 toMatchTrimmedSnapshot(received, length) {  
 return toMatchSnapshot.call(  
 this,  
 received.substring(0, length),  
 'toMatchTrimmedSnapshot',  
 );  
 },  
});  
  
it('stores only 10 characters', () => {  
 expect('extra long string oh my gerd').toMatchTrimmedSnapshot(10);  
});  
  
*/\**  
*Stored snapshot will look like:*  
  
*exports[`stores only 10 characters: toMatchTrimmedSnapshot 1`] = `"extra long"`;*  
*\*/*

It's also possible to create custom matchers for inline snapshots, the snapshots will be correctly added to the custom matchers. However, inline snapshot will always try to append to the first argument or the second when the first argument is the property matcher, so it's not possible to accept custom arguments in the custom matchers.

const {toMatchInlineSnapshot} = require('jest-snapshot');  
  
expect.extend({  
 toMatchTrimmedInlineSnapshot(received, ...rest) {  
 return toMatchInlineSnapshot.call(this, received.substring(0, 10), ...rest);  
 },  
});  
  
it('stores only 10 characters', () => {  
 expect('extra long string oh my gerd').toMatchTrimmedInlineSnapshot();  
 */\**  
 *The snapshot will be added inline like*  
 *expect('extra long string oh my gerd').toMatchTrimmedInlineSnapshot(*  
 *`"extra long"`*  
 *);*  
 *\*/*  
});

#### async[​](https://jestjs.io/docs/expect" \l "async" \o "Direct link to async)

If your custom inline snapshot matcher is async i.e. uses async-await you might encounter an error like "Multiple inline snapshots for the same call are not supported". Jest needs additional context information to find where the custom inline snapshot matcher was used to update the snapshots properly.

const {toMatchInlineSnapshot} = require('jest-snapshot');  
  
expect.extend({  
 async toMatchObservationInlineSnapshot(fn, ...rest) {  
 *// The error (and its stacktrace) must be created before any `await`*  
 this.error = new Error();  
  
 *// The implementation of `observe` doesn't matter.*  
 *// It only matters that the custom snapshot matcher is async.*  
 const observation = await observe(async () => {  
 await fn();  
 });  
  
 return toMatchInlineSnapshot.call(this, recording, ...rest);  
 },  
});  
  
it('observes something', async () => {  
 await expect(async () => {  
 return 'async action';  
 }).toMatchTrimmedInlineSnapshot();  
 */\**  
 *The snapshot will be added inline like*  
 *await expect(async () => {*  
 *return 'async action';*  
 *}).toMatchTrimmedInlineSnapshot(`"async action"`);*  
 *\*/*  
});

#### Bail out[​](https://jestjs.io/docs/expect#bail-out)

Usually jest tries to match every snapshot that is expected in a test.

Sometimes it might not make sense to continue the test if a prior snapshot failed. For example, when you make snapshots of a state-machine after various transitions you can abort the test once one transition produced the wrong state.

In that case you can implement a custom snapshot matcher that throws on the first mismatch instead of collecting every mismatch.

const {toMatchInlineSnapshot} = require('jest-snapshot');  
  
expect.extend({  
 toMatchStateInlineSnapshot(...args) {  
 this.dontThrow = () => {};  
  
 return toMatchInlineSnapshot.call(this, ...args);  
 },  
});  
  
let state = 'initial';  
  
function transition() {  
 *// Typo in the implementation should cause the test to fail*  
 if (state === 'INITIAL') {  
 state = 'pending';  
 } else if (state === 'pending') {  
 state = 'done';  
 }  
}  
  
it('transitions as expected', () => {  
 expect(state).toMatchStateInlineSnapshot(`"initial"`);  
  
 transition();  
 *// Already produces a mismatch. No point in continuing the test.*  
 expect(state).toMatchStateInlineSnapshot(`"loading"`);  
  
 transition();  
 expect(state).toMatchStateInlineSnapshot(`"done"`);  
});

# Jest Platform

You can cherry pick specific features of Jest and use them as standalone packages. Here's a list of the available packages:

## jest-changed-files[​](https://jestjs.io/docs/jest-platform" \l "jest-changed-files" \o "Direct link to jest-changed-files)

Tool for identifying modified files in a git/hg repository. Exports two functions:

* getChangedFilesForRoots returns a promise that resolves to an object with the changed files and repos.
* findRepos returns a promise that resolves to a set of repositories contained in the specified path.

### Example[​](https://jestjs.io/docs/jest-platform#example)

const {getChangedFilesForRoots} = require('jest-changed-files');  
  
*// print the set of modified files since last commit in the current repo*  
getChangedFilesForRoots(['./'], {  
 lastCommit: true,  
}).then(result => console.log(result.changedFiles));

You can read more about jest-changed-files in the [readme file](https://github.com/jestjs/jest/blob/main/packages/jest-changed-files/README.md).

