Powered by

Jump To: ajax.js boot.js custom_boot.js custom_equality.js custom_matcher.js custom_reporter.js focused_specs.js introduction.js node.js python_egg.py ruby_gem.rb upgrading.js

introduction.js

Jasmine is a behavior-driven development framework for testing JavaScript code. It does not depend on any other JavaScript frameworks. It does not require a DOM. And it has a clean, obvious syntax so that you can easily write tests. This guide is running against Jasmine version 2.4.1.

Standalone Distribution

The releases page has links to download the standalone distribution, which contains everything you need to start running Jasmine. After downloading a particular version and unzipping, opening `SpecRunner.html` will run the included specs. You'll note that both the source files and their respective specs are linked in the `<head>` of the `SpecRunner.html`. To start using Jasmine, replace the source/spec files with your own.

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. The string is a name or title for a spec suite - usually what is being tested. The function is a block of code that implements the suite.

```
describe("A suite", function() {
  it("contains spec with an expectation", function() {
    expect(true).toBe(true);
  });
});
```

Specs

Specs are defined by calling the global Jasmine function `it`, which, like `describe` takes a string and a function. The string is the title of the spec and the function is the spec, or test. A spec contains one or more expectations that test the state of the code. An expectation in Jasmine is an assertion that is either true or false. A spec with all true expectations is a passing spec. A spec with one or more false expectations is a failing spec.

It's Just Functions

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.

```
describe("A suite is just a function", function() {
  var a;

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

    expect(a).toBe(true);
  });
});
```

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.

Matchers

Each matcher implements a boolean comparison between the actual value and the expected value. It is responsible for reporting to Jasmine if the expectation is true or false. Jasmine will then pass or fail the spec.

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

```
describe("The 'toBe' matcher compares with ===", function() {

  it("and has a positive case", function() {
    expect(true).toBe(true);
  });

  it("and can have a negative case", function() {
    expect(false).not.toBe(true);
  });
});
```

Included Matchers

Jasmine has a rich set of matchers included. Each is used here - all expectations and specs pass. There is also the ability to write custom matchers for when a project's domain calls for specific assertions that are not included below.

```
describe("Included matchers:", function() {

  it("The 'toBe' matcher compares with ===", function() {
    var a = 12;
    var b = a;

    expect(a).toBe(b);
    expect(a).not.toBe(null);
  });

  describe("The 'toEqual' matcher", function() {

    it("works for simple literals and variables", function() {
      var a = 12;
      expect(a).toEqual(12);
    });

    it("should work for objects", function() {
```

introduction.js

```
    var foo = {
      a: 12,
      b: 34
    };
    var bar = {
      a: 12,
      b: 34
    };
    expect(foo).toEqual(bar);
  });
});

it("The 'toMatch' matcher is for regular expressions", function() {
  var message = "foo bar baz";

  expect(message).toMatch(/bar/);
  expect(message).toMatch("bar");
  expect(message).not.toMatch(/quux/);
});

it("The 'toBeDefined' matcher compares against `undefined`", function() {
  var a = {
    foo: "foo"
  };

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

it("The `toBeUndefined` matcher compares against `undefined`", function() {
  var a = {
    foo: "foo"
  };

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

it("The 'toBeNull' matcher compares against null", function() {
  var a = null;
  var foo = "foo";

  expect(null).toBeNull();
  expect(a).toBeNull();
  expect(foo).not.toBeNull();
});

it("The 'toBeTruthy' matcher is for boolean casting testing", function() {
  var a, foo = "foo";

  expect(foo).toBeTruthy();
  expect(a).not.toBeTruthy();
});

it("The 'toBeFalsy' matcher is for boolean casting testing", function() {
  var a, foo = "foo";

  expect(a).toBeFalsy();
  expect(foo).not.toBeFalsy();
});

describe("The 'toContain' matcher", function() {
  it("works for finding an item in an Array", function() {
    var a = ["foo", "bar", "baz"];

    expect(a).toContain("bar");
    expect(a).not.toContain("quux");
  });

  it("also works for finding a substring", function() {
    var a = "foo bar baz";

    expect(a).toContain("bar");
    expect(a).not.toContain("quux");
  });
});

it("The 'toBeLessThan' matcher is for mathematical comparisons", function() {
  var pi = 3.1415926,
    e = 2.78;

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

it("The 'toBeGreaterThan' matcher is for mathematical comparisons", function() {
  var pi = 3.1415926,
    e = 2.78;

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

it("The 'toBeCloseTo' matcher is for precision math comparison", function() {
  var pi = 3.1415926,
    e = 2.78;

  expect(pi).not.toBeCloseTo(e, 2);
  expect(pi).toBeCloseTo(e, 0);
```

