Global Objects

Node.js global objects are global in nature and they are available in all modules.

We do not need to include these objects in our application, rather we can use them directly.

These objects are modules, functions, strings and object itself as explained below.

# \_\_filename

The \_\_filename represents the filename of the code being executed.

This is the resolved absolute path of this code file. For a main program, this is not necessarily the same filename used in the command line. The value inside a module is the path to that module file.

**Example**

Create a js file named main.js with the following code −

// Let's try to print the value of \_\_filename  
console.log( \_\_filename );

Now run the main.js to see the result −

$ node main.js

Based on the location of your program, it will print the main file name as follows −

/web/com/1427091028\_21099/main.js

# \_\_dirname

The \_\_dirname represents the name of the directory that the currently executing script resides in.

**Example**

Create a js file named main.js with the following code −

// Let's try to print the value of \_\_dirname

console.log( \_\_dirname );

Now run the main.js to see the result −

$ node main.js

Based on the location of your program, it will print current directory name as follows −

/web/com/1427091028\_21099

# setTimeout(cb, ms)

The setTimeout(cb, ms) global function is used to run callback cb after at least ms milliseconds.

The actual delay depends on external factors like OS timer granularity and system load. A timer cannot span more than 24.8 days.

This function returns an opaque value that represents the timer which can be used to clear the timer.

**Example**

Create a js file named main.js with the following code −

function printHello() {  
 console.log( "Hello, World!");  
}

// Now call above function after 2 seconds  
setTimeout(printHello, 2000);

Now run the main.js to see the result −

$ node main.js

Verify the output is printed after a little delay.

Hello, World!

# clearTimeout(t)

The clearTimeout(t) global function is used to stop a timer that was previously created with setTimeout(). Here t is the timer returned by the setTimeout() function.

**Example**

Create a js file named main.js with the following code −

function printHello() {  
 console.log( "Hello, World!");  
}

// Now call above function after 2 seconds  
var t = setTimeout(printHello, 2000);

// Now clear the timer  
clearTimeout(t);

Now run the main.js to see the result −

$ node main.js

Verify the output where you will not find anything printed.

# setInterval(cb, ms)

The setInterval(cb, ms) global function is used to run callback cb repeatedly after at least ms milliseconds. The actual delay depends on external factors like OS timer granularity and system load. A timer cannot span more than 24.8 days.

This function returns an opaque value that represents the timer which can be used to clear the timer using the function clearInterval(t).

**Example**

Create a js file named main.js with the following code −

function printHello() {  
 console.log( "Hello, World!");  
}

// Now call above function after 2 seconds  
setInterval(printHello, 2000);

Now run the main.js to see the result −

$ node main.js

The above program will execute printHello() after every 2 second. Due to system limitation.

# Global Objects

The following table provides a list of other objects which we use frequently in our applications. For a more detail, you can refer to the official documentation.

|  |  |
| --- | --- |
| Sr. No. | Module Name & Description |
| 1 | Console  Used to print information on stdout and stderr. |
| 2 | Process  Used to get information on current process. Provides multiple events related to process activities. |

Console

Node.js console is a global object and is used to print different levels of messages to stdout and stderr.

FYI (wikipedia):

Die **Standard-Datenströme** (englisch: *standard streams*) sind drei [Datenströme](https://de.wikipedia.org/wiki/Datenstrom) für die [Ein- und Ausgabe](https://de.wikipedia.org/wiki/Eingabe_und_Ausgabe) im Betriebssystem [Unix](https://de.wikipedia.org/wiki/Unix) oder in verwandten [Betriebssystemen](https://de.wikipedia.org/wiki/Betriebssystem). Sie werden auch von der [C-Standard-Bibliothek](https://de.wikipedia.org/wiki/C-Standard-Bibliothek) unterstützt. Viele Programme verwenden automatisch die Standardein- bzw. Standardausgabe, wenn auf der [Kommandozeile](https://de.wikipedia.org/wiki/Kommandozeile) keine [Dateien](https://de.wikipedia.org/wiki/Datei) für die Ein- oder Ausgabe angegeben werden.

# **Standardeingabe (stdin)**

Über die Standardeingabe können Daten in ein Programm eingelesen werden. Normalerweise ist sie mit der Tastatur verbunden, d. h. Programme empfangen die Benutzereingaben über die Standardeingabe. Unter Unix entspricht der Standardeingabe die Gerätedatei */dev/stdin*, der Datei-Deskriptor hat die Nummer 0.

Beispiel: Hier wird die Standardeingabe für das Programm *less* aus der Datei *eingaben.txt* gelesen.

