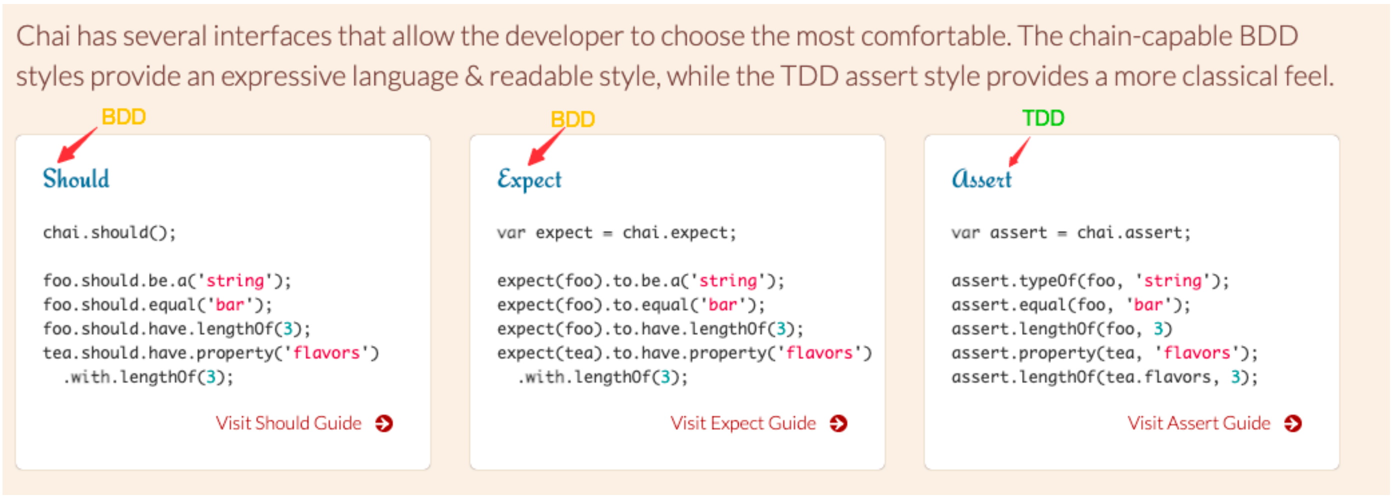
本文主要介绍Chai断言库的使用，mocha+chai是一个常用的组合

## Chai简介

在 Node.js 中，目前比较流行的单元测试组合是 mocha + chai。mocha 是一个测试框架，chai 是一个断言库，所以合称"抹茶"。

Chai 是一个 BDD / TDD 断言库，适用于Node.js和浏览器，可以与任何 javascript 测试框架完美搭配,它提供了两种风格的断言:BDD风格（行为驱动开发）和TDD风格（测试驱动开发）

BDD：Behavior Driven Development，行为驱动开发，注重测试逻辑
TDD：Test-Driven Development，测试驱动开发，注重输出结果


Chai has several interfaces that allow the developer to choose the most comfortable. The chain-capable BDD styles provide an expressive language & readable style, while the TDD assert style provides a more classical feel.

BDD

Should

```
chai.should();

foo.should.be.a('string');
foo.should.equal('bar');
foo.should.have.lengthOf(3);
tea.should.have.property('flavors')
  .with.lengthOf(3);
```

Visit Should Guide

BDD

Expect

```
var expect = chai.expect;

expect(foo).to.be.a('string');
expect(foo).to.equal('bar');
expect(foo).to.have.lengthOf(3);
expect(tea).to.have.property('flavors')
  .with.lengthOf(3);
```

Visit Expect Guide

TDD

Assert

```
var assert = chai.assert;

assert.typeOf(foo, 'string');
assert.equal(foo, 'bar');
assert.lengthOf(foo, 3)
assert.property(tea, 'flavors');
assert.lengthOf(tea.flavors, 3);
```

Visit Assert Guide

推荐的选择：expect(BDD),assert(TDD)

## Mocha+chai

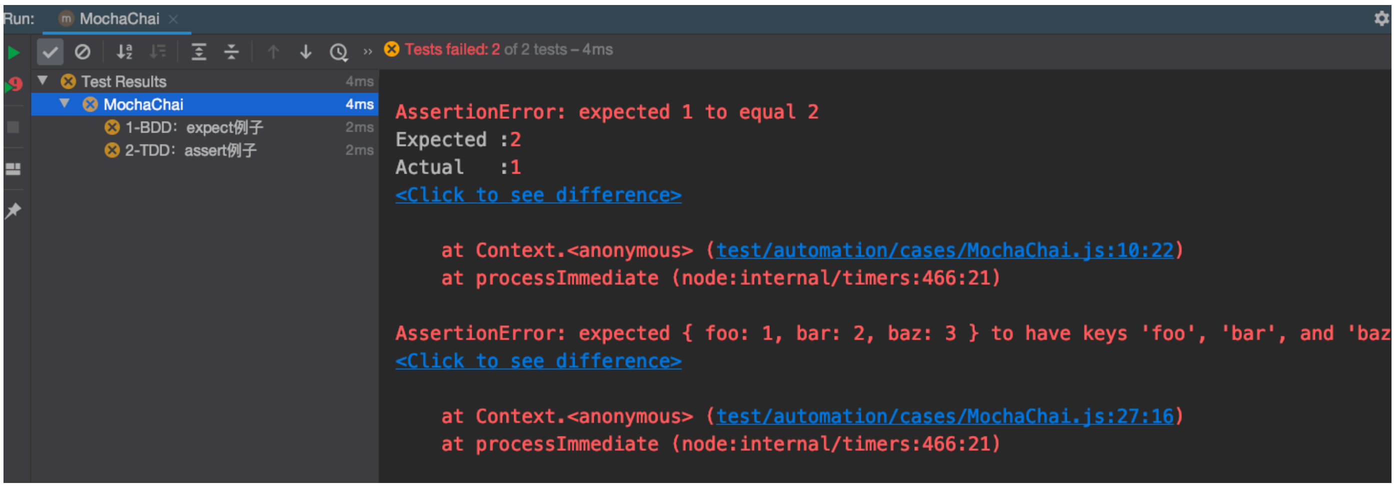
新建：test/automation/cases/MochaChai.js

```
const {describe,it}=require('mocha');
const {expect,assert}=require('chai');

describe('MochaChai',function () {
    this.timeout(10000);
    it('1-BDD: expect例子', async function () {
        expect('foo').to.be.a('string');
        expect(12).to.be.a('number');
        expect(1).to.equal(2);
        expect('foo').to.equal('foo');
        expect('foobar').to.have.string('bar');
        expect([1, 2, 3]).to.include(2);
        expect('foobar').to.match(/^foo/);
        expect({a: 1}).to.have.property('a');
        expect({a: 1, b: 2}).to.have.all.keys('a', 'b');
        expect({a: 1, b: 2}).to.have.any.keys('a');
    })
    it('2-TDD: assert例子', async function () {
        const teaServed = true;
        assert.isTrue(teaServed, 'the tea has been served');

        assert.deepEqual({ tea: 'green' }, { tea: 'green' });
        assert.isAtLeast(5, 2, '5 is greater or equal to 2');
        assert.isAtMost(3, 6, '3 is less than or equal to 6');
        assert.hasAnyKeys({foo: 1, bar: 2, baz: 3}, ['foo', 'iDontExist', 'baz']);
        assert.hasAllKeys({foo: 1, bar: 2, baz: 3}, {foo: 30, bar: 99, baz2: 1337});
        assert.containsAllKeys({foo: 1, bar: 2, baz: 3}, ['foo', 'baz']);
        assert.lengthOf('foobar', 6, 'string has length of 6');
    })
})
```

运行：



```
AssertionError: expected 1 to equal 2
Expected :2
Actual   :1
<Click to see difference>

    at Context.<anonymous> (test/automation/cases/MochaChai.js:10:22)
    at processImmediate (node:internal/timers:466:21)

AssertionError: expected { foo: 1, bar: 2, baz: 3 } to have keys 'foo', 'bar', and 'baz
<Click to see difference>

    at Context.<anonymous> (test/automation/cases/MochaChai.js:27:16)
    at processImmediate (node:internal/timers:466:21)
```

## assert-推荐

assert是一种TDD风格的断言库（类似于TestNG内置的断言库），对于TestNG熟悉的同学比较好理解

## assert(expression, message)

- **@param** *{ Mixed }* expression to test for truthiness
- **@param** *{ String }* message to display on error

Write your own test expressions.

```
assert('foo' !== 'bar', 'foo is not bar');
assert(Array.isArray([]), 'empty arrays are arrays');
```

## .fail([message])

## .fail(actual, expected, [message], [operator])

- **@param** *{ Mixed }* actual
- **@param** *{ Mixed }* expected
- **@param** *{ String }* message
- **@param** *{ String }* operator

Throw a failure. Node.js `assert` module-compatible.

```
assert.fail();
assert.fail("custom error message");
assert.fail(1, 2);
assert.fail(1, 2, "custom error message");
assert.fail(1, 2, "custom error message", ">");
assert.fail(1, 2, undefined, ">");
```

## .isOk(object, [message])

- **@param** *{ Mixed }* object to test
- **@param** *{ String }* message

Asserts that `object` is truthy.

```
assert.isOk('everything', 'everything is ok');
assert.isOk(false, 'this will fail');
```

## .isNotOk(object, [message])

- **@param** *{ Mixed }* object to test
- **@param** *{ String }* message

Asserts that `object` is falsy.

```
assert.isNotOk('everything', 'this will fail');
assert.isNotOk(false, 'this will pass');
```

## .equal(actual, expected, [message])

- **@param** *{ Mixed }* actual
- **@param** *{ Mixed }* expected
- **@param** *{ String }* message

Asserts non-strict equality (`==`) of `actual` and `expected`.

```
assert.equal(3, '3', '== coerces values to strings');
```

## .notEqual(actual, expected, [message])

- **@param** *{ Mixed }* actual
- **@param** *{ Mixed }* expected
- **@param** *{ String }* message

Asserts non-strict inequality (`!=`) of `actual` and `expected`.

```
assert.notEqual(3, 4, 'these numbers are not equal');
```

## .strictEqual(actual, expected, [message])

- **@param** *{ Mixed }* actual
- **@param** *{ Mixed }* expected
- **@param** *{ String }* message

Asserts strict equality (`===`) of `actual` and `expected`.

```
assert.strictEqual(true, true, 'these booleans are strictly equal');
```

## .notStrictEqual(actual, expected, [message])

- **@param** *{ Mixed }* actual
- **@param** *{ Mixed }* expected
- **@param** *{ String }* message

Asserts strict inequality (`!==`) of `actual` and `expected`.

```
assert.notStrictEqual(3, '3', 'no coercion for strict equality');
```

## .deepEqual(actual, expected, [message])

- **@param** *{ Mixed }* actual
- **@param** *{ Mixed }* expected
- **@param** *{ String }* message

Asserts that `actual` is deeply equal to `expected`.

```
assert.deepEqual({ tea: 'green' }, { tea: 'green' });
```

## .notDeepEqual(actual, expected, [message])

- **@param** *{ Mixed }* actual
- **@param** *{ Mixed }* expected
- **@param** *{ String }* message

Assert that `actual` is not deeply equal to `expected`.

```
assert.notDeepEqual({ tea: 'green' }, { tea: 'jasmine' });
```

## .isAbove(valueToCheck, valueToBeAbove, [message])

- **@param** *{ Mixed }* valueToCheck
- **@param** *{ Mixed }* valueToBeAbove
- **@param** *{ String }* message

Asserts `valueToCheck` is strictly greater than (>) `valueToBeAbove`.

```
assert.isAbove(5, 2, '5 is strictly greater than 2');
```

## .isAtLeast(valueToCheck, valueToBeAtLeast, [message])

- **@param** *{ Mixed }* valueToCheck
- **@param** *{ Mixed }* valueToBeAtLeast
- **@param** *{ String }* message

Asserts `valueToCheck` is greater than or equal to (>=) `valueToBeAtLeast`.

```
assert.isAtLeast(5, 2, '5 is greater or equal to 2');
assert.isAtLeast(3, 3, '3 is greater or equal to 3');
```

## .isBelow(valueToCheck, valueToBeBelow, [message])

- **@param** *{ Mixed }* valueToCheck
- **@param** *{ Mixed }* valueToBeBelow
- **@param** *{ String }* message

Asserts `valueToCheck` is strictly less than (<) `valueToBeBelow`.

```
assert.isBelow(3, 6, '3 is strictly less than 6');
```

## .isAtMost(valueToCheck, valueToBeAtMost, [message])

- **@param** *{ Mixed }* valueToCheck
- **@param** *{ Mixed }* valueToBeAtMost
- **@param** *{ String }* message

Asserts `valueToCheck` is less than or equal to (<=) `valueToBeAtMost`.

```
assert.isAtMost(3, 6, '3 is less than or equal to 6');
assert.isAtMost(4, 4, '4 is less than or equal to 4');
```

## .isTrue(value, [message])

- **@param** *{ Mixed }* value
- **@param** *{ String }* message

Asserts that `value` is true.

```
var teaServed = true;
assert.isTrue(teaServed, 'the tea has been served');
```

## .isNotTrue(value, [message])

- **@param** *{ Mixed }* value
- **@param** *{ String }* message

Asserts that `value` is not true.

```
var tea = 'tasty chai';
assert.isNotTrue(tea, 'great, time for tea!');
```

## .isFalse(value, [message])

- **@param** *{ Mixed }* value
- **@param** *{ String }* message

Asserts that `value` is false.

```
var teaServed = false;
assert.isFalse(teaServed, 'no tea yet? hmm...');
```

## .isNotFalse(value, [message])

- **@param** *{ Mixed }* value
- **@param** *{ String }* message

Asserts that `value` is not false.

```
var tea = 'tasty chai';
assert.isNotFalse(tea, 'great, time for tea!');
```

## .isNull(value, [message])

- **@param** *{ Mixed }* value
- **@param** *{ String }* message

Asserts that `value` is null.

```
assert.isNull(err, 'there was no error');
```

## .isNotNull(value, [message])

- **@param** *{ Mixed }* value
- **@param** *{ String }* message

Asserts that `value` is not null.

```
var tea = 'tasty chai';
assert.isNotNull(tea, 'great, time for tea!');
```

## .isNaN

- **@param** *{ Mixed }* value
- **@param** *{ String }* message

Asserts that value is NaN.

```
assert.isNaN(NaN, 'NaN is NaN');
```

## .isNotNaN

- **@param** *{ Mixed }* value
- **@param** *{ String }* message

Asserts that value is not NaN.

```
assert.isNotNaN(4, '4 is not NaN');
```

## .exists

- **@param** *{ Mixed }* value
- **@param** *{ String }* message

Asserts that the target is neither `null` nor `undefined`.

```
var foo = 'hi';

assert.exists(foo, 'foo is neither `null` nor `undefined`');
```