introduction.js

```
  });

  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;
    };
    var baz = function() {
      throw 'what';
    };

    expect(foo).not.toThrow();
    expect(bar).toThrow();
    expect(baz).toThrow('what');
  });

  it("The 'toThrowError' matcher is for testing a specific thrown exception", function() {
    var foo = function() {
      throw new TypeError("foo bar baz");
    };

    expect(foo).toThrowError("foo bar baz");
    expect(foo).toThrowError(/bar/);
    expect(foo).toThrowError(TypeError);
    expect(foo).toThrowError(TypeError, "foo bar baz");
  });
});
```

Manually failing a spec with `fail`

The `fail` function causes a spec to fail. It can take a failure message or an Error object as a parameter.

```
describe("A spec using the fail function", function() {
  var foo = function(x, callBack) {
    if (x) {
      callBack();
    }
  };

  it("should not call the callBack", function() {
    foo(false, function() {
      fail("Callback has been called");
    });
  });
});
```

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.

```
describe("A spec", function() {
  it("is just a function, so it can contain any code", function() {
    var foo = 0;
    foo += 1;

    expect(foo).toEqual(1);
  });

  it("can have more than one expectation", function() {
    var foo = 0;
    foo += 1;

    expect(foo).toEqual(1);
    expect(true).toEqual(true);
  });
});
```

Setup and Teardown

To help a test suite DRY up any duplicated setup and teardown code, Jasmine provides the global `beforeEach`, `afterEach`, `beforeAll`, and `afterAll` functions.

As the name implies, the `beforeEach` function is called once before each spec in the `describe` in which it is called, 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.

```
describe("A spec using beforeEach and afterEach", function() {
  var foo = 0;

  beforeEach(function() {
    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);
  });
});
```

The `beforeAll` function is called only once before all the specs in `describe` are run, and the `afterAll` function is called after all specs finish. These functions can be used to speed up test suites with expensive setup and teardown.

However, be careful using `beforeAll` and `afterAll`! Since they are not reset between specs, it is easy to accidentally leak state

```
describe("A spec using beforeAll and afterAll", function() {
  var foo;

  beforeAll(function() {
    foo = 1;
  });
```

introduction.js

between your specs so that they erroneously pass or fail.

```
  afterAll(function() {
    foo = 0;
  });

  it("sets the initial value of foo before specs run", function() {
    expect(foo).toEqual(1);
    foo += 1;
  });

  it("does not reset foo between specs", function() {
    expect(foo).toEqual(2);
  });
});
```

The `this` keyword

Another way to share variables between a `beforeEach`, `it`, and `afterEach` is through the `this` keyword. Each spec's `beforeEach`/`it`/`afterEach` has the `this` as the same empty object that is set back to empty for the next spec's `beforeEach`/`it`/`afterEach`.

```
describe("A spec", function() {
  beforeEach(function() {
    this.foo = 0;
  });

  it("can use the `this` to share state", function() {
    expect(this.foo).toEqual(0);
    this.bar = "test pollution?";
  });

  it("prevents test pollution by having an empty `this` created for the next spec", function() {
    expect(this.foo).toEqual(0);
    expect(this.bar).toBe(undefined);
  });
});
```

Nesting `describe` Blocks

Calls to `describe` can be nested, with specs defined at any level. This allows a suite to be composed as a tree of functions. 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.

```
describe("A spec", 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);
  });

  describe("nested inside a second describe", function() {
    var bar;

    beforeEach(function() {
      bar = 1;
    });

    it("can reference both scopes as needed", function() {
      expect(foo).toEqual(bar);
    });
  });
});
```

