```
// Node.js CheatSheet.
// Download the Node.js source code or a pre-built installer for your platform, and start
developing today.
// Download: http://nodejs.org/download/
// More: http://nodejs.org/api/all.html


// 0. Synopsis.
// http://nodejs.org/api/synopsis.html


var http = require('http');

// An example of a web server written with Node which responds with 'Hello World'.
// To run the server, put the code into a file called example.js and execute it with the
node program.
http.createServer(function (request, response) {
  response.writeHead(200, {'Content-Type': 'text/plain'});
  response.end('Hello World\n');
}).listen(8124);

console.log('Server running at http://127.0.0.1:8124/');


// 1. Global Objects.
// http://nodejs.org/api/globals.html


// In browsers, the top-level scope is the global scope.
// That means that in browsers if you're in the global scope var something will define a
global variable.
// In Node this is different. The top-level scope is not the global scope; var something
inside a Node module will be local to that module.

__filename;  // The filename of the code being executed. (absolute path)
__dirname;   // The name of the directory that the currently executing script resides in.
(absolute path)
module;      // A reference to the current module. In particular module.exports is used
for defining what a module exports and makes available through require().
exports;     // A reference to the module.exports that is shorter to type.
process;     // The process object is a global object and can be accessed from anywhere.
It is an instance of EventEmitter.
Buffer;      // The Buffer class is a global type for dealing with binary data directly.


// 2. Console.
// http://nodejs.org/api/console.html


console.log([data], [...]);             // Prints to stdout with newline.
console.info([data], [...]);            // Same as console.log.
console.error([data], [...]);           // Same as console.log but prints to stderr.
console.warn([data], [...]);            // Same as console.error.
console.dir(obj);                       // Uses util.inspect on obj and prints resulting
string to stdout.
console.time(label);                    // Mark a time.
console.timeEnd(label);                 // Finish timer, record output.
console.trace(label);                   // Print a stack trace to stderr of the current
position.
console.assert(expression, [message]);  // Same as assert.ok() where if the expression
evaluates as false throw an AssertionError with message.


// 3. Timers.
// http://nodejs.org/api/timers.html


setTimeout(callback, delay, [arg], [...]);   // To schedule execution of a one-time
callback after delay milliseconds. Optionally you can also pass arguments to the callback.
```

```
clearTimeout(t);                              // Stop a timer that was previously created
with setTimeout().
setInterval(callback, delay, [arg], [...]);  // To schedule the repeated execution of
callback every delay milliseconds. Optionally you can also pass arguments to the callback.
clearInterval(t);                             // Stop a timer that was previously created
with setInterval().
setImmediate(callback, [arg], [...]);        // To schedule the "immediate" execution of
callback after I/O events callbacks and before setTimeout and setInterval.
clearImmediate(immediateObject);              // Stop a timer that was previously created
with setImmediate().

unref();  // Allow you to create a timer that is active but if it is the only item left in
the event loop, node won't keep the program running.
ref();    // If you had previously unref()d a timer you can call ref() to explicitly
request the timer hold the program open.


// 4. Modules.
// http://nodejs.org/api/modules.html


var module = require('./module.js');    // Loads the module module.js in the same
directory.
module.require('./another_module.js');  // load another_module as if require() was called
from the module itself.

module.id;        // The identifier for the module. Typically this is the fully resolved
filename.
module.filename;  // The fully resolved filename to the module.
module.loaded;    // Whether or not the module is done loading, or is in the process of
loading.
module.parent;    // The module that required this one.
module.children;  // The module objects required by this one.

exports.area = function (r) {
  return 3.14 * r * r;
};

// If you want the root of your module's export to be a function (such as a constructor)
// or if you want to export a complete object in one assignment instead of building it one
property at a time,
// assign it to module.exports instead of exports.
module.exports = function(width) {
  return {
    area: function() {
      return width * width;
    }
  };
}


// 5. Process.
// http://nodejs.org/api/process.html


process.on('exit', function(code) {});             // Emitted when the process is about
to exit
process.on('uncaughtException', function(err) {});  // Emitted when an exception bubbles
all the way back to the event loop. (should not be used)

process.stdout;           // A writable stream to stdout.
process.stderr;           // A writable stream to stderr.
process.stdin;            // A readable stream for stdin.

