Assert

Stability: 2 - Stable

Source Code: lib/assert.js

The `node:assert` module provides a set of assertion functions for verifying invariants.

Strict assertion mode

▶ History

In strict assertion mode, non-strict methods behave like their corresponding strict methods. For example, `assert.deepEqual()` will behave like `assert.deepStrictEqual()`.

In strict assertion mode, error messages for objects display a diff. In legacy assertion mode, error messages for objects display the objects, often truncated.

To use strict assertion mode:

MJS modules

```
import { strict as assert } from 'node:assert';
```

CJS modules

```
const assert = require('node:assert').strict;
```

MJS modules

```
import assert from 'node:assert/strict';
```

CJS modules

```
const assert = require('node:assert/strict');
```

Example error diff:

MJS modules

```
import { strict as assert } from 'node:assert';

assert.deepEqual([[[1, 2, 3]], 4, 5], [[[1, 2, '3']], 4, 5]);
// AssertionError: Expected inputs to be strictly deep-equal:
// + actual - expected ... Lines skipped
//
```

```
//   [
//     [
// ...
//       2,
// +     3
// -     '3'
//     ],
// ...
//     5
//   ]
```

CJS modules

```cjs
const assert = require('node:assert/strict');

assert.deepEqual([[[1, 2, 3]], 4, 5], [[[1, 2, '3']], 4, 5]);
// AssertionError: Expected inputs to be strictly deep-equal:
// + actual - expected ... Lines skipped
//
//   [
//     [
// ...
//       2,
// +     3
// -     '3'
//     ],
// ...
//     5
//   ]
```

To deactivate the colors, use the `NO_COLOR` or `NODE_DISABLE_COLORS` environment variables. This will also deactivate the colors in the REPL. For more on color support in terminal environments, read the tty `getColorDepth()` documentation.

Legacy assertion mode

Legacy assertion mode uses the == operator in:

- `assert.deepEqual()`
- `assert.equal()`
- `assert.notDeepEqual()`
- `assert.notEqual()`

To use legacy assertion mode:

MJS modules

```mjs
import assert from 'node:assert';
```

CJS modules

```
const assert = require('node:assert');
```

Legacy assertion mode may have surprising results, especially when using `assert.deepEqual()`:

```
// WARNING: This does not throw an AssertionError in legacy assertion mode!
assert.deepEqual(/a/gi, new Date()); copy
```

Class: assert.AssertionError [src]

- Extends: <errors.Error>

Indicates the failure of an assertion. All errors thrown by the `node:assert` module will be instances of the `AssertionError` class.

```
new assert.AssertionError(options)
```

Added in: v0.1.21

- `options` <Object>
  - `message` <string> If provided, the error message is set to this value.
  - `actual` <any> The `actual` property on the error instance.
  - `expected` <any> The `expected` property on the error instance.
  - `operator` <string> The `operator` property on the error instance.
  - `stackStartFn` <Function> If provided, the generated stack trace omits frames before this function.

A subclass of `Error` that indicates the failure of an assertion.

All instances contain the built-in `Error` properties (`message` and `name`) and:

- `actual` <any> Set to the `actual` argument for methods such as `assert.strictEqual()`.
- `expected` <any> Set to the `expected` value for methods such as `assert.strictEqual()`.
- `generatedMessage` <boolean> Indicates if the message was auto-generated (`true`) or not.
- `code` <string> Value is always `ERR_ASSERTION` to show that the error is an assertion error.
- `operator` <string> Set to the passed in operator value.

MJS modules

```
import assert from 'node:assert';

// Generate an AssertionError to compare the error message later:
const { message } = new assert.AssertionError({
  actual: 1,
  expected: 2,
  operator: 'strictEqual',
```

```
});

// Verify error output:
try {
  assert.strictEqual(1, 2);
} catch (err) {
  assert(err instanceof assert.AssertionError);
  assert.strictEqual(err.message, message);
  assert.strictEqual(err.name, 'AssertionError');
  assert.strictEqual(err.actual, 1);
  assert.strictEqual(err.expected, 2);
  assert.strictEqual(err.code, 'ERR_ASSERTION');
  assert.strictEqual(err.operator, 'strictEqual');
  assert.strictEqual(err.generatedMessage, true);
}
```

CJS modules

```
const assert = require('node:assert');

// Generate an AssertionError to compare the error message later:
const { message } = new assert.AssertionError({
  actual: 1,
  expected: 2,
  operator: 'strictEqual',
});

// Verify error output:
try {
  assert.strictEqual(1, 2);
} catch (err) {
  assert(err instanceof assert.AssertionError);
  assert.strictEqual(err.message, message);
  assert.strictEqual(err.name, 'AssertionError');
  assert.strictEqual(err.actual, 1);
  assert.strictEqual(err.expected, 2);
  assert.strictEqual(err.code, 'ERR_ASSERTION');
  assert.strictEqual(err.operator, 'strictEqual');
  assert.strictEqual(err.generatedMessage, true);
}
```

Class: `assert.CallTracker`

▶ History

Stability: 0 - Deprecated

This feature is deprecated and will be removed in a future version. Please consider using alternatives such as the `mock` helper function.

```
new assert.CallTracker()
```

Added in: v14.2.0, v12.19.0

Creates a new `CallTracker` object which can be used to track if functions were called a specific number of times. The `tracker.verify()` must be called for the verification to take place. The usual pattern would be to call it in a `process.on('exit')` handler.

MJS modules

```mjs
import assert from 'node:assert';
import process from 'node:process';

const tracker = new assert.CallTracker();

function func() {}

// callsfunc() must be called exactly 1 time before tracker.verify().
const callsfunc = tracker.calls(func, 1);

callsfunc();

// Calls tracker.verify() and verifies if all tracker.calls() functions have
// been called exact times.
process.on('exit', () => {
  tracker.verify();
});
```

CJS modules

```cjs
const assert = require('node:assert');

const tracker = new assert.CallTracker();

function func() {}

// callsfunc() must be called exactly 1 time before tracker.verify().
const callsfunc = tracker.calls(func, 1);

callsfunc();

// Calls tracker.verify() and verifies if all tracker.calls() functions have
// been called exact times.
process.on('exit', () => {
  tracker.verify();
});
```

```
tracker.calls([fn][, exact])
```

Added in: v14.2.0, v12.19.0

- `fn` `<Function>` **Default:** A no-op function.
- `exact` `<number>` **Default:** `1`.
- Returns: `<Function>` A function that wraps `fn`.

The wrapper function is expected to be called exactly `exact` times. If the function has not been called exactly `exact` times when `tracker.verify()` is called, then `tracker.verify()` will throw an error.