Disabling Suites

Suites can be disabled with the `xdescribe` function. These suites and any specs inside them are skipped when run and thus their results will show as pending.

```
xdescribe("A spec", function() {
  var foo;

  beforeEach(function() {
    foo = 0;
    foo += 1;
  });

  it("is just a function, so it can contain any code", function() {
    expect(foo).toEqual(1);
  });
});
```

Pending Specs

Pending specs do not run, but their names will show up in the results as `pending`.

```
describe("Pending specs", function() {
```

Any spec declared with `xit` is marked as pending.

```
  xit("can be declared 'xit'", function() {
    expect(true).toBe(false);
  });
```

Any spec declared without a function body will also be marked pending in results.

```
  it("can be declared with 'it' but without a function");
```

And if you call the function `pending` anywhere in the spec body, no matter the expectations, the spec will be marked pending. A string passed to `pending` will be treated as a reason and displayed when the suite finishes.

```
  it("can be declared by calling 'pending' in the spec body", function() {
    expect(true).toBe(false);
    pending('this is why it is pending');
  });
});
```

introduction.js

Spies

Jasmine has test double functions called spies. A spy can stub any function and tracks calls to it and all arguments. A spy only exists in the `describe` or `it` block in which it is defined, and will be removed after each spec. There are special matchers for interacting with spies. *This syntax has changed for Jasmine 2.0*

```
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');
  });
```

The `toHaveBeenCalled` matcher will return true if the spy was called.

```
  it("tracks that the spy was called", function() {
    expect(foo.setBar).toHaveBeenCalled();
  });
```

The `toHaveBeenCalledTimes` matcher will pass if the spy was called the specified number of times.

```
  it("tracks that the spy was called x times", function() {
    expect(foo.setBar).toHaveBeenCalledTimes(2);
  });
```

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

```
  it("tracks all the arguments of its calls", function() {
    expect(foo.setBar).toHaveBeenCalledWith(123);
    expect(foo.setBar).toHaveBeenCalledWith(456, 'another param');
  });

  it("stops all execution on a function", function() {
    expect(bar).toBeNull();
  });
});
```

Spies: `and.callThrough`

By chaining the spy with `and.callThrough`, the spy will still track all calls to it but in addition it will delegate to the actual implementation.

```
describe("A spy, when configured to call through", function() {
  var foo, bar, fetchedBar;

  beforeEach(function() {
    foo = {
      setBar: function(value) {
        bar = value;
      },
      getBar: function() {
        return bar;
      }
    };

    spyOn(foo, 'getBar').and.callThrough();

    foo.setBar(123);
    fetchedBar = foo.getBar();
  });

  it("tracks that the spy was called", function() {
    expect(foo.getBar).toHaveBeenCalled();
  });

  it("should not affect other functions", function() {
    expect(bar).toEqual(123);
  });

  it("when called returns the requested value", function() {
    expect(fetchedBar).toEqual(123);
  });
});
```

Spies: `and.returnValue`

By chaining the spy with `and.returnValue`, all calls to the function will return a specific value.

```
describe("A spy, when configured to fake a return value", function() {
  var foo, bar, fetchedBar;

  beforeEach(function() {
    foo = {
      setBar: function(value) {
        bar = value;
      },
      getBar: function() {
        return bar;
      }
    };

    spyOn(foo, "getBar").and.returnValue(745);

    foo.setBar(123);
    fetchedBar = foo.getBar();
  });

  it("tracks that the spy was called", function() {
    expect(foo.getBar).toHaveBeenCalled();
  });

  it("should not affect other functions", function() {
    expect(bar).toEqual(123);
  });

  it("when called returns the requested value", function() {
    expect(fetchedBar).toEqual(745);
  });
```