process.argv;             // An array containing the command line arguments.
process.env;              // An object containing the user environment.
process.execPath;         // This is the absolute pathname of the executable that started
the process.
process.execArgv;         // This is the set of node-specific command line options from
```

```
the executable that started the process.

process.arch;             // What processor architecture you're running on: 'arm', 'ia32',
or 'x64'.
process.config;           // An Object containing the JavaScript representation of the
configure options that were used to compile the current node executable.
process.pid;              // The PID of the process.
process.platform;         // What platform you're running on: 'darwin', 'freebsd',
'linux', 'sunos' or 'win32'.
process.title;            // Getter/setter to set what is displayed in 'ps'.
process.version;          // A compiled-in property that exposes NODE_VERSION.
process.versions;         // A property exposing version strings of node and its
dependencies.

process.abort();          // This causes node to emit an abort. This will cause node to
exit and generate a core file.
process.chdir(dir);       // Changes the current working directory of the process or
throws an exception if that fails.
process.cwd();            // Returns the current working directory of the process.
process.exit([code]);     // Ends the process with the specified code. If omitted, exit
uses the 'success' code 0.
process.getgid();         // Gets the group identity of the process.
process.setgid(id);       // Sets the group identity of the process.
process.getuid();         // Gets the user identity of the process.
process.setuid(id);       // Sets the user identity of the process.
process.getgroups();      // Returns an array with the supplementary group IDs.
process.setgroups(grps);  // Sets the supplementary group IDs.

process.initgroups(user, extra_grp);  // Reads /etc/group and initializes the group access
list, using all groups of which the user is a member.
process.kill(pid, [signal]);          // Send a signal to a process. pid is the process id
and signal is the string describing the signal to send.
process.memoryUsage();                // Returns an object describing the memory usage of
the Node process measured in bytes.
process.nextTick(callback);           // On the next loop around the event loop call this
callback.
process.maxTickDepth;                 // Callbacks passed to process.nextTick will usually
be called at the end of the current flow of execution, and are thus approximately as fast
as calling a function synchronously.
process.umask([mask]);                // Sets or reads the process's file mode creation
mask.
process.uptime();                     // Number of seconds Node has been running.
process.hrtime();                     // Returns the current high-resolution real time in
a [seconds, nanoseconds] tuple Array.


// 6. Child Process.
// Node provides a tri-directional popen facility through the child_process module.
// It is possible to stream data through a child's stdin, stdout, and stderr in a fully
non-blocking way.
// http://nodejs.org/api/child_process.html


ChildProcess;                                         // Class. ChildProcess is an
EventEmitter.

child.stdin;                                          // A Writable Stream that
represents the child process's stdin
child.stdout;                                         // A Readable Stream that
represents the child process's stdout
child.stderr;                                         // A Readable Stream that
represents the child process's stderr.
child.pid;                                            // The PID of the child
process
child.connected;                                      // If .connected is false,
it is no longer possible to send messages
child.kill([signal]);                                 // Send a signal to the
child process
child.send(message, [sendHandle]);                    // When using
```

```
child_process.fork() you can write to the child using child.send(message, [sendHandle])
and messages are received by a 'message' event on the child.
child.disconnect();                                          // Close the IPC channel
between parent and child, allowing the child to exit gracefully once there are no other
connections keeping it alive.
child_process.spawn(command, [args], [options]);             // Launches a new process
with the given command, with command line arguments in args. If omitted, args defaults to
an empty Array.
child_process.exec(command, [options], callback);            // Runs a command in a shell
and buffers the output.
child_process.execFile(file, [args], [options], [callback]);  // Runs a command in a shell
and buffers the output.
child_process.fork(modulePath, [args], [options]);           // This is a special case of
the spawn() functionality for spawning Node processes. In addition to having all the
methods in a normal ChildProcess instance, the returned object has a communication channel
built-in.


// 7. Util.
// These functions are in the module 'util'. Use require('util') to access them.
