Util

Stability: 2 - Stable

The `util` module supports the needs of Node.js internal APIs. Many of the utilities are useful for application and module developers as well. To access it:

```
const util = require('util');
```

`util.callbackify(original)`

- `original` {Function} An `async` function
- Returns: {Function} a callback style function

Takes an `async` function (or a function that returns a `Promise`) and returns a function following the error-first callback style, i.e. taking an `(err, value) => ...` callback as the last argument. In the callback, the first argument will be the rejection reason (or `null` if the `Promise` resolved), and the second argument will be the resolved value.

```
const util = require('util');

async function fn() {
  return 'hello world';
}
const callbackFunction = util.callbackify(fn);

callbackFunction((err, ret) => {
  if (err) throw err;
  console.log(ret);
});
```

Will print:

```
hello world
```

The callback is executed asynchronously, and will have a limited stack trace. If the callback throws, the process will emit an `'uncaughtException'` event, and if not handled will exit.

Since `null` has a special meaning as the first argument to a callback, if a wrapped function rejects a `Promise` with a falsy value as a reason, the value is wrapped in an `Error` with the original value stored in a field named `reason`.

```
function fn() {
  return Promise.reject(null);
}
const callbackFunction = util.callbackify(fn);

callbackFunction((err, ret) => {
```

```
  // When the Promise was rejected with `null` it is wrapped with an Error and
  // the original value is stored in `reason`.
  err && Object.hasOwn(err, 'reason') && err.reason === null;  // true
});
```

util.debuglog(section[, callback])

- `section` {string} A string identifying the portion of the application for which the `debuglog` function is being created.
- `callback` {Function} A callback invoked the first time the logging function is called with a function argument that is a more optimized logging function.
- Returns: {Function} The logging function

The `util.debuglog()` method is used to create a function that conditionally writes debug messages to `stderr` based on the existence of the `NODE_DEBUG` environment variable. If the `section` name appears within the value of that environment variable, then the returned function operates similar to `console.error()`. If not, then the returned function is a no-op.

```
const util = require('util');
const debuglog = util.debuglog('foo');

debuglog('hello from foo [%d]', 123);
```

If this program is run with `NODE_DEBUG=foo` in the environment, then it will output something like:

```
FOO 3245: hello from foo [123]
```

where `3245` is the process id. If it is not run with that environment variable set, then it will not print anything.

The `section` supports wildcard also:

```
const util = require('util');
const debuglog = util.debuglog('foo-bar');

debuglog('hi there, it\'s foo-bar [%d]', 2333);
```

if it is run with `NODE_DEBUG=foo*` in the environment, then it will output something like:

```
FOO-BAR 3257: hi there, it's foo-bar [2333]
```

Multiple comma-separated `section` names may be specified in the `NODE_DEBUG` environment variable: `NODE_DEBUG=fs,net,tls`.

The optional `callback` argument can be used to replace the logging function with a different function that doesn't have any initialization or unnecessary wrapping.

```
const util = require('util');
let debuglog = util.debuglog('internals', (debug) => {
  // Replace with a logging function that optimizes out
  // testing if the section is enabled
  debuglog = debug;
});
```

`debuglog().enabled`

- {boolean}

The `util.debuglog().enabled` getter is used to create a test that can be used in conditionals based on the existence of the `NODE_DEBUG` environment variable. If the `section` name appears within the value of that environment variable, then the returned value will be `true`. If not, then the returned value will be `false`.

```
const util = require('util');
const enabled = util.debuglog('foo').enabled;
if (enabled) {
  console.log('hello from foo [%d]', 123);
}
```

If this program is run with `NODE_DEBUG=foo` in the environment, then it will output something like:

```
hello from foo [123]
```

`util.debug(section)`

Alias for `util.debuglog`. Usage allows for readability of that doesn't imply logging when only using `util.debuglog().enabled`.

`util.deprecate(fn, msg[, code])`

- `fn` {Function} The function that is being deprecated.
- `msg` {string} A warning message to display when the deprecated function is invoked.
- `code` {string} A deprecation code. See the list of deprecated APIs for a list of codes.
- Returns: {Function} The deprecated function wrapped to emit a warning.

The `util.deprecate()` method wraps `fn` (which may be a function or class) in such a way that it is marked as deprecated.

```
const util = require('util');

exports.obsoleteFunction = util.deprecate(() => {
  // Do something here.
}, 'obsoleteFunction() is deprecated. Use newShinyFunction() instead.');
```

When called, `util.deprecate()` will return a function that will emit a `DeprecationWarning` using the `'warning'` event. The warning will be emitted and printed to `stderr` the first time the returned function is called. After the warning is emitted, the wrapped function is called without emitting a warning.

If the same optional `code` is supplied in multiple calls to `util.deprecate()`, the warning will be emitted only once for that `code`.

```
const util = require('util');

const fn1 = util.deprecate(someFunction, someMessage, 'DEP0001');
const fn2 = util.deprecate(someOtherFunction, someOtherMessage, 'DEP0001');
fn1(); // Emits a deprecation warning with code DEP0001
fn2(); // Does not emit a deprecation warning because it has the same code
```

If either the `--no-deprecation` or `--no-warnings` command-line flags are used, or if the `process.noDeprecation` property is set to `true` *prior* to the first deprecation warning, the `util.deprecate()` method does nothing.

If the `--trace-deprecation` or `--trace-warnings` command-line flags are set, or the `process.traceDeprecation` property is set to `true`, a warning and a stack trace are printed to `stderr` the first time the deprecated function is called.

If the `--throw-deprecation` command-line flag is set, or the `process.throwDeprecation` property is set to `true`, then an exception will be thrown when the deprecated function is called.

The `--throw-deprecation` command-line flag and `process.throwDeprecation` property take precedence over `--trace-deprecation` and `process.traceDeprecation`.

`util.format(format[, ...args])`

- `format` {string} A `printf`-like format string.

The `util.format()` method returns a formatted string using the first argument as a `printf`-like format string which can contain zero or more format specifiers. Each specifier is replaced with the converted value from the corresponding argument. Supported specifiers are:

- `%s`: `String` will be used to convert all values except `BigInt`, `Object` and `-0`. `BigInt` values will be represented with an `n` and Objects that have no user defined `toString` function are inspected using `util.inspect()` with options `{ depth: 0, colors: false, compact: 3 }`.
- `%d`: `Number` will be used to convert all values except `BigInt` and `Symbol`.
- `%i`: `parseInt(value, 10)` is used for all values except `BigInt` and `Symbol`.
- `%f`: `parseFloat(value)` is used for all values expect `Symbol`.
- `%j`: JSON. Replaced with the string `'[Circular]'` if the argument contains circular references.
- `%o`: `Object`. A string representation of an object with generic JavaScript object formatting. Similar to `util.inspect()` with options `{ showHidden:`

`true, showProxy: true }`. This will show the full object including non-enumerable properties and proxies.
- `%O`: `Object`. A string representation of an object with generic JavaScript object formatting. Similar to `util.inspect()` without options. This will show the full object not including non-enumerable properties and proxies.
- `%c`: `CSS`. This specifier is ignored and will skip any CSS passed in.
- `%%`: single percent sign (`'%'`). This does not consume an argument.
- Returns: {string} The formatted string

If a specifier does not have a corresponding argument, it is not replaced:

```
util.format('%s:%s', 'foo');
// Returns: 'foo:%s'
```

Values that are not part of the format string are formatted using `util.inspect()` if their type is not `string`.

