JASMINE – UNIT TESTING

ROHAN RAJORE

JASMINE INTRO

LESSON 1

WHA IS JASMINE?

- Jasmine is a behavior-driven testing framework for JavaScript programming language.
- It’s a bunch of tools that you can use to test JavaScript code.

1. LEARNING THE SYNTAX

- Jasmine tests are primarily two parts: describe blocks and it blocks. Let’s see how this works.

```
describe('JavaScript addition operator', function () {
  it('adds two numbers together', function () {
    expect(1 + 2).toEqual(3);
  });
});
```

LEARNING THE SYNTAX

- Both the describe and it functions take two parameters: a text string and a function. Most test frameworks try to read as much like English as possible, and you can see this with Jasmine.
- First, notice that the string passed to describe and the string passed to it form a sentence (of sorts): “JavaScript addition operator adds two numbers together.” Then, we go on to show how.
- Inside that it block, you can write all the setup code you need for your test. We don’t need any for this simple example. Once you’re ready to write the actual test code, you’ll start with the expect function, passing it whatever you are testing.
- Notice how this forms a sentence as well: we “expect 1 + 2 to equal 3.”

2. SETTING UP THE PROJECT

- Download the standalone version of Jasmine.
- You’ll find the actual Jasmine framework files in the lib folder. If you prefer to structure your projects differently, please do so; but we’re going to keep this for now.
- There’s actually some sample code wired up in this project template. The “actual” JavaScript (the code we want to test) can be found in the src subdirectory; we’ll be putting ours there shortly. The testing code—the specs—go in the spec folder. Don’t worry about the SpecHelper.js file just yet; we’ll come back to that.
- That SpecRunner.html file is what runs the tests in a browser. Open it up (and check the “passed” checkbox in the upper right corner), and you should see something like this:

3. WRITING THE TEST

- describe("Convert library", function () {
- describe("distance converter", function () {
- });
- describe("volume converter", function () {
- });
- });
- We start with this; we're testing our Convert library. You'll notice that we're nesting describe statements here. This is perfectly legal. It's actually a great way to test seperate functionality chunks of the same codebase. Instead of two seperate describe calls for Convert library's distance conversions and volume conversions, we can have a more descriptive suite of tests like this.

DESCRIBE, IT AND EXPECT

- **An Example to Test**
- First, let’s create a simple function and test its behavior. It’ll say hello to the *entire world*. It could look something like this:
- function helloWorld() {
- return "Hello world!";
- }
- You’re pretty sure that this works, but you want to test it with Jasmine to see what it thinks. Start by saving this in the *src* directory as *hello.js*. Open up your *SpecRunner.html* file to include it:
- <!-- put this code somewhere in the <head>... -->
- <script type="text/javascript" src="src/hello.js"></script>

DESCRIBE, IT AND EXPECT

- Next is the Jasmine part. Get ready to get your money’s worth for this book.
- Make a file that includes the following code:

```
describe("Hello world", function() {
it("says hello", function() {
expect(helloWorld()).toEqual("Hello world!");
  });
});
```

- describe("Hello world"… is what is called a *suite*. The name of the suite (“Hello world” in this case) typically defines a component of your application; this could be a class, a function, or maybe something else fun. This suite is called “Hello world”; it’s a string of English, not code.

DESCRIBE, IT AND EXPECT

- Inside of that *suite* (technically, inside of an anonymous function), is the it() block. This is called a *specification*, or a *spec* for short. It’s a JavaScript function that says what some small piece of your component should do. It says it in plain English (“says hello”) and in code. For each suite, you can have any number of specs for the tests you want to run.
- In this case, you’re testing if helloWorld() does indeed return "Hello world!". This check is called a *matcher*. Jasmine includes a number of predefined matchers, but you can also define your own

MATCHERS

LESSON 2

Equality: toEqual

- Perhaps the simplest matcher in Jasmine is toEqual. It simply checks if two things are equal (and *not* necessarily the same exact object).
- The following expect functions will pass:

```
expect(true).toEqual(true);
expect([1, 2, 3]).toEqual([1, 2, 3]);
expect({}).toEqual({});
```

- Here are some examples of toEqual that will fail:

```
expect(5).toEqual(12);
expect([1, 2, 3, 4]).toEqual([1, 2, 3]);
expect(true).toEqual(100);
```

Identity: toBe

- At first, the toBe matcher looks a lot like the toEqual matcher, but it’s not exactly the same. toBe checks if two things are *the same object*, not just if they are equivalent.
- Here’s an example spec that illustrates the difference between toEqual and toBe:

```
var spot = { species: "Border Collie" };
var cosmo = { species: "Border Collie" };
expect(spot).toEqual(cosmo); // success; equivalent
expect(spot).toBe(cosmo); // failure; not the same object
expect(spot).toBe(spot); // success; the same object
```

- We see that, although spot and cosmo look really similar and are equal, *they aren’t the same object*. Because of that, they evaluate as *equal*, not *the same*.