introduction.js

Spies: `and.returnValues`

By chaining the spy with `and.returnValues`, all calls to the function will return specific values in order until it reaches the end of the return values list, at which point it will return undefined for all subsequent calls.

```
});

describe("A spy, when configured to fake a series of return values", function() {
  var foo, bar;

  beforeEach(function() {
    foo = {
      setBar: function(value) {
        bar = value;
      },
      getBar: function() {
        return bar;
      }
    };

    spyOn(foo, "getBar").and.returnValues("fetched first", "fetched second");

    foo.setBar(123);
  });

  it("tracks that the spy was called", function() {
    foo.getBar(123);
    expect(foo.getBar).toHaveBeenCalled();
  });

  it("should not affect other functions", function() {
    expect(bar).toEqual(123);
  });

  it("when called multiple times returns the requested values in order", function() {
    expect(foo.getBar()).toEqual("fetched first");
    expect(foo.getBar()).toEqual("fetched second");
    expect(foo.getBar()).toBeUndefined();
  });
});
```

Spies: `and.callFake`

By chaining the spy with `and.callFake`, all calls to the spy will delegate to the supplied function.

If the function being spied on receives arguments that the fake needs, you can get those as well

```
describe("A spy, when configured with an alternate implementation", function() {
  var foo, bar, fetchedBar;

  beforeEach(function() {
    foo = {
      setBar: function(value) {
        bar = value;
      },
      getBar: function() {
        return bar;
      }
    };

    spyOn(foo, "getBar").and.callFake(function(arguments, can, be, received) {
      return 1001;
    });

    foo.setBar(123);
    fetchedBar = foo.getBar();
  });

  it("tracks that the spy was called", function() {
    expect(foo.getBar).toHaveBeenCalled();
  });

  it("should not affect other functions", function() {
    expect(bar).toEqual(123);
  });

  it("when called returns the requested value", function() {
    expect(fetchedBar).toEqual(1001);
  });
});
```

Spies: `and.throwError`

By chaining the spy with `and.throwError`, all calls to the spy will `throw` the specified value as an error.

```
describe("A spy, when configured to throw an error", function() {
  var foo, bar;

  beforeEach(function() {
    foo = {
      setBar: function(value) {
        bar = value;
      }
    };

    spyOn(foo, "setBar").and.throwError("quux");
  });

  it("throws the value", function() {
    expect(function() {
      foo.setBar(123)
    }).toThrowError("quux");
  });
});
```

Spies: `and.stub`

When a calling strategy is used for a spy, the original stubbing behavior can be returned at any time with `and.stub`.

```
describe("A spy", function() {
  var foo, bar = null;

  beforeEach(function() {
    foo = {
      setBar: function(value) {
```

introduction.js

```
        bar = value;
      }
    };

    spyOn(foo, 'setBar').and.callThrough();
  });

  it("can call through and then stub in the same spec", function() {
    foo.setBar(123);
    expect(bar).toEqual(123);

    foo.setBar.and.stub();
    bar = null;

    foo.setBar(123);
    expect(bar).toBe(null);
  });
});
```

Other tracking properties

Every call to a spy is tracked and exposed on the `calls` property.

```
describe("A spy", function() {
  var foo, bar = null;

  beforeEach(function() {
    foo = {
      setBar: function(value) {
        bar = value;
      }
    };

    spyOn(foo, 'setBar');
  });
```

`.calls.any()`: returns `false` if the spy has not been called at all, and then `true` once at least one call happens.

```
  it("tracks if it was called at all", function() {
    expect(foo.setBar.calls.any()).toEqual(false);

    foo.setBar();

    expect(foo.setBar.calls.any()).toEqual(true);
  });
```

`.calls.count()`: returns the number of times the spy was called

```
  it("tracks the number of times it was called", function() {
    expect(foo.setBar.calls.count()).toEqual(0);

    foo.setBar();
    foo.setBar();

    expect(foo.setBar.calls.count()).toEqual(2);
  });
```

`.calls.argsFor(index)`: returns the arguments passed to call number `index`

```
  it("tracks the arguments of each call", function() {
    foo.setBar(123);
    foo.setBar(456, "baz");

    expect(foo.setBar.calls.argsFor(0)).toEqual([123]);
    expect(foo.setBar.calls.argsFor(1)).toEqual([456, "baz"]);
  });
```