If there are more arguments passed to the `util.format()` method than the number of specifiers, the extra arguments are concatenated to the returned string, separated by spaces:

```
util.format('%s:%s', 'foo', 'bar', 'baz');
// Returns: 'foo:bar baz'
```

If the first argument does not contain a valid format specifier, `util.format()` returns a string that is the concatenation of all arguments separated by spaces:

```
util.format(1, 2, 3);
// Returns: '1 2 3'
```

If only one argument is passed to `util.format()`, it is returned as it is without any formatting:

```
util.format('%% %s');
// Returns: '%% %s'
```

`util.format()` is a synchronous method that is intended as a debugging tool. Some input values can have a significant performance overhead that can block the event loop. Use this function with care and never in a hot code path.

`util.formatWithOptions(inspectOptions, format[, ...args])`

- `inspectOptions` {Object}
- `format` {string}

This function is identical to `util.format()`, except in that it takes an `inspectOptions` argument which specifies options that are passed along to `util.inspect()`.

```
util.formatWithOptions({ colors: true }, 'See object %O', { foo: 42 });
// Returns 'See object { foo: 42 }', where `42` is colored as a number
// when printed to a terminal.
```

util.getSystemErrorName(err)

- err {number}
- Returns: {string}

Returns the string name for a numeric error code that comes from a Node.js API. The mapping between error codes and error names is platform-dependent. See Common System Errors for the names of common errors.

```
fs.access('file/that/does/not/exist', (err) => {
  const name = util.getSystemErrorName(err.errno);
  console.error(name);  // ENOENT
});
```

util.getSystemErrorMap()

- Returns: {Map}

Returns a Map of all system error codes available from the Node.js API. The mapping between error codes and error names is platform-dependent. See Common System Errors for the names of common errors.

```
fs.access('file/that/does/not/exist', (err) => {
  const errorMap = util.getSystemErrorMap();
  const name = errorMap.get(err.errno);
  console.error(name);  // ENOENT
});
```

util.inherits(constructor, superConstructor)

> Stability: 3 - Legacy: Use ES2015 class syntax and `extends` keyword instead.

- constructor {Function}
- superConstructor {Function}

Usage of `util.inherits()` is discouraged. Please use the ES6 `class` and `extends` keywords to get language level inheritance support. Also note that the two styles are semantically incompatible.

Inherit the prototype methods from one constructor into another. The prototype of `constructor` will be set to a new object created from `superConstructor`.

This mainly adds some input validation on top of `Object.setPrototypeOf(constructor.prototype, superConstructor.prototype)`. As an additional convenience, `superConstructor` will be accessible through the `constructor.super_` property.

```
const util = require('util');
const EventEmitter = require('events');

function MyStream() {
```

```
  EventEmitter.call(this);
}

util.inherits(MyStream, EventEmitter);

MyStream.prototype.write = function(data) {
  this.emit('data', data);
};

const stream = new MyStream();

console.log(stream instanceof EventEmitter); // true
console.log(MyStream.super_ === EventEmitter); // true

stream.on('data', (data) => {
  console.log(`Received data: "${data}"`);
});
stream.write('It works!'); // Received data: "It works!"
```

ES6 example using `class` and `extends`:

```
const EventEmitter = require('events');

class MyStream extends EventEmitter {
  write(data) {
    this.emit('data', data);
  }
}

const stream = new MyStream();

stream.on('data', (data) => {
  console.log(`Received data: "${data}"`);
});
stream.write('With ES6');
```

`util.inspect(object[, options])`

`util.inspect(object[, showHidden[, depth[, colors]]])`

- `object` {any} Any JavaScript primitive or `Object`.
- `options` {Object}
  - `showHidden` {boolean} If `true`, `object`'s non-enumerable symbols and properties are included in the formatted result. `WeakMap` and `WeakSet` entries are also included as well as user defined prototype properties (excluding method properties). **Default:** `false`.
  - `depth` {number} Specifies the number of times to recurse while for-

matting `object`. This is useful for inspecting large objects. To recurse up to the maximum call stack size pass `Infinity` or `null`. **Default:** `2`.

- `colors` {boolean} If `true`, the output is styled with ANSI color codes. Colors are customizable. See Customizing `util.inspect` colors. **Default:** `false`.
- `customInspect` {boolean} If `false`, `[util.inspect.custom](depth, opts, inspect)` functions are not invoked. **Default:** `true`.
- `showProxy` {boolean} If `true`, `Proxy` inspection includes the `target` and `handler` objects. **Default:** `false`.
- `maxArrayLength` {integer} Specifies the maximum number of `Array`, `TypedArray`, `WeakMap` and `WeakSet` elements to include when formatting. Set to `null` or `Infinity` to show all elements. Set to `0` or negative to show no elements. **Default:** `100`.
- `maxStringLength` {integer} Specifies the maximum number of characters to include when formatting. Set to `null` or `Infinity` to show all elements. Set to `0` or negative to show no characters. **Default:** `10000`.
- `breakLength` {integer} The length at which input values are split across multiple lines. Set to `Infinity` to format the input as a single line (in combination with `compact` set to `true` or any number >= `1`). **Default:** `80`.
- `compact` {boolean|integer} Setting this to `false` causes each object key to be displayed on a new line. It will break on new lines in text that is longer than `breakLength`. If set to a number, the most `n` inner elements are united on a single line as long as all properties fit into `breakLength`. Short array elements are also grouped together. For more information, see the example below. **Default:** `3`.
- `sorted` {boolean|Function} If set to `true` or a function, all properties of an object, and `Set` and `Map` entries are sorted in the resulting string. If set to `true` the default sort is used. If set to a function, it is used as a compare function.
- `getters` {boolean|string} If set to `true`, getters are inspected. If set to `'get'`, only getters without a corresponding setter are inspected. If set to `'set'`, only getters with a corresponding setter are inspected. This might cause side effects depending on the getter function. **Default:** `false`.
- `numericSeparator` {boolean} If set to `true`, an underscore is used to separate every three digits in all bigints and numbers. **Default:** `false`.

- Returns: {string} The representation of `object`.

The `util.inspect()` method returns a string representation of `object` that is intended for debugging. The output of `util.inspect` may change at any time and should not be depended upon programmatically. Additional `options` may be passed that alter the result. `util.inspect()` will use the constructor's name

and/or `@@toStringTag` to make an identifiable tag for an inspected value.

```
class Foo {
  get [Symbol.toStringTag]() {
    return 'bar';
  }
}

class Bar {}

const baz = Object.create(null, { [Symbol.toStringTag]: { value: 'foo' } });

util.inspect(new Foo()); // 'Foo [bar] {}'
util.inspect(new Bar()); // 'Bar {}'
util.inspect(baz);       // '[foo] {}'
```

Circular references point to their anchor by using a reference index:

```
const { inspect } = require('util');

const obj = {};
obj.a = [obj];
obj.b = {};
obj.b.inner = obj.b;
obj.b.obj = obj;

console.log(inspect(obj));
// <ref *1> {
//   a: [ [Circular *1] ],
//   b: <ref *2> { inner: [Circular *2], obj: [Circular *1] }
// }
```

The following example inspects all properties of the `util` object:

```
const util = require('util');

console.log(util.inspect(util, { showHidden: true, depth: null }));
```