MJS modules

```mjs
import assert from 'node:assert';

// Creates call tracker.
const tracker = new assert.CallTracker();

function func() {}

// Returns a function that wraps func() that must be called exact times
// before tracker.verify().
const callsfunc = tracker.calls(func);
```

CJS modules

```cjs
const assert = require('node:assert');

// Creates call tracker.
const tracker = new assert.CallTracker();

function func() {}

// Returns a function that wraps func() that must be called exact times
// before tracker.verify().
const callsfunc = tracker.calls(func);
```

```
tracker.getCalls(fn)
```

Added in: v18.8.0, v16.18.0

- `fn` <Function>
- Returns: <Array> An array with all the calls to a tracked function.
- Object <Object>
  - `thisArg` <Object>
  - `arguments` <Array> the arguments passed to the tracked function

MJS modules

```mjs
import assert from 'node:assert';

const tracker = new assert.CallTracker();

function func() {}
```

```
const callsfunc = tracker.calls(func);
callsfunc(1, 2, 3);

assert.deepStrictEqual(tracker.getCalls(callsfunc),
                       [{ thisArg: undefined, arguments: [1, 2, 3] }]);
```

CJS modules

```
const assert = require('node:assert');

// Creates call tracker.
const tracker = new assert.CallTracker();

function func() {}
const callsfunc = tracker.calls(func);
callsfunc(1, 2, 3);

assert.deepStrictEqual(tracker.getCalls(callsfunc),
                       [{ thisArg: undefined, arguments: [1, 2, 3] }]);
```

```
tracker.report()
```

Added in: v14.2.0, v12.19.0

- Returns: <Array> An array of objects containing information about the wrapper functions returned by `tracker.calls()`.
- Object <Object>
  - `message` <string>
  - `actual` <number> The actual number of times the function was called.
  - `expected` <number> The number of times the function was expected to be called.
  - `operator` <string> The name of the function that is wrapped.
  - `stack` <Object> A stack trace of the function.

The arrays contains information about the expected and actual number of calls of the functions that have not been called the expected number of times.

MJS modules

```
import assert from 'node:assert';

// Creates call tracker.
const tracker = new assert.CallTracker();

function func() {}

// Returns a function that wraps func() that must be called exact times
// before tracker.verify().
const callsfunc = tracker.calls(func, 2);

// Returns an array containing information on callsfunc()
```

```
console.log(tracker.report());
// [
//  {
//    message: 'Expected the func function to be executed 2 time(s) but was
//    executed 0 time(s).',
//    actual: 0,
//    expected: 2,
//    operator: 'func',
//    stack: stack trace
//  }
// ]
```

CJS modules

```
const assert = require('node:assert');

// Creates call tracker.
const tracker = new assert.CallTracker();

function func() {}

// Returns a function that wraps func() that must be called exact times
// before tracker.verify().
const callsfunc = tracker.calls(func, 2);

// Returns an array containing information on callsfunc()
console.log(tracker.report());
// [
//  {
//    message: 'Expected the func function to be executed 2 time(s) but was
//    executed 0 time(s).',
//    actual: 0,
//    expected: 2,
//    operator: 'func',
//    stack: stack trace
//  }
// ]
```

```
tracker.reset([fn])
```

Added in: v18.8.0, v16.18.0

- `fn` <Function> a tracked function to reset.

Reset calls of the call tracker. If a tracked function is passed as an argument, the calls will be reset for it. If no arguments are passed, all tracked functions will be reset.

MJS modules

```mjs
import assert from 'node:assert';

const tracker = new assert.CallTracker();

function func() {}
const callsfunc = tracker.calls(func);

callsfunc();
// Tracker was called once
assert.strictEqual(tracker.getCalls(callsfunc).length, 1);

tracker.reset(callsfunc);
assert.strictEqual(tracker.getCalls(callsfunc).length, 0);
```

CJS modules

```cjs
const assert = require('node:assert');

const tracker = new assert.CallTracker();

function func() {}
const callsfunc = tracker.calls(func);

callsfunc();
// Tracker was called once
assert.strictEqual(tracker.getCalls(callsfunc).length, 1);

tracker.reset(callsfunc);
assert.strictEqual(tracker.getCalls(callsfunc).length, 0);
```

```
tracker.verify()
```

Iterates through the list of functions passed to `tracker.calls()` and will throw an error for functions that have not been called the expected number of times.

Added in: v14.2.0, v12.19.0

MJS modules

```mjs
import assert from 'node:assert';

// Creates call tracker.
const tracker = new assert.CallTracker();

function func() {}

// Returns a function that wraps func() that must be called exact times
// before tracker.verify().
const callsfunc = tracker.calls(func, 2);
```

```
callsfunc();

// Will throw an error since callsfunc() was only called once.
tracker.verify();
```

CJS modules

```
const assert = require('node:assert');

// Creates call tracker.
const tracker = new assert.CallTracker();

function func() {}

// Returns a function that wraps func() that must be called exact times
// before tracker.verify().
const callsfunc = tracker.calls(func, 2);

callsfunc();

// Will throw an error since callsfunc() was only called once.
tracker.verify();
```

```
assert(value[, message])
```

Added in: v0.5.9

- `value` <any> The input that is checked for being truthy.
- `message` <string> | <Error>

An alias of `assert.ok()`.

```
assert.deepEqual(actual, expected[, message])
```

▶ History

- `actual` <any>
- `expected` <any>
- `message` <string> | <Error>

Strict assertion mode

An alias of `assert.deepStrictEqual()`.

Legacy assertion mode

Stability: 3 - Legacy: Use `assert.deepStrictEqual()` instead.

Tests for deep equality between the `actual` and `expected` parameters. Consider using `assert.deepStrictEqual()` instead. `assert.deepEqual()` can have surprising results.

Deep equality means that the enumerable "own" properties of child objects are also recursively evaluated by the following rules.

Comparison details

- Primitive values are compared with the `==` operator, with the exception of `NaN`. It is treated as being identical in case both sides are `NaN`.
- Type tags of objects should be the same.
- Only enumerable "own" properties are considered.
- `Error` names, messages, causes, and errors are always compared, even if these are not enumerable properties.
- Object wrappers are compared both as objects and unwrapped values.
- `Object` properties are compared unordered.
- `Map` keys and `Set` items are compared unordered.
- Recursion stops when both sides differ or both sides encounter a circular reference.
- Implementation does not test the `[[Prototype]]` of objects.
- `Symbol` properties are not compared.
- `WeakMap` and `WeakSet` comparison does not rely on their values.
- `RegExp` lastIndex, flags, and source are always compared, even if these are not enumerable properties.

The following example does not throw an `AssertionError` because the primitives are compared using the `==` operator.

