1. Jasmine is a behavior-driven development **framework for testing JavaScript code**.
2. **Suites**: describe Your Tests. A test suite begins with a call to the global Jasmine function describe with two parameters. A string and a function. String is the name of the spec suite and function is the executable code.

Ex : describe("A suite", function() {

});

1. **Specs** : Specs are defined by calling the global Jasmine function it. The string is a title for this spec and the function is the spec, or test. A spec contains one or more expectations that test the state of the code under test.  
   Ex: it("and so is a spec", function() {

expect(a).toBe(true);

});

1. Since describe and it blocks are functions, they can contain any executable code necessary to implement the test. JavaScript scoping rules apply, so variables declared in a describe are available to any it block inside the suite.

Ex : describe("A suite is just a function", function() {

var a;

it("and so is a spec", function() {

a = true;

expect(a).toBe(true);

});

});

1. **Expectations** : Expectations are built with the function expect which takes a value, called the actual. It is chained with a Matcher function, which takes the expected value.
2. **Matchers** : Each matcher implements a boolean comparison between the actual value and the expected value.

Any matcher can evaluate to a negative assertion by chaining the call to expect with a not before calling the matcher.

Ex: it("and has a positive case ", function() {

expect(true).toBe(true);

});

it("and can have a negative case", function() {

expect(false).not.toBe(true);

});

1. **Included Matchers** : jasmine has a many set of matchers. Below are the list.

Ex:

expect(a).toBe(b); //for numerical or values

expect(a).not.toBe(null); //for numerical or values

expect(a).toEqual(12); //for numericals

expect(array).toEqual(someArray); // for arrays

expect(message).toMatch(/bar/); // for regular expressions

expect(message).toMatch('bar'); // for regular expressions expect(message).not.toMatch(/quux/); // for regular expressions

expect(a.foo).toBeDefined();

expect(a.bar).not.toBeDefined();

expect(a.foo).not.toBeUndefined();

expect(a.bar).toBeUndefined();

expect(null).toBeNull();

expect(a).toBeNull();

expect(foo).not.toBeNull();

expect(foo).toBeTruthy();

expect(a).not.toBeTruthy();

expect(a).toBeFalsy();

expect(foo).not.toBeFalsy();

expect(a).toContain('bar');

expect(a).not.toContain('quux');

expect(e).toBeLessThan(pi);

expect(pi).not.toBeLessThan(e);

expect(pi).toBeGreaterThan(e);

expect(e).not.toBeGreaterThan(pi);

expect(pi).not.toBeCloseTo(e, 2);

expect(pi).toBeCloseTo(e, 0);

it("The 'toThrow' matcher is for testing if a function throws an exception", function() {

var foo = function() {

return 1 + 2;

};

var bar = function() {

return a + 1;

};

expect(foo).not.toThrow();

expect(bar).toThrow();

});

1. **Writing a custom matcher :** There is also the ability to write custom matchers for when a project’s domain calls for specific assertions that are not included.

toBeGoofy takes an optional expected argument, so define it here if not passed in.

toBeGoofy tests for equality of the actual’s hyuk property to see if it matches the expectation.

1. **Failure Messages :** If left undefined, the expectation will attempt to craft a failure message for the matcher. However, if the message property is set, it will be used for a failed expectation.
2. **Grouping Related Specs with describe :** The describe function is for grouping related specs. The string parameter is for naming the collection of specs, and will be concatenated with specs to make a spec’s full name. This aids in finding specs in a large suite. If you name them well, your specs read as full sentences in traditional BDD style.

**BeforeEach and afterEach :** To help a test suite DRY up any duplicated setup and teardown code, Jasmine provides the global beforeEach and afterEach functions. As the name implies the beforeEach function is called once before each spec in the describe is run and the afterEach function is called once after each spec.

Here is the same set of specs written a little differently. The variable under test is defined at the top-level scope — the describe block — and initialization code is moved into a beforeEach function. The afterEach function resets the variable before continuing.

Ex : describe("A spec (with setup and tear-down)", function() {

var foo;

beforeEach(function() {

foo = 0;

foo += 1;

});

afterEach(function() {

foo = 0;

});

it("is just a function, so it can contain any code", function() {

expect(foo).toEqual(1);

});

it("can have more than one expectation", function() {

expect(foo).toEqual(1);

expect(true).toEqual(true);

});

});

1. **Nesting describe Blocks :** Calls to describe can be nested, with specs defined at any level.

Before a spec is executed, Jasmine walks down the tree executing each beforeEach function in order. After the spec is executed, Jasmine walks through the afterEach functions similarly.

**Disabling Specs and Suites :** Suites and specs can be disabled with the xdescribe and xit functions, respectively. These suites and specs are skipped when run and thus their results will not appear in the results.

Ex: xdescribe("A spec", function() {

var foo;

beforeEach(function() {

foo = 0;

foo += 1;

});

xit("is just a function, so it can contain any code", function() {

expect(foo).toEqual(1);

});

});

1. **Spies :** Jasmine’s test doubles are called spies. A spy can stub any function and tracks calls to it and all arguments. There are special matchers for interacting with spies.