## .notExists

- **@param** *{ Mixed }* value
- **@param** *{ String }* message

Asserts that the target is either `null` or `undefined`.

```
var bar = null
  , baz;

assert.notExists(bar);
assert.notExists(baz, 'baz is either null or undefined');
```

## .isUndefined(value, [message])

- **@param** *{ Mixed }* value
- **@param** *{ String }* message

Asserts that `value` is `undefined`.

```
var tea;
assert.isUndefined(tea, 'no tea defined');
```

## .isDefined(value, [message])

- **@param** *{ Mixed }* value
- **@param** *{ String }* message

Asserts that `value` is not `undefined`.

```
var tea = 'cup of chai';
assert.isDefined(tea, 'tea has been defined');
```

## .isFunction(value, [message])

- **@param** *{ Mixed }* value
- **@param** *{ String }* message

Asserts that `value` is a function.

```
function serveTea() { return 'cup of tea'; };
assert.isFunction(serveTea, 'great, we can have tea now');
```

## .isNotFunction(value, [message])

- **@param** *{ Mixed }* value
- **@param** *{ String }* message

Asserts that `value` is *not* a function.

```
var serveTea = [ 'heat', 'pour', 'sip' ];
assert.isNotFunction(serveTea, 'great, we have listed the steps');
```

## .isObject(value, [message])

- **@param** *{ Mixed }* value
- **@param** *{ String }* message

Asserts that `value` is an object of type ‘Object’ (as revealed by `Object.prototype.toString`). *The assertion does not match subclassed objects.*

```
var selection = { name: 'Chai', serve: 'with spices' };
assert.isObject(selection, 'tea selection is an object');
```

## .isNotObject(value, [message])

- **@param** *{ Mixed }* value
- **@param** *{ String }* message

Asserts that `value` is *not* an object of type ‘Object’ (as revealed by `Object.prototype.toString`).

```
var selection = 'chai'
assert.isNotObject(selection, 'tea selection is not an object');
assert.isNotObject(null, 'null is not an object');
```

## .isArray(value, [message])

- **@param** *{ Mixed }* value
- **@param** *{ String }* message

Asserts that `value` is an array.

```
var menu = [ 'green', 'chai', 'oolong' ];
assert.isArray(menu, 'what kind of tea do we want?');
```

## .isNotArray(value, [message])

- **@param** *{ Mixed }* value
- **@param** *{ String }* message

Asserts that `value` is *not* an array.

```
var menu = 'green|chai|oolong';
assert.isNotArray(menu, 'what kind of tea do we want?');
```

## .isString(value, [message])

- **@param** *{ Mixed }* value
- **@param** *{ String }* message

Asserts that `value` is a string.

```
var teaOrder = 'chai';
assert.isString(teaOrder, 'order placed');
```

## .isNotString(value, [message])

- **@param** *{ Mixed }* value
- **@param** *{ String }* message

Asserts that `value` is *not* a string.

```
var teaOrder = 4;
assert.isNotString(teaOrder, 'order placed');
```

## .isNumber(value, [message])

- **@param** *{ Number }* value
- **@param** *{ String }* message

Asserts that `value` is a number.

```
var cups = 2;
assert.isNumber(cups, 'how many cups');
```

## .isNotNumber(value, [message])

- **@param** *{ Mixed }* value
- **@param** *{ String }* message

Asserts that `value` is *not* a number.

```
var cups = '2 cups please';
assert.isNotNumber(cups, 'how many cups');
```

## .isFinite(value, [message])

- **@param** *{ Number }* value
- **@param** *{ String }* message

Asserts that `value` is a finite number. Unlike `.isNumber`, this will fail for `NaN` and `Infinity`.

```
var cups = 2;
assert.isFinite(cups, 'how many cups');

assert.isFinite(NaN); // throws
```

## .isBoolean(value, [message])

- **@param** *{ Mixed }* value
- **@param** *{ String }* message

Asserts that `value` is a boolean.

```
var teaReady = true
  , teaServed = false;

assert.isBoolean(teaReady, 'is the tea ready');
assert.isBoolean(teaServed, 'has tea been served');
```

## .isNotBoolean(value, [message])

- **@param** *{ Mixed }* value
- **@param** *{ String }* message

Asserts that `value` is *not* a boolean.

```
var teaReady = 'yep'
  , teaServed = 'nope';

assert.isNotBoolean(teaReady, 'is the tea ready');
assert.isNotBoolean(teaServed, 'has tea been served');
```

## .typeOf(value, name, [message])

- **@param** *{ Mixed }* value
- **@param** *{ String }* name
- **@param** *{ String }* message

Asserts that `value`'s type is `name`, as determined by `Object.prototype.toString`.

```
assert.typeOf({ tea: 'chai' }, 'object', 'we have an object');
assert.typeOf(['chai', 'jasmine'], 'array', 'we have an array');
assert.typeOf('tea', 'string', 'we have a string');
assert.typeOf(/tea/, 'regexp', 'we have a regular expression');
assert.typeOf(null, 'null', 'we have a null');
assert.typeOf(undefined, 'undefined', 'we have an undefined');
```

## .notTypeOf(value, name, [message])

- **@param** *{ Mixed }* value
- **@param** *{ String }* typeof name
- **@param** *{ String }* message

Asserts that `value`'s type is *not* `name`, as determined by `Object.prototype.toString`.

```
assert.notTypeOf('tea', 'number', 'strings are not numbers');
```

## .instanceOf(object, constructor, [message])

- **@param** *{ Object }* object
- **@param** *{ Constructor }* constructor
- **@param** *{ String }* message

Asserts that `value` is an instance of `constructor`.

```
var Tea = function (name) { this.name = name; }
  , chai = new Tea('chai');

assert.instanceOf(chai, Tea, 'chai is an instance of tea');
```

## .notInstanceOf(object, constructor, [message])

- **@param** *{ Object }* object
- **@param** *{ Constructor }* constructor
- **@param** *{ String }* message

Asserts `value` is not an instance of `constructor`.

```
var Tea = function (name) { this.name = name; }
  , chai = new String('chai');

assert.notInstanceOf(chai, Tea, 'chai is not an instance of tea');
```

## .include(haystack, needle, [message])

- **@param** *{ Array | String }* haystack
- **@param** *{ Mixed }* needle
- **@param** *{ String }* message

Asserts that `haystack` includes `needle`. Can be used to assert the inclusion of a value in an array, a substring in a string, or a subset of properties in an object.

```
assert.include([1,2,3], 2, 'array contains value');
assert.include('foobar', 'foo', 'string contains substring');
assert.include({ foo: 'bar', hello: 'universe' }, { foo: 'bar' }, 'object contains
property');
```

Strict equality (===) is used. When asserting the inclusion of a value in an array, the array is searched for an element that's strictly equal to the given value. When asserting a subset of properties in an object, the object is searched for the given property keys, checking that each one is present and strictly equal to the given property value. For instance:

```
var obj1 = {a: 1}
  , obj2 = {b: 2};
assert.include([obj1, obj2], obj1);
assert.include({foo: obj1, bar: obj2}, {foo: obj1});
assert.include({foo: obj1, bar: obj2}, {foo: obj1, bar: obj2});
```

## .notInclude(haystack, needle, [message])

- **@param** *{ Array | String }* haystack
- **@param** *{ Mixed }* needle
- **@param** *{ String }* message

Asserts that `haystack` does not include `needle`. Can be used to assert the absence of a value in an array, a substring in a string, or a subset of properties in an object.

```
assert.notInclude([1,2,3], 4, "array doesn't contain value");
assert.notInclude('foobar', 'baz', "string doesn't contain substring");
assert.notInclude({ foo: 'bar', hello: 'universe' }, { foo: 'baz' }, 'object doesn't
contain property');
```

Strict equality (===) is used. When asserting the absence of a value in an array, the array is searched to confirm the absence of an element that's strictly equal to the given value. When asserting a subset of properties in an object, the object is searched to confirm that at least one of the given property keys is either not present or not strictly equal to the given property value. For instance:

```
var obj1 = {a: 1}
  , obj2 = {b: 2};
assert.notInclude([obj1, obj2], {a: 1});
assert.notInclude({foo: obj1, bar: obj2}, {foo: {a: 1}});
assert.notInclude({foo: obj1, bar: obj2}, {foo: obj1, bar: {b: 2}});
```

## .deepInclude(haystack, needle, [message])

- **@param** *{ Array | String }* haystack
- **@param** *{ Mixed }* needle
- **@param** *{ String }* message

Asserts that `haystack` includes `needle`. Can be used to assert the inclusion of a value in an array or a subset of properties in an object. Deep equality is used.

```
var obj1 = {a: 1}
  , obj2 = {b: 2};
assert.deepInclude([obj1, obj2], {a: 1});
assert.deepInclude({foo: obj1, bar: obj2}, {foo: {a: 1}});
assert.deepInclude({foo: obj1, bar: obj2}, {foo: {a: 1}, bar: {b: 2}});
```

## .notDeepInclude(haystack, needle, [message])

- **@param** *{ Array | String }* haystack
- **@param** *{ Mixed }* needle
- **@param** *{ String }* message

Asserts that `haystack` does not include `needle`. Can be used to assert the absence of a value in an array or a subset of properties in an object. Deep equality is used.

```
var obj1 = {a: 1}
  , obj2 = {b: 2};
assert.notDeepInclude([obj1, obj2], {a: 9});
assert.notDeepInclude({foo: obj1, bar: obj2}, {foo: {a: 9}});
assert.notDeepInclude({foo: obj1, bar: obj2}, {foo: {a: 1}, bar: {b: 9}});
```

## .nestedInclude(haystack, needle, [message])

- **@param** *{ Object }* haystack
- **@param** *{ Object }* needle
- **@param** *{ String }* message

Asserts that 'haystack' includes 'needle'. Can be used to assert the inclusion of a subset of properties in an object. Enables the use of dot- and bracket-notation for referencing nested properties. '[]' and '.' in property names can be escaped using double backslashes.

```
assert.nestedInclude({'.a': {'b': 'x'}}, {'\\.a.[b]': 'x'});
assert.nestedInclude({'a': {'[b]': 'x'}}, {'a.\\[b\\]': 'x'});
```

## .notNestedInclude(haystack, needle, [message])

- **@param** *{ Object }* haystack
- **@param** *{ Object }* needle
- **@param** *{ String }* message

Asserts that ‘haystack’ does not include ‘needle’. Can be used to assert the absence of a subset of properties in an object. Enables the use of dot- and bracket-notation for referencing nested properties. ‘[]’ and ‘.’ in property names can be escaped using double backslashes.

```
assert.notNestedInclude({'.a': {'b': 'x'}}, {'\\.a.b': 'y'});
assert.notNestedInclude({'a': {'[b]': 'x'}}, {'a.\\[b\\]': 'y'});
```

## .deepNestedInclude(haystack, needle, [message])

- **@param** *{ Object }* haystack
- **@param** *{ Object }* needle
- **@param** *{ String }* message

Asserts that ‘haystack’ includes ‘needle’. Can be used to assert the inclusion of a subset of properties in an object while checking for deep equality. Enables the use of dot- and bracket-notation for referencing nested properties. ‘[]’ and ‘.’ in property names can be escaped using double backslashes.

```
assert.deepNestedInclude({a: {b: [{x: 1}]}}, {'a.b[0]': {x: 1}});
assert.deepNestedInclude({'.a': {'[b]': {x: 1}}}, {'\\.a.\\[b\\]': {x: 1}});
```

## .notDeepNestedInclude(haystack, needle, [message])

- **@param** *{ Object }* haystack
- **@param** *{ Object }* needle
- **@param** *{ String }* message

Asserts that ‘haystack’ does not include ‘needle’. Can be used to assert the absence of a subset of properties in an object while checking for deep equality. Enables the use of dot- and bracket-notation for referencing nested properties. ‘[]’ and ‘.’ in property names can be escaped using double backslashes.

```
assert.notDeepNestedInclude({a: {b: [{x: 1}]}}, {'a.b[0]': {y: 1}})
assert.notDeepNestedInclude({'.a': {'[b]': {x: 1}}}, {'\\.a.\\[b\\]': {y: 2}});
```

## .ownInclude(haystack, needle, [message])

- **@param** *{ Object }* haystack
- **@param** *{ Object }* needle
- **@param** *{ String }* message

Asserts that ‘haystack’ includes ‘needle’. Can be used to assert the inclusion of a subset of properties in an object while ignoring inherited properties.

```
assert.ownInclude({ a: 1 }, { a: 1 });
```

## .notOwnInclude(haystack, needle, [message])

- **@param** *{ Object }* haystack
- **@param** *{ Object }* needle
- **@param** *{ String }* message

Asserts that ‘haystack’ includes ‘needle’. Can be used to assert the absence of a subset of properties in an object while ignoring inherited properties.

```
Object.prototype.b = 2;

assert.notOwnInclude({ a: 1 }, { b: 2 });
```

## .deepOwnInclude(haystack, needle, [message])

- **@param** *{ Object }* haystack
- **@param** *{ Object }* needle
- **@param** *{ String }* message

Asserts that ‘haystack’ includes ‘needle’. Can be used to assert the inclusion of a subset of properties in an object while ignoring inherited properties and checking for deep equality.

```
 assert.deepOwnInclude({a: {b: 2}}, {a: {b: 2}});
```

## .notDeepOwnInclude(haystack, needle, [message])

- **@param** *{ Object }* haystack
- **@param** *{ Object }* needle
- **@param** *{ String }* message

Asserts that ‘haystack’ includes ‘needle’. Can be used to assert the absence of a subset of properties in an object while ignoring inherited properties and checking for deep equality.

```
 assert.notDeepOwnInclude({a: {b: 2}}, {a: {c: 3}});
```

## .match(value, regexp, [message])

- **@param** *{ Mixed }* value
- **@param** *{ RegExp }* regexp
- **@param** *{ String }* message

Asserts that `value` matches the regular expression `regexp`.

```
assert.match('foobar', /^foo/, 'regexp matches');
```

## .notMatch(value, regexp, [message])

- **@param** *{ Mixed }* value
- **@param** *{ RegExp }* regexp
- **@param** *{ String }* message

Asserts that `value` does not match the regular expression `regexp`.

```
assert.notMatch('foobar', /^foo/, 'regexp does not match');
```

## .property(object, property, [message])

- **@param** *{ Object }* object
- **@param** *{ String }* property
- **@param** *{ String }* message

Asserts that `object` has a direct or inherited property named by `property`.

```
assert.property({ tea: { green: 'matcha' }}, 'tea');
assert.property({ tea: { green: 'matcha' }}, 'toString');
```

## .notProperty(object, property, [message])

- **@param** *{ Object }* object
- **@param** *{ String }* property
- **@param** *{ String }* message