Identity: toBe

- The same is also true for arrays:

```
var arr = [1, 2, 3];
expect(arr).toEqual([1, 2, 3]); // success; equivalent
expect(arr).toBe([1, 2, 3]); // failure; not the same array
```

- You might notice that toBe works for primitives (numbers, Booleans, strings). This is because JavaScript’s === operator evaluates primitives as the same entity. Using toBe is essentially using the === operator.
- Use toEqual when checking the equivalence of primitive types, even if toBe will work.

Yes or No? toBeTruthy, toBeFalsy

- To test if something evaluates to true, you use the toBeTruthy matcher:

```
expect(true).toBeTruthy();
expect(12).toBeTruthy();
expect({}).toBeTruthy();
```

- Likewise, to test if something evaluates to false, you use toBeFalsy:

```
expect(false).toBeFalsy();
expect(null).toBeFalsy();
expect("").toBeFalsy();
```

- Note that Jasmine's evaluation of truthy and falsy are identical to JavaScript's. This means that true is truthy, but so is "Hello world", or the number 12, or an object. It's useful to think of all the things that are falsy, and then everything else as truthy.

Negate other Matchers with not

- It’s frequently useful to reverse Jasmine’s matchers to make sure that they aren’t true.
- To do that, simply prefix things with .not:

```
expect(foo).not.toEqual(bar);
expect("Hello planet").not.toContain("world");
```

Check If an Element Is Present with toContain

- Sometimes you want to verify that an element is a member of an array, *somewhere*. To do that, you can use the toContain matcher:

```
expect([1, 2, 3, 4]).toContain(3);
expect(["Penguin", "Turtle", "Pig", "Duck"]).toContain("Duck");
```

- Note that toContain doesn't check if the array contains *the exact same object*, so the following example will succeed:

```
var dog = { name: "Fido" };
expect([ { name: "Spike" }, { name: "Fido" }, { name: "Spot" } ]).toContain(dog);
```

- The toContain matcher also works in strings, as we saw in the first example of this book:

```
expect("Hello world").toContain("world");
expect(favoriteCandy).not.toContain("Almond");
```

Is It Defined? toBeDefined, toBeUndefined

- As with truthiness and falsiness, there are matchers to check if something is defined or undefined.
- Here are a few examples to demonstrate how these matchers work:

```
var somethingUndefined;
expect("Hello!").toBeDefined(); // success
expect(null).toBeDefined(); // success
expect(somethingUndefined).toBeDefined(); // failure
var somethingElseUndefined;
expect(somethingElseUndefined).toBeUndefined(); // success
expect(12).toBeUndefined(); // failure
expect(null).toBeUndefined(); // failure
```

Nullness: toBeNull

- The toBeNull matcher is fairly straightforward. If you hadn’t guessed by now, it checks
- if something is null:

```
expect(null).toBeNull(); // success
expect(false).toBeNull(); // failure
expect(somethingUndefined).toBeNull(); // failure
```

- Fairly simple!

Is It NaN? toBeNaN

- Like toBeNull, toBeNaN checks if something is NaN:

```
expect(5).not.toBeNaN(); // success
expect(0 / 0).toBeNaN(); // success
expect(parseInt("hello")).toBeNaN(); // success
```

- This is different from JavaScript’s built-in isNaN function. The built-in isNaN will return true for many nonnumber types, such as nonnumeric strings, objects, and arrays. Jasmine’s will be positive only if it’s the NaN value.

Comparators: toBeGreaterThan, toBeLessThan

- The toBeGreaterThan and toBeLessThan matchers check if something is greater than or less than something else. All of these will pass:

```
expect(8).toBeGreaterThan(5);
expect(5).toBeLessThan(12);
expect("a").toBeLessThan("z"); // Notice that it works for strings too!
```

Nearness: toBeCloseTo

- toBeCloseTo allows you to check if a number is close to another number, given a certain amount of decimal precision as the second argument.
- If you want to make sure that a variable is close to 12.3 within one decimal point, you'd code it like this:

```
expect(12.34).toBeCloseTo(12.3, 1); // success
```

- If you want it to be the same within two decimal points, you'd change the 1 to a 2. This spec will fail, though—they differ at the second decimal digit:

```
expect(12.34).toBeCloseTo(12.3, 2); // failure
```

Using toMatch with Regular Expressions

- toMatch checks if something is matched, given a regular expression. It can be passed as a regular expression *or* a string, which is then parsed as a regular expression. All of the following will succeed:

```
expect("foo bar").toMatch(/bar/);
expect("horse_ebooks.jpg").toMatch(/\w+.(jpg|gif|png|svg)/i);
expect("jasmine@example.com").toMatch("\w+@\w+\.\w+");
```

- For more on regular expressions in JavaScript, check out this very helpful article on the Mozilla Developer Network.