$ less < eingaben.txt

## **Standardausgabe (stdout)[**[**Bearbeiten**](https://de.wikipedia.org/w/index.php?title=Standard-Datenstr%C3%B6me&veaction=edit&section=2)**|**[**Quelltext bearbeiten**](https://de.wikipedia.org/w/index.php?title=Standard-Datenstr%C3%B6me&action=edit&section=2)**]**

Über die Standardausgabe kann ein Programm Daten ausgeben. Normalerweise ist sie mit dem Monitor verbunden, d. h. Programme senden Ausgabetexte über die Standardausgabe an den Benutzer. Unter Unix entspricht der Standardausgabe die Gerätedatei /dev/stdout, der Datei-Deskriptor hat die Nummer 1.

Beispiel: Hier wird die Standardausgabe des Programms find in die Datei ausgaben.txt umgeleitet.

$ find . -name '\*.html' > ausgaben.txt

gleichbedeutend:

$ find . -name '\*.html' 1> ausgaben.txt

## **Standardfehlerausgabe (stderr)[**[**Bearbeiten**](https://de.wikipedia.org/w/index.php?title=Standard-Datenstr%C3%B6me&veaction=edit&section=3)**|**[**Quelltext bearbeiten**](https://de.wikipedia.org/w/index.php?title=Standard-Datenstr%C3%B6me&action=edit&section=3)]

Die Standardfehlerausgabe ist ein zweiter Ausgabedatenstrom, der dazu gedacht ist, Fehler- und Statusmeldungen auszugeben. Normalerweise ist er ebenfalls mit dem Monitor verbunden; allerdings kann er getrennt von der Standardausgabe umgeleitet werden, so dass Fehlermeldungen nicht mit den ausgegebenen Nutzdaten vermischt werden. Unter Unix entspricht der Standardfehlerausgabe die Gerätedatei /dev/stderr, der Datei-Deskriptor hat die Nummer 2.

Beispiel 1: Hier werden die Fehlermeldungen des Programms find in die Datei fehlermeldungen.txt umgeleitet, während die Standardausgabe über eine [Pipe](https://de.wikipedia.org/wiki/Pipe_(Informatik)) an das Programm less weitergeleitet wird.

$ find . -name '\*.html' 2> fehlermeldungen.txt | less

In Node.js

There are built-in methods to be used for printing informational, warning, and error messages.

It is used in synchronous way when the destination is a file or a terminal and in asynchronous way when the destination is a pipe.

# Console Methods

Following is a list of methods available with the console global object.

|  |  |
| --- | --- |
| Sr.No. | Method & Description |
| 1 | **console.log([data][, ...])**  Prints to stdout with newline.  This function can take multiple arguments in a printf()-like way. |
| 2 | **console.info([data][, ...])**  Prints to stdout with newline.  This function can take multiple arguments in a printf()-like way. |
| 3 | **console.error([data][, ...])**  Prints to stderr with newline.  This function can take multiple arguments in a printf()-like way. |
| 4 | **console.warn([data][, ...])**  Prints to stderr with newline.  This function can take multiple arguments in a printf()-like way |
| 5 | **console.dir(obj[, options])**  Uses util.inspect on obj and prints resulting string to stdout. |
| 6 | **console.time(label)**  Mark a time. |
| 7 | **console.timeEnd(label)**  Finish timer, record output. |
| 8 | **console.trace(message[, ...])**  Print to stderr 'Trace :',  followed by the formatted message and stack trace to the current position. |
| 9 | **console.assert(value[, message][, ...])**  Similar to assert.ok(),  but the error message is formatted as util.format(message...).  FYI: The Node.js assert.ok() method tests if a given expression is true or not. If the expression evaluates to 0, or false, an assertion failure is being caused, and the program is terminated. |

**Example**

Let us create a js file named main.js with the following code −

console.info("Program Started");  
var counter = 10;

console.log("Counter: %d", counter);  
console.time("Getting data");

//  
// Do some processing here...  
//

console.timeEnd('Getting data');  
console.info("Program Ended")

Now run the main.js to see the result −

node main.js

Verify the Output.

Program Started  
Counter: 10  
Getting data: 0ms  
Program Ended

# Process

The process object is a global object and can be accessed from anywhere. There are several methods available in a process object.