MJS modules

```
import assert from 'node:assert';
// WARNING: This does not throw an AssertionError!

assert.deepEqual('+00000000', false);
```

CJS modules

```
const assert = require('node:assert');
// WARNING: This does not throw an AssertionError!

assert.deepEqual('+00000000', false);
```

"Deep" equality means that the enumerable "own" properties of child objects are evaluated also:

MJS modules

```mjs
import assert from 'node:assert';

const obj1 = {
  a: {
    b: 1,
  },
};
const obj2 = {
  a: {
    b: 2,
  },
};
const obj3 = {
  a: {
    b: 1,
  },
};
const obj4 = { __proto__: obj1 };

assert.deepEqual(obj1, obj1);
// OK

// Values of b are different:
assert.deepEqual(obj1, obj2);
// AssertionError: { a: { b: 1 } } deepEqual { a: { b: 2 } }

assert.deepEqual(obj1, obj3);
// OK

// Prototypes are ignored:
assert.deepEqual(obj1, obj4);
// AssertionError: { a: { b: 1 } } deepEqual {}
```

CJS modules

```cjs
const assert = require('node:assert');

const obj1 = {
  a: {
    b: 1,
  },
};
const obj2 = {
  a: {
    b: 2,
  },
};
const obj3 = {
  a: {
    b: 1,
  },
```

```
};
const obj4 = { __proto__: obj1 };

assert.deepEqual(obj1, obj1);
// OK

// Values of b are different:
assert.deepEqual(obj1, obj2);
// AssertionError: { a: { b: 1 } } deepEqual { a: { b: 2 } }

assert.deepEqual(obj1, obj3);
// OK

// Prototypes are ignored:
assert.deepEqual(obj1, obj4);
// AssertionError: { a: { b: 1 } } deepEqual {}
```

If the values are not equal, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned. If the `message` parameter is an instance of an `Error` then it will be thrown instead of the `AssertionError`.

```
assert.deepStrictEqual(actual, expected[, message])
```

▶ History

- `actual` <any>
- `expected` <any>
- `message` <string> | <Error>

Tests for deep equality between the `actual` and `expected` parameters. "Deep" equality means that the enumerable "own" properties of child objects are recursively evaluated also by the following rules.

Comparison details

- Primitive values are compared using `Object.is()`.
- Type tags of objects should be the same.
- `[[Prototype]]` of objects are compared using the `===` operator.
- Only enumerable "own" properties are considered.
- `Error` names, messages, causes, and errors are always compared, even if these are not enumerable properties. `errors` is also compared.
- Enumerable own `Symbol` properties are compared as well.
- Object wrappers are compared both as objects and unwrapped values.
- `Object` properties are compared unordered.
- `Map` keys and `Set` items are compared unordered.
- Recursion stops when both sides differ or both sides encounter a circular reference.
- `WeakMap` and `WeakSet` comparison does not rely on their values. See below for further details.

- RegExp lastIndex, flags, and source are always compared, even if these are not enumerable properties.

MJS modules

```
import assert from 'node:assert/strict';

// This fails because 1 !== '1'.
assert.deepStrictEqual({ a: 1 }, { a: '1' });
// AssertionError: Expected inputs to be strictly deep-equal:
// + actual - expected
//
//   {
// +   a: 1
// -   a: '1'
//   }

// The following objects don't have own properties
const date = new Date();
const object = {};
const fakeDate = {};
Object.setPrototypeOf(fakeDate, Date.prototype);

// Different [[Prototype]]:
assert.deepStrictEqual(object, fakeDate);
// AssertionError: Expected inputs to be strictly deep-equal:
// + actual - expected
//
// + {}
// - Date {}

// Different type tags:
assert.deepStrictEqual(date, fakeDate);
// AssertionError: Expected inputs to be strictly deep-equal:
// + actual - expected
//
// + 2018-04-26T00:49:08.604Z
// - Date {}

assert.deepStrictEqual(NaN, NaN);
// OK because Object.is(NaN, NaN) is true.

// Different unwrapped numbers:
assert.deepStrictEqual(new Number(1), new Number(2));
// AssertionError: Expected inputs to be strictly deep-equal:
// + actual - expected
//
// + [Number: 1]
// - [Number: 2]

assert.deepStrictEqual(new String('foo'), Object('foo'));
// OK because the object and the string are identical when unwrapped.

assert.deepStrictEqual(-0, -0);
// OK
```

```
// Different zeros:
assert.deepStrictEqual(0, -0);
// AssertionError: Expected inputs to be strictly deep-equal:
// + actual - expected
//
// + 0
// - -0

const symbol1 = Symbol();
const symbol2 = Symbol();
assert.deepStrictEqual({ [symbol1]: 1 }, { [symbol1]: 1 });
// OK, because it is the same symbol on both objects.

assert.deepStrictEqual({ [symbol1]: 1 }, { [symbol2]: 1 });
// AssertionError [ERR_ASSERTION]: Inputs identical but not reference equal:
//
// {
//   [Symbol()]: 1
// }

const weakMap1 = new WeakMap();
const weakMap2 = new WeakMap([[{}, {}]]);
const weakMap3 = new WeakMap();
weakMap3.unequal = true;

assert.deepStrictEqual(weakMap1, weakMap2);
// OK, because it is impossible to compare the entries

// Fails because weakMap3 has a property that weakMap1 does not contain:
assert.deepStrictEqual(weakMap1, weakMap3);
// AssertionError: Expected inputs to be strictly deep-equal:
// + actual - expected
//
//   WeakMap {
// +   [items unknown]
// -   [items unknown],
// -   unequal: true
//   }
```

CJS modules

```
const assert = require('node:assert/strict');

// This fails because 1 !== '1'.
assert.deepStrictEqual({ a: 1 }, { a: '1' });
// AssertionError: Expected inputs to be strictly deep-equal:
// + actual - expected
//
//   {
// +   a: 1
// -   a: '1'
//   }
```

```
// The following objects don't have own properties
const date = new Date();
const object = {};
const fakeDate = {};
Object.setPrototypeOf(fakeDate, Date.prototype);

// Different [[Prototype]]:
assert.deepStrictEqual(object, fakeDate);
// AssertionError: Expected inputs to be strictly deep-equal:
// + actual - expected
//
// + {}
// - Date {}

// Different type tags:
assert.deepStrictEqual(date, fakeDate);
// AssertionError: Expected inputs to be strictly deep-equal:
// + actual - expected
//
// + 2018-04-26T00:49:08.604Z
// - Date {}

assert.deepStrictEqual(NaN, NaN);
// OK because Object.is(NaN, NaN) is true.

// Different unwrapped numbers:
assert.deepStrictEqual(new Number(1), new Number(2));
// AssertionError: Expected inputs to be strictly deep-equal:
// + actual - expected
//
// + [Number: 1]
// - [Number: 2]

assert.deepStrictEqual(new String('foo'), Object('foo'));
// OK because the object and the string are identical when unwrapped.

assert.deepStrictEqual(-0, -0);
// OK

// Different zeros:
assert.deepStrictEqual(0, -0);
// AssertionError: Expected inputs to be strictly deep-equal:
// + actual - expected
//
// + 0
// - -0

const symbol1 = Symbol();
const symbol2 = Symbol();
assert.deepStrictEqual({ [symbol1]: 1 }, { [symbol1]: 1 });
// OK, because it is the same symbol on both objects.

assert.deepStrictEqual({ [symbol1]: 1 }, { [symbol2]: 1 });
// AssertionError [ERR_ASSERTION]: Inputs identical but not reference equal:
//
```

```
// {
//   [Symbol()]: 1
// }

const weakMap1 = new WeakMap();
const weakMap2 = new WeakMap([[{}, {}]]);
const weakMap3 = new WeakMap();
weakMap3.unequal = true;

assert.deepStrictEqual(weakMap1, weakMap2);
// OK, because it is impossible to compare the entries

// Fails because weakMap3 has a property that weakMap1 does not contain:
assert.deepStrictEqual(weakMap1, weakMap3);
// AssertionError: Expected inputs to be strictly deep-equal:
// + actual - expected
//
//   WeakMap {
// +   [items unknown]
// -   [items unknown],
// -   unequal: true
//   }
```

