

---

## should.js

gitter [join chat](#)

build [unknown](#)

 [unknown](#)

*should* is an expressive, readable, framework-agnostic assertion library. The main goals of this library are **to be expressive** and **to be helpful**. It keeps your test code clean, and your error messages helpful.

By default (when you `require('should')`) *should* extends the `Object.prototype` with a single non-enumerable getter that allows you to express how that object should behave. It also returns itself when required with `require`.

It is also possible to use *should.js* without getter (it will not even try to extend `Object.prototype`), just `require('should/as-function')`. Or if you already use version that auto add getter, you can call `.noConflict` function.

**Results of `(something).should` getter and `should(something)` in most situations are the same**

### Upgrading instructions

Please check wiki page for upgrading instructions.

### FAQ

You can take look in FAQ.

### Example

```
1 var should = require('should');
2
3 var user = {
4   name: 'tj'
5   , pets: ['tobi', 'loki', 'jane', 'bandit']
6 };
7
8 user.should.have.property('name', 'tj');
9 user.should.have.property('pets').with.lengthOf(4);
```

---

```
10
11 // If the object was created with Object.create(null)
12 // then it doesn't inherit `Object.prototype`, so it will not have `.
   should` getter
13 // so you can do:
14 should(user).have.property('name', 'tj');
15
16 // also you can test in that way for null's
17 should(null).not.be.ok();
18
19 someAsyncTask(foo, function(err, result){
20   should.not.exist(err);
21   should.exist(result);
22   result.bar.should.equal(foo);
23 });
```

## To begin

1. Install it:

```
1 $ npm install should --save-dev
```

2. Require it and use:

```
1 var should = require('should');
2
3 (5).should.be.exactly(5).and.be.a.Number();
```

```
1 var should = require('should/as-function');
2
3 should(10).be.exactly(5).and.be.a.Number();
```

3. For TypeScript users:

```
1 import * as should from 'should';
2
3 (0).should.be.Number();
```

## In browser

Well, even when browsers by complaints of authors have 100% es5 support, it does not mean it has no bugs. Please see wiki for known bugs.

If you want to use *should* in browser, use the `should.js` file in the root of this repository, or build it yourself. To build a fresh version:

---

```
1 $ npm install
2 $ npm run browser
```

The script is exported to `window.should`:

```
1 should(10).be.exactly(10)
```

You can easily install it with npm or bower:

```
1 npm install should -D
2 # or
3 bower install shouldjs/should.js
```

## API docs

Actual api docs generated by jsdoc comments and available at <http://shouldjs.github.io>.

## Usage examples

Please look on usage in examples

### **.not**

`.not` negates the current assertion.

### **.any**

`.any` allow for assertions with multiple parameters to assert any of the parameters (but not all). This is similar to the native JavaScript `array.some`.

## Assertions

### chaining assertions

Every assertion will return a `should.js`-wrapped Object, so assertions can be chained. To help chained assertions read more clearly, you can use the following helpers anywhere in your chain: `.an`, `.of`, `.a`, `.and`, `.be`, `.have`, `.with`, `.is`, `.which`. Use them for better readability; they do nothing at all. For example:

---

```
1 user.should.be.an.instanceOf(Object).and.have.property('name', 'tj');
2 user.pets.should.be.instanceof(Array).and.have.lengthOf(4);
```

Almost all assertions return the same object - so you can easily chain them. But some (eg: `.length` and `.property`) move the assertion object to a property value, so be careful.

## Adding own assertions

Adding own assertion is pretty easy. You need to call `should.Assertion.add` function. It accepts 2 arguments:

1. name of assertion method (string)
2. assertion function (function)

What assertion function should do. It should check only positive case. `should` will handle `.not` itself. `this` in assertion function will be instance of `should.Assertion` and you **must** define in any way this.params object in your assertion function call before assertion check happens.

`params` object can contain several fields:

- `operator` - it is string which describes your assertion
- `actual` it is actual value, you can assume it is your own `this.obj` if you need to define your own
- `expected` it is any value that is expected to be matched `this.obj`

You can assume its usage in generating `AssertionError` message like: expected `obj`? || `this.obj` not? `operator expected`?

In `should` sources appeared 2 kinds of usage of this method.

First not preferred and used **only** for shortcuts to other assertions, e.g. how `.should.be.true()` defined:

```
1 Assertion.add('true', function() {
2   this.is.exactly(true);
3 });
```

There you can see that assertion function does not define own `this.params` and instead calls within the same assertion `.exactly` that will fill `this.params`. **You should use this way very carefully, but you can use it.**

Second way preferred and I assume you will use it instead of first.

```
1 Assertion.add('true', function() {
2   this.params = { operator: 'to be true', expected: true };
3 });
```

---

```
4   should(this.obj).be.exactly(true);  
5 });
```

in this case `this.params` defined and then used new assertion context (because called `.should`). Internally this way does not create any edge cases as first.

```
1  Assertion.add('asset', function() {  
2    this.params = { operator: 'to be asset' };  
3  
4    this.obj.should.have.property('id').which.is.a.Number();  
5    this.obj.should.have.property('path');  
6  })  
7  
8  //then  
9  > ({ id: '10' }).should.be.an.asset();  
10 AssertionError: expected { id: '10' } to be asset  
11    expected '10' to be a number  
12  
13 > ({ id: 10 }).should.be.an.asset();  
14 AssertionError: expected { id: 10 } to be asset  
15    expected { id: 10 } to have property path
```

## Additional projects

- [should-sinon](#) - adds additional assertions for sinon.js
- [should-immutable](#) - extends different parts of should.js to make immutable.js first-class citizen in should.js
- [should-http](#) - adds small assertions for assertion on http responses for node only
- [should-jq](#) - assertions for jq (need maintainer)
- [karma-should](#) - make more or less easy to work karma with should.js
- [should-spies](#) - small and dirty simple zero dependencies spies

## Contributions

Actual list of contributors if you want to show it your friends.

To run the tests for *should* simply run:

```
1 $ npm test
```

See also CONTRIBUTING.

---

## OMG IT EXTENDS OBJECT???!?!@

Yes, yes it does, with a single getter *should*, and no it won't break your code, because it does this **properly** with a non-enumerable property.

Also it is possible use it without extension. Just use `require('should/as-function')` everywhere.

## License

MIT. See LICENSE for details.