// http://nodejs.org/api/util.html


util.format(format, [...]);    // Returns a formatted string using the first argument as a
printf-like format. (%s, %d, %j)
util.debug(string);            // A synchronous output function. Will block the process
and output string immediately to stderr.
util.error([...]);             // Same as util.debug() except this will output all
arguments immediately to stderr.
util.puts([...]);              // A synchronous output function. Will block the process
and output all arguments to stdout with newlines after each argument.
util.print([...]);             // A synchronous output function. Will block the process,
cast each argument to a string then output to stdout. (no newlines)
util.log(string);              // Output with timestamp on stdout.
util.inspect(object, [opts]);  // Return a string representation of object, which is
useful for debugging. (options: showHidden, depth, colors, customInspect)
util.isArray(object);          // Returns true if the given "object" is an Array. false
otherwise.
util.isRegExp(object);         // Returns true if the given "object" is a RegExp. false
otherwise.
util.isDate(object);           // Returns true if the given "object" is a Date. false
otherwise.
util.isError(object);          // Returns true if the given "object" is an Error. false
otherwise.

util.inherits(constructor, superConstructor);  // Inherit the prototype methods from one
constructor into another.


// 8. Events.
// All objects which emit events are instances of events.EventEmitter. You can access this
module by doing: require("events");
// To access the EventEmitter class, require('events').EventEmitter.
// All EventEmitters emit the event 'newListener' when new listeners are added and
'removeListener' when a listener is removed.
// http://nodejs.org/api/events.html


emitter.addListener(event, listener);        // Adds a listener to the end of the
listeners array for the specified event.
emitter.on(event, listener);                 // Same as emitter.addListener().
emitter.once(event, listener);               // Adds a one time listener for the event.
This listener is invoked only the next time the event is fired, after which it is removed.
emitter.removeListener(event, listener);     // Remove a listener from the listener array
for the specified event.
emitter.removeAllListeners([event]);         // Removes all listeners, or those of the
specified event.
emitter.setMaxListeners(n);                  // By default EventEmitters will print a
warning if more than 10 listeners are added for a particular event.
```

```
emitter.listeners(event);                    // Returns an array of listeners for the
specified event.
emitter.emit(event, [arg1], [arg2], [...]);  // Execute each of the listeners in order
with the supplied arguments. Returns true if event had listeners, false otherwise.

EventEmitter.listenerCount(emitter, event);  // Return the number of listeners for a given
event.


// 9. Stream.
// A stream is an abstract interface implemented by various objects in Node. For example a
request to an HTTP server is a stream, as is stdout.
// Streams are readable, writable, or both. All streams are instances of EventEmitter.
// http://nodejs.org/api/stream.html


// The Readable stream interface is the abstraction for a source of data that you are
reading from.
// In other words, data comes out of a Readable stream.
// A Readable stream will not start emitting data until you indicate that you are ready to
receive it.
// Examples of readable streams include: http responses on the client, http requests on
the server, fs read streams
// zlib streams, crypto streams, tcp sockets, child process stdout and stderr,
process.stdin.

var readable = getReadableStreamSomehow();

readable.on('readable', function() {});   // When a chunk of data can be read from the
stream, it will emit a 'readable' event.
readable.on('data', function(chunk) {});  // If you attach a data event listener, then it
will switch the stream into flowing mode, and data will be passed to your handler as soon
as it is available.
readable.on('end', function() {});        // This event fires when there will be no more
data to read.
readable.on('close', function() {});      // Emitted when the underlying resource (for
example, the backing file descriptor) has been closed. Not all streams will emit this.
readable.on('error', function() {});      // Emitted if there was an error receiving data.

// The read() method pulls some data out of the internal buffer and returns it. If there
is no data available, then it will return null.
// This method should only be called in non-flowing mode. In flowing-mode, this method is
called automatically until the internal buffer is drained.
readable.read([size]);

readable.setEncoding(encoding);           // Call this function to cause the stream to
return strings of the specified encoding instead of Buffer objects.
readable.resume();                        // This method will cause the readable stream to
resume emitting data events.
readable.pause();                         // This method will cause a stream in flowing-
mode to stop emitting data events.
readable.pipe(destination, [options]);    // This method pulls all the data out of a
readable stream, and writes it to the supplied destination, automatically managing the
flow so that the destination is not overwhelmed by a fast readable stream.
readable.unpipe([destination]);           // This method will remove the hooks set up for
a previous pipe() call. If the destination is not specified, then all pipes are removed.
readable.unshift(chunk);                  // This is useful in certain cases where a
stream is being consumed by a parser, which needs to "un-consume" some data that it has
optimistically pulled out of the source, so that the stream can be passed on to some other
party.