If the values are not equal, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned. If the `message` parameter is an instance of an `Error` then it will be thrown instead of the `AssertionError`.

```
assert.doesNotMatch(string, regexp[, message])
```

▶ History

- `string` `<string>`
- `regexp` `<RegExp>`
- `message` `<string>` | `<Error>`

Expects the `string` input not to match the regular expression.

MJS modules

```
import assert from 'node:assert/strict';

assert.doesNotMatch('I will fail', /fail/);
// AssertionError [ERR_ASSERTION]: The input was expected to not match the ...

assert.doesNotMatch(123, /pass/);
// AssertionError [ERR_ASSERTION]: The "string" argument must be of type string.

assert.doesNotMatch('I will pass', /different/);
// OK
```

CJS modules

```
const assert = require('node:assert/strict');

assert.doesNotMatch('I will fail', /fail/);
// AssertionError [ERR_ASSERTION]: The input was expected to not match the ...

assert.doesNotMatch(123, /pass/);
// AssertionError [ERR_ASSERTION]: The "string" argument must be of type string.

assert.doesNotMatch('I will pass', /different/);
// OK
```

If the values do match, or if the `string` argument is of another type than `string`, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned. If the `message` parameter is an instance of an `Error` then it will be thrown instead of the `AssertionError`.

```
assert.doesNotReject(asyncFn[, error][, message])
```

Added in: v10.0.0

- `asyncFn` `<Function>` | `<Promise>`
- `error` `<RegExp>` | `<Function>`
- `message` `<string>`

Awaits the `asyncFn` promise or, if `asyncFn` is a function, immediately calls the function and awaits the returned promise to complete. It will then check that the promise is not rejected.

If `asyncFn` is a function and it throws an error synchronously, `assert.doesNotReject()` will return a rejected `Promise` with that error. If the function does not return a promise, `assert.doesNotReject()` will return a rejected `Promise` with an `ERR_INVALID_RETURN_VALUE` error. In both cases the error handler is skipped.

Using `assert.doesNotReject()` is actually not useful because there is little benefit in catching a rejection and then rejecting it again. Instead, consider adding a comment next to the specific code path that should not reject and keep error messages as expressive as possible.

If specified, `error` can be a `Class`, `RegExp`, or a validation function. See `assert.throws()` for more details.

Besides the async nature to await the completion behaves identically to `assert.doesNotThrow()`.

MJS modules

```
import assert from 'node:assert/strict';

await assert.doesNotReject(
  async () => {
    throw new TypeError('Wrong value');
  },
  SyntaxError,
```

```
);
```

CJS modules

```
const assert = require('node:assert/strict');

(async () => {
  await assert.doesNotReject(
    async () => {
      throw new TypeError('Wrong value');
    },
    SyntaxError,
  );
})();
```

MJS modules

```
import assert from 'node:assert/strict';

assert.doesNotReject(Promise.reject(new TypeError('Wrong value')))
  .then(() => {
    // ...
  });
```

CJS modules

```
const assert = require('node:assert/strict');

assert.doesNotReject(Promise.reject(new TypeError('Wrong value')))
  .then(() => {
    // ...
  });
```

```
assert.doesNotThrow(fn[, error][, message])
```

▶ History

- `fn` <Function>
- `error` <RegExp> | <Function>
- `message` <string>

Asserts that the function `fn` does not throw an error.

Using `assert.doesNotThrow()` is actually not useful because there is no benefit in catching an error and then rethrowing it. Instead, consider adding a comment next to the specific code path that should not throw and keep error messages as expressive as possible.

When `assert.doesNotThrow()` is called, it will immediately call the `fn` function.

If an error is thrown and it is the same type as that specified by the `error` parameter, then an `AssertionError` is thrown. If the error is of a different type, or if the `error` parameter is undefined, the error is propagated back to the caller.

If specified, `error` can be a `Class`, `RegExp`, or a validation function. See `assert.throws()` for more details.

The following, for instance, will throw the `TypeError` because there is no matching error type in the assertion:

MJS modules

```
import assert from 'node:assert/strict';

assert.doesNotThrow(
  () => {
    throw new TypeError('Wrong value');
  },
  SyntaxError,
);
```

CJS modules

```
const assert = require('node:assert/strict');

assert.doesNotThrow(
  () => {
    throw new TypeError('Wrong value');
  },
  SyntaxError,
);
```

However, the following will result in an `AssertionError` with the message 'Got unwanted exception...':

MJS modules

```
import assert from 'node:assert/strict';

assert.doesNotThrow(
  () => {
    throw new TypeError('Wrong value');
  },
  TypeError,
);
```

CJS modules

```
const assert = require('node:assert/strict');

assert.doesNotThrow(
  () => {
    throw new TypeError('Wrong value');
```

```
  },
  TypeError,
);
```

If an `AssertionError` is thrown and a value is provided for the `message` parameter, the value of `message` will be appended to the `AssertionError` message:

MJS modules

```
import assert from 'node:assert/strict';

assert.doesNotThrow(
  () => {
    throw new TypeError('Wrong value');
  },
  /Wrong value/,
  'Whoops',
);
// Throws: AssertionError: Got unwanted exception: Whoops
```

CJS modules

```
const assert = require('node:assert/strict');

assert.doesNotThrow(
  () => {
    throw new TypeError('Wrong value');
  },
  /Wrong value/,
  'Whoops',
);
// Throws: AssertionError: Got unwanted exception: Whoops
```

```
assert.equal(actual, expected[, message])
```

▶ History

- `actual` `<any>`
- `expected` `<any>`
- `message` `<string>` | `<Error>`

Strict assertion mode

An alias of `assert.strictEqual()`.

Legacy assertion mode

> Stability: 3 - Legacy: Use `assert.strictEqual()` instead.

Tests shallow, coercive equality between the `actual` and `expected` parameters using the == operator. `NaN` is specially handled and treated as being identical if both sides are `NaN`.