The following example highlights the effect of the `compact` option:

```
const util = require('util');

const o = {
  a: [1, 2, [[
    'Lorem ipsum dolor sit amet,\nconsectetur adipiscing elit, sed do ' +
      'eiusmod \ntempor incididunt ut labore et dolore magna aliqua.',
    'test',
    'foo']], 4],
  b: new Map([['za', 1], ['zb', 'test']])
```

```
};
console.log(util.inspect(o, { compact: true, depth: 5, breakLength: 80 }));

// { a:
//   [ 1,
//     2,
//     [ [ 'Lorem ipsum dolor sit amet,\nconsectetur [...]', // A long line
//         'test',
//         'foo' ] ],
//     4 ],
//   b: Map(2) { 'za' => 1, 'zb' => 'test' } }

// Setting `compact` to false or an integer creates more reader friendly output.
console.log(util.inspect(o, { compact: false, depth: 5, breakLength: 80 }));

// {
//   a: [
//     1,
//     2,
//     [
//       [
//         'Lorem ipsum dolor sit amet,\n' +
//           'consectetur adipiscing elit, sed do eiusmod \n' +
//           'tempor incididunt ut labore et dolore magna aliqua.',
//         'test',
//         'foo'
//       ]
//     ],
//     4
//   ],
//   b: Map(2) {
//     'za' => 1,
//     'zb' => 'test'
//   }
// }

// Setting `breakLength` to e.g. 150 will print the "Lorem ipsum" text in a
// single line.
```

The `showHidden` option allows `WeakMap` and `WeakSet` entries to be inspected. If there are more entries than `maxArrayLength`, there is no guarantee which entries are displayed. That means retrieving the same `WeakSet` entries twice may result in different output. Furthermore, entries with no remaining strong references may be garbage collected at any time.

```
const { inspect } = require('util');
```

```
const obj = { a: 1 };
const obj2 = { b: 2 };
const weakSet = new WeakSet([obj, obj2]);

console.log(inspect(weakSet, { showHidden: true }));
// WeakSet { { a: 1 }, { b: 2 } }
```

The `sorted` option ensures that an object's property insertion order does not impact the result of `util.inspect()`.

```
const { inspect } = require('util');
const assert = require('assert');

const o1 = {
  b: [2, 3, 1],
  a: '`a` comes before `b`',
  c: new Set([2, 3, 1])
};
console.log(inspect(o1, { sorted: true }));
// { a: '`a` comes before `b`', b: [ 2, 3, 1 ], c: Set(3) { 1, 2, 3 } }
console.log(inspect(o1, { sorted: (a, b) => b.localeCompare(a) }));
// { c: Set(3) { 3, 2, 1 }, b: [ 2, 3, 1 ], a: '`a` comes before `b`' }

const o2 = {
  c: new Set([2, 1, 3]),
  a: '`a` comes before `b`',
  b: [2, 3, 1]
};
assert.strict.equal(
  inspect(o1, { sorted: true }),
  inspect(o2, { sorted: true })
);
```

The `numericSeparator` option adds an underscore every three digits to all numbers.

```
const { inspect } = require('util');

const thousand = 1_000;
const million = 1_000_000;
const bigNumber = 123_456_789n;
const bigDecimal = 1_234.123_45;

console.log(thousand, million, bigNumber, bigDecimal);
// 1_000 1_000_000 123_456_789n 1_234.123_45
```

`util.inspect()` is a synchronous method intended for debugging. Its maximum output length is approximately 128 MB. Inputs that result in longer output will

be truncated.

Customizing `util.inspect` colors

Color output (if enabled) of `util.inspect` is customizable globally via the `util.inspect.styles` and `util.inspect.colors` properties.

`util.inspect.styles` is a map associating a style name to a color from `util.inspect.colors`.

The default styles and associated colors are:

- `bigint`: `yellow`
- `boolean`: `yellow`
- `date`: `magenta`
- `module`: `underline`
- `name`: (no styling)
- `null`: `bold`
- `number`: `yellow`
- `regexp`: `red`
- `special`: `cyan` (e.g., `Proxies`)
- `string`: `green`
- `symbol`: `green`
- `undefined`: `grey`

Color styling uses ANSI control codes that may not be supported on all terminals. To verify color support use `tty.hasColors()`.

Predefined control codes are listed below (grouped as "Modifiers", "Foreground colors", and "Background colors").

Modifiers Modifier support varies throughout different terminals. They will mostly be ignored, if not supported.

- `reset` - Resets all (color) modifiers to their defaults
- **bold** - Make text bold
- *italic* - Make text italic
- underline - Make text underlined
- ~~strikethrough~~ - Puts a horizontal line through the center of the text (Alias: `strikeThrough`, `crossedout`, `crossedOut`)
- `hidden` - Prints the text, but makes it invisible (Alias: conceal)
- dim - Decreased color intensity (Alias: `faint`)
- overlined - Make text overlined
- blink - Hides and shows the text in an interval
- inverse - Swap foreground and background colors (Alias: `swapcolors`, `swapColors`)
- doubleunderline - Make text double underlined (Alias: `doubleUnderline`)
- framed - Draw a frame around the text

Foreground colors

- `black`
- `red`
- `green`
- `yellow`
- `blue`
- `magenta`
- `cyan`
- `white`
- `gray` (alias: `grey`, `blackBright`)
- `redBright`
- `greenBright`
- `yellowBright`
- `blueBright`
- `magentaBright`
- `cyanBright`
- `whiteBright`

Background colors

- `bgBlack`
- `bgRed`
- `bgGreen`
- `bgYellow`
- `bgBlue`
- `bgMagenta`
- `bgCyan`
- `bgWhite`
- `bgGray` (alias: `bgGrey`, `bgBlackBright`)
- `bgRedBright`
- `bgGreenBright`
- `bgYellowBright`
- `bgBlueBright`
- `bgMagentaBright`
- `bgCyanBright`
- `bgWhiteBright`

Custom inspection functions on objects

Objects may also define their own `[util.inspect.custom](depth, opts, inspect)` function, which `util.inspect()` will invoke and use the result of when inspecting the object.

```
const util = require('util');

class Box {
```

```
  constructor(value) {
    this.value = value;
  }

  [util.inspect.custom](depth, options, inspect) {
    if (depth < 0) {
      return options.stylize('[Box]', 'special');
    }

    const newOptions = Object.assign({}, options, {
      depth: options.depth === null ? null : options.depth - 1
    });

    // Five space padding because that's the size of "Box< ".
    const padding = ' '.repeat(5);
    const inner = inspect(this.value, newOptions)
                  .replace(/\n/g, `\n${padding}`);
    return `${options.stylize('Box', 'special')}< ${inner} >`;
  }
}

const box = new Box(true);

util.inspect(box);
// Returns: "Box< true >"
```

Custom `[util.inspect.custom](depth, opts, inspect)` functions typically return a string but may return a value of any type that will be formatted accordingly by `util.inspect()`.

```
const util = require('util');

const obj = { foo: 'this will not show up in the inspect() output' };
obj[util.inspect.custom] = (depth) => {
  return { bar: 'baz' };
};

util.inspect(obj);
// Returns: "{ bar: 'baz' }"
```

`util.inspect.custom`

- {symbol} that can be used to declare custom inspect functions.