Asserts that `object` does *not* have a direct or inherited property named by `property`.

```
assert.notProperty({ tea: { green: 'matcha' }}, 'coffee');
```

## .propertyVal(object, property, value, [message])

- **@param** *{ Object }* object
- **@param** *{ String }* property
- **@param** *{ Mixed }* value
- **@param** *{ String }* message

Asserts that `object` has a direct or inherited property named by `property` with a value given by `value`. Uses a strict equality check (===).

```
assert.propertyVal({ tea: 'is good' }, 'tea', 'is good');
```

## .notPropertyVal(object, property, value, [message])

- **@param** *{ Object }* object
- **@param** *{ String }* property
- **@param** *{ Mixed }* value
- **@param** *{ String }* message

Asserts that `object` does *not* have a direct or inherited property named by `property` with value given by `value`. Uses a strict equality check (===).

```
assert.notPropertyVal({ tea: 'is good' }, 'tea', 'is bad');
assert.notPropertyVal({ tea: 'is good' }, 'coffee', 'is good');
```

## .deepPropertyVal(object, property, value, [message])

- **@param** *{ Object }* object
- **@param** *{ String }* property
- **@param** *{ Mixed }* value
- **@param** *{ String }* message

Asserts that `object` has a direct or inherited property named by `property` with a value given by `value`. Uses a deep equality check.

```
assert.deepPropertyVal({ tea: { green: 'matcha' } }, 'tea', { green: 'matcha' });
```

## .notDeepPropertyVal(object, property, value, [message])

- **@param** *{ Object }* object
- **@param** *{ String }* property
- **@param** *{ Mixed }* value
- **@param** *{ String }* message

Asserts that `object` does *not* have a direct or inherited property named by `property` with value given by `value`. Uses a deep equality check.

```
assert.notDeepPropertyVal({ tea: { green: 'matcha' } }, 'tea', { black: 'matcha' });
assert.notDeepPropertyVal({ tea: { green: 'matcha' } }, 'tea', { green: 'oolong' });
assert.notDeepPropertyVal({ tea: { green: 'matcha' } }, 'coffee', { green: 'matcha' });
```

## .nestedProperty(object, property, [message])

- **@param** *{ Object }* object
- **@param** *{ String }* property
- **@param** *{ String }* message

Asserts that `object` has a direct or inherited property named by `property`, which can be a string using dot- and bracket-notation for nested reference.

```
assert.nestedProperty({ tea: { green: 'matcha' }}, 'tea.green');
```

## .notNestedProperty(object, property, [message])

- **@param** *{ Object }* object
- **@param** *{ String }* property
- **@param** *{ String }* message

Asserts that `object` does *not* have a property named by `property`, which can be a string using dot- and bracket-notation for nested reference. The property cannot exist on the object nor anywhere in its prototype chain.

```
assert.notNestedProperty({ tea: { green: 'matcha' }}, 'tea.oolong');
```

## .nestedPropertyVal(object, property, value, [message])

- **@param** *{ Object }* object
- **@param** *{ String }* property
- **@param** *{ Mixed }* value
- **@param** *{ String }* message

Asserts that `object` has a property named by `property` with value given by `value`. `property` can use dot- and bracket-notation for nested reference. Uses a strict equality check (===).

```
assert.nestedPropertyVal({ tea: { green: 'matcha' }}, 'tea.green', 'matcha');
```

## .notNestedPropertyVal(object, property, value, [message])

- **@param** *{ Object }* object
- **@param** *{ String }* property
- **@param** *{ Mixed }* value
- **@param** *{ String }* message

Asserts that `object` does *not* have a property named by `property` with value given by `value`. `property` can use dot- and bracket-notation for nested reference. Uses a strict equality check (===).

```
assert.notNestedPropertyVal({ tea: { green: 'matcha' }}, 'tea.green', 'konacha');
assert.notNestedPropertyVal({ tea: { green: 'matcha' }}, 'coffee.green', 'matcha');
```

## .deepNestedPropertyVal(object, property, value, [message])

- **@param** *{ Object }* object
- **@param** *{ String }* property
- **@param** *{ Mixed }* value
- **@param** *{ String }* message

Asserts that `object` has a property named by `property` with a value given by `value`. `property` can use dot- and bracket-notation for nested reference. Uses a deep equality check.

```
assert.deepNestedPropertyVal({ tea: { green: { matcha: 'yum' } } }, 'tea.green', {
matcha: 'yum' });
```

## .notDeepNestedPropertyVal(object, property, value, [message])

- **@param** *{ Object }* object
- **@param** *{ String }* property
- **@param** *{ Mixed }* value
- **@param** *{ String }* message

Asserts that `object` does *not* have a property named by `property` with value given by `value`. `property` can use dot- and bracket-notation for nested reference. Uses a deep equality check.

```
assert.notDeepNestedPropertyVal({ tea: { green: { matcha: 'yum' } } }, 'tea.green', {
oolong: 'yum' });
assert.notDeepNestedPropertyVal({ tea: { green: { matcha: 'yum' } } }, 'tea.green', {
matcha: 'yuck' });
assert.notDeepNestedPropertyVal({ tea: { green: { matcha: 'yum' } } }, 'tea.black', {
matcha: 'yum' });
```

## .lengthOf(object, length, [message])

- **@param** *{ Mixed }* object
- **@param** *{ Number }* length
- **@param** *{ String }* message

Asserts that `object` has a `length` or `size` with the expected value.

```
assert.lengthOf([1,2,3], 3, 'array has length of 3');
assert.lengthOf('foobar', 6, 'string has length of 6');
assert.lengthOf(new Set([1,2,3]), 3, 'set has size of 3');
assert.lengthOf(new Map([['a',1],['b',2],['c',3]]), 3, 'map has size of 3');
```

## .hasAnyKeys(object, [keys], [message])

- **@param** *{ Mixed }* object
- **@param** *{ Array | Object }* keys
- **@param** *{ String }* message

Asserts that `object` has at least one of the `keys` provided. You can also provide a single object instead of a `keys` array and its keys will be used as the expected set of keys.

```
assert.hasAnyKeys({foo: 1, bar: 2, baz: 3}, ['foo', 'iDontExist', 'baz']);
assert.hasAnyKeys({foo: 1, bar: 2, baz: 3}, {foo: 30, iDontExist: 99, baz: 1337});
assert.hasAnyKeys(new Map([[{foo: 1}, 'bar'], ['key', 'value']]), [{foo: 1}, 'key']);
assert.hasAnyKeys(new Set([{foo: 'bar'}, 'anotherKey']), [{foo: 'bar'}, 'anotherKey']);
```

## .hasAllKeys(object, [keys], [message])

- **@param** *{ Mixed }* object
- **@param** *{ Array. }* keys
- **@param** *{ String }* message

Asserts that `object` has all and only all of the `keys` provided. You can also provide a single object instead of a `keys` array and its keys will be used as the expected set of keys.

```
assert.hasAllKeys({foo: 1, bar: 2, baz: 3}, ['foo', 'bar', 'baz']);
assert.hasAllKeys({foo: 1, bar: 2, baz: 3}, {foo: 30, bar: 99, baz: 1337]);
assert.hasAllKeys(new Map([[{foo: 1}, 'bar'], ['key', 'value']]), [{foo: 1}, 'key']);
assert.hasAllKeys(new Set([{foo: 'bar'}, 'anotherKey'], [{foo: 'bar'}, 'anotherKey']);
```

## .containsAllKeys(object, [keys], [message])

- **@param** *{ Mixed }* object
- **@param** *{ Array. }* keys
- **@param** *{ String }* message

Asserts that `object` has all of the `keys` provided but may have more keys not listed. You can also provide a single object instead of a `keys` array and its keys will be used as the expected set of keys.

```
assert.containsAllKeys({foo: 1, bar: 2, baz: 3}, ['foo', 'baz']);
assert.containsAllKeys({foo: 1, bar: 2, baz: 3}, ['foo', 'bar', 'baz']);
assert.containsAllKeys({foo: 1, bar: 2, baz: 3}, {foo: 30, baz: 1337});
assert.containsAllKeys({foo: 1, bar: 2, baz: 3}, {foo: 30, bar: 99, baz: 1337});
assert.containsAllKeys(new Map([[{foo: 1}, 'bar'], ['key', 'value']]), [{foo: 1}]);
assert.containsAllKeys(new Map([[{foo: 1}, 'bar'], ['key', 'value']]), [{foo: 1},
'key']);
assert.containsAllKeys(new Set([{foo: 'bar'}, 'anotherKey'], [{foo: 'bar'}]);
assert.containsAllKeys(new Set([{foo: 'bar'}, 'anotherKey'], [{foo: 'bar'},
'anotherKey']);
```

## .doesNotHaveAnyKeys(object, [keys], [message])

- **@param** *{ Mixed }* object
- **@param** *{ Array. }* keys
- **@param** *{ String }* message

Asserts that `object` has none of the `keys` provided. You can also provide a single object instead of a `keys` array and its keys will be used as the expected set of keys.

```
assert.doesNotHaveAnyKeys({foo: 1, bar: 2, baz: 3}, ['one', 'two', 'example']);
assert.doesNotHaveAnyKeys({foo: 1, bar: 2, baz: 3}, {one: 1, two: 2, example: 'foo'});
assert.doesNotHaveAnyKeys(new Map([[{foo: 1}, 'bar'], ['key', 'value']]), [{one:
'two'}, 'example']);
assert.doesNotHaveAnyKeys(new Set([{foo: 'bar'}, 'anotherKey'], [{one: 'two'},
'example']);
```

## .doesNotHaveAllKeys(object, [keys], [message])

- **@param** *{ Mixed }* object
- **@param** *{ Array. }* keys
- **@param** *{ String }* message

Asserts that `object` does not have at least one of the `keys` provided. You can also provide a single object instead of a `keys` array and its keys will be used as the expected set of keys.

```
assert.doesNotHaveAllKeys({foo: 1, bar: 2, baz: 3}, ['one', 'two', 'example']);
assert.doesNotHaveAllKeys({foo: 1, bar: 2, baz: 3}, {one: 1, two: 2, example: 'foo'});
assert.doesNotHaveAllKeys(new Map([[{foo: 1}, 'bar'], ['key', 'value']]), [{one:
'two'}, 'example']);
assert.doesNotHaveAllKeys(new Set([{foo: 'bar'}, 'anotherKey'], [{one: 'two'},
'example']);
```

## .hasAnyDeepKeys(object, [keys], [message])

- **@param** *{ Mixed }* object
- **@param** *{ Array | Object }* keys
- **@param** *{ String }* message

Asserts that `object` has at least one of the `keys` provided. Since Sets and Maps can have objects as keys you can use this assertion to perform a deep comparison. You can also provide a single object instead of a `keys` array and its keys will be used as the expected set of keys.

```
assert.hasAnyDeepKeys(new Map([[{one: 'one'}, 'valueOne'], [1, 2]]), {one: 'one'});
assert.hasAnyDeepKeys(new Map([[{one: 'one'}, 'valueOne'], [1, 2]]), [{one: 'one'},
{two: 'two'}]);
assert.hasAnyDeepKeys(new Map([[{one: 'one'}, 'valueOne'], [{two: 'two'},
'valueTwo']]), [{one: 'one'}, {two: 'two'}]);
assert.hasAnyDeepKeys(new Set([{one: 'one'}, {two: 'two'}]), {one: 'one'});
assert.hasAnyDeepKeys(new Set([{one: 'one'}, {two: 'two'}]), [{one: 'one'}, {three:
'three'}]);
assert.hasAnyDeepKeys(new Set([{one: 'one'}, {two: 'two'}]), [{one: 'one'}, {two:
'two'}]);
```

## .hasAllDeepKeys(object, [keys], [message])

- **@param** *{ Mixed }* object
- **@param** *{ Array | Object }* keys
- **@param** *{ String }* message

Asserts that `object` has all and only all of the `keys` provided. Since Sets and Maps can have objects as keys you can use this assertion to perform a deep comparison. You can also provide a single object instead of a `keys` array and its keys will be used as the expected set of keys.

```
assert.hasAllDeepKeys(new Map([[{one: 'one'}, 'valueOne']]), {one: 'one'});
assert.hasAllDeepKeys(new Map([[{one: 'one'}, 'valueOne'], [{two: 'two'},
'valueTwo']]), [{one: 'one'}, {two: 'two'}]);
assert.hasAllDeepKeys(new Set([{one: 'one'}]), {one: 'one'});
assert.hasAllDeepKeys(new Set([{one: 'one'}, {two: 'two'}]), [{one: 'one'}, {two:
'two'}]);
```

## .containsAllDeepKeys(object, [keys], [message])

- **@param** *{ Mixed }* object
- **@param** *{ Array | Object }* keys
- **@param** *{ String }* message

Asserts that `object` contains all of the `keys` provided. Since Sets and Maps can have objects as keys you can use this assertion to perform a deep comparison. You can also provide a single object instead of a `keys` array and its keys will be used as the expected set of keys.

```
assert.containsAllDeepKeys(new Map([[{one: 'one'}, 'valueOne'], [1, 2]]), {one:
'one'});
assert.containsAllDeepKeys(new Map([[{one: 'one'}, 'valueOne'], [{two: 'two'},
'valueTwo']]), [{one: 'one'}, {two: 'two'}]);
assert.containsAllDeepKeys(new Set([{one: 'one'}, {two: 'two'}]), {one: 'one'});
assert.containsAllDeepKeys(new Set([{one: 'one'}, {two: 'two'}]), [{one: 'one'}, {two:
'two'}]);
```

## .doesNotHaveAnyDeepKeys(object, [keys], [message])

- **@param** *{ Mixed }* object
- **@param** *{ Array | Object }* keys
- **@param** *{ String }* message

Asserts that `object` has none of the `keys` provided. Since Sets and Maps can have objects as keys you can use this assertion to perform a deep comparison. You can also provide a single object instead of a `keys` array and its keys will be used as the expected set of keys.

```
assert.doesNotHaveAnyDeepKeys(new Map([[{one: 'one'}, 'valueOne'], [1, 2]]),
{thisDoesNot: 'exist'});
assert.doesNotHaveAnyDeepKeys(new Map([[{one: 'one'}, 'valueOne'], [{two: 'two'},
'valueTwo']]), [{twenty: 'twenty'}, {fifty: 'fifty'}]);
assert.doesNotHaveAnyDeepKeys(new Set([{one: 'one'}, {two: 'two'}]), {twenty:
'twenty'});
assert.doesNotHaveAnyDeepKeys(new Set([{one: 'one'}, {two: 'two'}]), [{twenty:
'twenty'}, {fifty: 'fifty'}]);
```

## .doesNotHaveAllDeepKeys(object, [keys], [message])

- **@param** *{ Mixed }* object
- **@param** *{ Array | Object }* keys
- **@param** *{ String }* message