MJS modules

```
import assert from 'node:assert';

assert.equal(1, 1);
// OK, 1 == 1
assert.equal(1, '1');
// OK, 1 == '1'
assert.equal(NaN, NaN);
// OK

assert.equal(1, 2);
// AssertionError: 1 == 2
assert.equal({ a: { b: 1 } }, { a: { b: 1 } });
// AssertionError: { a: { b: 1 } } == { a: { b: 1 } }
```

CJS modules

```
const assert = require('node:assert');

assert.equal(1, 1);
// OK, 1 == 1
assert.equal(1, '1');
// OK, 1 == '1'
assert.equal(NaN, NaN);
// OK

assert.equal(1, 2);
// AssertionError: 1 == 2
assert.equal({ a: { b: 1 } }, { a: { b: 1 } });
// AssertionError: { a: { b: 1 } } == { a: { b: 1 } }
```

If the values are not equal, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned. If the `message` parameter is an instance of an `Error` then it will be thrown instead of the `AssertionError`.

```
assert.fail([message])
```

- `message` <string> | <Error> **Default:** `'Failed'`

Added in: v0.1.21

Throws an `AssertionError` with the provided error message or a default error message. If the `message` parameter is an instance of an `Error` then it will be thrown instead of the `AssertionError`.

MJS modules

```
import assert from 'node:assert/strict';

assert.fail();
// AssertionError [ERR_ASSERTION]: Failed

assert.fail('boom');
// AssertionError [ERR_ASSERTION]: boom

assert.fail(new TypeError('need array'));
// TypeError: need array
```

CJS modules

```
const assert = require('node:assert/strict');

assert.fail();
// AssertionError [ERR_ASSERTION]: Failed

assert.fail('boom');
// AssertionError [ERR_ASSERTION]: boom

assert.fail(new TypeError('need array'));
// TypeError: need array
```

Using `assert.fail()` with more than two arguments is possible but deprecated. See below for further details.

```
assert.fail(actual, expected[, message[, operator[, stackStartFn]]])
```

▶ History

Stability: 0 - Deprecated: Use `assert.fail([message])` or other assert functions instead.

- `actual` <any>
- `expected` <any>
- `message` <string> | <Error>
- `operator` <string> **Default:** `'!='`
- `stackStartFn` <Function> **Default:** `assert.fail`

If `message` is falsy, the error message is set as the values of `actual` and `expected` separated by the provided `operator`. If just the two `actual` and `expected` arguments are provided, `operator` will default to `'!='`. If `message` is provided as third argument it will be used as the error message and the other arguments will be stored as properties on the thrown object. If `stackStartFn` is provided, all stack frames above that function will be removed from stacktrace (see `Error.captureStackTrace`). If no arguments are given, the default message `Failed` will be used.

MJS modules

```mjs
import assert from 'node:assert/strict';

assert.fail('a', 'b');
// AssertionError [ERR_ASSERTION]: 'a' != 'b'

assert.fail(1, 2, undefined, '>');
// AssertionError [ERR_ASSERTION]: 1 > 2

assert.fail(1, 2, 'fail');
// AssertionError [ERR_ASSERTION]: fail

assert.fail(1, 2, 'whoops', '>');
// AssertionError [ERR_ASSERTION]: whoops

assert.fail(1, 2, new TypeError('need array'));
// TypeError: need array
```

CJS modules

```cjs
const assert = require('node:assert/strict');

assert.fail('a', 'b');
// AssertionError [ERR_ASSERTION]: 'a' != 'b'

assert.fail(1, 2, undefined, '>');
// AssertionError [ERR_ASSERTION]: 1 > 2

assert.fail(1, 2, 'fail');
// AssertionError [ERR_ASSERTION]: fail

assert.fail(1, 2, 'whoops', '>');
// AssertionError [ERR_ASSERTION]: whoops

assert.fail(1, 2, new TypeError('need array'));
// TypeError: need array
```

In the last three cases `actual`, `expected`, and `operator` have no influence on the error message.

Example use of `stackStartFn` for truncating the exception's stacktrace:

MJS modules

```mjs
import assert from 'node:assert/strict';

function suppressFrame() {
  assert.fail('a', 'b', undefined, '!==', suppressFrame);
}
```

```
suppressFrame();
// AssertionError [ERR_ASSERTION]: 'a' !== 'b'
//     at repl:1:1
//     at ContextifyScript.Script.runInThisContext (vm.js:44:33)
//     ...
```

CJS modules

```
const assert = require('node:assert/strict');

function suppressFrame() {
  assert.fail('a', 'b', undefined, '!==', suppressFrame);
}
suppressFrame();
// AssertionError [ERR_ASSERTION]: 'a' !== 'b'
//     at repl:1:1
//     at ContextifyScript.Script.runInThisContext (vm.js:44:33)
//     ...
```

```
assert.ifError(value)
```

▶ History

- `value` <any>

Throws `value` if `value` is not `undefined` or `null`. This is useful when testing the `error` argument in callbacks. The stack trace contains all frames from the error passed to `ifError()` including the potential new frames for `ifError()` itself.

MJS modules

```
import assert from 'node:assert/strict';

assert.ifError(null);
// OK
assert.ifError(0);
// AssertionError [ERR_ASSERTION]: ifError got unwanted exception: 0
assert.ifError('error');
// AssertionError [ERR_ASSERTION]: ifError got unwanted exception: 'error'
assert.ifError(new Error());
// AssertionError [ERR_ASSERTION]: ifError got unwanted exception: Error

// Create some random error frames.
let err;
(function errorFrame() {
  err = new Error('test error');
})();

(function ifErrorFrame() {
  assert.ifError(err);
})();
// AssertionError [ERR_ASSERTION]: ifError got unwanted exception: test error
//     at ifErrorFrame
//     at errorFrame
```

CJS modules

```
const assert = require('node:assert/strict');

assert.ifError(null);
// OK
assert.ifError(0);
// AssertionError [ERR_ASSERTION]: ifError got unwanted exception: 0
assert.ifError('error');
// AssertionError [ERR_ASSERTION]: ifError got unwanted exception: 'error'
assert.ifError(new Error());
// AssertionError [ERR_ASSERTION]: ifError got unwanted exception: Error

// Create some random error frames.
let err;
(function errorFrame() {
  err = new Error('test error');
})();

(function ifErrorFrame() {
  assert.ifError(err);
})();
// AssertionError [ERR_ASSERTION]: ifError got unwanted exception: test error
//     at ifErrorFrame
//     at errorFrame
```

```
assert.match(string, regexp[, message])
```

▶ History

- `string` <string>
- `regexp` <RegExp>
- `message` <string> | <Error>

Expects the `string` input to match the regular expression.