In addition to being accessible through `util.inspect.custom`, this symbol is registered globally and can be accessed in any environment as `Symbol.for('nodejs.util.inspect.custom')`.

Using this allows code to be written in a portable fashion, so that the custom inspect function is used in an Node.js environment and ignored in the browser. The `util.inspect()` function itself is passed as third argument to the custom inspect function to allow further portability.

```
const customInspectSymbol = Symbol.for('nodejs.util.inspect.custom');

class Password {
  constructor(value) {
    this.value = value;
  }

  toString() {
    return 'xxxxxxxx';
  }

  [customInspectSymbol](depth, inspectOptions, inspect) {
    return `Password <${this.toString()}>`;
  }
}

const password = new Password('r0sebud');
console.log(password);
// Prints Password <xxxxxxxx>
```

See Custom inspection functions on Objects for more details.

`util.inspect.defaultOptions`

The `defaultOptions` value allows customization of the default options used by `util.inspect`. This is useful for functions like `console.log` or `util.format` which implicitly call into `util.inspect`. It shall be set to an object containing one or more valid `util.inspect()` options. Setting option properties directly is also supported.

```
const util = require('util');
const arr = Array(101).fill(0);

console.log(arr); // Logs the truncated array
util.inspect.defaultOptions.maxArrayLength = null;
console.log(arr); // logs the full array
```

`util.isDeepStrictEqual(val1, val2)`

- `val1` {any}
- `val2` {any}
- Returns: {boolean}

Returns `true` if there is deep strict equality between `val1` and `val2`. Otherwise, returns `false`.

See `assert.deepStrictEqual()` for more information about deep strict equality.

`util.promisify(original)`

- `original` {Function}
- Returns: {Function}

Takes a function following the common error-first callback style, i.e. taking an `(err, value) => ...` callback as the last argument, and returns a version that returns promises.

```
const util = require('util');
const fs = require('fs');

const stat = util.promisify(fs.stat);
stat('.').then((stats) => {
  // Do something with `stats`
}).catch((error) => {
  // Handle the error.
});
```

Or, equivalently using `async functions`:

```
const util = require('util');
const fs = require('fs');

const stat = util.promisify(fs.stat);

async function callStat() {
  const stats = await stat('.');
  console.log(`This directory is owned by ${stats.uid}`);
}
```

If there is an `original[util.promisify.custom]` property present, `promisify` will return its value, see Custom promisified functions.

`promisify()` assumes that `original` is a function taking a callback as its final argument in all cases. If `original` is not a function, `promisify()` will throw an error. If `original` is a function but its last argument is not an error-first callback, it will still be passed an error-first callback as its last argument.

Using `promisify()` on class methods or other methods that use `this` may not work as expected unless handled specially:

```
const util = require('util');

class Foo {
```

```
  constructor() {
    this.a = 42;
  }

  bar(callback) {
    callback(null, this.a);
  }
}

const foo = new Foo();

const naiveBar = util.promisify(foo.bar);
// TypeError: Cannot read property 'a' of undefined
// naiveBar().then(a => console.log(a));

naiveBar.call(foo).then((a) => console.log(a)); // '42'

const bindBar = naiveBar.bind(foo);
bindBar().then((a) => console.log(a)); // '42'
```

Custom promisified functions

Using the `util.promisify.custom` symbol one can override the return value of `util.promisify()`:

```
const util = require('util');

function doSomething(foo, callback) {
  // ...
}

doSomething[util.promisify.custom] = (foo) => {
  return getPromiseSomehow();
};

const promisified = util.promisify(doSomething);
console.log(promisified === doSomething[util.promisify.custom]);
// prints 'true'
```

This can be useful for cases where the original function does not follow the standard format of taking an error-first callback as the last argument.

For example, with a function that takes in `(foo, onSuccessCallback, onErrorCallback)`:

```
doSomething[util.promisify.custom] = (foo) => {
  return new Promise((resolve, reject) => {
    doSomething(foo, resolve, reject);
```

```
  });
};
```

If `promisify.custom` is defined but is not a function, `promisify()` will throw an error.

`util.promisify.custom`

- {symbol} that can be used to declare custom promisified variants of functions, see Custom promisified functions.

In addition to being accessible through `util.promisify.custom`, this symbol is registered globally and can be accessed in any environment as `Symbol.for('nodejs.util.promisify.custom')`.

For example, with a function that takes in `(foo, onSuccessCallback, onErrorCallback)`:

```
const kCustomPromisifiedSymbol = Symbol.for('nodejs.util.promisify.custom');

doSomething[kCustomPromisifiedSymbol] = (foo) => {
  return new Promise((resolve, reject) => {
    doSomething(foo, resolve, reject);
  });
};
```

`util.stripVTControlCharacters(str)`

- `str` {string}
- Returns: {string}

Returns `str` with any ANSI escape codes removed.

```
console.log(util.stripVTControlCharacters('\u001B[4mvalue\u001B[0m'));
// Prints "value"
```

Class: `util.TextDecoder`

An implementation of the WHATWG Encoding Standard `TextDecoder` API.

```
const decoder = new TextDecoder();
const u8arr = new Uint8Array([72, 101, 108, 108, 111]);
console.log(decoder.decode(u8arr)); // Hello
```

WHATWG supported encodings

Per the WHATWG Encoding Standard, the encodings supported by the `TextDecoder` API are outlined in the tables below. For each encoding, one or more aliases may be used.

Different Node.js build configurations support different sets of encodings. (see Internationalization)

Encodings supported by default (with full ICU data)

Encoding	Aliases
'ibm866'	'866', 'cp866', 'csibm866'
'iso-8859-2'	'csisolatin2', 'iso-ir-101', 'iso8859-2', 'iso88592', 'iso_8859-2', 'iso_8859-2:1987', 'l2', 'latin2'
'iso-8859-3'	'csisolatin3', 'iso-ir-109', 'iso8859-3', 'iso88593', 'iso_8859-3', 'iso_8859-3:1988', 'l3', 'latin3'
'iso-8859-4'	'csisolatin4', 'iso-ir-110', 'iso8859-4', 'iso88594', 'iso_8859-4', 'iso_8859-4:1988', 'l4', 'latin4'
'iso-8859-5'	'csisolatincyrillic', 'cyrillic', 'iso-ir-144', 'iso8859-5', 'iso88595', 'iso_8859-5', 'iso_8859-5:1988'
'iso-8859-6'	'arabic', 'asmo-708', 'csiso88596e', 'csiso88596i', 'csisolatinarabic', 'ecma-114', 'iso-8859-6-e', 'iso-8859-6-i', 'iso-ir-127', 'iso8859-6', 'iso88596', 'iso_8859-6', 'iso_8859-6:1987'
'iso-8859-7'	'csisolatingreek', 'ecma-118', 'elot_928', 'greek', 'greek8', 'iso-ir-126', 'iso8859-7', 'iso88597', 'iso_8859-7', 'iso_8859-7:1987', 'sun_eu_greek'
'iso-8859-8'	'csiso88598e', 'csisolatinhebrew', 'hebrew', 'iso-8859-8-e', 'iso-ir-138', 'iso8859-8', 'iso88598', 'iso_8859-8', 'iso_8859-8:1988', 'visual'
'iso-8859-8-i'	'csiso88598i', 'logical'
'iso-8859-10'	'csisolatin6', 'iso-ir-157', 'iso8859-10', 'iso885910', 'l6', 'latin6'
'iso-8859-13'	'iso8859-13', 'iso885913'
'iso-8859-14'	'iso8859-14', 'iso885914'
'iso-8859-15'	'csisolatin9', 'iso8859-15', 'iso885915', 'iso_8859-15', 'l9'
'koi8-r'	'cskoi8r', 'koi', 'koi8', 'koi8_r'
'koi8-u'	'koi8-ru'
'macintosh'	'csmacintosh', 'mac', 'x-mac-roman'
'windows-874'	'iso-8859-11', 'iso8859-11', 'iso885911', 'tis-620'
'windows-1250'	'x-cp1250'
'windows-1251'	'x-cp1251'
'windows-1252'	'ansi_x3.4-1968', 'ascii', 'cp1252', 'cp819', 'csisolatin1', 'ibm819', 'iso-8859-1', 'iso-ir-100', 'iso8859-1', 'iso88591', 'iso_8859-1', 'iso_8859-1:1987', 'l1', 'latin1', 'us-ascii', 'x-cp1252'
'windows-1253'	'x-cp1253'
'windows-1254'	'csisolatin5', 'iso-8859-9', 'iso-ir-148', 'iso8859-9', 'iso88599', 'iso_8859-9', 'iso_8859-9:1989', 'l5', 'latin5', 'x-cp1254'

