**BDD**

The BDD styles are expect and should. Both use the same chainable language to construct assertions, but they differ in the way an assertion is initially constructed. Check out the [Style Guide](https://www.chaijs.com/guide/styles) for a comparison.

**API Reference**

**Language Chains**

The following are provided as chainable getters to improve the readability of your assertions.

**Chains**

* to
* be
* been
* is
* that
* which
* and
* has
* have
* with
* at
* of
* same
* but
* does
* still
* also

**.not**

Negates all assertions that follow in the chain.

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);

Just because you can negate any assertion with .not doesn’t mean you should. With great power comes great responsibility. It’s often best to assert that the one expected output was produced, rather than asserting that one of countless unexpected outputs wasn’t produced. See individual assertions for specific guidance.

expect(2).to.equal(2); // Recommended

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

**.deep**

Causes all .equal, .include, .members, .keys, and .property assertions that follow in the chain to use deep equality instead of strict (===) equality. See the deep-eql project page for info on the deep equality algorithm: https://github.com/chaijs/deep-eql.

// 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**

Enables dot- and bracket-notation in all .property and .include assertions that follow in the chain.

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

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

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

.nested cannot be combined with .own.

**.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**

Causes all .members assertions that follow in the chain to require that members be in the same order.

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**

Causes all .keys assertions that follow in the chain to only require that the target have at least one of the given keys. This is the opposite of .all, which requires that the target have all of the given keys.

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

See the .keys doc for guidance on when to use .any or .all.

**.all**

Causes all .keys assertions that follow in the chain to require that the target have all of the given keys. This is the opposite of .any, which only requires that the target have at least one of the given keys.

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

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

See the .keys doc for guidance on when to use .any or .all.

**.a(type[, msg])**

* @param { String } type
* @param { String } msg \_optional\_

Asserts that the target’s type is equal to the given string type. Types are case insensitive. See the type-detect project page for info on the type detection algorithm: https://github.com/chaijs/type-detect.

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');

.a supports objects that have a custom type set via Symbol.toStringTag.

var myObj = {

[Symbol.toStringTag]: 'myCustomType'

};

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

It’s often best to use .a to check a target’s type before making more assertions on the same target. That way, you avoid unexpected behavior from any assertion that does different things based on the target’s type.

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

expect([]).to.be.an('array').that.is.empty;

Add .not earlier in the chain to negate .a. However, it’s often best to assert that the target is 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.an('array'); // Not recommended

.a 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.a('string', 'nooo why fail??');

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

.a can also be used as a language chain to improve the readability of your assertions.

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

The alias .an can be used interchangeably with .a.

**.include(val[, msg])**

* @param { Mixed } val
* @param { String } msg \_optional\_

When the target is a string, .include asserts that the given string val is a substring of the target.

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

When the target is an array, .include asserts that the given val is a member of the target.

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

When the target is an object, .include asserts that the given object val’s properties are a subset of the target’s properties.

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

When the target is a Set or WeakSet, .include asserts that the given val is a member of the target. SameValueZero equality algorithm is used.

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

When the target is a Map, .include asserts that the given val is one of the values of the target. SameValueZero equality algorithm is used.

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

Because .include does different things based on the target’s type, it’s important to check the target’s type before using .include. See the .a doc for info on testing a target’s type.

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

By default, strict (===) equality is used to compare array members and object properties. Add .deep earlier in the chain to use deep equality instead (WeakSet targets are not supported). See the deep-eql project page for info on the deep equality algorithm: https://github.com/chaijs/deep-eql.

// 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}});

By default, all of the target’s properties are searched when working with objects. This includes properties that are inherited and/or non-enumerable. Add .own earlier in the chain to exclude the target’s inherited properties from the search.

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});

Note that a target object is always only searched for val’s own enumerable properties.

.deep and .own can be combined.

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

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

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

.deep and .nested can be combined.

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

.own and .nested cannot be combined.

Add .not earlier in the chain to negate .include.

expect('foobar').to.not.include('taco');

expect([1, 2, 3]).to.not.include(4);

However, it’s dangerous to negate .include when the target is an object. The problem is that it creates uncertain expectations by asserting that the target object doesn’t have all of val’s key/value pairs 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.