Process Events

The process object is an instance of EventEmitter and emits the following events –

|  |  |
| --- | --- |
| Sr.No. | Event & Description |
| 1 | **exit**  Emitted when the process is about to exit.  There is no way to prevent the exiting of the event loop at this point,  and once all exit listeners have finished running, the process will exit. |
| 2 | **beforeExit**  This event is emitted when node empties its event loop and has nothing else to schedule.  Normally, the node exits when there is no work scheduled, but a listener for 'beforeExit' can make asynchronous calls, and cause the node to continue. |
| 3 | **uncaughtException**  Emitted when an exception bubbles all the way back to the event loop.  If a listener is added for this exception,  the default action (which is to print a stack trace and exit) will not occur. |
| 4 | **Signal** **Events**  Emitted when the processes receives a signal (system notification) such as SIGINT, SIGHUP, etc. FYI:  <https://de.wikipedia.org/wiki/Signal_(Unix)> SIGHUP - Hangup detected on controlling terminal or death of controlling process  SIGINT - Interrupt from keyboard; interactive attention signal. |

**Example**

Create a js file named main.js with the following code for listening for exit event −

process.on('exit', function(code) {  
 // Following code will never execute.  
 setTimeout(function() {  
 console.log("This will not run");  
 }, 0);

console.log('About to exit with code:', code);

});

console.log("Program Ended");

Now run the main.js to see the result −

$ node main.js

Verify the Output.

Program Ended

About to exit with code: 0

# Exit Codes

Node normally exits with a 0 status code when no more async operations are pending. There are other exit codes which are described below –

|  |  |
| --- | --- |
| **Code** | **Name & Description** |
| 1 | **Uncaught Fatal Exception**  There was an uncaught exception, and it was not handled by a domain or an uncaughtException event handler. |
| 2 | **Unused**  reserved by Bash for built in misuse. |
| 3 | **Internal JavaScript Parse Error**  The JavaScript source code internal in Node's bootstrapping process caused a parse error.  This is extremely rare, and generally can only happen during the development of Node itself. |
| 4 | **Internal JavaScript Evaluation Failure**  The JavaScript source code internal in Node's bootstrapping process failed to return a function value when evaluated.  This is extremely rare, and generally can only happen during the development of Node itself. |
| 5 | **Fatal Error**  There was a fatal unrecoverable error in V8.  Typically, a message will be printed to stderr with the prefix FATAL ERROR. |
| 6 | **Non-function Internal Exception Handler**  There was an uncaught exception, but the internal fatal exception handler function was somehow set to a non-function, and could not be called. |
| 7 | **Internal Exception Handler Run-Time Failure**  There was an uncaught exception, and the internal fatal exception handler function itself threw an error while attempting to handle it. |
| 8 | **Unused** |
| 9 | **Invalid Argument**  Either an unknown option was specified, or an option requiring a value was provided without a value. |
| 10 | **Internal JavaScript Run-Time Failure**  The JavaScript source code internal in Node's bootstrapping process threw an error when the bootstrapping function was called.  This is extremely rare, and generally can only happen during the development of Node itself. |
| 11 | **Invalid Debug Argument**  The --debug and/or --debug-brk options were set,  but an invalid port number was chosen. |
| 12 | **Signal Exits**  If Node receives a fatal signal such as SIGKILL or SIGHUP,  then its exit code will be 128 plus the value of the signal code.  This is a standard Unix practice, since exit codes are defined to be 7-bit integers, and signal exits set the high-order bit, and then contain the value of the signal code. |

# Process Properties

Process provides many useful properties to get better control over system interactions.

|  |  |
| --- | --- |
| **Sr.No.** | **Property& Description** |
| 1 | **stdout**  A Writable Stream to stdout. |
| 2 | **stderr**  A Writable Stream to stderr. |
| 3 | **stdin**  A Writable Stream to stdin. |
| 4 | **argv**  An array containing the command line arguments.  The first element will be 'node',  the second element will be the name of the JavaScript file.  The next elements will be any additional command line arguments. |
| 5 | **execPath**  This is the absolute pathname of the executable that started the process. |
| 6 | **execArgv**  This is the set of node-specific command line options from the executable that started the process. |
| 7 | **env**  An object containing the user environment. |
| 8 | **exitCode**  A number which will be the process exit code,  when the process either exits gracefully,  or is exited via process.exit() without specifying a code. |
| 9 | **version**  A compiled-in property that exposes NODE\_VERSION. |
| 10 | **versions**  A property exposing the version strings of node and its dependencies. |
| 11 | **config**  An Object containing the JavaScript representation of the configure options  that were used to compile the current node executable.  This is the same as the "config.gypi" file that was produced when running the ./configure script. |
| 12 | **pid**  The PID (process id) of the process. |
| 13 | **title**  Getter/setter to set what is displayed in 'ps' (process status). |
| 14 | **arch**  What processor architecture you're running on: 'arm', 'ia32', or 'x64'. |
| 15 | **platform**  What platform you're running on: 'darwin', 'freebsd', 'linux', 'sunos' or 'win32' |
| 16 | **mainModule**  Alternate way to retrieve require.main.  The difference is that if the main module changes at runtime,  require.main might still refer to the original main module in modules that were required before the change occurred.  Generally it's safe to assume that the two refer to the same module. |