Encoding	Aliases
'windows-1255'	'cp1255', 'x-cp1255'
'windows-1256'	'cp1256', 'x-cp1256'
'windows-1257'	'cp1257', 'x-cp1257'
'windows-1258'	'cp1258', 'x-cp1258'
'x-mac-cyrillic'	'x-mac-ukrainian'
'gbk'	'chinese', 'csgb2312', 'csiso58gb231280', 'gb2312', 'gb_2312', 'gb_2312-80', 'iso-ir-58', 'x-gbk'
'gb18030'	
'big5'	'big5-hkscs', 'cn-big5', 'csbig5', 'x-x-big5'
'euc-jp'	'cseucpkdfmtjapanese', 'x-euc-jp'
'iso-2022-jp'	'csiso2022jp'
'shift_jis'	'csshiftjis', 'ms932', 'ms_kanji', 'shift-jis', 'sjis', 'windows-31j', 'x-sjis'
'euc-kr'	'cseuckr', 'csksc56011987', 'iso-ir-149', 'korean', 'ks_c_5601-1987', 'ks_c_5601-1989', 'ksc5601', 'ksc_5601', 'windows-949'

Encodings supported when Node.js is built with the `small-icu` option

Encoding	Aliases
'utf-8'	'unicode-1-1-utf-8', 'utf8'
'utf-16le'	'utf-16'
'utf-16be'	

Encodings supported when ICU is disabled

Encoding	Aliases
'utf-8'	'unicode-1-1-utf-8', 'utf8'
'utf-16le'	'utf-16'

The `'iso-8859-16'` encoding listed in the WHATWG Encoding Standard is not supported.

`new TextDecoder([encoding[, options]])`

- `encoding` {string} Identifies the `encoding` that this `TextDecoder` instance supports. **Default:** `'utf-8'`.
- `options` {Object}

  - `fatal` {boolean} `true` if decoding failures are fatal. This option is not supported when ICU is disabled (see Internationalization). **Default:** `false`.
  - `ignoreBOM` {boolean} When `true`, the `TextDecoder` will include the byte order mark in the decoded result. When `false`, the byte order mark will be removed from the output. This option is only used when `encoding` is `'utf-8'`, `'utf-16be'` or `'utf-16le'`. **Default:** `false`.

Creates a new `TextDecoder` instance. The `encoding` may specify one of the supported encodings or an alias.

The `TextDecoder` class is also available on the global object.

`textDecoder.decode([input[, options]])`

- `input` {ArrayBuffer|DataView|TypedArray} An `ArrayBuffer`, `DataView` or `TypedArray` instance containing the encoded data.
- `options` {Object}
  - `stream` {boolean} `true` if additional chunks of data are expected. **Default:** `false`.
- Returns: {string}

Decodes the `input` and returns a string. If `options.stream` is `true`, any incomplete byte sequences occurring at the end of the `input` are buffered internally and emitted after the next call to `textDecoder.decode()`.

If `textDecoder.fatal` is `true`, decoding errors that occur will result in a `TypeError` being thrown.

`textDecoder.encoding`

- {string}

The encoding supported by the `TextDecoder` instance.

`textDecoder.fatal`

- {boolean}

The value will be `true` if decoding errors result in a `TypeError` being thrown.

`textDecoder.ignoreBOM`

- {boolean}

The value will be `true` if the decoding result will include the byte order mark.

Class: `util.TextEncoder`

An implementation of the WHATWG Encoding Standard `TextEncoder` API. All instances of `TextEncoder` only support UTF-8 encoding.

```
const encoder = new TextEncoder();
const uint8array = encoder.encode('this is some data');
```

The `TextEncoder` class is also available on the global object.

`textEncoder.encode([input])`

- `input` {string} The text to encode. **Default:** an empty string.
- Returns: {Uint8Array}

UTF-8 encodes the `input` string and returns a `Uint8Array` containing the encoded bytes.

`textEncoder.encodeInto(src, dest)`

- `src` {string} The text to encode.
- `dest` {Uint8Array} The array to hold the encode result.
- Returns: {Object}
  - `read` {number} The read Unicode code units of src.
  - `written` {number} The written UTF-8 bytes of dest.

UTF-8 encodes the `src` string to the `dest` Uint8Array and returns an object containing the read Unicode code units and written UTF-8 bytes.

```
const encoder = new TextEncoder();
const src = 'this is some data';
const dest = new Uint8Array(10);
const { read, written } = encoder.encodeInto(src, dest);
```

`textEncoder.encoding`

- {string}

The encoding supported by the `TextEncoder` instance. Always set to `'utf-8'`.

`util.toUSVString(string)`

- `string` {string}

Returns the `string` after replacing any surrogate code points (or equivalently, any unpaired surrogate code units) with the Unicode "replacement character" U+FFFD.

`util.types`

`util.types` provides type checks for different kinds of built-in objects. Unlike `instanceof` or `Object.prototype.toString.call(value)`, these checks do not inspect properties of the object that are accessible from JavaScript (like their prototype), and usually have the overhead of calling into C++.

The result generally does not make any guarantees about what kinds of properties or behavior a value exposes in JavaScript. They are primarily useful for addon developers who prefer to do type checking in JavaScript.

The API is accessible via `require('util').types` or `require('util/types')`.

`util.types.isAnyArrayBuffer(value)`

- `value` {any}
- Returns: {boolean}

Returns `true` if the value is a built-in `ArrayBuffer` or `SharedArrayBuffer` instance.