When the target object isn’t even expected to have val’s keys, it’s often best to assert exactly that.

expect({c: 3}).to.not.have.any.keys('a', 'b'); // Recommended

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

When the target object is expected to have val’s keys, it’s often best to assert that each of the properties has its expected value, rather than asserting that each property doesn’t have one of many unexpected values.

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

.include 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, 3]).to.include(4, 'nooo why fail??');

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

.include can also be used as a language chain, causing all .members and .keys assertions that follow in the chain to require the target to be a superset of the expected set, rather than an identical set. Note that .members ignores duplicates in the subset when .include is added.

// 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]);

Note that adding .any earlier in the chain causes the .keys assertion to ignore .include.

// Both assertions are identical

expect({a: 1}).to.include.any.keys('a', 'b');

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

The aliases .includes, .contain, and .contains can be used interchangeably with .include.

**.ok**

Asserts that the target is a truthy value (considered true in boolean context). However, it’s often best to assert that the target is strictly (===) or deeply equal to its expected value.

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

Add .not earlier in the chain to negate .ok.

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

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

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

**.true**

Asserts that the target is strictly (===) equal to true.

expect(true).to.be.true;

Add .not earlier in the chain to negate .true. However, it’s often best to assert that the target is equal to its expected value, rather than not equal to 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

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

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

**.false**

Asserts that the target is strictly (===) equal to false.

expect(false).to.be.false;

Add .not earlier in the chain to negate .false. However, it’s often best to assert that the target is equal to its expected value, rather than not equal to 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

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

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

**.null**

Asserts that the target is strictly (===) equal to null.

expect(null).to.be.null;

Add .not earlier in the chain to negate .null. However, it’s often best to assert that the target is equal to its expected value, rather than not equal to null.

expect(1).to.equal(1); // Recommended

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

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

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

**.undefined**

Asserts that the target is strictly (===) equal to undefined.

expect(undefined).to.be.undefined;

Add .not earlier in the chain to negate .undefined. However, it’s often best to assert that the target is equal to its expected value, rather than not equal to 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**

Asserts that the target is exactly NaN.

expect(NaN).to.be.NaN;

Add .not earlier in the chain to negate .NaN. However, it’s often best to assert that the target is equal to its expected value, rather than not equal to 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**

Asserts that the target is not strictly (===) equal to either null or undefined. 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.exist; // Not recommended

expect(0).to.equal(0); // Recommended

expect(0).to.exist; // Not recommended

Add .not earlier in the chain to negate .exist.

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

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

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

The alias .exists can be used interchangeably with .exist.

**.empty**

When the target is a string or array, .empty asserts that the target’s length property is strictly (===) equal to 0.

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

expect('').to.be.empty;

When the target is a map or set, .empty asserts that the target’s size property is strictly equal to 0.

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

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

When the target is a non-function object, .empty asserts that the target doesn’t have any own enumerable properties. Properties with Symbol-based keys are excluded from the count.

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

Because .empty does different things based on the target’s type, it’s important to check the target’s type before using .empty. See the .a doc for info on testing a target’s type.

expect([]).to.be.an('array').that.is.empty;

Add .not earlier in the chain to negate .empty. However, it’s often best to assert that the target contains its expected number of values, rather than asserting that it’s not 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

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

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

**.arguments**

Asserts that the target is an arguments object.

function test () {

expect(arguments).to.be.arguments;

}

test();

Add .not earlier in the chain to negate .arguments. However, it’s often best to assert which type the target is expected to be, rather than asserting that it’s not an arguments object.

expect('foo').to.be.a('string'); // Recommended

expect('foo').to.not.be.arguments; // Not recommended

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

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

The alias .Arguments can be used interchangeably with .arguments.

**.equal(val[, msg])**

* @param { Mixed } val
* @param { String } msg \_optional\_

Asserts that the target is strictly (===) equal to the given val.

expect(1).to.equal(1);

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

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.

// 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]);

Add .not earlier in the chain to negate .equal. However, it’s often best to assert that the target is equal to its expected value, rather than not equal to one of countless unexpected values.

expect(1).to.equal(1); // Recommended

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

.equal 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.equal(2, 'nooo why fail??');

expect(1, 'nooo why fail??').to.equal(2);

The aliases .equals and eq can be used interchangeably with .equal.