Asserts that `object` does not have at least one of the `keys` provided. Since Sets and Maps can have objects as keys you can use this assertion to perform a deep comparison. You can also provide a single object instead of a `keys` array and its keys will be used as the expected set of keys.

```
assert.doesNotHaveAllDeepKeys(new Map([[{one: 'one'}, 'valueOne'], [1, 2]]),
{thisDoesNot: 'exist'});
assert.doesNotHaveAllDeepKeys(new Map([[{one: 'one'}, 'valueOne'], [{two: 'two'},
'valueTwo']]), [{twenty: 'twenty'}, {one: 'one'}]);
assert.doesNotHaveAllDeepKeys(new Set([{one: 'one'}, {two: 'two'}]), {twenty:
'twenty'});
assert.doesNotHaveAllDeepKeys(new Set([{one: 'one'}, {two: 'two'}]), [{one: 'one'},
{fifty: 'fifty'}]);
```

## .throws(fn, [errorLike/string/regexp], [string/regexp], [message])

- **@param** *{ Function }* fn
- **@param** *{ ErrorConstructor | Error }* errorLike
- **@param** *{ RegExp | String }* errMsgMatcher
- **@param** *{ String }* message
- **@see** [https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Error#Error_types](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Error#Error_types)

If `errorLike` is an `Error` constructor, asserts that `fn` will throw an error that is an instance of `errorLike`. If `errorLike` is an `Error` instance, asserts that the error thrown is the same instance as `errorLike`. If `errMsgMatcher` is provided, it also asserts that the error thrown will have a message matching `errMsgMatcher`.

```
assert.throws(fn, 'Error thrown must have this msg');
assert.throws(fn, /Error thrown must have a msg that matches this/);
assert.throws(fn, ReferenceError);
assert.throws(fn, errorInstance);
assert.throws(fn, ReferenceError, 'Error thrown must be a ReferenceError and have this
msg');
assert.throws(fn, errorInstance, 'Error thrown must be the same errorInstance and have
this msg');
assert.throws(fn, ReferenceError, /Error thrown must be a ReferenceError and match
this/);
assert.throws(fn, errorInstance, /Error thrown must be the same errorInstance and match
this/);
```

## .doesNotThrow(fn, [errorLike/string/regexp], [string/regexp], [message])

- **@param** *{ Function }* fn
- **@param** *{ ErrorConstructor }* errorLike
- **@param** *{ RegExp | String }* errMsgMatcher
- **@param** *{ String }* message
- **@see** https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Error#Error_types

If `errorLike` is an `Error` constructor, asserts that `fn` will *not* throw an error that is an instance of `errorLike`. If `errorLike` is an `Error` instance, asserts that the error thrown is *not* the same instance as `errorLike`. If `errMsgMatcher` is provided, it also asserts that the error thrown will *not* have a message matching `errMsgMatcher`.

```
assert.doesNotThrow(fn, 'Any Error thrown must not have this message');
assert.doesNotThrow(fn, /Any Error thrown must not match this/);
assert.doesNotThrow(fn, Error);
assert.doesNotThrow(fn, errorInstance);
assert.doesNotThrow(fn, Error, 'Error must not have this message');
assert.doesNotThrow(fn, errorInstance, 'Error must not have this message');
assert.doesNotThrow(fn, Error, /Error must not match this/);
assert.doesNotThrow(fn, errorInstance, /Error must not match this/);
```

## .operator(val1, operator, val2, [message])

- **@param** *{ Mixed }* val1
- **@param** *{ String }* operator
- **@param** *{ Mixed }* val2
- **@param** *{ String }* message

Compares two values using `operator`.

```
assert.operator(1, '<', 2, 'everything is ok');
assert.operator(1, '>', 2, 'this will fail');
```

## .closeTo(actual, expected, delta, [message])

- **@param** *{ Number }* actual
- **@param** *{ Number }* expected
- **@param** *{ Number }* delta
- **@param** *{ String }* message

Asserts that the target is equal `expected`, to within a +/- `delta` range.

```
assert.closeTo(1.5, 1, 0.5, 'numbers are close');
```

## .approximately(actual, expected, delta, [message])

- **@param** *{ Number }* actual
- **@param** *{ Number }* expected
- **@param** *{ Number }* delta
- **@param** *{ String }* message

Asserts that the target is equal `expected`, to within a +/- `delta` range.

```
assert.approximately(1.5, 1, 0.5, 'numbers are close');
```

## .sameMembers(set1, set2, [message])

- **@param** *{ Array }* set1
- **@param** *{ Array }* set2
- **@param** *{ String }* message

Asserts that `set1` and `set2` have the same members in any order. Uses a strict equality check (===).

```
assert.sameMembers([ 1, 2, 3 ], [ 2, 1, 3 ], 'same members');
```

## .notSameMembers(set1, set2, [message])

- **@param** *{ Array }* set1
- **@param** *{ Array }* set2
- **@param** *{ String }* message

Asserts that `set1` and `set2` don’t have the same members in any order. Uses a strict equality check (===).

```
assert.notSameMembers([ 1, 2, 3 ], [ 5, 1, 3 ], 'not same members');
```

## .sameDeepMembers(set1, set2, [message])

- **@param** *{ Array }* set1
- **@param** *{ Array }* set2
- **@param** *{ String }* message

Asserts that `set1` and `set2` have the same members in any order. Uses a deep equality check.

```
assert.sameDeepMembers([ { a: 1 }, { b: 2 }, { c: 3 } ], [{ b: 2 }, { a: 1 }, { c: 3
}], 'same deep members');
```

## .notSameDeepMembers(set1, set2, [message])

- **@param** *{ Array }* set1
- **@param** *{ Array }* set2
- **@param** *{ String }* message

Asserts that `set1` and `set2` don't have the same members in any order. Uses a deep equality check.

```
assert.notSameDeepMembers([ { a: 1 }, { b: 2 }, { c: 3 } ], [{ b: 2 }, { a: 1 }, { f: 5
}], 'not same deep members');
```

## .sameOrderedMembers(set1, set2, [message])

- **@param** *{ Array }* set1
- **@param** *{ Array }* set2
- **@param** *{ String }* message

Asserts that `set1` and `set2` have the same members in the same order. Uses a strict equality check (===).

```
assert.sameOrderedMembers([ 1, 2, 3 ], [ 1, 2, 3 ], 'same ordered members');
```

## .notSameOrderedMembers(set1, set2, [message])

- **@param** *{ Array }* set1
- **@param** *{ Array }* set2
- **@param** *{ String }* message

Asserts that `set1` and `set2` don't have the same members in the same order. Uses a strict equality check (===).

```
assert.notSameOrderedMembers([ 1, 2, 3 ], [ 2, 1, 3 ], 'not same ordered members');
```

## .sameDeepOrderedMembers(set1, set2, [message])

- **@param** *{ Array }* set1
- **@param** *{ Array }* set2
- **@param** *{ String }* message

Asserts that `set1` and `set2` have the same members in the same order. Uses a deep equality check.

```
assert.sameDeepOrderedMembers([ { a: 1 }, { b: 2 }, { c: 3 } ], [ { a: 1 }, { b: 2 }, {
c: 3 } ], 'same deep ordered members');
```

## .notSameDeepOrderedMembers(set1, set2, [message])

- **@param** *{ Array }* set1
- **@param** *{ Array }* set2
- **@param** *{ String }* message

Asserts that `set1` and `set2` don’t have the same members in the same order. Uses a deep equality check.

```
assert.notSameDeepOrderedMembers([ { a: 1 }, { b: 2 }, { c: 3 } ], [ { a: 1 }, { b: 2
}, { z: 5 } ], 'not same deep ordered members');
assert.notSameDeepOrderedMembers([ { a: 1 }, { b: 2 }, { c: 3 } ], [ { b: 2 }, { a: 1
}, { c: 3 } ], 'not same deep ordered members');
```

## .includeMembers(superset, subset, [message])

- **@param** *{ Array }* superset
- **@param** *{ Array }* subset
- **@param** *{ String }* message

Asserts that `subset` is included in `superset` in any order. Uses a strict equality check (===). Duplicates are ignored.

```
assert.includeMembers([ 1, 2, 3 ], [ 2, 1, 2 ], 'include members');
```

## .notIncludeMembers(superset, subset, [message])

- **@param** *{ Array }* superset
- **@param** *{ Array }* subset
- **@param** *{ String }* message

Asserts that `subset` isn’t included in `superset` in any order. Uses a strict equality check (===). Duplicates are ignored.

```
assert.notIncludeMembers([ 1, 2, 3 ], [ 5, 1 ], 'not include members');
```

## .includeDeepMembers(superset, subset, [message])

- **@param** *{ Array }* superset
- **@param** *{ Array }* subset
- **@param** *{ String }* message

Asserts that `subset` is included in `superset` in any order. Uses a deep equality check. Duplicates are ignored.

```
assert.includeDeepMembers([ { a: 1 }, { b: 2 }, { c: 3 } ], [ { b: 2 }, { a: 1 }, { b: 2 } ], 'include deep members');
```

## .notIncludeDeepMembers(superset, subset, [message])

- **@param** *{ Array }* superset
- **@param** *{ Array }* subset
- **@param** *{ String }* message

Asserts that `subset` isn't included in `superset` in any order. Uses a deep equality check. Duplicates are ignored.

```
assert.notIncludeDeepMembers([ { a: 1 }, { b: 2 }, { c: 3 } ], [ { b: 2 }, { f: 5 } ], 'not include deep members');
```

## .includeOrderedMembers(superset, subset, [message])

- **@param** *{ Array }* superset
- **@param** *{ Array }* subset
- **@param** *{ String }* message

Asserts that `subset` is included in `superset` in the same order beginning with the first element in `superset`. Uses a strict equality check (===).

```
assert.includeOrderedMembers([ 1, 2, 3 ], [ 1, 2 ], 'include ordered members');
```

## .notIncludeOrderedMembers(superset, subset, [message])

- **@param** *{ Array }* superset
- **@param** *{ Array }* subset
- **@param** *{ String }* message

Asserts that `subset` isn't included in `superset` in the same order beginning with the first element in `superset`. Uses a strict equality check (===).

```
assert.notIncludeOrderedMembers([ 1, 2, 3 ], [ 2, 1 ], 'not include ordered members');
assert.notIncludeOrderedMembers([ 1, 2, 3 ], [ 2, 3 ], 'not include ordered members');
```

## .includeDeepOrderedMembers(superset, subset, [message])

- **@param** *{ Array }* superset
- **@param** *{ Array }* subset
- **@param** *{ String }* message

Asserts that `subset` is included in `superset` in the same order beginning with the first element in `superset`. Uses a deep equality check.

```
assert.includeDeepOrderedMembers([ { a: 1 }, { b: 2 }, { c: 3 } ], [ { a: 1 }, { b: 2 }
], 'include deep ordered members');
```

## .notIncludeDeepOrderedMembers(superset, subset, [message])

- **@param** *{ Array }* superset
- **@param** *{ Array }* subset
- **@param** *{ String }* message

Asserts that `subset` isn't included in `superset` in the same order beginning with the first element in `superset`. Uses a deep equality check.

```
assert.notIncludeDeepOrderedMembers([ { a: 1 }, { b: 2 }, { c: 3 } ], [ { a: 1 }, { f:
5 } ], 'not include deep ordered members');
assert.notIncludeDeepOrderedMembers([ { a: 1 }, { b: 2 }, { c: 3 } ], [ { b: 2 }, { a:
1 } ], 'not include deep ordered members');
assert.notIncludeDeepOrderedMembers([ { a: 1 }, { b: 2 }, { c: 3 } ], [ { b: 2 }, { c:
3 } ], 'not include deep ordered members');
```

## .oneOf(inList, list, [message])

- **@param** *{ }* inList
- **@param** *{ Array.<*> }* list
- **@param** *{ String }* message

Asserts that non-object, non-array value `inList` appears in the flat array `list`.

```
assert.oneOf(1, [ 2, 1 ], 'Not found in list');
```

## .changes(function, object, property, [message])

- **@param** *{ Function }* modifier function
- **@param** *{ Object }* object or getter function
- **@param** *{ String }* property name *optional*
- **@param** *{ String }* message *optional*

Asserts that a function changes the value of a property.

```
var obj = { val: 10 };
var fn = function() { obj.val = 22 };
assert.changes(fn, obj, 'val');
```

## .changesBy(function, object, property, delta, [message])

- **@param** *{ Function }* modifier function
- **@param** *{ Object }* object or getter function
- **@param** *{ String }* property name *optional*
- **@param** *{ Number }* change amount (delta)
- **@param** *{ String }* message *optional*

Asserts that a function changes the value of a property by an amount (delta).

```
var obj = { val: 10 };
var fn = function() { obj.val += 2 };
assert.changesBy(fn, obj, 'val', 2);
```

## .doesNotChange(function, object, property, [message])

- **@param** *{ Function }* modifier function
- **@param** *{ Object }* object or getter function
- **@param** *{ String }* property name *optional*
- **@param** *{ String }* message *optional*

Asserts that a function does not change the value of a property.

```
var obj = { val: 10 };
var fn = function() { console.log('foo'); };
assert.doesNotChange(fn, obj, 'val');
```

## .changesButNotBy(function, object, property, delta, [message])

- **@param** *{ Function }* modifier function
- **@param** *{ Object }* object or getter function
- **@param** *{ String }* property name *optional*
- **@param** *{ Number }* change amount (delta)
- **@param** *{ String }* message *optional*

Asserts that a function does not change the value of a property or of a function’s return value by an amount (delta)

```
var obj = { val: 10 };
var fn = function() { obj.val += 10 };
assert.changesButNotBy(fn, obj, 'val', 5);
```

## .increases(function, object, property, [message])

- **@param** *{ Function }* modifier function
- **@param** *{ Object }* object or getter function
- **@param** *{ String }* property name *optional*
- **@param** *{ String }* message *optional*

Asserts that a function increases a numeric object property.

```
var obj = { val: 10 };
var fn = function() { obj.val = 13 };
assert.increases(fn, obj, 'val');
```

## .increasesBy(function, object, property, delta, [message])

- **@param** *{ Function }* modifier function
- **@param** *{ Object }* object or getter function
- **@param** *{ String }* property name *optional*
- **@param** *{ Number }* change amount (delta)
- **@param** *{ String }* message *optional*

Asserts that a function increases a numeric object property or a function’s return value by an amount (delta).

```
var obj = { val: 10 };
var fn = function() { obj.val += 10 };
assert.increasesBy(fn, obj, 'val', 10);
```

## .doesNotIncrease(function, object, property, [message])

- **@param** *{ Function }* modifier function
- **@param** *{ Object }* object or getter function
- **@param** *{ String }* property name *optional*
- **@param** *{ String }* message *optional*

Asserts that a function does not increase a numeric object property.

```
var obj = { val: 10 };
var fn = function() { obj.val = 8 };
assert.doesNotIncrease(fn, obj, 'val');
```