See also `util.types.isArrayBuffer()` and `util.types.isSharedArrayBuffer()`.

```
util.types.isAnyArrayBuffer(new ArrayBuffer());  // Returns true
util.types.isAnyArrayBuffer(new SharedArrayBuffer());  // Returns true
```

`util.types.isArrayBufferView(value)`

- `value` {any}
- Returns: {boolean}

Returns `true` if the value is an instance of one of the `ArrayBuffer` views, such as typed array objects or `DataView`. Equivalent to `ArrayBuffer.isView()`.

```
util.types.isArrayBufferView(new Int8Array());  // true
util.types.isArrayBufferView(Buffer.from('hello world')); // true
util.types.isArrayBufferView(new DataView(new ArrayBuffer(16)));  // true
util.types.isArrayBufferView(new ArrayBuffer());  // false
```

`util.types.isArgumentsObject(value)`

- `value` {any}
- Returns: {boolean}

Returns `true` if the value is an `arguments` object.

```
function foo() {
  util.types.isArgumentsObject(arguments);  // Returns true
}
```

`util.types.isArrayBuffer(value)`

- `value` {any}
- Returns: {boolean}

Returns `true` if the value is a built-in `ArrayBuffer` instance. This does *not* include `SharedArrayBuffer` instances. Usually, it is desirable to test for both; See `util.types.isAnyArrayBuffer()` for that.

```
util.types.isArrayBuffer(new ArrayBuffer());  // Returns true
util.types.isArrayBuffer(new SharedArrayBuffer());  // Returns false
```

util.types.isAsyncFunction(value)

- value {any}
- Returns: {boolean}

Returns true if the value is an async function. This only reports back what the JavaScript engine is seeing; in particular, the return value may not match the original source code if a transpilation tool was used.

```
util.types.isAsyncFunction(function foo() {});  // Returns false
util.types.isAsyncFunction(async function foo() {});  // Returns true
```

util.types.isBigInt64Array(value)

- value {any}
- Returns: {boolean}

Returns true if the value is a BigInt64Array instance.

```
util.types.isBigInt64Array(new BigInt64Array());   // Returns true
util.types.isBigInt64Array(new BigUint64Array());  // Returns false
```

util.types.isBigUint64Array(value)

- value {any}
- Returns: {boolean}

Returns true if the value is a BigUint64Array instance.

```
util.types.isBigUint64Array(new BigInt64Array());   // Returns false
util.types.isBigUint64Array(new BigUint64Array());  // Returns true
```

util.types.isBooleanObject(value)

- value {any}
- Returns: {boolean}

Returns true if the value is a boolean object, e.g. created by new Boolean().

```
util.types.isBooleanObject(false);  // Returns false
util.types.isBooleanObject(true);   // Returns false
util.types.isBooleanObject(new Boolean(false)); // Returns true
util.types.isBooleanObject(new Boolean(true));  // Returns true
util.types.isBooleanObject(Boolean(false)); // Returns false
util.types.isBooleanObject(Boolean(true));  // Returns false
```

`util.types.isBoxedPrimitive(value)`

- `value` {any}
- Returns: {boolean}

Returns `true` if the value is any boxed primitive object, e.g. created by `new Boolean()`, `new String()` or `Object(Symbol())`.

For example:

```
util.types.isBoxedPrimitive(false); // Returns false
util.types.isBoxedPrimitive(new Boolean(false)); // Returns true
util.types.isBoxedPrimitive(Symbol('foo')); // Returns false
util.types.isBoxedPrimitive(Object(Symbol('foo'))); // Returns true
util.types.isBoxedPrimitive(Object(BigInt(5))); // Returns true
```

`util.types.isCryptoKey(value)`

- `value` {Object}
- Returns: {boolean}

Returns `true` if `value` is a {CryptoKey}, `false` otherwise.

`util.types.isDataView(value)`

- `value` {any}
- Returns: {boolean}

Returns `true` if the value is a built-in `DataView` instance.

```
const ab = new ArrayBuffer(20);
util.types.isDataView(new DataView(ab));  // Returns true
util.types.isDataView(new Float64Array());  // Returns false
```

See also `ArrayBuffer.isView()`.

`util.types.isDate(value)`

- `value` {any}
- Returns: {boolean}

Returns `true` if the value is a built-in `Date` instance.

```
util.types.isDate(new Date());  // Returns true
```

`util.types.isExternal(value)`

- `value` {any}
- Returns: {boolean}

Returns `true` if the value is a native `External` value.

A native `External` value is a special type of object that contains a raw C++ pointer (`void*`) for access from native code, and has no other properties. Such objects are created either by Node.js internals or native addons. In JavaScript, they are frozen objects with a `null` prototype.

```c
#include <js_native_api.h>
#include <stdlib.h>
napi_value result;
static napi_value MyNapi(napi_env env, napi_callback_info info) {
  int* raw = (int*) malloc(1024);
  napi_status status = napi_create_external(env, (void*) raw, NULL, NULL, &result);
  if (status != napi_ok) {
    napi_throw_error(env, NULL, "napi_create_external failed");
    return NULL;
  }
  return result;
}
...
DECLARE_NAPI_PROPERTY("myNapi", MyNapi)
...
```

```js
const native = require('napi_addon.node');
const data = native.myNapi();
util.types.isExternal(data); // returns true
util.types.isExternal(0); // returns false
util.types.isExternal(new String('foo')); // returns false
```

For further information on `napi_create_external`, refer to `napi_create_external()`.

`util.types.isFloat32Array(value)`

- `value` {any}
- Returns: {boolean}

Returns `true` if the value is a built-in `Float32Array` instance.

```js
util.types.isFloat32Array(new ArrayBuffer());  // Returns false
util.types.isFloat32Array(new Float32Array());  // Returns true
util.types.isFloat32Array(new Float64Array());  // Returns false
```

`util.types.isFloat64Array(value)`

- `value` {any}
- Returns: {boolean}

Returns `true` if the value is a built-in `Float64Array` instance.

```js
util.types.isFloat64Array(new ArrayBuffer());  // Returns false
util.types.isFloat64Array(new Uint8Array());  // Returns false
util.types.isFloat64Array(new Float64Array());  // Returns true
```

`util.types.isGeneratorFunction(value)`

- `value` {any}
- Returns: {boolean}

Returns `true` if the value is a generator function. This only reports back what the JavaScript engine is seeing; in particular, the return value may not match the original source code if a transpilation tool was used.

```
util.types.isGeneratorFunction(function foo() {});  // Returns false
util.types.isGeneratorFunction(function* foo() {});  // Returns true
```

`util.types.isGeneratorObject(value)`

- `value` {any}
- Returns: {boolean}

Returns `true` if the value is a generator object as returned from a built-in generator function. This only reports back what the JavaScript engine is seeing; in particular, the return value may not match the original source code if a transpilation tool was used.

```
function* foo() {}
const generator = foo();
util.types.isGeneratorObject(generator);  // Returns true
```

`util.types.isInt8Array(value)`

- `value` {any}
- Returns: {boolean}

Returns `true` if the value is a built-in `Int8Array` instance.

```
util.types.isInt8Array(new ArrayBuffer());  // Returns false
util.types.isInt8Array(new Int8Array());  // Returns true
util.types.isInt8Array(new Float64Array());  // Returns false
```

`util.types.isInt16Array(value)`

- `value` {any}
- Returns: {boolean}

Returns `true` if the value is a built-in `Int16Array` instance.

```
util.types.isInt16Array(new ArrayBuffer());  // Returns false
util.types.isInt16Array(new Int16Array());  // Returns true
util.types.isInt16Array(new Float64Array());  // Returns false
```

`util.types.isInt32Array(value)`

- `value` {any}

- Returns: {boolean}

Returns `true` if the value is a built-in `Int32Array` instance.

```
util.types.isInt32Array(new ArrayBuffer());  // Returns false
util.types.isInt32Array(new Int32Array());  // Returns true
util.types.isInt32Array(new Float64Array());  // Returns false
```

`util.types.isKeyObject(value)`

- `value` {Object}
- Returns: {boolean}

Returns `true` if `value` is a {KeyObject}, `false` otherwise.