**.eql(obj[, msg])**

* @param { Mixed } obj
* @param { String } msg \_optional\_

Asserts that the target is deeply equal to the given obj. See the deep-eql project page for info on the deep equality algorithm: https://github.com/chaijs/deep-eql.

// 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]);

Add .not earlier in the chain to negate .eql. However, it’s often best to assert that the target is deeply equal to its expected value, rather than not deeply equal to one of countless unexpected values.

expect({a: 1}).to.eql({a: 1}); // Recommended

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

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

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

The alias .eqls can be used interchangeably with .eql.

The .deep.equal assertion is almost identical to .eql but with one difference: .deep.equal causes deep equality comparisons to also be used for any other assertions that follow in the chain.

**.above(n[, msg])**

* @param { Number } n
* @param { String } msg \_optional\_

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

expect(2).to.equal(2); // Recommended

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

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

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

Add .not earlier in the chain to negate .above.

expect(2).to.equal(2); // Recommended

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

.above 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.above(2, 'nooo why fail??');

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

The aliases .gt and .greaterThan can be used interchangeably with .above.

**.least(n[, msg])**

* @param { Number } n
* @param { String } msg \_optional\_

Asserts that the target is a number or a date greater 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(2).to.equal(2); // Recommended

expect(2).to.be.at.least(1); // Not recommended

expect(2).to.be.at.least(2); // Not recommended

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

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

Add .not earlier in the chain to negate .least.

expect(1).to.equal(1); // Recommended

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

.least 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.at.least(2, 'nooo why fail??');

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

The aliases .gte and .greaterThanOrEqual can be used interchangeably with .least.

**.below(n[, msg])**

* @param { Number } n
* @param { String } msg \_optional\_

Asserts that the target is a number or a date less than 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.below(2); // Not recommended

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

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

Add .not earlier in the chain to negate .below.

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])**

* @param { Number } start lower bound inclusive
* @param { Number } finish upper bound inclusive
* @param { String } msg \_optional\_

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

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

Add .lengthOf earlier in the chain to assert that the target’s length or size is greater than or equal to the given number start, and less than or equal to the given number finish.

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

Add .not earlier in the chain to negate .within.

expect(1).to.equal(1); // Recommended

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

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

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

**.instanceof(constructor[, msg])**

* @param { Constructor } constructor
* @param { String } msg \_optional\_

Asserts that the target is an instance of the given constructor.

function Cat () { }

expect(new Cat()).to.be.an.instanceof(Cat);

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

Add .not earlier in the chain to negate .instanceof.

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

.instanceof 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.an.instanceof(Array, 'nooo why fail??');

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

Due to limitations in ES5, .instanceof may not always work as expected when using a transpiler such as Babel or TypeScript. In particular, it may produce unexpected results when subclassing built-in object such as Array, Error, and Map. See your transpiler’s docs for details:

* ([Babel](https://babeljs.io/docs/usage/caveats/#classes))
* ([TypeScript](https://github.com/Microsoft/TypeScript/wiki/Breaking-Changes#extending-built-ins-like-error-array-and-map-may-no-longer-work))

The alias .instanceOf can be used interchangeably with .instanceof.

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

* @param { String } name
* @param { Mixed } val (optional)
* @param { String } msg \_optional\_

Asserts that the target has a property with the given key name.

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

When val is provided, .property also asserts that the property’s value is equal to the given val.

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

By default, strict (===) equality is used. 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.

// 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});

The target’s enumerable and non-enumerable properties are always included in the search. By default, both own and inherited properties are included. Add .own earlier in the chain to exclude inherited properties from the search.

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]])**

* @param { String } name
* @param { Object } descriptor \_optional\_
* @param { String } msg \_optional\_

Asserts that the target has its own property descriptor with the given key name. Enumerable and non-enumerable properties are included in the search.

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

When descriptor is provided, .ownPropertyDescriptor also asserts that the property’s descriptor is deeply equal to the given descriptor. See the deep-eql project page for info on the deep equality algorithm: https://github.com/chaijs/deep-eql.

