Console

> *Stability: 2 - Stable*

The `console` module provides a simple debugging console that is similar to the JavaScript console mechanism provided by web browsers.

The module exports two specific components:

- A `Console` class with methods such as `console.log()`, `console.error()` and `console.warn()` that can be used to write to any Node.js stream.
- A global `console` instance configured to write to `process.stdout` and `process.stderr`. The global `console` can be used without calling `require('console')`.

Warning: The global console object's methods are neither consistently synchronous like the browser APIs they resemble, nor are they consistently asynchronous like all other Node.js streams. See the note on process I/O for more information.

Example using the global `console`:

```
console.log('hello world');
// Prints: hello world, to stdout
console.log('hello %s', 'world');
// Prints: hello world, to stdout
console.error(new Error('Whoops, something bad happened'));
// Prints error message and stack trace to stderr:
//   Error: Whoops, something bad happened
//     at [eval]:5:15
//     at Script.runInThisContext (node:vm:132:18)
//     at Object.runInThisContext (node:vm:309:38)
//     at node:internal/process/execution:77:19
//     at [eval]-wrapper:6:22
//     at evalScript (node:internal/process/execution:76:60)
//     at node:internal/main/eval_string:23:3

const name = 'Will Robinson';
console.warn(`Danger ${name}! Danger!`);
// Prints: Danger Will Robinson! Danger!, to stderr
```

Example using the `Console` class:

```
const out = getStreamSomehow();
const err = getStreamSomehow();
const myConsole = new console.Console(out, err);

myConsole.log('hello world');
// Prints: hello world, to out
myConsole.log('hello %s', 'world');
// Prints: hello world, to out
myConsole.error(new Error('Whoops, something bad happened'));
// Prints: [Error: Whoops, something bad happened], to err
```

```
const name = 'Will Robinson';
myConsole.warn(`Danger ${name}! Danger!`);
// Prints: Danger Will Robinson! Danger!, to err
```

Class: `Console`

The `Console` class can be used to create a simple logger with configurable output streams and can be accessed using either `require('console').Console` or `console.Console` (or their destructured counterparts):

```
const { Console } = require('console');
```

```
const { Console } = console;
```

`new Console(stdout[, stderr][, ignoreErrors])`

`new Console(options)`

- `options` {Object}
  - `stdout` {stream.Writable}
  - `stderr` {stream.Writable}
  - `ignoreErrors` {boolean} Ignore errors when writing to the underlying streams. **Default:** `true`.
  - `colorMode` {boolean|string} Set color support for this `Console` instance. Setting to `true` enables coloring while inspecting values. Setting to `false` disables coloring while inspecting values. Setting to `'auto'` makes color support depend on the value of the `isTTY` property and the value returned by `getColorDepth()` on the respective stream. This option can not be used, if `inspectOptions.colors` is set as well. **Default:** `'auto'`.
  - `inspectOptions` {Object} Specifies options that are passed along to `util.inspect()`.
  - `groupIndentation` {number} Set group indentation. **Default:** `2`.

Creates a new `Console` with one or two writable stream instances. `stdout` is a writable stream to print log or info output. `stderr` is used for warning or error output. If `stderr` is not provided, `stdout` is used for `stderr`.

```
const output = fs.createWriteStream('./stdout.log');
const errorOutput = fs.createWriteStream('./stderr.log');
// Custom simple logger
const logger = new Console({ stdout: output, stderr: errorOutput });
// use it like console
const count = 5;
logger.log('count: %d', count);
// In stdout.log: count 5
```

The global `console` is a special `Console` whose output is sent to `process.stdout` and `process.stderr`. It is equivalent to calling:

```
new Console({ stdout: process.stdout, stderr: process.stderr });
```

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

- `value` {any} The value tested for being truthy.
- `...message` {any} All arguments besides `value` are used as error message.

`console.assert()` writes a message if `value` is falsy or omitted. It only writes a message and does not otherwise affect execution. The output always starts with `"Assertion failed"`. If provided, `message` is formatted using `util.format()`.