MJS modules

```
import assert from 'node:assert/strict';

assert.match('I will fail', /pass/);
// AssertionError [ERR_ASSERTION]: The input did not match the regular ...

assert.match(123, /pass/);
// AssertionError [ERR_ASSERTION]: The "string" argument must be of type string.

assert.match('I will pass', /pass/);
// OK
```

CJS modules

```
const assert = require('node:assert/strict');

assert.match('I will fail', /pass/);
// AssertionError [ERR_ASSERTION]: The input did not match the regular ...

assert.match(123, /pass/);
// AssertionError [ERR_ASSERTION]: The "string" argument must be of type string.

assert.match('I will pass', /pass/);
// OK
```

If the values do not match, or if the `string` argument is of another type than `string`, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned. If the `message` parameter is an instance of an `Error` then it will be thrown instead of the `AssertionError`.

```
assert.notDeepEqual(actual, expected[, message])
```

▶ History

- `actual` <any>
- `expected` <any>
- `message` <string> | <Error>

Strict assertion mode

An alias of `assert.notDeepStrictEqual()`.

Legacy assertion mode

Stability: 3 - Legacy: Use `assert.notDeepStrictEqual()` instead.

Tests for any deep inequality. Opposite of `assert.deepEqual()`.

MJS modules

```
import assert from 'node:assert';

const obj1 = {
  a: {
    b: 1,
  },
};
const obj2 = {
  a: {
    b: 2,
  },
};
```

```
const obj3 = {
  a: {
    b: 1,
  },
};
const obj4 = { __proto__: obj1 };

assert.notDeepEqual(obj1, obj1);
// AssertionError: { a: { b: 1 } } notDeepEqual { a: { b: 1 } }

assert.notDeepEqual(obj1, obj2);
// OK

assert.notDeepEqual(obj1, obj3);
// AssertionError: { a: { b: 1 } } notDeepEqual { a: { b: 1 } }

assert.notDeepEqual(obj1, obj4);
// OK
```

CJS modules

```
const assert = require('node:assert');

const obj1 = {
  a: {
    b: 1,
  },
};
const obj2 = {
  a: {
    b: 2,
  },
};
const obj3 = {
  a: {
    b: 1,
  },
};
const obj4 = { __proto__: obj1 };

assert.notDeepEqual(obj1, obj1);
// AssertionError: { a: { b: 1 } } notDeepEqual { a: { b: 1 } }

assert.notDeepEqual(obj1, obj2);
// OK

assert.notDeepEqual(obj1, obj3);
// AssertionError: { a: { b: 1 } } notDeepEqual { a: { b: 1 } }

assert.notDeepEqual(obj1, obj4);
// OK
```

If the values are deeply equal, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned. If the `message` parameter is an instance of an `Error` then it will be thrown instead of the `AssertionError`.

```
assert.notDeepStrictEqual(actual, expected[, message])
```

▶ History

- `actual` <any>
- `expected` <any>
- `message` <string> | <Error>

Tests for deep strict inequality. Opposite of `assert.deepStrictEqual()`.

MJS modules

```
import assert from 'node:assert/strict';

assert.notDeepStrictEqual({ a: 1 }, { a: '1' });
// OK
```

CJS modules

```
const assert = require('node:assert/strict');

assert.notDeepStrictEqual({ a: 1 }, { a: '1' });
// OK
```

If the values are deeply and strictly equal, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned. If the `message` parameter is an instance of an `Error` then it will be thrown instead of the `AssertionError`.

```
assert.notEqual(actual, expected[, message])
```

▶ History

- `actual` <any>
- `expected` <any>
- `message` <string> | <Error>

Strict assertion mode

An alias of `assert.notStrictEqual()`.

Legacy assertion mode

Stability: 3 - Legacy: Use `assert.notStrictEqual()` instead.

Tests shallow, coercive inequality with the `!=` operator. `NaN` is specially handled and treated as being identical if both sides are `NaN`.

MJS modules

```
import assert from 'node:assert';

assert.notEqual(1, 2);
// OK

assert.notEqual(1, 1);
// AssertionError: 1 != 1

assert.notEqual(1, '1');
// AssertionError: 1 != '1'
```

CJS modules

```
const assert = require('node:assert');

assert.notEqual(1, 2);
// OK

assert.notEqual(1, 1);
// AssertionError: 1 != 1

assert.notEqual(1, '1');
// AssertionError: 1 != '1'
```

If the values are equal, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned. If the `message` parameter is an instance of an `Error` then it will be thrown instead of the `AssertionError`.

```
assert.notStrictEqual(actual, expected[, message])
```

▶ History

- `actual` <any>
- `expected` <any>
- `message` <string> | <Error>

Tests strict inequality between the `actual` and `expected` parameters as determined by `Object.is()`.

MJS modules

```
import assert from 'node:assert/strict';

assert.notStrictEqual(1, 2);
// OK

assert.notStrictEqual(1, 1);
```

```
// AssertionError [ERR_ASSERTION]: Expected "actual" to be strictly unequal to:
//
// 1

assert.notStrictEqual(1, '1');
// OK
```

CJS modules

```
const assert = require('node:assert/strict');

assert.notStrictEqual(1, 2);
// OK

assert.notStrictEqual(1, 1);
// AssertionError [ERR_ASSERTION]: Expected "actual" to be strictly unequal to:
//
// 1

assert.notStrictEqual(1, '1');
// OK
```

If the values are strictly equal, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned. If the `message` parameter is an instance of an `Error` then it will be thrown instead of the `AssertionError`.

```
assert.ok(value[, message])
```

▶ History

- `value` `<any>`
- `message` `<string>` | `<Error>`

Tests if `value` is truthy. It is equivalent to `assert.equal(!!value, true, message)`.

If `value` is not truthy, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is `undefined`, a default error message is assigned. If the `message` parameter is an instance of an `Error` then it will be thrown instead of the `AssertionError`. If no arguments are passed in at all `message` will be set to the string: `'No value argument passed to `assert.ok()`'`.

Be aware that in the `repl` the error message will be different to the one thrown in a file! See below for further details.

MJS modules

```
import assert from 'node:assert/strict';

assert.ok(true);
// OK
assert.ok(1);
// OK
```

```
assert.ok();
// AssertionError: No value argument passed to `assert.ok()`

assert.ok(false, 'it\'s false');
// AssertionError: it's false

// In the repl:
assert.ok(typeof 123 === 'string');
// AssertionError: false == true

// In a file (e.g. test.js):
assert.ok(typeof 123 === 'string');
// AssertionError: The expression evaluated to a falsy value:
//
//   assert.ok(typeof 123 === 'string')

assert.ok(false);
// AssertionError: The expression evaluated to a falsy value:
//
//   assert.ok(false)

assert.ok(0);
// AssertionError: The expression evaluated to a falsy value:
//
//   assert.ok(0)
```

CJS modules

```
const assert = require('node:assert/strict');

assert.ok(true);
// OK
assert.ok(1);
// OK

assert.ok();
// AssertionError: No value argument passed to `assert.ok()`

assert.ok(false, 'it\'s false');
// AssertionError: it's false

// In the repl:
assert.ok(typeof 123 === 'string');
// AssertionError: false == true

// In a file (e.g. test.js):
assert.ok(typeof 123 === 'string');
// AssertionError: The expression evaluated to a falsy value:
//
//   assert.ok(typeof 123 === 'string')

assert.ok(false);
// AssertionError: The expression evaluated to a falsy value:
//
```

```
//   assert.ok(false)

assert.ok(0);
// AssertionError: The expression evaluated to a falsy value:
//
//   assert.ok(0)
```