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

configurable: true,

enumerable: true,

writable: true,

value: 1,

});

Add .not earlier in the chain to negate .ownPropertyDescriptor.

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

However, it’s dangerous to negate .ownPropertyDescriptor when providing a descriptor. The problem is that it creates uncertain expectations by asserting that the target either doesn’t have a property descriptor with the given key name, or that it does have a property descriptor with the given key name but it’s not deeply equal to the given descriptor. 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 descriptor with the given key name, it’s often best to assert exactly that.

// 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,

});

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

// 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,

});

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

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

.that.has.property('enumerable', true);

.ownPropertyDescriptor 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 descriptor, only use the second form.

// 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??');

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

The alias .haveOwnPropertyDescriptor can be used interchangeably with .ownPropertyDescriptor.

**.lengthOf(n[, msg])**

* @param { Number } n
* @param { String } msg \_optional\_

Asserts that the target’s length or size is equal to the given number n.

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);

Add .not earlier in the chain to negate .lengthOf. However, it’s often best to assert that the target’s length property is equal to its expected value, rather than not equal to one of many unexpected values.

expect('foo').to.have.lengthOf(3); // Recommended

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

.lengthOf 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, 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])**

* @param { RegExp } re
* @param { String } msg \_optional\_

Asserts that the target matches the given regular expression re.

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])**

* @param { String } str
* @param { String } msg \_optional\_

Asserts that the target string contains the given substring str.

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

Add .not earlier in the chain to negate .string.

expect('foobar').to.not.have.string('taco');

.string 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.have.string('taco', 'nooo why fail??');

expect('foobar', 'nooo why fail??').to.have.string('taco');

**.keys(key1[, key2[, …]])**

* @param { String | Array | Object } keys

Asserts that the target object, array, map, or set has the given keys. Only the target’s own inherited properties are included in the search.

When the target is an object or array, keys can be provided as one or more string arguments, a single array argument, or a single object argument. In the latter case, only the keys in the given object matter; the values are ignored.

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

When the target is a map or set, each key must be provided as a separate argument.

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

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

Because .keys does different things based on the target’s type, it’s important to check the target’s type before using .keys. See the .a doc for info on testing a target’s type.

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

By default, strict (===) equality is used to compare keys of maps and sets. 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.

// 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])**

* @param { Error | ErrorConstructor } errorLike
* @param { String | RegExp } errMsgMatcher error message
* @param { String } msg \_optional\_
* @see <https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Error#Error_types>

When no arguments are provided, .throw invokes the target function and asserts that an error is thrown.

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

expect(badFn).to.throw();

When one argument is provided, and it’s an error constructor, .throw invokes the target function and asserts that an error is thrown that’s an instance of that error constructor.

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

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

When one argument is provided, and it’s an error instance, .throw invokes the target function and asserts that an error is thrown that’s strictly (===) equal to that error instance.

var err = new TypeError('Illegal salmon!');

var badFn = function () { throw err; };

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

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

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/);

When two arguments are provided, and the first is an error instance or constructor, and the second is a string or regular expression, .throw invokes the function and asserts that an error is thrown that fulfills both conditions as described above.

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/);

Add .not earlier in the chain to negate .throw.

var goodFn = function () {};

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

However, it’s dangerous to negate .throw when providing any arguments. The problem is that it creates uncertain expectations by asserting that the target either doesn’t throw an error, or that it throws an error but of a different type than the given type, or that it throws an error of the given type but with a message that doesn’t include the given string. 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 throw an error, it’s often best to assert exactly that.

var goodFn = function () {};

expect(goodFn).to.not.throw(); // Recommended

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

When the target is expected to throw an error, it’s often best to assert that the error is of its expected type, and has a message that includes an expected string, rather than asserting that it doesn’t have one of many unexpected types, and doesn’t have a message that includes some string.

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

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

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

.throw changes the target of any assertions that follow in the chain to be the error object that’s thrown.

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](https://babeljs.io/docs/usage/caveats/#classes))
* ([TypeScript](https://github.com/Microsoft/TypeScript/wiki/Breaking-Changes#extending-built-ins-like-error-array-and-map-may-no-longer-work))

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.

// 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, ">");