If `value` is truthy, nothing happens.

```
console.assert(true, 'does nothing');

console.assert(false, 'Whoops %s work', 'didn\'t');
// Assertion failed: Whoops didn't work

console.assert();
// Assertion failed
```

`console.clear()`

When `stdout` is a TTY, calling `console.clear()` will attempt to clear the TTY. When `stdout` is not a TTY, this method does nothing.

The specific operation of `console.clear()` can vary across operating systems and terminal types. For most Linux operating systems, `console.clear()` operates similarly to the `clear` shell command. On Windows, `console.clear()` will clear only the output in the current terminal viewport for the Node.js binary.

`console.count([label])`

- `label` {string} The display label for the counter. **Default:** `'default'`.

Maintains an internal counter specific to `label` and outputs to `stdout` the number of times `console.count()` has been called with the given `label`.

```
> console.count()
default: 1
undefined
> console.count('default')
default: 2
undefined
> console.count('abc')
abc: 1
undefined
> console.count('xyz')
xyz: 1
undefined
> console.count('abc')
abc: 2
```

```
undefined
> console.count()
default: 3
undefined
>
```

`console.countReset([label])`

- `label` {string} The display label for the counter. **Default:** `'default'`.

Resets the internal counter specific to `label`.

```
> console.count('abc');
abc: 1
undefined
> console.countReset('abc');
undefined
> console.count('abc');
abc: 1
undefined
>
```

`console.debug(data[, ...args])`

- `data` {any}
- `...args` {any}

The `console.debug()` function is an alias for `console.log()`.

`console.dir(obj[, options])`

- `obj` {any}
- `options` {Object}
  - `showHidden` {boolean} If `true` then the object's non-enumerable and symbol properties will be shown too. **Default:** `false`.
  - `depth` {number} Tells `util.inspect()` how many times to recurse while formatting the object. This is useful for inspecting large complicated objects. To make it recurse indefinitely, pass `null`. **Default:** `2`.
  - `colors` {boolean} If `true`, then the output will be styled with ANSI color codes. Colors are customizable; see customizing `util.inspect()` colors. **Default:** `false`.

Uses `util.inspect()` on `obj` and prints the resulting string to `stdout`. This function bypasses any custom `inspect()` function defined on `obj`.

`console.dirxml(...data)`

- `...data` {any}

This method calls `console.log()` passing it the arguments received. This method does not produce any XML formatting.

`console.error([data][, ...args])`

- `data` {any}
- `...args` {any}

Prints to `stderr` with newline. Multiple arguments can be passed, with the first used as the primary message and all additional used as substitution values similar to printf(3) (the arguments are all passed to `util.format()`).

```
const code = 5;
console.error('error #%d', code);
// Prints: error #5, to stderr
console.error('error', code);
// Prints: error 5, to stderr
```

If formatting elements (e.g. `%d`) are not found in the first string then `util.inspect()` is called on each argument and the resulting string values are concatenated. See `util.format()` for more information.

`console.group([...label])`

- `...label` {any}

Increases indentation of subsequent lines by spaces for `groupIndentation` length.

If one or more `label`s are provided, those are printed first without the additional indentation.

`console.groupCollapsed()`

An alias for `console.group()`.

`console.groupEnd()`

Decreases indentation of subsequent lines by spaces for `groupIndentation` length.

`console.info([data][, ...args])`

- `data` {any}
- `...args` {any}

The `console.info()` function is an alias for `console.log()`.

`console.log([data][, ...args])`

- `data` {any}
- `...args` {any}

Prints to `stdout` with newline. Multiple arguments can be passed, with the first used as the primary message and all additional used as substitution values similar to printf(3) (the arguments are all passed to `util.format()`).

```
const count = 5;
console.log('count: %d', count);
// Prints: count: 5, to stdout
console.log('count:', count);
// Prints: count: 5, to stdout
```

See `util.format()` for more information.

`console.table(tabularData[, properties])`

- `tabularData` {any}
- `properties` {string[]} Alternate properties for constructing the table.