`.calls.allArgs()`: returns the arguments to all calls

```
  it("tracks the arguments of all calls", function() {
    foo.setBar(123);
    foo.setBar(456, "baz");

    expect(foo.setBar.calls.allArgs()).toEqual([[123],[456, "baz"]]);
  });
```

`.calls.all()`: returns the context (the `this`) and arguments passed all calls

```
  it("can provide the context and arguments to all calls", function() {
    foo.setBar(123);

    expect(foo.setBar.calls.all()).toEqual([{object: foo, args: [123], returnValue: undefined}]);
  });
```

`.calls.mostRecent()`: returns the context (the `this`) and arguments for the most recent call

```
  it("has a shortcut to the most recent call", function() {
    foo.setBar(123);
    foo.setBar(456, "baz");

    expect(foo.setBar.calls.mostRecent()).toEqual({object: foo, args: [456, "baz"], returnValue: undefined});
  });
```

`.calls.first()`: returns the context (the `this`) and arguments for the first call

```
  it("has a shortcut to the first call", function() {
    foo.setBar(123);
    foo.setBar(456, "baz");

    expect(foo.setBar.calls.first()).toEqual({object: foo, args: [123], returnValue: undefined});
  });
```

When inspecting the return from `all()`, `mostRecent()` and `first()`, the `object` property is set to the value of `this` when the spy was called.

```
  it("tracks the context", function() {
    var spy = jasmine.createSpy('spy');
    var baz = {
      fn: spy
    };
    var quux = {
      fn: spy
    };
    baz.fn(123);
```

introduction.js

```
    quux.fn(456);

    expect(spy.calls.first().object).toBe(baz);
    expect(spy.calls.mostRecent().object).toBe(quux);
  });
```

`.calls.reset()`: clears all tracking for a spy

```
  it("can be reset", function() {
    foo.setBar(123);
    foo.setBar(456, "baz");

    expect(foo.setBar.calls.any()).toBe(true);

    foo.setBar.calls.reset();

    expect(foo.setBar.calls.any()).toBe(false);
  });
});
```

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.

```
describe("A spy, when created manually", function() {
  var whatAmI;

  beforeEach(function() {
    whatAmI = jasmine.createSpy('whatAmI');

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

  it("is named, which helps in error reporting", function() {
    expect(whatAmI.and.identity()).toEqual('whatAmI');
  });

  it("tracks that the spy was called", function() {
    expect(whatAmI).toHaveBeenCalled();
  });

  it("tracks its number of calls", function() {
    expect(whatAmI.calls.count()).toEqual(1);
  });

  it("tracks all the arguments of its calls", function() {
    expect(whatAmI).toHaveBeenCalledWith("I", "am", "a", "spy");
  });

  it("allows access to the most recent call", function() {
    expect(whatAmI.calls.mostRecent().args[0]).toEqual("I");
  });
});
```

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.

```
describe("Multiple spies, when created manually", function() {
  var tape;

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

    tape.play();
    tape.pause();
    tape.rewind(0);
  });

  it("creates spies for each requested function", function() {
    expect(tape.play).toBeDefined();
    expect(tape.pause).toBeDefined();
    expect(tape.stop).toBeDefined();
    expect(tape.rewind).toBeDefined();
  });

  it("tracks that the spies were called", function() {
    expect(tape.play).toHaveBeenCalled();
    expect(tape.pause).toHaveBeenCalled();
    expect(tape.rewind).toHaveBeenCalled();
    expect(tape.stop).not.toHaveBeenCalled();
  });

  it("tracks all the arguments of its calls", function() {
    expect(tape.rewind).toHaveBeenCalledWith(0);
  });
});
```