## jest-diff[​](https://jestjs.io/docs/jest-platform" \l "jest-diff" \o "Direct link to jest-diff)

Tool for visualizing changes in data. Exports a function that compares two values of any type and returns a "pretty-printed" string illustrating the difference between the two arguments.

### Example[​](https://jestjs.io/docs/jest-platform#example-1)

const {diff} = require('jest-diff');  
  
const a = {a: {b: {c: 5}}};  
const b = {a: {b: {c: 6}}};  
  
const result = diff(a, b);  
  
*// print diff*  
console.log(result);

## jest-docblock[​](https://jestjs.io/docs/jest-platform" \l "jest-docblock" \o "Direct link to jest-docblock)

Tool for extracting and parsing the comments at the top of a JavaScript file. Exports various functions to manipulate the data inside the comment block.

### Example[​](https://jestjs.io/docs/jest-platform#example-2)

const {parseWithComments} = require('jest-docblock');  
  
const code = `  
/\*\*  
 \* This is a sample  
 \*  
 \* @flow  
 \*/  
  
 console.log('Hello World!');  
`;  
  
const parsed = parseWithComments(code);  
  
*// prints an object with two attributes: comments and pragmas.*  
console.log(parsed);

You can read more about jest-docblock in the [readme file](https://github.com/jestjs/jest/blob/main/packages/jest-docblock/README.md).

## jest-get-type[​](https://jestjs.io/docs/jest-platform" \l "jest-get-type" \o "Direct link to jest-get-type)

Module that identifies the primitive type of any JavaScript value. Exports a function that returns a string with the type of the value passed as argument.

### Example[​](https://jestjs.io/docs/jest-platform#example-3)

const {getType} = require('jest-get-type');  
  
const array = [1, 2, 3];  
const nullValue = null;  
const undefinedValue = undefined;  
  
*// prints 'array'*  
console.log(getType(array));  
*// prints 'null'*  
console.log(getType(nullValue));  
*// prints 'undefined'*  
console.log(getType(undefinedValue));

## jest-validate[​](https://jestjs.io/docs/jest-platform#jest-validate)

Tool for validating configurations submitted by users. Exports a function that takes two arguments: the user's configuration and an object containing an example configuration and other options. The return value is an object with two attributes:

* hasDeprecationWarnings, a boolean indicating whether the submitted configuration has deprecation warnings,
* isValid, a boolean indicating whether the configuration is correct or not.

### Example[​](https://jestjs.io/docs/jest-platform#example-4)

const {validate} = require('jest-validate');  
  
const configByUser = {  
 transform: '<rootDir>/node\_modules/my-custom-transform',  
};  
  
const result = validate(configByUser, {  
 comment: ' Documentation: http://custom-docs.com',  
 exampleConfig: {transform: '<rootDir>/node\_modules/babel-jest'},  
});  
  
console.log(result);

You can read more about jest-validate in the [readme file](https://github.com/jestjs/jest/blob/main/packages/jest-validate/README.md).

## jest-worker[​](https://jestjs.io/docs/jest-platform" \l "jest-worker" \o "Direct link to jest-worker)

Module used for parallelization of tasks. Exports a class JestWorker that takes the path of Node.js module and lets you call the module's exported methods as if they were class methods, returning a promise that resolves when the specified method finishes its execution in a forked process.

### Example[​](https://jestjs.io/docs/jest-platform#example-5)

heavy-task.js

module.exports = {  
 myHeavyTask: args => {  
 *// long running CPU intensive task.*  
 },  
};

main.js

async function main() {  
 const worker = new Worker(require.resolve('./heavy-task.js'));  
  
 *// run 2 tasks in parallel with different arguments*  
 const results = await Promise.all([  
 worker.myHeavyTask({foo: 'bar'}),  
 worker.myHeavyTask({bar: 'foo'}),  
 ]);  
  
 console.log(results);  
}  
  
main();

You can read more about jest-worker in the [readme file](https://github.com/jestjs/jest/blob/main/packages/jest-worker/README.md).

## pretty-format[​](https://jestjs.io/docs/jest-platform" \l "pretty-format" \o "Direct link to pretty-format)

Exports a function that converts any JavaScript value into a human-readable string. Supports all built-in JavaScript types out of the box and allows extension for application-specific types via user-defined plugins.

### Example[​](https://jestjs.io/docs/jest-platform#example-6)

const {format: prettyFormat} = require('pretty-format');  
  
const val = {object: {}};  
val.circularReference = val;  
val[Symbol('foo')] = 'foo';  
val.map = new Map([['prop', 'value']]);  
val.array = [-0, Infinity, NaN];  
  
console.log(prettyFormat(val));