/\*\*

\* The `assert` module provides a set of assertion functions for verifying

\* invariants.

\* @see [source](https://github.com/nodejs/node/blob/v17.0.0/lib/assert.js)

\*/

declare module 'assert' {

/\*\*

\* An alias of {@link ok}.

\* @since v0.5.9

\* @param value The input that is checked for being truthy.

\*/

function assert(value: unknown, message?: string | Error): asserts value;

namespace assert {

/\*\*

\* Indicates the failure of an assertion. All errors thrown by the `assert` module

\* will be instances of the `AssertionError` class.

\*/

class AssertionError extends Error {

actual: unknown;

expected: unknown;

operator: string;

generatedMessage: boolean;

code: 'ERR\_ASSERTION';

constructor(options?: {

/\*\* If provided, the error message is set to this value. \*/

message?: string | undefined;

/\*\* The `actual` property on the error instance. \*/

actual?: unknown | undefined;

/\*\* The `expected` property on the error instance. \*/

expected?: unknown | undefined;

/\*\* The `operator` property on the error instance. \*/

operator?: string | undefined;

/\*\* If provided, the generated stack trace omits frames before this function. \*/

// tslint:disable-next-line:ban-types

stackStartFn?: Function | undefined;

});

}

/\*\*

\* This feature is currently experimental and behavior might still change.

\* @since v14.2.0, v12.19.0

\* @experimental

\*/

class CallTracker {

/\*\*

\* 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.

\*

\* ```js

\* import assert from '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);

\* ```

\* @since v14.2.0, v12.19.0

\* @param [fn='A no-op function']

\* @param [exact=1]

\* @return that wraps `fn`.

\*/

calls(exact?: number): () => void;

calls<Func extends (...args: any[]) => any>(fn?: Func, exact?: number): Func;

/\*\*

\* 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.

\*

\* ```js

\* import assert from 'assert';

\*

\* // Creates call tracker.

\* const tracker = new assert.CallTracker();

\*

\* function func() {}

\*

\* function foo() {}

\*

\* // 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()

\* 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

\* // }

\* // ]

\* ```

\* @since v14.2.0, v12.19.0

\* @return of objects containing information about the wrapper functions returned by `calls`.

\*/

report(): CallTrackerReportInformation[];

/\*\*

\* 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.

\*

\* ```js

\* import assert from '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();

\* ```

\* @since v14.2.0, v12.19.0

\*/

verify(): void;

}

interface CallTrackerReportInformation {

message: string;

/\*\* The actual number of times the function was called. \*/

actual: number;

/\*\* The number of times the function was expected to be called. \*/

expected: number;

/\*\* The name of the function that is wrapped. \*/

operator: string;

/\*\* A stack trace of the function. \*/

stack: object;

}

type AssertPredicate = RegExp | (new () => object) | ((thrown: unknown) => boolean) | object | Error;

/\*\*

\* 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`.

\*

\* ```js

\* import assert from '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.

\* @since v0.1.21

\* @param [message='Failed']

\*/

function fail(message?: string | Error): never;

/\*\* @deprecated since v10.0.0 - use fail([message]) or other assert functions instead. \*/

function fail(

actual: unknown,

expected: unknown,

message?: string | Error,

operator?: string,

// tslint:disable-next-line:ban-types

stackStartFn?: Function

): never;

/\*\*

\* 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.

\*

\* ```js

\* import assert from '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)

\* ```

\*

\* ```js

\* import assert from 'assert/strict';

\*

\* // Using `assert()` works the same:

\* assert(0);

\* // AssertionError: The expression evaluated to a falsy value:

\* //

\* // assert(0)

\* ```

\* @since v0.1.21

\*/

function ok(value: unknown, message?: string | Error): asserts value;

/\*\*

\* \*\*Strict assertion mode\*\*

\*

\* An alias of {@link strictEqual}.

\*

\* \*\*Legacy assertion mode\*\*

\*

\* > Stability: 3 - Legacy: Use {@link strictEqual} instead.

\*

\* Tests shallow, coercive equality between the `actual` and `expected` parameters

\* using the [Abstract Equality Comparison](https://tc39.github.io/ecma262/#sec-abstract-equality-comparison) ( `==` ). `NaN` is special handled

\* and treated as being identical in case both sides are `NaN`.

\*

\* ```js

\* import assert from '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`.