## .increasesButNotBy(function, object, property, delta, [message])

- **@param** *{ Function }* modifier function
- **@param** *{ Object }* object or getter function
- **@param** *{ String }* property name *optional*
- **@param** *{ Number }* change amount (delta)

- **@param** *{ String }* message *optional*

Asserts that a function does not increase a numeric object property or function’s return value by an amount (delta).

```
var obj = { val: 10 };
var fn = function() { obj.val = 15 };
assert.increasesButNotBy(fn, obj, 'val', 10);
```

## .decreases(function, object, property, [message])

- **@param** *{ Function }* modifier function
- **@param** *{ Object }* object or getter function
- **@param** *{ String }* property name *optional*
- **@param** *{ String }* message *optional*

Asserts that a function decreases a numeric object property.

```
var obj = { val: 10 };
var fn = function() { obj.val = 5 };
assert.decreases(fn, obj, 'val');
```

## .decreasesBy(function, object, property, delta, [message])

- **@param** *{ Function }* modifier function
- **@param** *{ Object }* object or getter function
- **@param** *{ String }* property name *optional*
- **@param** *{ Number }* change amount (delta)
- **@param** *{ String }* message *optional*

Asserts that a function decreases a numeric object property or a function’s return value by an amount (delta)

```
var obj = { val: 10 };
var fn = function() { obj.val -= 5 };
assert.decreasesBy(fn, obj, 'val', 5);
```

## .doesNotDecrease(function, object, property, [message])

- **@param** *{ Function }* modifier function
- **@param** *{ Object }* object or getter function
- **@param** *{ String }* property name *optional*
- **@param** *{ String }* message *optional*

Asserts that a function does not decreases a numeric object property.

```
var obj = { val: 10 };
var fn = function() { obj.val = 15 };
assert.doesNotDecrease(fn, obj, 'val');
```

## .doesNotDecreaseBy(function, object, property, delta, [message])

- **@param** *{ Function }* modifier function
- **@param** *{ Object }* object or getter function
- **@param** *{ String }* property name *optional*
- **@param** *{ Number }* change amount (delta)
- **@param** *{ String }* message *optional*

Asserts that a function does not decreases a numeric object property or a function’s return value by an amount (delta)

```
var obj = { val: 10 };
var fn = function() { obj.val = 5 };
assert.doesNotDecreaseBy(fn, obj, 'val', 1);
```

## .decreasesButNotBy(function, object, property, delta, [message])

- **@param** *{ Function }* modifier function
- **@param** *{ Object }* object or getter function
- **@param** *{ String }* property name *optional*
- **@param** *{ Number }* change amount (delta)
- **@param** *{ String }* message *optional*

Asserts that a function does not decreases a numeric object property or a function’s return value by an amount (delta)

```
var obj = { val: 10 };
var fn = function() { obj.val = 5 };
assert.decreasesButNotBy(fn, obj, 'val', 1);
```

## .ifError(object)

- **@param** *{ Object }* object

Asserts if value is not a false value, and throws if it is a true value. This is added to allow for chai to be a drop-in replacement for Node’s assert class.

```
var err = new Error('I am a custom error');
assert.ifError(err); // Rethrows err!
```

## .isExtensible(object)

- **@param** *{ Object }* object
- **@param** *{ String }* message *optional*

Asserts that `object` is extensible (can have new properties added to it).

```
assert.isExtensible({});
```

## .isNotExtensible(object)

- **@param** *{ Object }* object
- **@param** *{ String }* message *optional*

Asserts that `object` is *not* extensible.

```
var nonExtensibleObject = Object.preventExtensions({});
var sealedObject = Object.seal({});
var frozenObject = Object.freeze({});

assert.isNotExtensible(nonExtensibleObject);
assert.isNotExtensible(sealedObject);
assert.isNotExtensible(frozenObject);
```

## .isSealed(object)

- **@param** *{ Object }* object
- **@param** *{ String }* message *optional*

Asserts that `object` is sealed (cannot have new properties added to it and its existing properties cannot be removed).

```
var sealedObject = Object.seal({});
var frozenObject = Object.seal({});

assert.isSealed(sealedObject);
assert.isSealed(frozenObject);
```

## .isNotSealed(object)

- **@param** *{ Object }* object
- **@param** *{ String }* message *optional*

Asserts that `object` is *not* sealed.

```
assert.isNotSealed({});
```

## .isFrozen(object)

- **@param** *{ Object }* object
- **@param** *{ String }* message *optional*

Asserts that `object` is frozen (cannot have new properties added to it and its existing properties cannot be modified).

```
var frozenObject = Object.freeze({});
assert.frozen(frozenObject);
```

## .isNotFrozen(object)

- **@param** *{ Object }* object
- **@param** *{ String }* message *optional*

Asserts that `object` is *not* frozen.

```
assert.isNotFrozen({});
```

## .isEmpty(target)

- **@param** *{ Object | Array | String | Map | Set }* target
- **@param** *{ String }* message *optional*

Asserts that the target does not contain any values. For arrays and strings, it checks the `length` property. For `Map` and `Set` instances, it checks the `size` property. For non-function objects, it gets the count of own enumerable string keys.

```
assert.isEmpty([]);
assert.isEmpty('');
assert.isEmpty(new Map);
assert.isEmpty({});
```

## .isNotEmpty(target)

- **@param** *{ Object | Array | String | Map | Set }* target
- **@param** *{ String }* message *optional*

Asserts that the target contains values. For arrays and strings, it checks the `length` property. For `Map` and `Set` instances, it checks the `size` property. For non-function objects, it gets the count of own enumerable string keys.

```
assert.isNotEmpty([1, 2]);
assert.isNotEmpty('34');
assert.isNotEmpty(new Set([5, 6]));
assert.isNotEmpty({ key: 7 });
```

## expect-推荐

expect是一种BDD风格的断言库

### .not

```
expect(function () {}).to.not.throw();
expect({a: 1}).to.not.have.property('b');
expect([1, 2]).to.be.an('array').that.does.not.include(3);
```

```
expect(2).to.equal(2); // Recommended
expect(2).to.not.equal(1); // Not recommended
```

### .deep

```
// Target object deeply (but not strictly) equals `{a: 1}`
expect({a: 1}).to.deep.equal({a: 1});
expect({a: 1}).to.not.equal({a: 1});

// Target array deeply (but not strictly) includes `{a: 1}`
expect([{a: 1}]).to.deep.include({a: 1});
expect([{a: 1}]).to.not.include({a: 1});

// Target object deeply (but not strictly) includes `x: {a: 1}`
expect({x: {a: 1}}).to.deep.include({x: {a: 1}});
expect({x: {a: 1}}).to.not.include({x: {a: 1}});

// Target array deeply (but not strictly) has member `{a: 1}`
expect([{a: 1}]).to.have.deep.members([{a: 1}]);
expect([{a: 1}]).to.not.have.members([{a: 1}]);

// Target set deeply (but not strictly) has key `{a: 1}`
expect(new Set([{a: 1}])).to.have.deep.keys([{a: 1}]);
expect(new Set([{a: 1}])).to.not.have.keys([{a: 1}]);

// Target object deeply (but not strictly) has property `x: {a: 1}`
expect({x: {a: 1}}).to.have.deep.property('x', {a: 1});
expect({x: {a: 1}}).to.not.have.property('x', {a: 1});
```

### .nested

```
expect({a: {b: ['x', 'y']}}).to.have.nested.property('a.b[1]');
expect({a: {b: ['x', 'y']}}).to.nested.include({'a.b[1]': 'y'});
expect({'.a': {'[b]': 'x'}}).to.have.nested.property('\\.a.\\[b\\]');
expect({'.a': {'[b]': 'x'}}).to.nested.include({'\\.a.\\[b\\]': 'x'});
```

## .own

Causes all `.property` and `.include` assertions that follow in the chain to ignore inherited properties.

```
Object.prototype.b = 2;

expect({a: 1}).to.have.own.property('a');
expect({a: 1}).to.have.property('b');
expect({a: 1}).to.not.have.own.property('b');

expect({a: 1}).to.own.include({a: 1});
expect({a: 1}).to.include({b: 2}).but.not.own.include({b: 2});
```

`.own` cannot be combined with `.nested`.

## .ordered

```
expect([1, 2]).to.have.ordered.members([1, 2])
  .but.not.have.ordered.members([2, 1]);
```

When `.include` and `.ordered` are combined, the ordering begins at the start of both arrays.

```
expect([1, 2, 3]).to.include.ordered.members([1, 2])
  .but.not.include.ordered.members([2, 3]);
```

## .any

```
expect({a: 1, b: 2}).to.not.have.any.keys('c', 'd');
```

## .all

```
expect({a: 1, b: 2}).to.have.all.keys('a', 'b');
```

## .a(type[, msg])

```
expect('foo').to.be.a('string');
expect({a: 1}).to.be.an('object');
expect(null).to.be.a('null');
expect(undefined).to.be.an('undefined');
expect(new Error).to.be.an('error');
expect(Promise.resolve()).to.be.a('promise');
expect(new Float32Array).to.be.a('float32array');
expect(Symbol()).to.be.a('symbol');
```

```
var myObj = {
  [Symbol.toStringTag]: 'myCustomType'
};

expect(myObj).to.be.a('myCustomType').but.not.an('object');
```

```
expect([1, 2, 3]).to.be.an('array').that.includes(2);
expect([]).to.be.an('array').that.is.empty;
```

```
expect('foo').to.be.a('string'); // Recommended
expect('foo').to.not.be.an('array'); // Not recommended
```

```
expect(1).to.be.a('string', 'nooo why fail??');
expect(1, 'nooo why fail??').to.be.a('string');
```

```
expect({b: 2}).to.have.a.property('b');
```

## .include(val[, msg])

```
expect('foobar').to.include('foo');
```

```
expect([1, 2, 3]).to.include(2);
```

```
expect({a: 1, b: 2, c: 3}).to.include({a: 1, b: 2});
```

```
expect(new Set([1, 2])).to.include(2);
```

```
expect(new Map([['a', 1], ['b', 2]])).to.include(2);
```

```
expect([1, 2, 3]).to.be.an('array').that.includes(2);
```

```
// Target array deeply (but not strictly) includes `{a: 1}`
expect([{a: 1}]).to.deep.include({a: 1});
expect([{a: 1}]).to.not.include({a: 1});

// Target object deeply (but not strictly) includes `x: {a: 1}`
expect({x: {a: 1}}).to.deep.include({x: {a: 1}});
expect({x: {a: 1}}).to.not.include({x: {a: 1}});
```

```
Object.prototype.b = 2;

expect({a: 1}).to.own.include({a: 1});
expect({a: 1}).to.include({b: 2}).but.not.own.include({b: 2});
```

```
expect({a: {b: 2}}).to.deep.own.include({a: {b: 2}});
```

```
expect({a: {b: ['x', 'y']}}).to.nested.include({'a.b[1]': 'y'});
```

```
expect({'.a': {'[b]': 2}}).to.nested.include({'\\.a.\\[b\\]': 2});
```

```
expect({a: {b: [{c: 3}]}}).to.deep.nested.include({'a.b[0]': {c: 3}});
```

```
expect('foobar').to.not.include('taco');
expect([1, 2, 3]).to.not.include(4);
```

```
expect({c: 3}).to.not.have.any.keys('a', 'b'); // Recommended
expect({c: 3}).to.not.include({a: 1, b: 2}); // Not recommended
```

```
expect({a: 3, b: 4}).to.include({a: 3, b: 4}); // Recommended
expect({a: 3, b: 4}).to.not.include({a: 1, b: 2}); // Not recommended
```

```
expect([1, 2, 3]).to.include(4, 'nooo why fail??');
expect([1, 2, 3], 'nooo why fail??').to.include(4);
```

```
// Target object's keys are a superset of ['a', 'b'] but not identical
expect({a: 1, b: 2, c: 3}).to.include.all.keys('a', 'b');
expect({a: 1, b: 2, c: 3}).to.not.have.all.keys('a', 'b');

// Target array is a superset of [1, 2] but not identical
expect([1, 2, 3]).to.include.members([1, 2]);
expect([1, 2, 3]).to.not.have.members([1, 2]);

// Duplicates in the subset are ignored
expect([1, 2, 3]).to.include.members([1, 2, 2, 2]);
```

```
// Both assertions are identical
expect({a: 1}).to.include.any.keys('a', 'b');
expect({a: 1}).to.have.any.keys('a', 'b');
```

## .ok

```
expect(1).to.equal(1); // Recommended
expect(1).to.be.ok; // Not recommended

expect(true).to.be.true; // Recommended
expect(true).to.be.ok; // Not recommended
```

```
expect(0).to.equal(0); // Recommended
expect(0).to.not.be.ok; // Not recommended

expect(false).to.be.false; // Recommended
expect(false).to.not.be.ok; // Not recommended

expect(null).to.be.null; // Recommended
expect(null).to.not.be.ok; // Not recommended

expect(undefined).to.be.undefined; // Recommended
expect(undefined).to.not.be.ok; // Not recommended
```

```
expect(false, 'nooo why fail??').to.be.ok;
```

## .true

```
expect(true).to.be.true;
expect(false).to.be.false; // Recommended
expect(false).to.not.be.true; // Not recommended

expect(1).to.equal(1); // Recommended
expect(1).to.not.be.true; // Not recommended
```

```
expect(false, 'nooo why fail??').to.be.true;
```

## .false

```
expect(false).to.be.false;
```

```
expect(true).to.be.true; // Recommended
expect(true).to.not.be.false; // Not recommended

expect(1).to.equal(1); // Recommended
expect(1).to.not.be.false; // Not recommended
```

```
expect(true, 'nooo why fail??').to.be.false;
```

## .null

```
expect(null).to.be.null;
```

```
expect(1).to.equal(1); // Recommended
expect(1).to.not.be.null; // Not recommended
```

```
expect(42, 'nooo why fail??').to.be.null;
```

## .undefined

```
expect(undefined).to.be.undefined;
```

```
expect(1).to.equal(1); // Recommended
expect(1).to.not.be.undefined; // Not recommended
```

A custom error message can be given as the second argument to `expect`.

```
expect(42, 'nooo why fail??').to.be.undefined;
```

## .NaN

```
expect(NaN).to.be.NaN;
```

```
expect('foo').to.equal('foo'); // Recommended
expect('foo').to.not.be.NaN; // Not recommended
```

A custom error message can be given as the second argument to `expect`.

```
expect(42, 'nooo why fail??').to.be.NaN;
```

## .exist

```
expect(1).to.equal(1); // Recommended
expect(1).to.exist; // Not recommended

expect(0).to.equal(0); // Recommended
expect(0).to.exist; // Not recommended
```

```
expect(null).to.be.null; // Recommended
expect(null).to.not.exist; // Not recommended

expect(undefined).to.be.undefined; // Recommended
expect(undefined).to.not.exist; // Not recommended
```

```
expect(null, 'nooo why fail??').to.exist;
```