`util.types.isMap(value)`

- `value` {any}
- Returns: {boolean}

Returns `true` if the value is a built-in `Map` instance.

```
util.types.isMap(new Map());  // Returns true
```

`util.types.isMapIterator(value)`

- `value` {any}
- Returns: {boolean}

Returns `true` if the value is an iterator returned for a built-in `Map` instance.

```
const map = new Map();
util.types.isMapIterator(map.keys());  // Returns true
util.types.isMapIterator(map.values());  // Returns true
util.types.isMapIterator(map.entries());  // Returns true
util.types.isMapIterator(map[Symbol.iterator]());  // Returns true
```

`util.types.isModuleNamespaceObject(value)`

- `value` {any}
- Returns: {boolean}

Returns `true` if the value is an instance of a Module Namespace Object.

```
import * as ns from './a.js';

util.types.isModuleNamespaceObject(ns);  // Returns true
```

`util.types.isNativeError(value)`

- `value` {any}
- Returns: {boolean}

Returns `true` if the value is an instance of a built-in `Error` type.

```
util.types.isNativeError(new Error());  // Returns true
util.types.isNativeError(new TypeError());  // Returns true
util.types.isNativeError(new RangeError());  // Returns true
```

`util.types.isNumberObject(value)`

- `value` {any}
- Returns: {boolean}

Returns `true` if the value is a number object, e.g. created by `new Number()`.

```
util.types.isNumberObject(0);  // Returns false
util.types.isNumberObject(new Number(0));   // Returns true
```

`util.types.isPromise(value)`

- `value` {any}
- Returns: {boolean}

Returns `true` if the value is a built-in `Promise`.

```
util.types.isPromise(Promise.resolve(42));  // Returns true
```

`util.types.isProxy(value)`

- `value` {any}
- Returns: {boolean}

Returns `true` if the value is a `Proxy` instance.

```
const target = {};
const proxy = new Proxy(target, {});
util.types.isProxy(target);  // Returns false
util.types.isProxy(proxy);  // Returns true
```

`util.types.isRegExp(value)`

- `value` {any}
- Returns: {boolean}

Returns `true` if the value is a regular expression object.

```
util.types.isRegExp(/abc/);  // Returns true
util.types.isRegExp(new RegExp('abc'));  // Returns true
```

`util.types.isSet(value)`

- `value` {any}
- Returns: {boolean}

Returns `true` if the value is a built-in `Set` instance.

```
util.types.isSet(new Set());  // Returns true
```

`util.types.isSetIterator(value)`

- `value` {any}
- Returns: {boolean}

Returns `true` if the value is an iterator returned for a built-in `Set` instance.

```
const set = new Set();
util.types.isSetIterator(set.keys());  // Returns true
util.types.isSetIterator(set.values());  // Returns true
util.types.isSetIterator(set.entries());  // Returns true
util.types.isSetIterator(set[Symbol.iterator]());  // Returns true
```

`util.types.isSharedArrayBuffer(value)`

- `value` {any}
- Returns: {boolean}

Returns `true` if the value is a built-in `SharedArrayBuffer` instance. This does *not* include `ArrayBuffer` instances. Usually, it is desirable to test for both; See `util.types.isAnyArrayBuffer()` for that.

```
util.types.isSharedArrayBuffer(new ArrayBuffer());  // Returns false
util.types.isSharedArrayBuffer(new SharedArrayBuffer());  // Returns true
```

`util.types.isStringObject(value)`

- `value` {any}
- Returns: {boolean}

Returns `true` if the value is a string object, e.g. created by `new String()`.

```
util.types.isStringObject('foo');  // Returns false
util.types.isStringObject(new String('foo'));   // Returns true
```

`util.types.isSymbolObject(value)`

- `value` {any}
- Returns: {boolean}

Returns `true` if the value is a symbol object, created by calling `Object()` on a `Symbol` primitive.

```
const symbol = Symbol('foo');
util.types.isSymbolObject(symbol);  // Returns false
util.types.isSymbolObject(Object(symbol));   // Returns true
```

`util.types.isTypedArray(value)`

- `value` {any}
- Returns: {boolean}

Returns `true` if the value is a built-in `TypedArray` instance.

```
util.types.isTypedArray(new ArrayBuffer());  // Returns false
util.types.isTypedArray(new Uint8Array());  // Returns true
util.types.isTypedArray(new Float64Array());  // Returns true
```

See also `ArrayBuffer.isView()`.

`util.types.isUint8Array(value)`

- `value` {any}
- Returns: {boolean}

Returns `true` if the value is a built-in `Uint8Array` instance.

```
util.types.isUint8Array(new ArrayBuffer());  // Returns false
util.types.isUint8Array(new Uint8Array());  // Returns true
util.types.isUint8Array(new Float64Array());  // Returns false
```

`util.types.isUint8ClampedArray(value)`

- `value` {any}
- Returns: {boolean}

Returns `true` if the value is a built-in `Uint8ClampedArray` instance.

```
util.types.isUint8ClampedArray(new ArrayBuffer());  // Returns false
util.types.isUint8ClampedArray(new Uint8ClampedArray());  // Returns true
util.types.isUint8ClampedArray(new Float64Array());  // Returns false
```

`util.types.isUint16Array(value)`

- `value` {any}
- Returns: {boolean}

Returns `true` if the value is a built-in `Uint16Array` instance.

```
util.types.isUint16Array(new ArrayBuffer());  // Returns false
util.types.isUint16Array(new Uint16Array());  // Returns true
util.types.isUint16Array(new Float64Array());  // Returns false
```

`util.types.isUint32Array(value)`

- `value` {any}
- Returns: {boolean}

Returns `true` if the value is a built-in `Uint32Array` instance.

```
util.types.isUint32Array(new ArrayBuffer());  // Returns false
util.types.isUint32Array(new Uint32Array());  // Returns true
util.types.isUint32Array(new Float64Array());  // Returns false
```

util.types.isWeakMap(value)

- value {any}
- Returns: {boolean}

Returns true if the value is a built-in WeakMap instance.

```
util.types.isWeakMap(new WeakMap());  // Returns true
```

util.types.isWeakSet(value)

- value {any}
- Returns: {boolean}

Returns true if the value is a built-in WeakSet instance.

```
util.types.isWeakSet(new WeakSet());  // Returns true
```

util.types.isWebAssemblyCompiledModule(value)

> Stability: 0 - Deprecated: Use value instanceof WebAssembly.Module instead.

- value {any}
- Returns: {boolean}

Returns true if the value is a built-in WebAssembly.Module instance.

```
const module = new WebAssembly.Module(wasmBuffer);
util.types.isWebAssemblyCompiledModule(module);  // Returns true
```

Deprecated APIs

The following APIs are deprecated and should no longer be used. Existing applications and modules should be updated to find alternative approaches.

util._extend(target, source)

> Stability: 0 - Deprecated: Use Object.assign() instead.

- target {Object}
- source {Object}

The util._extend() method was never intended to be used outside of internal Node.js modules. The community found and used it anyway.

It is deprecated and should not be used in new code. JavaScript comes with very similar built-in functionality through Object.assign().