\* @since v0.1.21

\*/

function equal(actual: unknown, expected: unknown, message?: string | Error): void;

/\*\*

\* \*\*Strict assertion mode\*\*

\*

\* An alias of {@link notStrictEqual}.

\*

\* \*\*Legacy assertion mode\*\*

\*

\* > Stability: 3 - Legacy: Use {@link notStrictEqual} instead.

\*

\* Tests shallow, coercive inequality with the [Abstract Equality Comparison](https://tc39.github.io/ecma262/#sec-abstract-equality-comparison)(`!=` ). `NaN` is special handled and treated as

\* being identical in case both

\* sides are `NaN`.

\*

\* ```js

\* import assert from '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`.

\* @since v0.1.21

\*/

function notEqual(actual: unknown, expected: unknown, message?: string | Error): void;

/\*\*

\* \*\*Strict assertion mode\*\*

\*

\* An alias of {@link deepStrictEqual}.

\*

\* \*\*Legacy assertion mode\*\*

\*

\* > Stability: 3 - Legacy: Use {@link deepStrictEqual} instead.

\*

\* Tests for deep equality between the `actual` and `expected` parameters. Consider

\* using {@link deepStrictEqual} instead. {@link deepEqual} can have

\* surprising results.

\*

\* \_Deep equality\_ means that the enumerable "own" properties of child objects

\* are also recursively evaluated by the following rules.

\* @since v0.1.21

\*/

function deepEqual(actual: unknown, expected: unknown, message?: string | Error): void;

/\*\*

\* \*\*Strict assertion mode\*\*

\*

\* An alias of {@link notDeepStrictEqual}.

\*

\* \*\*Legacy assertion mode\*\*

\*

\* > Stability: 3 - Legacy: Use {@link notDeepStrictEqual} instead.

\*

\* Tests for any deep inequality. Opposite of {@link deepEqual}.

\*

\* ```js

\* import assert from 'assert';

\*

\* const obj1 = {

\* a: {

\* b: 1

\* }

\* };

\* const obj2 = {

\* a: {

\* b: 2

\* }

\* };

\* const obj3 = {

\* a: {

\* b: 1

\* }

\* };

\* const obj4 = Object.create(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`.

\* @since v0.1.21

\*/

function notDeepEqual(actual: unknown, expected: unknown, message?: string | Error): void;

/\*\*

\* Tests strict equality between the `actual` and `expected` parameters as

\* determined by the [SameValue Comparison](https://tc39.github.io/ecma262/#sec-samevalue).

\*

\* ```js

\* import assert from '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`.

\* @since v0.1.21

\*/

function strictEqual<T>(actual: unknown, expected: T, message?: string | Error): asserts actual is T;

/\*\*

\* Tests strict inequality between the `actual` and `expected` parameters as

\* determined by the [SameValue Comparison](https://tc39.github.io/ecma262/#sec-samevalue).

\*

\* ```js

\* import assert from '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`.

\* @since v0.1.21

\*/

function notStrictEqual(actual: unknown, expected: unknown, message?: string | Error): void;

/\*\*

\* 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.

\* @since v1.2.0

\*/

function deepStrictEqual<T>(actual: unknown, expected: T, message?: string | Error): asserts actual is T;

/\*\*

\* Tests for deep strict inequality. Opposite of {@link deepStrictEqual}.

\*

\* ```js

\* import assert from '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`.

\* @since v1.2.0

\*/

function notDeepStrictEqual(actual: unknown, expected: unknown, message?: string | Error): void;

/\*\*

\* Expects the function `fn` to throw an error.

\*

\* If specified, `error` can be a [`Class`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes),

\* [`RegExp`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular\_Expressions), 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:

\*

\* ```js

\* import assert from '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:

\* 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:

\* 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:

\*

\* ```js

\* import assert from 'assert/strict';

\*

\* assert.throws(

\* () => {

\* throw new Error('Wrong value');

\* },

\* Error

\* );

\* ```

\*

\* Validate error message using [`RegExp`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular\_Expressions):

\*

\* Using a regular expression runs `.toString` on the error object, and will

\* therefore also include the error name.

\*

\* ```js

\* import assert from '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`.

\*

\* ```js

\* import assert from '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:

\*

\* ```js

\* import assert from '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.

\* @since v0.1.21

\*/

function throws(block: () => unknown, message?: string | Error): void;

function throws(block: () => unknown, error: AssertPredicate, message?: string | Error): void;

/\*\*

\* 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`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes),

\* [`RegExp`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular\_Expressions) or a validation

\* function. See {@link throws} for more details.