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.

```
describe("jasmine.any", function() {
  it("matches any value", function() {
    expect({}).toEqual(jasmine.any(Object));
    expect(12).toEqual(jasmine.any(Number));
  });

  describe("when used with a spy", function() {
    it("is useful for comparing arguments", function() {
      var foo = jasmine.createSpy('foo');
      foo(12, function() {
        return true;
      });

      expect(foo).toHaveBeenCalledWith(jasmine.any(Number), jasmine.any(Function));
    });
  });
});
```

introduction.js

Matching existence with `jasmine.anything`

`jasmine.anything` returns `true` if the actual value is not `null` or `undefined`.

```
describe("jasmine.anything", function() {
  it("matches anything", function() {
    expect(1).toEqual(jasmine.anything());
  });

  describe("when used with a spy", function() {
    it("is useful when the argument can be ignored", function() {
      var foo = jasmine.createSpy('foo');
      foo(12, function() {
        return false;
      });

      expect(foo).toHaveBeenCalledWith(12, jasmine.anything());
    });
  });
});
```

Partial Matching with `jasmine.objectContaining`

`jasmine.objectContaining` is for those times when an expectation only cares about certain key/value pairs in the actual.

```
describe("jasmine.objectContaining", function() {
  var foo;

  beforeEach(function() {
    foo = {
      a: 1,
      b: 2,
      bar: "baz"
    };
  });

  it("matches objects with the expect key/value pairs", function() {
    expect(foo).toEqual(jasmine.objectContaining({
      bar: "baz"
    }));
    expect(foo).not.toEqual(jasmine.objectContaining({
      c: 37
    }));
  });

  describe("when used with a spy", function() {
    it("is useful for comparing arguments", function() {
      var callback = jasmine.createSpy('callback');

      callback({
        bar: "baz"
      });

      expect(callback).toHaveBeenCalledWith(jasmine.objectContaining({
        bar: "baz"
      }));
      expect(callback).not.toHaveBeenCalledWith(jasmine.objectContaining({
        c: 37
      }));
    });
  });
});
```

Partial Array Matching with `jasmine.arrayContaining`

`jasmine.arrayContaining` is for those times when an expectation only cares about some of the values in an array.

```
describe("jasmine.arrayContaining", function() {
  var foo;

  beforeEach(function() {
    foo = [1, 2, 3, 4];
  });

  it("matches arrays with some of the values", function() {
    expect(foo).toEqual(jasmine.arrayContaining([3, 1]));
    expect(foo).not.toEqual(jasmine.arrayContaining([6]));
  });

  describe("when used with a spy", function() {
    it("is useful when comparing arguments", function() {
      var callback = jasmine.createSpy('callback');

      callback([1, 2, 3, 4]);

      expect(callback).toHaveBeenCalledWith(jasmine.arrayContaining([4, 2, 3]));
      expect(callback).not.toHaveBeenCalledWith(jasmine.arrayContaining([5, 2]));
    });
  });
});
```

String Matching with `jasmine.stringMatching`

`jasmine.stringMatching` is for when you don't want to match a string in a larger object exactly, or match a portion of a string in a spy expectation.

```
describe('jasmine.stringMatching', function() {
  it("matches as a regexp", function() {
    expect({foo: 'bar'}).toEqual({foo: jasmine.stringMatching(/^bar$/)});
    expect({foo: 'foobarbaz'}).toEqual({foo: jasmine.stringMatching('bar')});
  });

  describe("when used with a spy", function() {
    it("is useful for comparing arguments", function() {
      var callback = jasmine.createSpy('callback');

      callback('foobarbaz');

      expect(callback).toHaveBeenCalledWith(jasmine.stringMatching('bar'));
      expect(callback).not.toHaveBeenCalledWith(jasmine.stringMatching(/^bar$/));
    });
  });
});
```

introduction.js

Custom asymmetric equality tester

When you need to check that something meets a certain criteria, without being strictly equal, you can also specify a custom asymmetric equality tester simply by providing an object that has an `asymmetricMatch` function.

```
describe("custom asymmetry", function() {
  var tester = {
    asymmetricMatch: function(actual) {
      var secondValue = actual.split(',')[1];
      return secondValue === 'bar';
    }
  };

  it("dives in deep", function() {
    expect("foo,bar,baz,quux").toEqual(tester);
  });

  describe("when used with a spy", function() {
    it("is useful for comparing arguments", function() {
      var callback = jasmine.createSpy('callback');

      callback('foo,bar,baz');

      expect(callback).toHaveBeenCalledWith(tester);
    });
  });
});
```

Jasmine Clock

This syntax has changed for Jasmine 2.0. The Jasmine Clock is available for testing time dependent code.