`util.isArray(object)`

Stability: 0 - Deprecated: Use `Array.isArray()` instead.

- `object` {any}
- Returns: {boolean}

Alias for `Array.isArray()`.

Returns `true` if the given `object` is an `Array`. Otherwise, returns `false`.

```
const util = require('util');

util.isArray([]);
// Returns: true
util.isArray(new Array());
// Returns: true
util.isArray({});
// Returns: false
```

`util.isBoolean(object)`

Stability: 0 - Deprecated: Use `typeof value === 'boolean'` instead.

- `object` {any}
- Returns: {boolean}

Returns `true` if the given `object` is a `Boolean`. Otherwise, returns `false`.

```
const util = require('util');

util.isBoolean(1);
// Returns: false
util.isBoolean(0);
// Returns: false
util.isBoolean(false);
// Returns: true
```

`util.isBuffer(object)`

Stability: 0 - Deprecated: Use `Buffer.isBuffer()` instead.

- `object` {any}
- Returns: {boolean}

Returns `true` if the given `object` is a `Buffer`. Otherwise, returns `false`.

```
const util = require('util');

util.isBuffer({ length: 0 });
// Returns: false
```

```
util.isBuffer([]);
// Returns: false
util.isBuffer(Buffer.from('hello world'));
// Returns: true
```

util.isDate(object)

> Stability: 0 - Deprecated: Use `util.types.isDate()` instead.

- `object` {any}
- Returns: {boolean}

Returns `true` if the given `object` is a `Date`. Otherwise, returns `false`.

```
const util = require('util');

util.isDate(new Date());
// Returns: true
util.isDate(Date());
// false (without 'new' returns a String)
util.isDate({});
// Returns: false
```

util.isError(object)

> Stability: 0 - Deprecated: Use `util.types.isNativeError()` instead.

- `object` {any}
- Returns: {boolean}

Returns `true` if the given `object` is an `Error`. Otherwise, returns `false`.

```
const util = require('util');

util.isError(new Error());
// Returns: true
util.isError(new TypeError());
// Returns: true
util.isError({ name: 'Error', message: 'an error occurred' });
// Returns: false
```

This method relies on `Object.prototype.toString()` behavior. It is possible to obtain an incorrect result when the `object` argument manipulates `@@toStringTag`.

```
const util = require('util');
const obj = { name: 'Error', message: 'an error occurred' };

util.isError(obj);
```

```
// Returns: false
obj[Symbol.toStringTag] = 'Error';
util.isError(obj);
// Returns: true
```

util.isFunction(object)

> Stability: 0 - Deprecated: Use `typeof value === 'function'` instead.

- `object` {any}
- Returns: {boolean}

Returns `true` if the given `object` is a `Function`. Otherwise, returns `false`.

```
const util = require('util');

function Foo() {}
const Bar = () => {};

util.isFunction({});
// Returns: false
util.isFunction(Foo);
// Returns: true
util.isFunction(Bar);
// Returns: true
```

util.isNull(object)

> Stability: 0 - Deprecated: Use `value === null` instead.

- `object` {any}
- Returns: {boolean}

Returns `true` if the given `object` is strictly `null`. Otherwise, returns `false`.

```
const util = require('util');

util.isNull(0);
// Returns: false
util.isNull(undefined);
// Returns: false
util.isNull(null);
// Returns: true
```

util.isNullOrUndefined(object)

> Stability: 0 - Deprecated: Use `value === undefined || value === null` instead.

- `object` {any}
- Returns: {boolean}

Returns `true` if the given `object` is `null` or `undefined`. Otherwise, returns `false`.

```
const util = require('util');

util.isNullOrUndefined(0);
// Returns: false
util.isNullOrUndefined(undefined);
// Returns: true
util.isNullOrUndefined(null);
// Returns: true
```

`util.isNumber(object)`

> Stability: 0 - Deprecated: Use `typeof value === 'number'` instead.

- `object` {any}
- Returns: {boolean}

Returns `true` if the given `object` is a `Number`. Otherwise, returns `false`.

```
const util = require('util');

util.isNumber(false);
// Returns: false
util.isNumber(Infinity);
// Returns: true
util.isNumber(0);
// Returns: true
util.isNumber(NaN);
// Returns: true
```

`util.isObject(object)`

> Stability: 0 - Deprecated: Use `value !== null && typeof value === 'object'` instead.

- `object` {any}
- Returns: {boolean}

Returns `true` if the given `object` is strictly an `Object` **and** not a `Function` (even though functions are objects in JavaScript). Otherwise, returns `false`.

```
const util = require('util');

util.isObject(5);
```

```
// Returns: false
util.isObject(null);
// Returns: false
util.isObject({});
// Returns: true
util.isObject(() => {});
// Returns: false
```

util.isPrimitive(object)

> Stability: 0 - Deprecated: Use `(typeof value !== 'object' && typeof value !== 'function') || value === null` instead.

- `object` {any}
- Returns: {boolean}

Returns `true` if the given `object` is a primitive type. Otherwise, returns `false`.

```
const util = require('util');

util.isPrimitive(5);
// Returns: true
util.isPrimitive('foo');
// Returns: true
util.isPrimitive(false);
// Returns: true
util.isPrimitive(null);
// Returns: true
util.isPrimitive(undefined);
// Returns: true
util.isPrimitive({});
// Returns: false
util.isPrimitive(() => {});
// Returns: false
util.isPrimitive(/^$/);
// Returns: false
util.isPrimitive(new Date());
// Returns: false
```

util.isRegExp(object)

> Stability: 0 - Deprecated

- `object` {any}
- Returns: {boolean}

Returns `true` if the given `object` is a `RegExp`. Otherwise, returns `false`.

```
const util = require('util');

util.isRegExp(/some regexp/);
// Returns: true
util.isRegExp(new RegExp('another regexp'));
// Returns: true
util.isRegExp({});
// Returns: false
```

util.isString(object)

> Stability: 0 - Deprecated: Use `typeof value === 'string'` instead.

- `object` {any}
- Returns: {boolean}

Returns `true` if the given `object` is a `string`. Otherwise, returns `false`.

```
const util = require('util');

util.isString('');
// Returns: true
util.isString('foo');
// Returns: true
util.isString(String('foo'));
// Returns: true
util.isString(5);
// Returns: false
```

util.isSymbol(object)

> Stability: 0 - Deprecated: Use `typeof value === 'symbol'` instead.

- `object` {any}
- Returns: {boolean}

Returns `true` if the given `object` is a `Symbol`. Otherwise, returns `false`.

```
const util = require('util');

util.isSymbol(5);
// Returns: false
util.isSymbol('foo');
// Returns: false
util.isSymbol(Symbol('foo'));
// Returns: true
```

`util.isUndefined(object)`

Stability: 0 - Deprecated: Use `value === undefined` instead.

- `object` {any}
- Returns: {boolean}

Returns `true` if the given `object` is `undefined`. Otherwise, returns `false`.

```
const util = require('util');

const foo = undefined;
util.isUndefined(5);
// Returns: false
util.isUndefined(foo);
// Returns: true
util.isUndefined(null);
// Returns: false
```

`util.log(string)`

Stability: 0 - Deprecated: Use a third party module instead.

- `string` {string}

The `util.log()` method prints the given `string` to `stdout` with an included timestamp.

```
const util = require('util');

util.log('Timestamped message.');
```