\*

\* The following, for instance, will throw the `TypeError` because there is no

\* matching error type in the assertion:

\*

\* ```js

\* import assert from 'assert/strict';

\*

\* assert.doesNotThrow(

\* () => {

\* throw new TypeError('Wrong value');

\* },

\* SyntaxError

\* );

\* ```

\*

\* However, the following will result in an `AssertionError` with the message

\* 'Got unwanted exception...':

\*

\* ```js

\* import assert from '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:

\*

\* ```js

\* import assert from 'assert/strict';

\*

\* assert.doesNotThrow(

\* () => {

\* throw new TypeError('Wrong value');

\* },

\* /Wrong value/,

\* 'Whoops'

\* );

\* // Throws: AssertionError: Got unwanted exception: Whoops

\* ```

\* @since v0.1.21

\*/

function doesNotThrow(block: () => unknown, message?: string | Error): void;

function doesNotThrow(block: () => unknown, error: AssertPredicate, message?: string | Error): void;

/\*\*

\* 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.

\*

\* ```js

\* import assert from '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

\* ```

\* @since v0.1.97

\*/

function ifError(value: unknown): asserts value is null | undefined;

/\*\*

\* 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 {@link throws}.

\*

\* If specified, `error` can be a [`Class`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes),

\* [`RegExp`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular\_Expressions), 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.

\*

\* ```js

\* import assert from 'assert/strict';

\*

\* await assert.rejects(

\* async () => {

\* throw new TypeError('Wrong value');

\* },

\* {

\* name: 'TypeError',

\* message: 'Wrong value'

\* }

\* );

\* ```

\*

\* ```js

\* import assert from '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;

\* }

\* );

\* ```

\*

\* ```js

\* import assert from '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 {@link throws} carefully if using a string as the second

\* argument gets considered.

\* @since v10.0.0

\*/

function rejects(block: (() => Promise<unknown>) | Promise<unknown>, message?: string | Error): Promise<void>;

function rejects(block: (() => Promise<unknown>) | Promise<unknown>, error: AssertPredicate, message?: string | Error): Promise<void>;

/\*\*

\* 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`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes),

\* [`RegExp`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular\_Expressions) or a validation

\* function. See {@link throws} for more details.

\*

\* Besides the async nature to await the completion behaves identically to {@link doesNotThrow}.

\*

\* ```js

\* import assert from 'assert/strict';

\*

\* await assert.doesNotReject(

\* async () => {

\* throw new TypeError('Wrong value');

\* },

\* SyntaxError

\* );

\* ```

\*

\* ```js

\* import assert from 'assert/strict';

\*

\* assert.doesNotReject(Promise.reject(new TypeError('Wrong value')))

\* .then(() => {

\* // ...

\* });

\* ```

\* @since v10.0.0

\*/

function doesNotReject(block: (() => Promise<unknown>) | Promise<unknown>, message?: string | Error): Promise<void>;

function doesNotReject(block: (() => Promise<unknown>) | Promise<unknown>, error: AssertPredicate, message?: string | Error): Promise<void>;

/\*\*

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

\*

\* ```js

\* import assert from '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`.

\* @since v13.6.0, v12.16.0

\*/

function match(value: string, regExp: RegExp, message?: string | Error): void;

/\*\*

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

\*

\* ```js

\* import assert from '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`.

\* @since v13.6.0, v12.16.0

\*/

function doesNotMatch(value: string, regExp: RegExp, message?: string | Error): void;

const strict: Omit<typeof assert, 'equal' | 'notEqual' | 'deepEqual' | 'notDeepEqual' | 'ok' | 'strictEqual' | 'deepStrictEqual' | 'ifError' | 'strict'> & {

(value: unknown, message?: string | Error): asserts value;

equal: typeof strictEqual;

notEqual: typeof notStrictEqual;

deepEqual: typeof deepStrictEqual;

notDeepEqual: typeof notDeepStrictEqual;

// Mapped types and assertion functions are incompatible?

// TS2775: Assertions require every name in the call target

// to be declared with an explicit type annotation.

ok: typeof ok;

strictEqual: typeof strictEqual;

deepStrictEqual: typeof deepStrictEqual;

ifError: typeof ifError;

strict: typeof strict;

};

}

export = assert;

}

declare module 'node:assert' {

import assert = require('assert');

export = assert;

}