MJS modules

```
import assert from 'node:assert/strict';

// Using `assert()` works the same:
assert(0);
// AssertionError: The expression evaluated to a falsy value:
//
//   assert(0)
```

CJS modules

```
const assert = require('node:assert');

// Using `assert()` works the same:
assert(0);
// AssertionError: The expression evaluated to a falsy value:
//
//   assert(0)
```

```
assert.rejects(asyncFn[, error][, message])
```

Added in: v10.0.0

- `asyncFn` <Function> | <Promise>
- `error` <RegExp> | <Function> | <Object> | <Error>
- `message` <string>

Awaits the `asyncFn` promise or, if `asyncFn` is a function, immediately calls the function and awaits the returned promise to complete. It will then check that the promise is rejected.

If `asyncFn` is a function and it throws an error synchronously, `assert.rejects()` will return a rejected `Promise` with that error. If the function does not return a promise, `assert.rejects()` will return a rejected `Promise` with an `ERR_INVALID_RETURN_VALUE` error. In both cases the error handler is skipped.

Besides the async nature to await the completion behaves identically to `assert.throws()`.

If specified, `error` can be a `Class`, `RegExp`, a validation function, an object where each property will be tested for, or an instance of error where each property will be tested for including the non-enumerable `message` and `name` properties.

If specified, `message` will be the message provided by the `AssertionError` if the `asyncFn` fails to reject.

MJS modules

```
import assert from 'node:assert/strict';

await assert.rejects(
  async () => {
    throw new TypeError('Wrong value');
  },
  {
    name: 'TypeError',
    message: 'Wrong value',
  },
);
```

CJS modules

```
const assert = require('node:assert/strict');

(async () => {
  await assert.rejects(
    async () => {
      throw new TypeError('Wrong value');
    },
    {
      name: 'TypeError',
      message: 'Wrong value',
    },
  );
})();
```

MJS modules

```
import assert from 'node:assert/strict';

await assert.rejects(
  async () => {
    throw new TypeError('Wrong value');
  },
  (err) => {
    assert.strictEqual(err.name, 'TypeError');
    assert.strictEqual(err.message, 'Wrong value');
    return true;
  },
);
```

CJS modules

```
const assert = require('node:assert/strict');

(async () => {
  await assert.rejects(
    async () => {
      throw new TypeError('Wrong value');
    },
    (err) => {
      assert.strictEqual(err.name, 'TypeError');
      assert.strictEqual(err.message, 'Wrong value');
      return true;
    },
  );
})();
```

MJS modules

```
import assert from 'node:assert/strict';

assert.rejects(
  Promise.reject(new Error('Wrong value')),
  Error,
).then(() => {
  // ...
});
```

CJS modules

```
const assert = require('node:assert/strict');

assert.rejects(
  Promise.reject(new Error('Wrong value')),
  Error,
).then(() => {
  // ...
});
```

`error` cannot be a string. If a string is provided as the second argument, then `error` is assumed to be omitted and the string will be used for `message` instead. This can lead to easy-to-miss mistakes. Please read the example in `assert.throws()` carefully if using a string as the second argument gets considered.

```
assert.strictEqual(actual, expected[, message])
```

▶ History

- `actual` <any>
- `expected` <any>
- `message` <string> | <Error>

Tests strict equality between the `actual` and `expected` parameters as determined by `Object.is()`.

MJS modules

```
import assert from 'node:assert/strict';

assert.strictEqual(1, 2);
// AssertionError [ERR_ASSERTION]: Expected inputs to be strictly equal:
//
// 1 !== 2

assert.strictEqual(1, 1);
// OK

assert.strictEqual('Hello foobar', 'Hello World!');
// AssertionError [ERR_ASSERTION]: Expected inputs to be strictly equal:
// + actual - expected
//
// + 'Hello foobar'
// - 'Hello World!'
//          ^

const apples = 1;
const oranges = 2;
assert.strictEqual(apples, oranges, `apples ${apples} !== oranges ${oranges}`);
// AssertionError [ERR_ASSERTION]: apples 1 !== oranges 2

assert.strictEqual(1, '1', new TypeError('Inputs are not identical'));
// TypeError: Inputs are not identical
```

CJS modules

```
const assert = require('node:assert/strict');

assert.strictEqual(1, 2);
// AssertionError [ERR_ASSERTION]: Expected inputs to be strictly equal:
//
// 1 !== 2

assert.strictEqual(1, 1);
// OK

assert.strictEqual('Hello foobar', 'Hello World!');
// AssertionError [ERR_ASSERTION]: Expected inputs to be strictly equal:
// + actual - expected
//
// + 'Hello foobar'
// - 'Hello World!'
//          ^

const apples = 1;
const oranges = 2;
```

```
assert.strictEqual(apples, oranges, `apples ${apples} !== oranges ${oranges}`);
// AssertionError [ERR_ASSERTION]: apples 1 !== oranges 2

assert.strictEqual(1, '1', new TypeError('Inputs are not identical'));
// TypeError: Inputs are not identical
```

If the values are not strictly equal, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned. If the `message` parameter is an instance of an `Error` then it will be thrown instead of the `AssertionError`.

```
assert.throws(fn[, error][, message])
```

▶ History

- `fn` `<Function>`
- `error` `<RegExp>` | `<Function>` | `<Object>` | `<Error>`
- `message` `<string>`

Expects the function `fn` to throw an error.

If specified, `error` can be a `Class`, `RegExp`, a validation function, a validation object where each property will be tested for strict deep equality, or an instance of error where each property will be tested for strict deep equality including the non-enumerable `message` and `name` properties. When using an object, it is also possible to use a regular expression, when validating against a string property. See below for examples.

If specified, `message` will be appended to the message provided by the `AssertionError` if the `fn` call fails to throw or in case the error validation fails.