The toHaveBeenCalled matcher will return true if the spy was called. The toHaveBeenCalledWith matcher will return true if the argument list matches any of the recorded calls to the spy.

Ex : describe("A spy", function() {

var foo, bar = null;

beforeEach(function() {

foo = {

setBar: function(value) {

bar = value;

}

};

spyOn(foo, 'setBar');

foo.setBar(123);

foo.setBar(456, 'another param');

});

it("tracks that the spy was called", function() {

expect(foo.setBar).toHaveBeenCalled();

});

it("tracks its number of calls", function() {

expect(foo.setBar.calls.length).toEqual(2);

});

it("tracks all the arguments of its calls", function() {

expect(foo.setBar).toHaveBeenCalledWith(123);

expect(foo.setBar).toHaveBeenCalledWith(456, 'another param');

});

it("allows access to the most recent call", function() {

expect(foo.setBar.mostRecentCall.args[0]).toEqual(456);

});

it("allows access to other calls", function() {

expect(foo.setBar.calls[0].args[0]).toEqual(123);

});

it("stops all execution on a function", function() {

expect(bar).toBeNull();

});

});

1. **Spies: andCallThrough :** By chaining the spy with andCallThrough, the spy will still track all calls to it but in addition it will delegate to the actual implementation.

Ex: spyOn(foo, 'getBar').andCallThrough();

1. **Spies: andReturn :** By chaining the spy with andReturn, all calls to the function will return a specific value.

Ex: spyOn(foo, 'getBar').andReturn(745);

1. **Spies: andCallFake :** By chaining the spy with andCallFake, all calls to the spy will delegate to the supplied function.  
   Ex : spyOn(foo, 'getBar').andCallFake(function() {

return 1001;

});

1. **Spies: createSpy :** When there is not a function to spy on, jasmine.createSpy can create a “bare” spy. This spy acts as any other spy – tracking calls, arguments, etc. But there is no implementation behind it. Spies are JavaScript objects and can be used as such.

Ex **:**  beforeEach(function() {

whatAmI = jasmine.createSpy('whatAmI');

whatAmI("I", "am", "a", "spy");

});

1. **Spies: createSpyObj :** In order to create a mock with multiple spies, use jasmine.createSpyObj and pass an array of strings. It returns an object that has a property for each string that is a spy.

Ex : beforeEach(function() {

tape = jasmine.createSpyObj('tape', ['play', 'pause', 'stop', 'rewind']);

tape.play();

tape.pause();

tape.rewind(0);

});

1. **Matching Anything with jasmine.any :** jasmine.any takes a constructor or “class” name as an expected value. It returns true if the constructor matches the constructor of the actual value.

Ex : describe("jasmine.any", function() {

it("matches any value", function() {

expect({}).toEqual(jasmine.any(Object));

expect(12).toEqual(jasmine.any(Number));

});

1. **Mocking the JavaScript Clock :** The Jasmine Mock Clock is available for test suites that need the ability to use setTimeout or setInterval callbacks. It makes the timer callbacks synchronous, thus making them easier to test.

It is installed with a call to jasmine.Clock.useMock in a spec or suite that needs to call the timer functions.

Calls to any registered callback are triggered when the clock is ticked forward via the jasmine.Clock.tick function, which takes a number of milliseconds.

Ex: describe("Manually ticking the Jasmine Mock Clock", function() {

var timerCallback;

beforeEach(function() {

timerCallback = jasmine.createSpy('timerCallback');

jasmine.Clock.useMock();

});

it("causes a timeout to be called synchronously", function() {

setTimeout(function() {

timerCallback();

}, 100);

expect(timerCallback).not.toHaveBeenCalled();

jasmine.Clock.tick(101);

expect(timerCallback).toHaveBeenCalled();

});

it("causes an interval to be called synchronously", function() {

setInterval(function() {

timerCallback();

}, 100);

expect(timerCallback).not.toHaveBeenCalled();

jasmine.Clock.tick(101);

expect(timerCallback.callCount).toEqual(1);

jasmine.Clock.tick(50);

expect(timerCallback.callCount).toEqual(1);

jasmine.Clock.tick(50);

expect(timerCallback.callCount).toEqual(2);

});

});

1. **Asynchronous Support :** Specs are written by defining a set of blocks with calls to runs, which usually finish with an asynchronous call.

The waitsFor block takes a latch function, a failure message, and a timeout.

1. **Testing Directive**:
2. <https://www.sitepoint.com/angular-testing-tips-testing-directives/>
3. Scope of directive can be fetched as below.

**var** element = angular.*element*(**'<aab-auto-complete-item display-mode="heading"></aab-auto-complete-item>'**);

**var** directiveElement = $compile(element)($scope);

**$scope.$apply();**

**var** scope = directiveElement.isolateScope();

1. Controller can be fetched as below.

**var** controller = element.controller(**"aabAutoCompleteItem"**);

**Note** : **aabAutoCompleteItem** is directive name.