## .empty

```
expect([]).to.be.empty;
expect('').to.be.empty;
```

```
expect(new Set()).to.be.empty;
expect(new Map()).to.be.empty;
```

```
expect({}).to.be.empty;
```

```
expect([]).to.be.an('array').that.is.empty;
expect([1, 2, 3]).to.have.lengthOf(3); // Recommended
expect([1, 2, 3]).to.not.be.empty; // Not recommended

expect(new Set([1, 2, 3])).to.have.property('size', 3); // Recommended
expect(new Set([1, 2, 3])).to.not.be.empty; // Not recommended

expect(Object.keys({a: 1})).to.have.lengthOf(1); // Recommended
expect({a: 1}).to.not.be.empty; // Not recommended
```

```
expect([1, 2, 3], 'nooo why fail??').to.be.empty;
```

## .arguments

```
function test () {
  expect(arguments).to.be.arguments;
}

test();
```

```
expect('foo').to.be.a('string'); // Recommended
expect('foo').to.not.be.arguments; // Not recommended
```

```
expect({}, 'nooo why fail??').to.be.arguments;
```

## .equal(val[, msg])

```
expect(1).to.equal(1);
expect('foo').to.equal('foo');
```

```
// Target object deeply (but not strictly) equals `{a: 1}`
expect({a: 1}).to.deep.equal({a: 1});
expect({a: 1}).to.not.equal({a: 1});

// Target array deeply (but not strictly) equals `[1, 2]`
expect([1, 2]).to.deep.equal([1, 2]);
expect([1, 2]).to.not.equal([1, 2]);
```

```
expect(1).to.equal(1); // Recommended
expect(1).to.not.equal(2); // Not recommended
expect(1).to.equal(2, 'nooo why fail??');
expect(1, 'nooo why fail??').to.equal(2);
```

## .eql(obj[, msg])

```
// Target object is deeply (but not strictly) equal to {a: 1}
expect({a: 1}).to.eql({a: 1}).but.not.equal({a: 1});

// Target array is deeply (but not strictly) equal to [1, 2]
expect([1, 2]).to.eql([1, 2]).but.not.equal([1, 2]);
```

```
expect({a: 1}).to.eql({a: 1}); // Recommended
expect({a: 1}).to.not.eql({b: 2}); // Not recommended
```

```
expect({a: 1}).to.eql({b: 2}, 'nooo why fail??');
expect({a: 1}, 'nooo why fail??').to.eql({b: 2});
```

## .above(n[, msg])

```
expect(2).to.equal(2); // Recommended
expect(2).to.be.above(1); // Not recommended
```

```
expect('foo').to.have.lengthOf(3); // Recommended
expect('foo').to.have.lengthOf.above(2); // Not recommended

expect([1, 2, 3]).to.have.lengthOf(3); // Recommended
expect([1, 2, 3]).to.have.lengthOf.above(2); // Not recommended
```

```
expect(2).to.equal(2); // Recommended
expect(1).to.not.be.above(2); // Not recommended
```

```
expect(1).to.be.above(2, 'nooo why fail??');
expect(1, 'nooo why fail??').to.be.above(2);
```

## .least(n[, msg])

```
expect(2).to.equal(2); // Recommended
expect(2).to.be.at.least(1); // Not recommended
expect(2).to.be.at.least(2); // Not recommended
expect('foo').to.have.lengthOf(3); // Recommended
expect('foo').to.have.lengthOf.at.least(2); // Not recommended

expect([1, 2, 3]).to.have.lengthOf(3); // Recommended
expect([1, 2, 3]).to.have.lengthOf.at.least(2); // Not recommended
```

```
expect(1).to.equal(1); // Recommended
expect(1).to.not.be.at.least(2); // Not recommended
```

```
expect(1).to.be.at.least(2, 'nooo why fail??');
expect(1, 'nooo why fail??').to.be.at.least(2);
```

## .below(n[, msg])

```
expect(1).to.equal(1); // Recommended
expect(1).to.be.below(2); // Not recommended
```

```
expect('foo').to.have.lengthOf(3); // Recommended
expect('foo').to.have.lengthOf.below(4); // Not recommended

expect([1, 2, 3]).to.have.length(3); // Recommended
expect([1, 2, 3]).to.have.lengthOf.below(4); // Not recommended
```

```
expect(2).to.equal(2); // Recommended
expect(2).to.not.be.below(1); // Not recommended
```

`.below` accepts an optional `msg` argument which is a custom error message to show when the assertion fails. The message can also be given as the second argument to `expect`.

```
expect(2).to.be.below(1, 'nooo why fail??');
expect(2, 'nooo why fail??').to.be.below(1);
```

The aliases `.lt` and `.lessThan` can be used interchangeably with `.below`.

## .most(n[, msg])

- **@param** *{ Number }* n
- **@param** *{ String }* msg *optional*

Asserts that the target is a number or a date less than or equal to the given number or date `n` respectively. However, it’s often best to assert that the target is equal to its expected value.

```
expect(1).to.equal(1); // Recommended
expect(1).to.be.at.most(2); // Not recommended
expect(1).to.be.at.most(1); // Not recommended
```

Add `.lengthOf` earlier in the chain to assert that the target’s `length` or `size` is less than or equal to the given number `n`.

```
expect('foo').to.have.lengthOf(3); // Recommended
expect('foo').to.have.lengthOf.at.most(4); // Not recommended

expect([1, 2, 3]).to.have.lengthOf(3); // Recommended
expect([1, 2, 3]).to.have.lengthOf.at.most(4); // Not recommended
```

Add `.not` earlier in the chain to negate `.most`.

```
expect(2).to.equal(2); // Recommended
expect(2).to.not.be.at.most(1); // Not recommended
```

`.most` accepts an optional `msg` argument which is a custom error message to show when the assertion fails. The message can also be given as the second argument to `expect`.

```
expect(2).to.be.at.most(1, 'nooo why fail??');
expect(2, 'nooo why fail??').to.be.at.most(1);
```

The aliases `.lte` and `.lessThanOrEqual` can be used interchangeably with `.most`.

## .within(start, finish[, msg])

```
expect(2).to.equal(2); // Recommended
expect(2).to.be.within(1, 3); // Not recommended
expect(2).to.be.within(2, 3); // Not recommended
expect(2).to.be.within(1, 2); // Not recommended
```

```
expect('foo').to.have.lengthOf(3); // Recommended
expect('foo').to.have.lengthOf.within(2, 4); // Not recommended

expect([1, 2, 3]).to.have.lengthOf(3); // Recommended
expect([1, 2, 3]).to.have.lengthOf.within(2, 4); // Not recommended
```

```
expect(1).to.equal(1); // Recommended
expect(1).to.not.be.within(2, 4); // Not recommended
```

```
expect(4).to.be.within(1, 3, 'nooo why fail??');
expect(4, 'nooo why fail??').to.be.within(1, 3);
```

## .instanceof(constructor[, msg])

```
function Cat () { }

expect(new Cat()).to.be.an.instanceof(Cat);
expect([1, 2]).to.be.an.instanceof(Array);
```

```
expect({a: 1}).to.not.be.an.instanceof(Array);
```

```
expect(1).to.be.an.instanceof(Array, 'nooo why fail??');
expect(1, 'nooo why fail??').to.be.an.instanceof(Array);
```

## .property(name[, val[, msg]])

```
expect({a: 1}).to.have.property('a');
```

```
expect({a: 1}).to.have.property('a', 1);
```

```
// Target object deeply (but not strictly) has property `x: {a: 1}`
expect({x: {a: 1}}).to.have.deep.property('x', {a: 1});
expect({x: {a: 1}}).to.not.have.property('x', {a: 1});
```

```
Object.prototype.b = 2;

expect({a: 1}).to.have.own.property('a');
expect({a: 1}).to.have.own.property('a', 1);
expect({a: 1}).to.have.property('b');
expect({a: 1}).to.not.have.own.property('b');
```

`.deep` and `.own` can be combined.

```
expect({x: {a: 1}}).to.have.deep.own.property('x', {a: 1});
```

Add `.nested` earlier in the chain to enable dot- and bracket-notation when referencing nested properties.

```
expect({a: {b: ['x', 'y']}}).to.have.nested.property('a.b[1]');
expect({a: {b: ['x', 'y']}}).to.have.nested.property('a.b[1]', 'y');
```

If `.` or `[]` are part of an actual property name, they can be escaped by adding two backslashes before them.

```
expect({'.a': {'[b]': 'x'}}).to.have.nested.property('\\.a.\\[b\\]');
```

`.deep` and `.nested` can be combined.

```
expect({a: {b: [{c: 3}]}})
  .to.have.deep.nested.property('a.b[0]', {c: 3});
```

`.own` and `.nested` cannot be combined.

Add `.not` earlier in the chain to negate `.property`.

```
expect({a: 1}).to.not.have.property('b');
```

However, it's dangerous to negate `.property` when providing `val`. The problem is that it creates uncertain expectations by asserting that the target either doesn't have a property with the given key `name`, or that it does have a property with the given key `name` but its value isn't equal to the given `val`. It's often best to identify the exact output that's expected, and then write an assertion that only accepts that exact output.

When the target isn't expected to have a property with the given key `name`, it's often best to assert exactly that.

```
expect({b: 2}).to.not.have.property('a'); // Recommended
expect({b: 2}).to.not.have.property('a', 1); // Not recommended
```

When the target is expected to have a property with the given key `name`, it's often best to assert that the property has its expected value, rather than asserting that it doesn't have one of many unexpected values.

```
expect({a: 3}).to.have.property('a', 3); // Recommended
expect({a: 3}).to.not.have.property('a', 1); // Not recommended
```

`.property` changes the target of any assertions that follow in the chain to be the value of the property from the original target object.

```
expect({a: 1}).to.have.property('a').that.is.a('number');
```

`.property` accepts an optional `msg` argument which is a custom error message to show when the assertion fails. The message can also be given as the second argument to `expect`. When not providing `val`, only use the second form.

```
// Recommended
expect({a: 1}).to.have.property('a', 2, 'nooo why fail??');
expect({a: 1}, 'nooo why fail??').to.have.property('a', 2);
expect({a: 1}, 'nooo why fail??').to.have.property('b');

// Not recommended
expect({a: 1}).to.have.property('b', undefined, 'nooo why fail??');
```

The above assertion isn't the same thing as not providing `val`. Instead, it's asserting that the target object has a `b` property that's equal to `undefined`.

The assertions `.ownProperty` and `.haveOwnProperty` can be used interchangeably with `.own.property`.

## .ownPropertyDescriptor(name[, descriptor[, msg]])

```
expect({a: 1}).to.have.ownPropertyDescriptor('a');
```

```
expect({a: 1}).to.have.ownPropertyDescriptor('a', {
  configurable: true,
  enumerable: true,
  writable: true,
  value: 1,
});
```

```
expect({a: 1}).to.not.have.ownPropertyDescriptor('b');
```

```
// Recommended
expect({b: 2}).to.not.have.ownPropertyDescriptor('a');

// Not recommended
expect({b: 2}).to.not.have.ownPropertyDescriptor('a', {
  configurable: true,
  enumerable: true,
  writable: true,
  value: 1,
});
```

```
// Recommended
expect({a: 3}).to.have.ownPropertyDescriptor('a', {
  configurable: true,
  enumerable: true,
  writable: true,
  value: 3,
});

// Not recommended
```

```
expect({a: 3}).to.not.have.ownPropertyDescriptor('a', {
  configurable: true,
  enumerable: true,
  writable: true,
  value: 1,
});
```

```
expect({a: 1}).to.have.ownPropertyDescriptor('a')
  .that.has.property('enumerable', true);
```

```
// Recommended
expect({a: 1}).to.have.ownPropertyDescriptor('a', {
  configurable: true,
  enumerable: true,
  writable: true,
  value: 2,
}, 'nooo why fail??');

// Recommended
expect({a: 1}, 'nooo why fail??').to.have.ownPropertyDescriptor('a', {
  configurable: true,
  enumerable: true,
  writable: true,
  value: 2,
});

// Recommended
expect({a: 1}, 'nooo why fail??').to.have.ownPropertyDescriptor('b');

// Not recommended
expect({a: 1})
  .to.have.ownPropertyDescriptor('b', undefined, 'nooo why fail??');
```

## .lengthOf(n[, msg])

```
expect([1, 2, 3]).to.have.lengthOf(3);
expect('foo').to.have.lengthOf(3);
expect(new Set([1, 2, 3])).to.have.lengthOf(3);
expect(new Map([['a', 1], ['b', 2], ['c', 3]])).to.have.lengthOf(3);
```

```
expect('foo').to.have.lengthOf(3); // Recommended
expect('foo').to.not.have.lengthOf(4); // Not recommended
```

```
expect([1, 2, 3]).to.have.lengthOf(2, 'nooo why fail??');
expect([1, 2, 3], 'nooo why fail??').to.have.lengthOf(2);
```

`.lengthOf` can also be used as a language chain, causing all `.above`, `.below`, `.least`, `.most`, and `.within` assertions that follow in the chain to use the target's `length` property as the target. However, it's often best to assert that the target's `length` property is equal to its expected length, rather than asserting that its `length` property falls within some range of values.

```
// Recommended
expect([1, 2, 3]).to.have.lengthOf(3);

// Not recommended
expect([1, 2, 3]).to.have.lengthOf.above(2);
expect([1, 2, 3]).to.have.lengthOf.below(4);
expect([1, 2, 3]).to.have.lengthOf.at.least(3);
expect([1, 2, 3]).to.have.lengthOf.at.most(3);
expect([1, 2, 3]).to.have.lengthOf.within(2,4);
```

Due to a compatibility issue, the alias `.length` can't be chained directly off of an uninvoked method such as `.a`. Therefore, `.length` can't be used interchangeably with `.lengthOf` in every situation. It's recommended to always use `.lengthOf` instead of `.length`.

```
expect([1, 2, 3]).to.have.a.length(3); // incompatible; throws error
expect([1, 2, 3]).to.have.a.lengthOf(3);  // passes as expected
```

## .match(re[, msg])

```
expect('foobar').to.match(/^foo/);
```

Add `.not` earlier in the chain to negate `.match`.

```
expect('foobar').to.not.match(/taco/);
```

`.match` accepts an optional `msg` argument which is a custom error message to show when the assertion fails. The message can also be given as the second argument to `expect`.

```
expect('foobar').to.match(/taco/, 'nooo why fail??');
expect('foobar', 'nooo why fail??').to.match(/taco/);
```

The alias `.matches` can be used interchangeably with `.match`.