Try to construct a table with the columns of the properties of `tabularData` (or use `properties`) and rows of `tabularData` and log it. Falls back to just logging the argument if it can't be parsed as tabular.

```
// These can't be parsed as tabular data
console.table(Symbol());
// Symbol()

console.table(undefined);
// undefined

console.table([{ a: 1, b: 'Y' }, { a: 'Z', b: 2 }]);
// ┌─────────┬─────┬─────┐
// │ (index) │  a  │  b  │
// ├─────────┼─────┼─────┤
// │    0    │  1  │ 'Y' │
// │    1    │ 'Z' │  2  │
// └─────────┴─────┴─────┘

console.table([{ a: 1, b: 'Y' }, { a: 'Z', b: 2 }], ['a']);
// ┌─────────┬─────┐
// │ (index) │  a  │
// ├─────────┼─────┤
// │    0    │  1  │
// │    1    │ 'Z' │
// └─────────┴─────┘
```

`console.time([label])`

- `label` {string} **Default:** `'default'`

Starts a timer that can be used to compute the duration of an operation. Timers are identified by a unique `label`. Use the same `label` when calling `console.timeEnd()` to stop the timer and output the elapsed time in suitable time units to `stdout`. For example, if the elapsed time is 3869ms, `console.timeEnd()` displays "3.869s".

`console.timeEnd([label])`

- `label` {string} **Default:** `'default'`

Stops a timer that was previously started by calling `console.time()` and prints the result to `stdout`:

```
console.time('bunch-of-stuff');
// Do a bunch of stuff.
console.timeEnd('bunch-of-stuff');
// Prints: bunch-of-stuff: 225.438ms
```

`console.timeLog([label][, ...data])`

- `label` {string} **Default:** `'default'`
- `...data` {any}

For a timer that was previously started by calling `console.time()`, prints the elapsed time and other `data` arguments to `stdout`:

```
console.time('process');
const value = expensiveProcess1(); // Returns 42
console.timeLog('process', value);
// Prints "process: 365.227ms 42".
doExpensiveProcess2(value);
console.timeEnd('process');
```

`console.trace([message][, ...args])`

- `message` {any}
- `...args` {any}

Prints to `stderr` the string `'Trace: '`, followed by the `util.format()` formatted message and stack trace to the current position in the code.

```
console.trace('Show me');
// Prints: (stack trace will vary based on where trace is called)
//  Trace: Show me
//    at repl:2:9
//    at REPLServer.defaultEval (repl.js:248:27)
//    at bound (domain.js:287:14)
//    at REPLServer.runBound [as eval] (domain.js:300:12)
//    at REPLServer.<anonymous> (repl.js:412:12)
//    at emitOne (events.js:82:20)
//    at REPLServer.emit (events.js:169:7)
//    at REPLServer.Interface._onLine (readline.js:210:10)
//    at REPLServer.Interface._line (readline.js:549:8)
//    at REPLServer.Interface._ttyWrite (readline.js:826:14)
```

`console.warn([data][, ...args])`

- `data` {any}
- `...args` {any}

The `console.warn()` function is an alias for `console.error()`.

Inspector only methods

The following methods are exposed by the V8 engine in the general API but do not display anything unless used in conjunction with the inspector (`--inspect` flag).

`console.profile([label])`

- `label` {string}

This method does not display anything unless used in the inspector. The `console.profile()` method starts a JavaScript CPU profile with an optional label until `console.profileEnd()` is called. The profile is then added to the **Profile** panel of the inspector.

```
console.profile('MyLabel');
// Some code
console.profileEnd('MyLabel');
// Adds the profile 'MyLabel' to the Profiles panel of the inspector.
```

`console.profileEnd([label])`

- `label` {string}

This method does not display anything unless used in the inspector. Stops the current JavaScript CPU profiling session if one has been started and prints the report to the **Profiles** panel of the inspector. See `console.profile()` for an example.

If this method is called without a label, the most recently started profile is stopped.

`console.timeStamp([label])`

- `label` {string}

This method does not display anything unless used in the inspector. The `console.timeStamp()` method adds an event with the label `'label'` to the **Timeline** panel of the inspector.