Checking If a Function Throws an Error with toThrow

- toThrow lets you express, “Hey, I expect this function to throw an error”:

```
var throwMeAnError = function() {
throw new Error();
};
expect(throwMeAnError).toThrow();
```

- You can use this with anonymous functions too, which can be more useful. For example, let’s say there’s a function that should throw an exception with bad input, like so:

```
calculate("BAD INPUT"); // This should throw some exciting exception
```

- To test that, we use Jasmine like so:

```
expect(function() {
calculate("BAD INPUT");
}).toThrow();
```

- Whether you use an anonymous or named function, you still need to pass a function because Jasmine will call it when running your specs.

Custom Matchers

- Let’s say you want to add a matcher called toBeLarge, which checks if a number is greater than 100. At the very top of a file (or at the top of a describe), you can add the following:

```
beforeEach(function() {
  this.addMatchers({
    toBeLarge: function() {
      this.message = function() {
        return "Expected " + this.actual + " to be large";
      };
      return this.actual > 100;
    }
  });
});
```

Custom Matchers

- Now we can do the following in our specs:

```
expect(5).toBeLarge(); // failure
expect(200).toBeLarge(); // success
expect(12).not.toBeLarge(); // success
```

MORE JASMINE FEATURES

LESSON 3

Before and After

- Another useful feature of Jasmine is actually a twofer: beforeEach and afterEach. They allow you to execute some code—you guessed it—before and after each spec. This can be very useful for factoring out common code or cleaning up variables after tests. To execute some code *before* every spec, simply put it in a beforeEach. Note that you have to scope variables properly in order to have them throughout each spec:

```
describe("employee", function() {
var employee; // Note the scoping of this variable.
beforeEach(function() {
  employee = new Employee;
});
  it("has a name", function() {
    expect(employee.name).toBeDefined();
  });
  it("has a role", function() {
   expect(employee.role).toBeDefined();
  });
});
```

Nested Suites

- As your code gets more complex, you might want to organize your suites into groups, subgroups sub-subgroups, and so on. Jasmine makes it very easy for you to do that by simply nesting the specs. Put a describe block inside another describe block like this:

```
describe("chat", function() {
  describe("buddy list", function() {
    it("contains a list of users", function() {
      expect(chat.buddyList instanceof Array).toBeTruthy();
      expect(chat.buddyList[0] instanceof chat.User).toBeTruthy(); }); });
  describe("messages object", function() {
    it("contains a sender and a body", function() {
      var message = new chat.Message;
      expect(message.body).toEqual("");
      expect(message.sender instanceof chat.User).toBeTruthy(); }); }); });
```

Skipping Specs and Suites

- When you code specs, sometimes you might want to skip a few. Maybe a spec isn’t finished; maybe it’s too slow; maybe you’re just not in the mood to see one red spec in a sea of green ones.
- Instead of commenting specs out, just add an x before the word it, and the code will behave as though you had commented the spec out. In the following example, we haven’t finished the spec that tests a double rainbow’s brightness, so we’ll x it out:

```
describe("double rainbow", function() {
it("is all the way across the sky", function() {
// This spec will run. });
xit("is so bright", function() {
// Because we've x'd this spec out, it won't run. });
});
```

Skipping Specs and Suites

- You can also x out entire suites, which will skip all of the specs inside. In this example, none of the Leonardo DiCaprio specs will run:

```
xdescribe("Leonardo DiCaprio", function() {
  it("is not named after Leonardo da Vinci", function() {
    expect("Leonardo DiCaprio").not.toEqual("Leonardo da Vinci");
  });
  it("is in the movie Inception", function() {
    expect(Inception.cast).toContain("Leonardo DiCaprio");
  });
});
```

Skipping Specs and Suites

- Because all of your specs and suites are defined in functions, you can skip all specs and suites after a certain point in the function with a clever return. Because return halts a function’s execution, you can stop some specs from running, like so:

```
describe("I'm only going to run SOME of these", function() {
  it("will run this spec", function() {});
  it("will run this spec", function() {});
  return; // This will stop the function from doing anything else.
  it("will not run this spec", function() {});
  it("will not run this spec", function() {});
});
```

Matching Class Names

- Sometimes you don’t care what the value is; you care what *type* it is. To indicate this, use jasmine.any.

Let’s say that we create a function called rand that generates a random number. We want to make sure that, no matter what, it returns a number. We don’t really care *what* the number is—we just care that it’s a number:

```
expect(rand()).toEqual(jasmine.any(Number));
```

- Of course, this doesn’t just work for numbers. All of these specs succeed:

```
expect("Hello world").toEqual(jasmine.any(String));
expect({}).toEqual(jasmine.any(Object));
expect(new MyObject).toEqual(jasmine.any(MyObject));
```

- These are incredibly useful when you want your results to be of a certain type but don’t need to be more specific than that.