**Example**

Create a js file named main.js with the following code −

// Printing to console  
process.stdout.write("Hello World!" + "\n");

// Reading passed parameter  
process.argv.forEach(function(val, index, array) {  
 console.log(index + ': ' + val);  
});

// Getting executable path  
console.log(process.execPath);

// Platform Information   
console.log(process.platform);

Now run the main.js to see the result −

$ node main.js

Verify the Output while running your program on Linux machine −

Hello World!  
0: node  
1: /web/com/1427106219\_25089/main.js  
/usr/bin/node  
linux

# Methods Reference

Process provides many useful methods to get better control over system interactions.

|  |  |
| --- | --- |
| Sr.No. | Method & Description |
| 1 | **abort()**  It causes the node to emit an abort.  It causes the node to exit and generate a core file. |
| 2 | **chdir(directory)**  Changes the current working directory of the process or throws an exception if that fails. |
| 3 | **cwd()**  Returns the current working directory of the process. |
| 4 | **exit([code])**  Ends the process with the specified code.  If omitted, exit uses the 'success' code 0. |
| 5 | **getgid()**  Gets the group identity of the process.  This is the numerical group id, not the group name.  This function is available only POSIX platforms (i.e. not Windows, Android). |
| 6 | **setgid(id)**  Sets the group identity of the process. (See setgid(2)).  It accepts either a numerical ID or a groupname string.  If a groupname is specified, this method blocks while resolving it to a numerical ID.  This function is available only POSIX platforms (i.e. not Windows, Android). |
| 7 | **getuid()**  Gets the user identity of the process.  This is the numerical id, not the username.  This function is only available on POSIX platforms (i.e. not Windows, Android). |
| 8 | **setuid(id)**  Sets the user identity of the process (See setgid(2)).  It accepts either a numerical ID or a username string.  If a username is specified, this method blocks while resolving it to a numerical ID.  This function is available only POSIX platforms (i.e. not Windows, Android). |
| 9 | **getgroups()**  Returns an array with the supplementary group IDs.  POSIX (portable operating system interface) leaves it unspecified  if the effective group ID is included, but node.js ensures it always is.  This function is available only on POSIX platforms (i.e. not Windows, Android). |
| 10 | **setgroups(groups)**  Sets the supplementary group IDs.  This is a privileged operation, which implies that you have to be at the root or have the CAP\_SETGID capability.  This function is available only on POSIX platforms (i.e. not Windows, Android). |
| 11 | **initgroups(user, extra\_group)**  Reads /etc/group and initializes the group access list, using all the groups of which the user is a member.  This is a privileged operation, which implies that you have to be at the root or have the CAP\_SETGID capability.  This function is available only on POSIX platforms (i.e. not Windows, Android). |
| 12 | **kill(pid[, signal])**  Send a signal to a process.  pid is the process id and signal is the string describing the signal to send.  Signal names are strings like 'SIGINT' or 'SIGHUP'. If omitted, the signal will be 'SIGTERM'. |
| 13 | **memoryUsage()**  Returns an object describing the memory usage of the Node process measured in bytes. |
| 14 | **nextTick(callback)**  Once the current event loop turn runs to completion, call the callback function. |
| 15 | **umask([mask])**  Sets or reads the process's file mode creation mask.  Child processes inherit the mask from the parent process.  Returns the old mask if mask argument is given, otherwise returns the current mask. |
| 16 | **uptime()**  Number of seconds Node has been running. |
| 17 | **hrtime()**  Returns the current high-resolution real time in a [seconds, nanoseconds] tuple Array.  It is relative to an arbitrary time in the past.  It is not related to the time of day and therefore not subject to clock drift.  The primary use is for measuring performance between intervals. |

**Example**

Create a js file named main.js with the following code −

// Print the current directory  
console.log('Current directory: ' + process.cwd());

// Print the process version  
console.log('Current version: ' + process.version);

// Print the memory usage  
console.log(process.memoryUsage());

Now run the main.js to see the result −

$ node main.js

Verify the Output while running your program on Linux machine −

Current directory: /web/com/1427106219\_25089  
Current version: v0.10.33  
{ rss: 11505664, heapTotal: 4083456, heapUsed: 2157704 }