It is installed with a call to `jasmine.clock().install` in a spec or suite that needs to manipulate time.

Be sure to uninstall the clock after you are done to restore the original functions.

Mocking the JavaScript Timeout Functions

You can make `setTimeout` or `setInterval` synchronous executing the registered functions only once the clock is ticked forward in time.

To execute registered functions, move time forward via the `jasmine.clock().tick` function, which takes a number of milliseconds.

```
describe("Manually ticking the Jasmine Clock", function() {
  var timerCallback;

  beforeEach(function() {
    timerCallback = jasmine.createSpy("timerCallback");
    jasmine.clock().install();
  });

  afterEach(function() {
    jasmine.clock().uninstall();
  });

  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.calls.count()).toEqual(1);

    jasmine.clock().tick(50);
    expect(timerCallback.calls.count()).toEqual(1);

    jasmine.clock().tick(50);
    expect(timerCallback.calls.count()).toEqual(2);
  });
```

Mocking the Date

The Jasmine Clock can also be used to mock the current date.

If you do not provide a base time to `mockDate` it will use the current date.

```
  describe("Mocking the Date object", function(){
    it("mocks the Date object and sets it to a given time", function() {
      var baseTime = new Date(2013, 9, 23);

      jasmine.clock().mockDate(baseTime);

      jasmine.clock().tick(50);
      expect(new Date().getTime()).toEqual(baseTime.getTime() + 50);
    });
  });
});
```

Asynchronous Support

This syntax has changed for Jasmine 2.0. Jasmine also has support for running specs that require testing asynchronous operations.

Calls to `beforeAll`, `afterAll`, `beforeEach`, `afterEach`, and `it` can take an optional single argument that should be called when the async work is complete.

This spec will not start until the `done` function is called in the call to `beforeEach` above. And this spec will not complete until its

```
describe("Asynchronous specs", function() {
  var value;

  beforeEach(function(done) {
    setTimeout(function() {
      value = 0;
      done();
    }, 1);
  });

  it("should support async execution of test preparation and expectations", function(done) {
```

introduction.js

`done` is called.

By default jasmine will wait for 5 seconds for an asynchronous spec to finish before causing a timeout failure. If the timeout expires before `done` is called, the current spec will be marked as failed and suite execution will continue as if `done` was called.

If specific specs should fail faster or need more time this can be adjusted by passing a timeout value to `it`, etc.

If the entire suite should have a different timeout, `jasmine.DEFAULT_TIMEOUT_INTERVAL` can be set globally, outside of any given `describe`.

The `done.fail` function fails the spec and indicates that it has completed.

```
  value++;
  expect(value).toBeGreaterThan(0);
  done();
});


describe("long asynchronous specs", function() {
  beforeEach(function(done) {
    done();
  }, 1000);

  it("takes a long time", function(done) {
    setTimeout(function() {
      done();
    }, 9000);
  }, 10000);

  afterEach(function(done) {
    done();
  }, 1000);
});


describe("A spec using done.fail", function() {
  var foo = function(x, callBack1, callBack2) {
    if (x) {
      setTimeout(callBack1, 0);
    } else {
      setTimeout(callBack2, 0);
    }
  };

  it("should not call the second callBack", function(done) {
    foo(true,
      done,
      function() {
        done.fail("Second callback has been called");
      }
    );
  });
});
});
```

Downloads

- The Standalone Release (available on the releases page) is for simple, browser page, or console projects
- The Jasmine Ruby Gem is for Rails, Ruby, or Ruby-friendly development
- Other Environments are supported as well

Support

- Mailing list at Google Groups - a great first stop to ask questions, propose features, or discuss pull requests
- Report Issues at Github
- The Backlog lives at Pivotal Tracker
- Follow @JasmineBDD on Twitter

Thanks

Running documentation inspired by @mjackson and the 2012 Fluent Summit.