## .string(str[, msg])

```
expect('foobar').to.have.string('bar');
```

```
expect('foobar').to.not.have.string('taco');
expect('foobar').to.have.string('taco', 'nooo why fail??');
expect('foobar', 'nooo why fail??').to.have.string('taco');
```

## .keys(key1[, key2[, ...]])

```
expect({a: 1, b: 2}).to.have.all.keys('a', 'b');
expect(['x', 'y']).to.have.all.keys(0, 1);

expect({a: 1, b: 2}).to.have.all.keys(['a', 'b']);
expect(['x', 'y']).to.have.all.keys([0, 1]);

expect({a: 1, b: 2}).to.have.all.keys({a: 4, b: 5}); // ignore 4 and 5
expect(['x', 'y']).to.have.all.keys({0: 4, 1: 5}); // ignore 4 and 5
```

```
expect(new Map([['a', 1], ['b', 2]])).to.have.all.keys('a', 'b');
expect(new Set(['a', 'b'])).to.have.all.keys('a', 'b');
```

```
expect({a: 1, b: 2}).to.be.an('object').that.has.all.keys('a', 'b');
```

```
// Target set deeply (but not strictly) has key `{a: 1}`
expect(new Set([{a: 1}])).to.have.all.deep.keys([{a: 1}]);
expect(new Set([{a: 1}])).to.not.have.all.keys([{a: 1}]);
```

By default, the target must have all of the given keys and no more. Add `.any` earlier in the chain to only require that the target have at least one of the given keys. Also, add `.not` earlier in the chain to negate `.keys`. It's often best to add `.any` when negating `.keys`, and to use `.all` when asserting `.keys` without negation.

When negating `.keys`, `.any` is preferred because `.not.any.keys` asserts exactly what's expected of the output, whereas `.not.all.keys` creates uncertain expectations.

```
// Recommended; asserts that target doesn't have any of the given keys
expect({a: 1, b: 2}).to.not.have.any.keys('c', 'd');

// Not recommended; asserts that target doesn't have all of the given
// keys but may or may not have some of them
expect({a: 1, b: 2}).to.not.have.all.keys('c', 'd');
```

When asserting `.keys` without negation, `.all` is preferred because `.all.keys` asserts exactly what's expected of the output, whereas `.any.keys` creates uncertain expectations.

```
// Recommended; asserts that target has all the given keys
expect({a: 1, b: 2}).to.have.all.keys('a', 'b');

// Not recommended; asserts that target has at least one of the given
// keys but may or may not have more of them
expect({a: 1, b: 2}).to.have.any.keys('a', 'b');
```

Note that `.all` is used by default when neither `.all` nor `.any` appear earlier in the chain. However, it's often best to add `.all` anyway because it improves readability.

```
// Both assertions are identical
expect({a: 1, b: 2}).to.have.all.keys('a', 'b'); // Recommended
expect({a: 1, b: 2}).to.have.keys('a', 'b'); // Not recommended
```

Add `.include` earlier in the chain to require that the target's keys be a superset of the expected keys, rather than identical sets.

```
// Target object's keys are a superset of ['a', 'b'] but not identical
expect({a: 1, b: 2, c: 3}).to.include.all.keys('a', 'b');
expect({a: 1, b: 2, c: 3}).to.not.have.all.keys('a', 'b');
```

However, if `.any` and `.include` are combined, only the `.any` takes effect. The `.include` is ignored in this case.

```
// Both assertions are identical
expect({a: 1}).to.have.any.keys('a', 'b');
expect({a: 1}).to.include.any.keys('a', 'b');
```

A custom error message can be given as the second argument to `expect`.

```
expect({a: 1}, 'nooo why fail??').to.have.key('b');
```

The alias `.key` can be used interchangeably with `.keys`.

## .throw([errorLike], [errMsgMatcher], [msg])

```
var badFn = function () { throw new TypeError('Illegal salmon!'); };

expect(badFn).to.throw();
```

```
var badFn = function () { throw new TypeError('Illegal salmon!'); };

expect(badFn).to.throw(TypeError);
```

```
var err = new TypeError('Illegal salmon!');
var badFn = function () { throw err; };

expect(badFn).to.throw(err);
```

```
var badFn = function () { throw new TypeError('Illegal salmon!'); };

expect(badFn).to.throw('salmon');
```

When one argument is provided, and it’s a regular expression, `.throw` invokes the target function and asserts that an error is thrown with a message that matches that regular expression.

```
var badFn = function () { throw new TypeError('Illegal salmon!'); };

expect(badFn).to.throw(/salmon/);
```

```
var err = new TypeError('Illegal salmon!');
var badFn = function () { throw err; };

expect(badFn).to.throw(TypeError, 'salmon');
expect(badFn).to.throw(TypeError, /salmon/);
expect(badFn).to.throw(err, 'salmon');
expect(badFn).to.throw(err, /salmon/);
```

```
var goodFn = function () {};

expect(goodFn).to.not.throw();
```

```
var goodFn = function () {};

expect(goodFn).to.not.throw(); // Recommended
expect(goodFn).to.not.throw(ReferenceError, 'x'); // Not recommended
```

```
var badFn = function () { throw new TypeError('Illegal salmon!'); };

expect(badFn).to.throw(TypeError, 'salmon'); // Recommended
expect(badFn).to.not.throw(ReferenceError, 'x'); // Not recommended
```

```
var err = new TypeError('Illegal salmon!');
err.code = 42;
var badFn = function () { throw err; };

expect(badFn).to.throw(TypeError).with.property('code', 42);
```

`.throw` accepts an optional `msg` argument which is a custom error message to show when the assertion fails. The message can also be given as the second argument to `expect`. When not providing two arguments, always use the second form.

```
var goodFn = function () {};

expect(goodFn).to.throw(TypeError, 'x', 'nooo why fail??');
expect(goodFn, 'nooo why fail??').to.throw();
```

Due to limitations in ES5, `.throw` may not always work as expected when using a transpiler such as Babel or TypeScript. In particular, it may produce unexpected results when subclassing the built-in `Error` object and then passing the subclassed constructor to `.throw`. See your transpiler's docs for details:

- (Babel)
- (TypeScript)

Beware of some common mistakes when using the `throw` assertion. One common mistake is to accidentally invoke the function yourself instead of letting the `throw` assertion invoke the function for you. For example, when testing if a function named `fn` throws, provide `fn` instead of `fn()` as the target for the assertion.

```
expect(fn).to.throw();     // Good! Tests `fn` as desired
expect(fn()).to.throw();   // Bad! Tests result of `fn()`, not `fn`
```

If you need to assert that your function `fn` throws when passed certain arguments, then wrap a call to `fn` inside of another function.

```
expect(function () { fn(42); }).to.throw();  // Function expression
expect(() => fn(42)).to.throw();             // ES6 arrow function
```

Another common mistake is to provide an object method (or any stand-alone function that relies on `this`) as the target of the assertion. Doing so is problematic because the `this` context will be lost when the function is invoked by `.throw`; there's no way for it to know what `this` is supposed to be. There are two ways around this problem. One solution is to wrap the method or function call inside of another function. Another solution is to use `bind`.

```
expect(function () { cat.meow(); }).to.throw();  // Function expression
expect(() => cat.meow()).to.throw();             // ES6 arrow function
expect(cat.meow.bind(cat)).to.throw();           // Bind
```

Finally, it's worth mentioning that it's a best practice in JavaScript to only throw `Error` and derivatives of `Error` such as `ReferenceError`, `TypeError`, and user-defined objects that extend `Error`. No other type of value will generate a stack trace when initialized. With that said, the `throw` assertion does technically support any type of value being thrown, not just `Error` and its derivatives.

The aliases `.throws` and `.Throw` can be used interchangeably with `.throw`.

## .respondTo(method[, msg])

- **@param** *{ String }* method
- **@param** *{ String }* msg *optional*

When the target is a non-function object, `.respondTo` asserts that the target has a method with the given name `method`. The method can be own or inherited, and it can be enumerable or non-enumerable.

```
function Cat () {}
Cat.prototype.meow = function () {};

expect(new Cat()).to.respondTo('meow');
```

When the target is a function, `.respondTo` asserts that the target's `prototype` property has a method with the given name `method`. Again, the method can be own or inherited, and it can be enumerable or non-enumerable.

```
function Cat () {}
Cat.prototype.meow = function () {};

expect(Cat).to.respondTo('meow');
```

Add `.itself` earlier in the chain to force `.respondTo` to treat the target as a non-function object, even if it's a function. Thus, it asserts that the target has a method with the given name `method`, rather than asserting that the target's `prototype` property has a method with the given name `method`.

```
function Cat () {}
Cat.prototype.meow = function () {};
Cat.hiss = function () {};

expect(Cat).itself.to.respondTo('hiss').but.not.respondTo('meow');
```

When not adding `.itself`, it's important to check the target's type before using `.respondTo`. See the `.a` doc for info on checking a target's type.

```
function Cat () {}
Cat.prototype.meow = function () {};

expect(new Cat()).to.be.an('object').that.respondsTo('meow');
```

Add `.not` earlier in the chain to negate `.respondTo`.

```
function Dog () {}
Dog.prototype.bark = function () {};

expect(new Dog()).to.not.respondTo('meow');
```

`.respondTo` accepts an optional `msg` argument which is a custom error message to show when the assertion fails. The message can also be given as the second argument to `expect`.

```
expect({}).to.respondTo('meow', 'nooo why fail??');
expect({}, 'nooo why fail??').to.respondTo('meow');
```

The alias `.respondsTo` can be used interchangeably with `.respondTo`.

## .itself

Forces all `.respondTo` assertions that follow in the chain to behave as if the target is a non-function object, even if it's a function. Thus, it causes `.respondTo` to assert that the target has a method with the given name, rather than asserting that the target's `prototype` property has a method with the given name.

```
function Cat () {}
Cat.prototype.meow = function () {};
Cat.hiss = function () {};

expect(Cat).itself.to.respondTo('hiss').but.not.respondTo('meow');
```

## .satisfy(matcher[, msg])

- **@param** *{ Function }* matcher
- **@param** *{ String }* msg *optional*

Invokes the given `matcher` function with the target being passed as the first argument, and asserts that the value returned is truthy.

```
expect(1).to.satisfy(function(num) {
  return num > 0;
});
```

Add `.not` earlier in the chain to negate `.satisfy`.

```
expect(1).to.not.satisfy(function(num) {
  return num > 2;
});
```

`.satisfy` accepts an optional `msg` argument which is a custom error message to show when the assertion fails. The message can also be given as the second argument to `expect`.

```
expect(1).to.satisfy(function(num) {
  return num > 2;
}, 'nooo why fail??');

expect(1, 'nooo why fail??').to.satisfy(function(num) {
  return num > 2;
});
```

The alias `.satisfies` can be used interchangeably with `.satisfy`.

## .closeTo(expected, delta[, msg])

- **@param** *{ Number }* expected
- **@param** *{ Number }* delta
- **@param** *{ String }* msg *optional*

Asserts that the target is a number that's within a given +/- `delta` range of the given number `expected`. However, it's often best to assert that the target is equal to its expected value.

```
// Recommended
expect(1.5).to.equal(1.5);

// Not recommended
expect(1.5).to.be.closeTo(1, 0.5);
expect(1.5).to.be.closeTo(2, 0.5);
expect(1.5).to.be.closeTo(1, 1);
```

Add `.not` earlier in the chain to negate `.closeTo`.

```
expect(1.5).to.equal(1.5); // Recommended
expect(1.5).to.not.be.closeTo(3, 1); // Not recommended
```

`.closeTo` accepts an optional `msg` argument which is a custom error message to show when the assertion fails. The message can also be given as the second argument to `expect`.

```
expect(1.5).to.be.closeTo(3, 1, 'nooo why fail??');
expect(1.5, 'nooo why fail??').to.be.closeTo(3, 1);
```

The alias `.approximately` can be used interchangeably with `.closeTo`.

## .members(set[, msg])

- **@param** *{ Array }* set
- **@param** *{ String }* msg *optional*

Asserts that the target array has the same members as the given array `set`.

```
expect([1, 2, 3]).to.have.members([2, 1, 3]);
expect([1, 2, 2]).to.have.members([2, 1, 2]);
```

By default, members are compared using strict (`===`) equality. Add `.deep` earlier in the chain to use deep equality instead. See the `deep-eql` project page for info on the deep equality algorithm: [https://github.com/chaijs/deep-eql](https://github.com/chaijs/deep-eql).

```
// Target array deeply (but not strictly) has member `{a: 1}`
expect([{a: 1}]).to.have.deep.members([{a: 1}]);
expect([{a: 1}]).to.not.have.members([{a: 1}]);
```

By default, order doesn't matter. Add `.ordered` earlier in the chain to require that members appear in the same order.

```
expect([1, 2, 3]).to.have.ordered.members([1, 2, 3]);
expect([1, 2, 3]).to.have.members([2, 1, 3])
  .but.not.ordered.members([2, 1, 3]);
```

By default, both arrays must be the same size. Add `.include` earlier in the chain to require that the target's members be a superset of the expected members. Note that duplicates are ignored in the subset when `.include` is added.

```
// Target array is a superset of [1, 2] but not identical
expect([1, 2, 3]).to.include.members([1, 2]);
expect([1, 2, 3]).to.not.have.members([1, 2]);

// Duplicates in the subset are ignored
expect([1, 2, 3]).to.include.members([1, 2, 2, 2]);
```

`.deep`, `.ordered`, and `.include` can all be combined. However, if `.include` and `.ordered` are combined, the ordering begins at the start of both arrays.

```
expect([{a: 1}, {b: 2}, {c: 3}])
  .to.include.deep.ordered.members([{a: 1}, {b: 2}])
  .but.not.include.deep.ordered.members([{b: 2}, {c: 3}]);
```

Add `.not` earlier in the chain to negate `.members`. However, it's dangerous to do so. The problem is that it creates uncertain expectations by asserting that the target array doesn't have all of the same members as the given array `set` but may or may not have some of them. It's often best to identify the exact output that's expected, and then write an assertion that only accepts that exact output.

```
expect([1, 2]).to.not.include(3).and.not.include(4); // Recommended
expect([1, 2]).to.not.have.members([3, 4]); // Not recommended
```

`.members` accepts an optional `msg` argument which is a custom error message to show when the assertion fails. The message can also be given as the second argument to `expect`.

```
expect([1, 2]).to.have.members([1, 2, 3], 'nooo why fail??');
expect([1, 2], 'nooo why fail??').to.have.members([1, 2, 3]);
```

## .oneOf(list[, msg])

- **@param** *{ Array.<*> }* list
- **@param** *{ String }* msg *optional*

Asserts that the target is a member of the given array `list`. However, it's often best to assert that the target is equal to its expected value.

```
expect(1).to.equal(1); // Recommended
expect(1).to.be.oneOf([1, 2, 3]); // Not recommended
```

Comparisons are performed using strict (`===`) equality.

