/\*\*

\* 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`:

\*

\* ```js

\* 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:

\*

\* ```js

\* 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

\* ```

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

\*/

declare module 'console' {

import console = require('node:console');

export = console;

}

declare module 'node:console' {

import { InspectOptions } from 'node:util';

global {

// This needs to be global to avoid TS2403 in case lib.dom.d.ts is present in the same build

interface Console {

Console: console.ConsoleConstructor;

/\*\*

\* `console.assert()` writes a message if `value` is [falsy](https://developer.mozilla.org/en-US/docs/Glossary/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](https://developer.mozilla.org/en-US/docs/Glossary/Truthy), nothing happens.

\*

\* ```js

\* console.assert(true, 'does nothing');

\*

\* console.assert(false, 'Whoops %s work', 'didn\'t');

\* // Assertion failed: Whoops didn't work

\*

\* console.assert();

\* // Assertion failed

\* ```

\* @since v0.1.101

\* @param value The value tested for being truthy.

\* @param message All arguments besides `value` are used as error message.

\*/

assert(value: any, message?: string, ...optionalParams: any[]): void;

/\*\*

\* 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.

\* @since v8.3.0

\*/

clear(): void;

/\*\*

\* Maintains an internal counter specific to `label` and outputs to `stdout` the

\* number of times `console.count()` has been called with the given `label`.

\*

\* ```js

\* > 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

\* >

\* ```

\* @since v8.3.0

\* @param label The display label for the counter.

\*/

count(label?: string): void;

/\*\*

\* Resets the internal counter specific to `label`.

\*

\* ```js

\* > console.count('abc');

\* abc: 1

\* undefined

\* > console.countReset('abc');

\* undefined

\* > console.count('abc');

\* abc: 1

\* undefined

\* >

\* ```

\* @since v8.3.0

\* @param label The display label for the counter.

\*/

countReset(label?: string): void;

/\*\*

\* The `console.debug()` function is an alias for {@link log}.

\* @since v8.0.0

\*/

debug(message?: any, ...optionalParams: any[]): void;

/\*\*

\* Uses `util.inspect()` on `obj` and prints the resulting string to `stdout`.

\* This function bypasses any custom `inspect()` function defined on `obj`.

\* @since v0.1.101

\*/

dir(obj: any, options?: InspectOptions): void;

/\*\*

\* This method calls `console.log()` passing it the arguments received.

\* This method does not produce any XML formatting.

\* @since v8.0.0

\*/

dirxml(...data: any[]): void;

/\*\*

\* 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)`](http://man7.org/linux/man-pages/man3/printf.3.html) (the arguments are all passed to `util.format()`).

\*

\* ```js

\* 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.

\* @since v0.1.100

\*/

error(message?: any, ...optionalParams: any[]): void;

/\*\*

\* 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.

\* @since v8.5.0

\*/

group(...label: any[]): void;

/\*\*

\* An alias for {@link group}.

\* @since v8.5.0

\*/

groupCollapsed(...label: any[]): void;

/\*\*

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

\* @since v8.5.0

\*/

groupEnd(): void;

/\*\*

\* The `console.info()` function is an alias for {@link log}.

\* @since v0.1.100

\*/

info(message?: any, ...optionalParams: any[]): void;

/\*\*

\* 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)`](http://man7.org/linux/man-pages/man3/printf.3.html) (the arguments are all passed to `util.format()`).

\*

\* ```js

\* 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.

\* @since v0.1.100

\*/

log(message?: any, ...optionalParams: any[]): void;

/\*\*

\* 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.

\*

\* ```js

\* // 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' │

\* // └─────────┴─────┘

\* ```

\* @since v10.0.0

\* @param properties Alternate properties for constructing the table.

\*/

table(tabularData: any, properties?: ReadonlyArray<string>): void;

/\*\*

\* 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 {@link 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".

\* @since v0.1.104

\*/

time(label?: string): void;

/\*\*

\* Stops a timer that was previously started by calling {@link time} and

\* prints the result to `stdout`:

\*

\* ```js

\* console.time('100-elements');

\* for (let i = 0; i < 100; i++) {}

\* console.timeEnd('100-elements');

\* // prints 100-elements: 225.438ms

\* ```

\* @since v0.1.104

\*/

timeEnd(label?: string): void;

/\*\*

\* For a timer that was previously started by calling {@link time}, prints

\* the elapsed time and other `data` arguments to `stdout`:

\*

\* ```js

\* console.time('process');

\* const value = expensiveProcess1(); // Returns 42

\* console.timeLog('process', value);

\* // Prints "process: 365.227ms 42".

\* doExpensiveProcess2(value);

\* console.timeEnd('process');

\* ```

\* @since v10.7.0

\*/

timeLog(label?: string, ...data: any[]): void;

/\*\*

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

\*

\* ```js

\* 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)

\* ```

\* @since v0.1.104

\*/

trace(message?: any, ...optionalParams: any[]): void;

/\*\*

\* The `console.warn()` function is an alias for {@link error}.

\* @since v0.1.100

\*/

warn(message?: any, ...optionalParams: any[]): void;

// --- Inspector mode only ---

/\*\*

\* This method does not display anything unless used in the inspector.

\* Starts a JavaScript CPU profile with an optional label.

\*/

profile(label?: string): void;

/\*\*

\* 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.

\*/

profileEnd(label?: string): void;

/\*\*

\* This method does not display anything unless used in the inspector.

\* Adds an event with the label `label` to the Timeline panel of the inspector.

\*/

timeStamp(label?: string): void;

}

/\*\*

\* 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`:

\*

\* ```js

\* 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:

\*

\* ```js

\* 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

\* ```

\* @see [source](https://github.com/nodejs/node/blob/v16.4.2/lib/console.js)

\*/

namespace console {

interface ConsoleConstructorOptions {

stdout: NodeJS.WritableStream;

stderr?: NodeJS.WritableStream | undefined;

ignoreErrors?: boolean | undefined;

colorMode?: boolean | 'auto' | undefined;

inspectOptions?: InspectOptions | undefined;

/\*\*

\* Set group indentation

\* @default 2

\*/

groupIndentation?: number | undefined;

}

interface ConsoleConstructor {

prototype: Console;

new (stdout: NodeJS.WritableStream, stderr?: NodeJS.WritableStream, ignoreErrors?: boolean): Console;

new (options: ConsoleConstructorOptions): Console;

}

}

var console: Console;

}

export = globalThis.console;

}