Custom validation object/error instance:

MJS modules

```
import assert from 'node:assert/strict';

const err = new TypeError('Wrong value');
err.code = 404;
err.foo = 'bar';
err.info = {
  nested: true,
  baz: 'text',
};
err.reg = /abc/i;

assert.throws(
  () => {
    throw err;
  },
  {
    name: 'TypeError',
    message: 'Wrong value',
    info: {
      nested: true,
```

```
      baz: 'text',
    },
    // Only properties on the validation object will be tested for.
    // Using nested objects requires all properties to be present. Otherwise
    // the validation is going to fail.
  },
);

// Using regular expressions to validate error properties:
assert.throws(
  () => {
    throw err;
  },
  {
    // The `name` and `message` properties are strings and using regular
    // expressions on those will match against the string. If they fail, an
    // error is thrown.
    name: /^TypeError$/,
    message: /Wrong/,
    foo: 'bar',
    info: {
      nested: true,
      // It is not possible to use regular expressions for nested properties!
      baz: 'text',
    },
    // The `reg` property contains a regular expression and only if the
    // validation object contains an identical regular expression, it is going
    // to pass.
    reg: /abc/i,
  },
);

// Fails due to the different `message` and `name` properties:
assert.throws(
  () => {
    const otherErr = new Error('Not found');
    // Copy all enumerable properties from `err` to `otherErr`.
    for (const [key, value] of Object.entries(err)) {
      otherErr[key] = value;
    }
    throw otherErr;
  },
  // The error's `message` and `name` properties will also be checked when using
  // an error as validation object.
  err,
);
```

CJS modules

```
const assert = require('node:assert/strict');

const err = new TypeError('Wrong value');
err.code = 404;
err.foo = 'bar';
```

```
err.info = {
  nested: true,
  baz: 'text',
};
err.reg = /abc/i;

assert.throws(
  () => {
    throw err;
  },
  {
    name: 'TypeError',
    message: 'Wrong value',
    info: {
      nested: true,
      baz: 'text',
    },
    // Only properties on the validation object will be tested for.
    // Using nested objects requires all properties to be present. Otherwise
    // the validation is going to fail.
  },
);

// Using regular expressions to validate error properties:
assert.throws(
  () => {
    throw err;
  },
  {
    // The `name` and `message` properties are strings and using regular
    // expressions on those will match against the string. If they fail, an
    // error is thrown.
    name: /^TypeError$/,
    message: /Wrong/,
    foo: 'bar',
    info: {
      nested: true,
      // It is not possible to use regular expressions for nested properties!
      baz: 'text',
    },
    // The `reg` property contains a regular expression and only if the
    // validation object contains an identical regular expression, it is going
    // to pass.
    reg: /abc/i,
  },
);

// Fails due to the different `message` and `name` properties:
assert.throws(
  () => {
    const otherErr = new Error('Not found');
    // Copy all enumerable properties from `err` to `otherErr`.
    for (const [key, value] of Object.entries(err)) {
      otherErr[key] = value;
    }
```

```
    throw otherErr;
  },
  // The error's `message` and `name` properties will also be checked when using
  // an error as validation object.
  err,
);
```

Validate instanceof using constructor:

MJS modules

```
import assert from 'node:assert/strict';

assert.throws(
  () => {
    throw new Error('Wrong value');
  },
  Error,
);
```

CJS modules

```
const assert = require('node:assert/strict');

assert.throws(
  () => {
    throw new Error('Wrong value');
  },
  Error,
);
```

Validate error message using `RegExp`:

Using a regular expression runs `.toString` on the error object, and will therefore also include the error name.

MJS modules

```
import assert from 'node:assert/strict';

assert.throws(
  () => {
    throw new Error('Wrong value');
  },
  /^Error: Wrong value$/,
);
```

CJS modules

```
const assert = require('node:assert/strict');

assert.throws(
  () => {
    throw new Error('Wrong value');
  },
  /^Error: Wrong value$/,
);
```

Custom error validation:

The function must return `true` to indicate all internal validations passed. It will otherwise fail with an `AssertionError`.

MJS modules

```
import assert from 'node:assert/strict';

assert.throws(
  () => {
    throw new Error('Wrong value');
  },
  (err) => {
    assert(err instanceof Error);
    assert(/value/.test(err));
    // Avoid returning anything from validation functions besides `true`.
    // Otherwise, it's not clear what part of the validation failed. Instead,
    // throw an error about the specific validation that failed (as done in this
    // example) and add as much helpful debugging information to that error as
    // possible.
    return true;
  },
  'unexpected error',
);
```

CJS modules

```
const assert = require('node:assert/strict');

assert.throws(
  () => {
    throw new Error('Wrong value');
  },
  (err) => {
    assert(err instanceof Error);
    assert(/value/.test(err));
    // Avoid returning anything from validation functions besides `true`.
    // Otherwise, it's not clear what part of the validation failed. Instead,
    // throw an error about the specific validation that failed (as done in this
    // example) and add as much helpful debugging information to that error as
```

```
    // possible.
    return true;
  },
  'unexpected error',
);
```

`error` cannot be a string. If a string is provided as the second argument, then `error` is assumed to be omitted and the string will be used for `message` instead. This can lead to easy-to-miss mistakes. Using the same message as the thrown error message is going to result in an `ERR_AMBIGUOUS_ARGUMENT` error. Please read the example below carefully if using a string as the second argument gets considered:

MJS modules

```
import assert from 'node:assert/strict';

function throwingFirst() {
  throw new Error('First');
}

function throwingSecond() {
  throw new Error('Second');
}

function notThrowing() {}

// The second argument is a string and the input function threw an Error.
// The first case will not throw as it does not match for the error message
// thrown by the input function!
assert.throws(throwingFirst, 'Second');
// In the next example the message has no benefit over the message from the
// error and since it is not clear if the user intended to actually match
// against the error message, Node.js throws an `ERR_AMBIGUOUS_ARGUMENT` error.
assert.throws(throwingSecond, 'Second');
// TypeError [ERR_AMBIGUOUS_ARGUMENT]

// The string is only used (as message) in case the function does not throw:
assert.throws(notThrowing, 'Second');
// AssertionError [ERR_ASSERTION]: Missing expected exception: Second

// If it was intended to match for the error message do this instead:
// It does not throw because the error messages match.
assert.throws(throwingSecond, /Second$/);

// If the error message does not match, an AssertionError is thrown.
assert.throws(throwingFirst, /Second$/);
// AssertionError [ERR_ASSERTION]
```

CJS modules

```
const assert = require('node:assert/strict');

function throwingFirst() {
  throw new Error('First');
}

function throwingSecond() {
  throw new Error('Second');
}

function notThrowing() {}

// The second argument is a string and the input function threw an Error.
// The first case will not throw as it does not match for the error message
// thrown by the input function!
assert.throws(throwingFirst, 'Second');
// In the next example the message has no benefit over the message from the
// error and since it is not clear if the user intended to actually match
// against the error message, Node.js throws an `ERR_AMBIGUOUS_ARGUMENT` error.
assert.throws(throwingSecond, 'Second');
// TypeError [ERR_AMBIGUOUS_ARGUMENT]

// The string is only used (as message) in case the function does not throw:
assert.throws(notThrowing, 'Second');
// AssertionError [ERR_ASSERTION]: Missing expected exception: Second

// If it was intended to match for the error message do this instead:
// It does not throw because the error messages match.
assert.throws(throwingSecond, /Second$/);

// If the error message does not match, an AssertionError is thrown.
assert.throws(throwingFirst, /Second$/);
// AssertionError [ERR_ASSERTION]
```

Due to the confusing error-prone notation, avoid a string as the second argument.