Add `.not` earlier in the chain to negate `.oneOf`.

```
expect(1).to.equal(1); // Recommended
expect(1).to.not.be.oneOf([2, 3, 4]); // Not recommended
```

It can also be chained with `.contain` or `.include`, which will work with both arrays and strings:

```
expect('Today is sunny').to.contain.oneOf(['sunny', 'cloudy'])
expect('Today is rainy').to.not.contain.oneOf(['sunny', 'cloudy'])
expect([1,2,3]).to.contain.oneOf([3,4,5])
expect([1,2,3]).to.not.contain.oneOf([4,5,6])
```

`.oneOf` accepts an optional `msg` argument which is a custom error message to show when the assertion fails. The message can also be given as the second argument to `expect`.

```
expect(1).to.be.oneOf([2, 3, 4], 'nooo why fail??');
expect(1, 'nooo why fail??').to.be.oneOf([2, 3, 4]);
```

## .change(subject[, prop[, msg]])

- **@param** *{ String }* subject
- **@param** *{ String }* prop name *optional*
- **@param** *{ String }* msg *optional*

When one argument is provided, `.change` asserts that the given function `subject` returns a different value when it's invoked before the target function compared to when it's invoked afterward. However, it's often best to assert that `subject` is equal to its expected value.

```
var dots = ''
  , addDot = function () { dots += '.'; }
  , getDots = function () { return dots; };

// Recommended
expect(getDots()).to.equal('');
addDot();
expect(getDots()).to.equal('.');

// Not recommended
expect(addDot).to.change(getDots);
```

When two arguments are provided, `.change` asserts that the value of the given object `subject`'s `prop` property is different before invoking the target function compared to afterward.

```
var myObj = {dots: ''}
  , addDot = function () { myObj.dots += '.'; };

// Recommended
expect(myObj).to.have.property('dots', '');
addDot();
expect(myObj).to.have.property('dots', '.');

// Not recommended
expect(addDot).to.change(myObj, 'dots');
```

Strict (`===`) equality is used to compare before and after values.

Add `.not` earlier in the chain to negate `.change`.

```
var dots = ''
  , noop = function () {}
  , getDots = function () { return dots; };

expect(noop).to.not.change(getDots);

var myObj = {dots: ''}
  , noop = function () {};

expect(noop).to.not.change(myObj, 'dots');
```

`.change` accepts an optional `msg` argument which is a custom error message to show when the assertion fails. The message can also be given as the second argument to `expect`. When not providing two arguments, always use the second form.

```
var myObj = {dots: ''}
  , addDot = function () { myObj.dots += '.'; };

expect(addDot).to.not.change(myObj, 'dots', 'nooo why fail??');

var dots = ''
  , addDot = function () { dots += '.'; }
  , getDots = function () { return dots; };

expect(addDot, 'nooo why fail??').to.not.change(getDots);
```

`.change` also causes all `.by` assertions that follow in the chain to assert how much a numeric subject was increased or decreased by. However, it's dangerous to use `.change.by`. The problem is that it creates uncertain expectations by asserting that the subject either increases by the given delta, or that it decreases by the given delta. It's often best to identify the exact output that's expected, and then write an assertion that only accepts that exact output.

```
var myObj = {val: 1}
  , addTwo = function () { myObj.val += 2; }
  , subtractTwo = function () { myObj.val -= 2; };

expect(addTwo).to.increase(myObj, 'val').by(2); // Recommended
expect(addTwo).to.change(myObj, 'val').by(2); // Not recommended

expect(subtractTwo).to.decrease(myObj, 'val').by(2); // Recommended
expect(subtractTwo).to.change(myObj, 'val').by(2); // Not recommended
```

The alias `.changes` can be used interchangeably with `.change`.

## .increase(subject[, prop[, msg]])

- **@param** *{ String | Function }* subject
- **@param** *{ String }* prop name *optional*
- **@param** *{ String }* msg *optional*

When one argument is provided, `.increase` asserts that the given function `subject` returns a greater number when it's invoked after invoking the target function compared to when it's invoked beforehand. `.increase` also causes all `.by` assertions that follow in the chain to assert how much greater of a number is returned. It's often best to assert that the return value increased by the expected amount, rather than asserting it increased by any amount.

```
var val = 1
  , addTwo = function () { val += 2; }
  , getVal = function () { return val; };

expect(addTwo).to.increase(getVal).by(2); // Recommended
expect(addTwo).to.increase(getVal); // Not recommended
```

When two arguments are provided, `.increase` asserts that the value of the given object `subject`'s `prop` property is greater after invoking the target function compared to beforehand.

```
var myObj = {val: 1}
  , addTwo = function () { myObj.val += 2; };

expect(addTwo).to.increase(myObj, 'val').by(2); // Recommended
expect(addTwo).to.increase(myObj, 'val'); // Not recommended
```

Add `.not` earlier in the chain to negate `.increase`. However, it's dangerous to do so. The problem is that it creates uncertain expectations by asserting that the subject either decreases, or that it stays the same. It's often best to identify the exact output that's expected, and then write an assertion that only accepts that exact output.

When the subject is expected to decrease, it's often best to assert that it decreased by the expected amount.

```
var myObj = {val: 1}
  , subtractTwo = function () { myObj.val -= 2; };

expect(subtractTwo).to.decrease(myObj, 'val').by(2); // Recommended
expect(subtractTwo).to.not.increase(myObj, 'val'); // Not recommended
```

When the subject is expected to stay the same, it's often best to assert exactly that.

```
var myObj = {val: 1}
  , noop = function () {};

expect(noop).to.not.change(myObj, 'val'); // Recommended
expect(noop).to.not.increase(myObj, 'val'); // Not recommended
```

`.increase` accepts an optional `msg` argument which is a custom error message to show when the assertion fails. The message can also be given as the second argument to `expect`. When not providing two arguments, always use the second form.

```
var myObj = {val: 1}
  , noop = function () {};

expect(noop).to.increase(myObj, 'val', 'nooo why fail??');

var val = 1
  , noop = function () {}
  , getVal = function () { return val; };

expect(noop, 'nooo why fail??').to.increase(getVal);
```

The alias `.increases` can be used interchangeably with `.increase`.

## .decrease(subject[, prop[, msg]])

- **@param** *{ String | Function }* subject
- **@param** *{ String }* prop name *optional*
- **@param** *{ String }* msg *optional*

When one argument is provided, `.decrease` asserts that the given function `subject` returns a lesser number when it's invoked after invoking the target function compared to when it's invoked beforehand. `.decrease` also causes all `.by` assertions that follow in the chain to assert how much lesser of a number is returned. It's often best to assert that the return value decreased by the expected amount, rather than asserting it decreased by any amount.

```
var val = 1
  , subtractTwo = function () { val -= 2; }
  , getVal = function () { return val; };

expect(subtractTwo).to.decrease(getVal).by(2); // Recommended
expect(subtractTwo).to.decrease(getVal); // Not recommended
```

When two arguments are provided, `.decrease` asserts that the value of the given object `subject`'s `prop` property is lesser after invoking the target function compared to beforehand.

```
var myObj = {val: 1}
  , subtractTwo = function () { myObj.val -= 2; };

expect(subtractTwo).to.decrease(myObj, 'val').by(2); // Recommended
expect(subtractTwo).to.decrease(myObj, 'val'); // Not recommended
```

Add `.not` earlier in the chain to negate `.decrease`. However, it's dangerous to do so. The problem is that it creates uncertain expectations by asserting that the subject either increases, or that it stays the same. It's often best to identify the exact output that's expected, and then write an assertion that only accepts that exact output.

When the subject is expected to increase, it's often best to assert that it increased by the expected amount.

```
var myObj = {val: 1}
  , addTwo = function () { myObj.val += 2; };

expect(addTwo).to.increase(myObj, 'val').by(2); // Recommended
expect(addTwo).to.not.decrease(myObj, 'val'); // Not recommended
```

When the subject is expected to stay the same, it's often best to assert exactly that.

```
var myObj = {val: 1}
  , noop = function () {};

expect(noop).to.not.change(myObj, 'val'); // Recommended
expect(noop).to.not.decrease(myObj, 'val'); // Not recommended
```

`.decrease` accepts an optional `msg` argument which is a custom error message to show when the assertion fails. The message can also be given as the second argument to `expect`. When not providing two arguments, always use the second form.

```
var myObj = {val: 1}
  , noop = function () {};

expect(noop).to.decrease(myObj, 'val', 'nooo why fail??');

var val = 1
  , noop = function () {}
  , getVal = function () { return val; };

expect(noop, 'nooo why fail??').to.decrease(getVal);
```

The alias `.decreases` can be used interchangeably with `.decrease`.

## .by(delta[, msg])

- **@param** *{ Number }* delta
- **@param** *{ String }* msg *optional*

When following an `.increase` assertion in the chain, `.by` asserts that the subject of the `.increase` assertion increased by the given `delta`.

```
var myObj = {val: 1}
  , addTwo = function () { myObj.val += 2; };

expect(addTwo).to.increase(myObj, 'val').by(2);
```

When following a `.decrease` assertion in the chain, `.by` asserts that the subject of the `.decrease` assertion decreased by the given `delta`.

```
var myObj = {val: 1}
  , subtractTwo = function () { myObj.val -= 2; };

expect(subtractTwo).to.decrease(myObj, 'val').by(2);
```

When following a `.change` assertion in the chain, `.by` asserts that the subject of the `.change` assertion either increased or decreased by the given `delta`. However, it’s dangerous to use `.change.by`. The problem is that it creates uncertain expectations. It’s often best to identify the exact output that’s expected, and then write an assertion that only accepts that exact output.

```
var myObj = {val: 1}
  , addTwo = function () { myObj.val += 2; }
  , subtractTwo = function () { myObj.val -= 2; };

expect(addTwo).to.increase(myObj, 'val').by(2); // Recommended
expect(addTwo).to.change(myObj, 'val').by(2); // Not recommended

expect(subtractTwo).to.decrease(myObj, 'val').by(2); // Recommended
expect(subtractTwo).to.change(myObj, 'val').by(2); // Not recommended
```

Add `.not` earlier in the chain to negate `.by`. However, it’s often best to assert that the subject changed by its expected delta, rather than asserting that it didn’t change by one of countless unexpected deltas.

```
var myObj = {val: 1}
  , addTwo = function () { myObj.val += 2; };

// Recommended
expect(addTwo).to.increase(myObj, 'val').by(2);

// Not recommended
expect(addTwo).to.increase(myObj, 'val').but.not.by(3);
```

`.by` accepts an optional `msg` argument which is a custom error message to show when the assertion fails. The message can also be given as the second argument to `expect`.

```
var myObj = {val: 1}
  , addTwo = function () { myObj.val += 2; };

expect(addTwo).to.increase(myObj, 'val').by(3, 'nooo why fail??');
expect(addTwo, 'nooo why fail??').to.increase(myObj, 'val').by(3);
```

## .extensible

Asserts that the target is extensible, which means that new properties can be added to it. Primitives are never extensible.

```
expect({a: 1}).to.be.extensible;
```

Add `.not` earlier in the chain to negate `.extensible`.

```
var nonExtensibleObject = Object.preventExtensions({})
  , sealedObject = Object.seal({})
  , frozenObject = Object.freeze({});

expect(nonExtensibleObject).to.not.be.extensible;
expect(sealedObject).to.not.be.extensible;
expect(frozenObject).to.not.be.extensible;
expect(1).to.not.be.extensible;
```

A custom error message can be given as the second argument to `expect`.

```
expect(1, 'nooo why fail??').to.be.extensible;
```

## .sealed

Asserts that the target is sealed, which means that new properties can’t be added to it, and its existing properties can’t be reconfigured or deleted. However, it’s possible that its existing properties can still be reassigned to different values. Primitives are always sealed.

```
var sealedObject = Object.seal({});
var frozenObject = Object.freeze({});

expect(sealedObject).to.be.sealed;
expect(frozenObject).to.be.sealed;
expect(1).to.be.sealed;
```

Add `.not` earlier in the chain to negate `.sealed`.

```
expect({a: 1}).to.not.be.sealed;
```

A custom error message can be given as the second argument to `expect`.

```
expect({a: 1}, 'nooo why fail??').to.be.sealed;
```

## .frozen

Asserts that the target is frozen, which means that new properties can’t be added to it, and its existing properties can’t be reassigned to different values, reconfigured, or deleted. Primitives are always frozen.

```
var frozenObject = Object.freeze({});

expect(frozenObject).to.be.frozen;
expect(1).to.be.frozen;
```

Add `.not` earlier in the chain to negate `.frozen`.

```
expect({a: 1}).to.not.be.frozen;
```

A custom error message can be given as the second argument to `expect`.

```
expect({a: 1}, 'nooo why fail??').to.be.frozen;
```

## .finite

Asserts that the target is a number, and isn't `NaN` or positive/negative `Infinity`.

```
expect(1).to.be.finite;
```

Add `.not` earlier in the chain to negate `.finite`. However, it's dangerous to do so. The problem is that it creates uncertain expectations by asserting that the subject either isn't a number, or that it's `NaN`, or that it's positive `Infinity`, or that it's negative `Infinity`. It's often best to identify the exact output that's expected, and then write an assertion that only accepts that exact output.

When the target isn't expected to be a number, it's often best to assert that it's the expected type, rather than asserting that it isn't one of many unexpected types.

```
expect('foo').to.be.a('string'); // Recommended
expect('foo').to.not.be.finite; // Not recommended
```

When the target is expected to be `NaN`, it's often best to assert exactly that.

```
expect(NaN).to.be.NaN; // Recommended
expect(NaN).to.not.be.finite; // Not recommended
```

When the target is expected to be positive infinity, it's often best to assert exactly that.

```
expect(Infinity).to.equal(Infinity); // Recommended
expect(Infinity).to.not.be.finite; // Not recommended
```

When the target is expected to be negative infinity, it's often best to assert exactly that.

```
expect(-Infinity).to.equal(-Infinity); // Recommended
expect(-Infinity).to.not.be.finite; // Not recommended
```

A custom error message can be given as the second argument to `expect`.

```
expect('foo', 'nooo why fail??').to.be.finite;
```

## .fail([message])

## .fail(actual, expected, [message], [operator])

- **@param** *{ Mixed }* actual
- **@param** *{ Mixed }* expected
- **@param** *{ String }* message
- **@param** *{ String }* operator

Throw a failure.

```
expect.fail();
expect.fail("custom error message");
expect.fail(1, 2);
expect.fail(1, 2, "custom error message");
expect.fail(1, 2, "custom error message", ">");
expect.fail(1, 2, undefined, ">");
```

## .fail([message])

## .fail(actual, expected, [message], [operator])

- **@param** *{ Mixed }* actual
- **@param** *{ Mixed }* expected
- **@param** *{ String }* message
- **@param** *{ String }* operator

Throw a failure.

```
should.fail();
should.fail("custom error message");
should.fail(1, 2);
should.fail(1, 2, "custom error message");
should.fail(1, 2, "custom error message", ">");
should.fail(1, 2, undefined, ">");
```