2.4.1 Options

89 specs, 0 failures, 4 pending specs finished in 9.193s

```
A suite
  contains spec with an expectation

A suite is just a function
  and so is a spec

The 'toBe' matcher compares with ===
  and has a positive case
  and can have a negative case

Included matchers:
  The 'toBe' matcher compares with ===

  The 'toEqual' matcher
    works for simple literals and variables
    should work for objects

  The 'toMatch' matcher is for regular expressions
  The 'toBeDefined' matcher compares against `undefined`
  The `toBeUndefined` matcher compares against `undefined`
  The 'toBeNull' matcher compares against null
  The 'toBeTruthy' matcher is for boolean casting testing
  The 'toBeFalsy' matcher is for boolean casting testing

  The 'toContain' matcher
    works for finding an item in an Array
    also works for finding a substring

  The 'toBeLessThan' matcher is for mathematical comparisons
  The 'toBeGreaterThan' matcher is for mathematical comparisons
  The 'toBeCloseTo' matcher is for precision math comparison
  The 'toThrow' matcher is for testing if a function throws an exception
  The 'toThrowError' matcher is for testing a specific thrown exception

A spec using the fail function
  SPEC HAS NO EXPECTATIONS should not call the callBack

A spec
  is just a function, so it can contain any code
  can have more than one expectation
```

A spec using beforeEach and afterEach
  is just a function, so it can contain any code
  can have more than one expectation

A spec using beforeAll and afterAll
  sets the initial value of foo before specs run
  does not reset foo between specs

A spec
  can use the `this` to share state
  prevents test pollution by having an empty `this` created for the next spec

A spec
  is just a function, so it can contain any code
  can have more than one expectation

  nested inside a second describe
    can reference both scopes as needed

A spec
  is just a function, so it can contain any code

Pending specs
  can be declared 'xit' PENDING WITH MESSAGE: Temporarily disabled with xit
  can be declared with 'it' but without a function
  can be declared by calling 'pending' in the spec body PENDING WITH MESSAGE: this is why it is pending

A spy
  tracks that the spy was called
  tracks that the spy was called x times
  tracks all the arguments of its calls
  stops all execution on a function

A spy, when configured to call through
  tracks that the spy was called
  should not affect other functions
  when called returns the requested value

A spy, when configured to fake a return value
  tracks that the spy was called
  should not affect other functions
  when called returns the requested value

A spy, when configured to fake a series of return values
  tracks that the spy was called
  should not affect other functions
  when called multiple times returns the requested values in order

A spy, when configured with an alternate implementation
  tracks that the spy was called
  should not affect other functions
  when called returns the requested value

A spy, when configured to throw an error
  throws the value

A spy
  can call through and then stub in the same spec

A spy
  tracks if it was called at all
  tracks the number of times it was called
  tracks the arguments of each call
  tracks the arguments of all calls
  can provide the context and arguments to all calls
  has a shortcut to the most recent call
  has a shortcut to the first call
  tracks the context
  can be reset

A spy, when created manually
  is named, which helps in error reporting
  tracks that the spy was called
  tracks its number of calls
  tracks all the arguments of its calls
  allows access to the most recent call

Multiple spies, when created manually
  creates spies for each requested function
  tracks that the spies were called
  tracks all the arguments of its calls

jasmine.any
  matches any value

  when used with a spy
    is useful for comparing arguments

jasmine.anything
  matches anything

  when used with a spy
    is useful when the argument can be ignored

jasmine.objectContaining
  matches objects with the expect key/value pairs

  when used with a spy
    is useful for comparing arguments

jasmine.arrayContaining
  matches arrays with some of the values

  when used with a spy
    is useful when comparing arguments

jasmine.stringMatching
  matches as a regexp

  when used with a spy
    is useful for comparing arguments

custom asymmetry
  dives in deep

  when used with a spy
    is useful for comparing arguments

Manually ticking the Jasmine Clock
  causes a timeout to be called synchronously
  causes an interval to be called synchronously

  Mocking the Date object
    mocks the Date object and sets it to a given time

Asynchronous specs
  should support async execution of test preparation and expectations

  long asynchronous specs
    SPEC HAS NO EXPECTATIONS takes a long time

  A spec using done.fail
    SPEC HAS NO EXPECTATIONS should not call the second callBack