Jest Unit Testing

Learning Objectives

- Understand the purpose of tests
- Understand what a unit test is
- Understand how to write unit tests in Jest

Material

Tests

When we write code, we need to check it works. As beginners, we often do this by trying out some dummy examples and printing the results to the console. This process, however, is slow and manual; it would be better if we could have the code check itself.

This is where tests come in. Tests take a piece of your code, give it some input and check that the output is correct.

```
expect(sum(9, 4)).toBe(13);
```

The above is a *unit test*. Unit tests test the smallest possible testable pieces of your code - in practice, these are usually functions. The opposite end of the spectrum is end-to-end testing; these check the whole application works properly.

Jest

You'll almost always use a library for testing. Jest is an example library created by Facebook. To install it, use `npm init -y` to create a `package.json` then run:

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

The `--save-dev` isn't strictly necessary - it just marks jest as a dev-dependency (a dependency that isn't actually needed for the application to run). We then need to go into our package.json and change the test script:

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

```
}
```

Now, when we create some tests and want to run them, we just enter `npm run test` and jest will find all the project's tests and run them.

I would also recommend running the following:

```
npm install @types/jest
```

This is optional but it will mean that VSCode recognises Jest and won't try and autocorrect keywords like `test`.

Unit Testing with Jest

Let's test our Person class. To create tests, we first need to create a file with the .test.js extension - when we run jest, it will recognise these as tests and run them.

```
app/
├─ person.js
├─ person.test.js
├─ node_modules/
├─ package.json
├─ package-lock.json
```

We then need export our Person class so it can be imported and tested:

```
// person.js
class Person {
    constructor(name, age) {
        if (age < 0) {
            throw new Error("age must be >= 0");
        }
        this.name = name;
        this.age = age;
    }
    greet() {
        console.log("Hello, I am", this.name);
    }
}

module.exports = Person;
```

Our person.js file should only contain the class: if we'd previously created some dummy Person objects in this file to test our class we should remove them now to tidy up our code.

We can now import our Person class to our test file to check it works. Remember, classes are blueprints for creating objects so we need to use it to create an object, then test the object:

```
// person.test.js
const Person = require("./person");

test("Person sets name argument as property", () => {
    const jon = new Person("Jon", 22);
    expect(jon.name).toBe("Jon");
});
```

As you can see, jest's `test` function takes 2 arguments:

- A description of what the test does
- A callback function which is the actual test

`toBe` is a matcher: `expect(jon.name).toBe("Jon")` essentially checks `jon.name === "Jon"` - if this is true, the test passes.

Jest has a handy page explaining the most common matchers you'll need.

Let's add a second test to check that our constructor throws an error if we input a negative age:

```
// person.test.js
test("Person throws error if age is negative", () => {
    const expectedError = new Error("age must be >= 0");
    expect(() => new Person("Jill", -99)).toThrow(expectedError);
});
```

When checking for an error, we need to give `expect` a function to call that will produce the error, so that it can handle it without breaking the test.

One last thing to watch out for when using jest is when checking two objects or two arrays are equal. The JavaScript `===` operator doesn't work as expected for arrays and objects:

```
> { a: 5 } === { a: 5 }
false
> [1] === [1]
```

```
false
```

If we are checking for equality of arrays or objects in jest, we need to use the `toEqual` matcher:

```
test("object equality", () => {
    const obj = { name: "Maya" };
    expect(obj).toBe({ name: "Maya" }); // × FAILS
    expect(obj).toEqual({ name: "Maya" }); // √ PASSES
});
```

Core Assignments

Sum of Odd

Create a function `sumOfOdd` which takes an array of integers as its argument and returns the sum of all the odd integers in the array.

E.g.

```
sumOfOdd([1, 7, 2]) = 8
```

Write some unit tests to ensure your function works correctly.

Great Expectations

Create unit tests for the Book and Author classes you created in the *Classic Literature* assignment. You should test **all** the features specified in that assignment (e.g. a test for "Books should have an Author").

You will need to install jest and export your Book and Author classes to be imported to your test file.

Extension Assignments

Ways to Make

Create a function waysToMake which, given a target value and an array of unique numbers, calculates the number of different ways to sum the numbers to make the target value. All targets and numbers must be positive integers.

E.g.

```
waysToMake(4, [2, 1]) = 3:
2 + 2,
2 + 1 + 1,
1 + 1 + 1 + 1
```

Write some unit tests to ensure your function works.

Hint: if you haven't used recursion before, research it and attempt some simpler problems before moving onto this one. It's tricky.

Additional Resources

- The Jest docs have a beginner-friendly section