// The Writable stream interface is an abstraction for a destination that you are writing
data to.
// Examples of writable streams include: http requests on the client, http responses on
the server, fs write streams,
// zlib streams, crypto streams, tcp sockets, child process stdin, process.stdout,
process.stderr.
```

```
var writer = getWritableStreamSomehow();

writable.write(chunk, [encoding], [callback]);  // This method writes some data to the
underlying system, and calls the supplied callback once the data has been fully handled.
writer.once('drain', write);                    // If a writable.write(chunk) call returns
false, then the drain event will indicate when it is appropriate to begin writing more
data to the stream.

writable.end([chunk], [encoding], [callback]);  // Call this method when no more data will
be written to the stream.
writer.on('finish', function() {});             // When the end() method has been called,
and all data has been flushed to the underlying system, this event is emitted.
writer.on('pipe', function(src) {});            // This is emitted whenever the pipe()
method is called on a readable stream, adding this writable to its set of destinations.
writer.on('unpipe', function(src) {});          // This is emitted whenever the unpipe()
method is called on a readable stream, removing this writable from its set of
destinations.
writer.on('error', function(src) {});           // Emitted if there was an error when
writing or piping data.


// Duplex streams are streams that implement both the Readable and Writable interfaces.
See above for usage.
// Examples of Duplex streams include: tcp sockets, zlib streams, crypto streams.

// Transform streams are Duplex streams where the output is in some way computed from the
input. They implement both the Readable and Writable interfaces. See above for usage.
// Examples of Transform streams include: zlib streams, crypto streams.


// 10. File System.
// To use this module do require('fs').
// All the methods have asynchronous and synchronous forms.
// http://nodejs.org/api/fs.html


fs.rename(oldPath, newPath, callback);  // Asynchronous rename. No arguments other than a
possible exception are given to the completion callback.Asynchronous ftruncate. No
arguments other than a possible exception are given to the completion callback.
fs.renameSync(oldPath, newPath);        // Synchronous rename.

fs.ftruncate(fd, len, callback);        // Asynchronous ftruncate. No arguments other than
a possible exception are given to the completion callback.
fs.ftruncateSync(fd, len);              // Synchronous ftruncate.
fs.truncate(path, len, callback);       // Asynchronous truncate. No arguments other than
a possible exception are given to the completion callback.
fs.truncateSync(path, len);             // Synchronous truncate.

fs.chown(path, uid, gid, callback);     // Asynchronous chown. No arguments other than a
possible exception are given to the completion callback.
fs.chownSync(path, uid, gid);           // Synchronous chown.
fs.fchown(fd, uid, gid, callback);      // Asynchronous fchown. No arguments other than a
possible exception are given to the completion callback.
fs.fchownSync(fd, uid, gid);            // Synchronous fchown.
fs.lchown(path, uid, gid, callback);    // Asynchronous lchown. No arguments other than a
possible exception are given to the completion callback.
fs.lchownSync(path, uid, gid);          // Synchronous lchown.

fs.chmod(path, mode, callback);         // Asynchronous chmod. No arguments other than a
possible exception are given to the completion callback.
fs.chmodSync(path, mode);               // Synchronous chmod.
fs.fchmod(fd, mode, callback);          // Asynchronous fchmod. No arguments other than a
possible exception are given to the completion callback.
fs.fchmodSync(fd, mode);                // Synchronous fchmod.
fs.lchmod(path, mode, callback);        // Asynchronous lchmod. No arguments other than a
possible exception are given to the completion callback.
fs.lchmodSync(path, mode);              // Synchronous lchmod.

fs.stat(path, callback);                // Asynchronous stat. The callback gets two
```

```
arguments (err, stats) where stats is a fs.Stats object.
fs.statSync(path);                      // Synchronous stat. Returns an instance of
fs.Stats.
fs.lstat(path, callback);               // Asynchronous lstat. The callback gets two
arguments (err, stats) where stats is a fs.Stats object. lstat() is identical to stat(),
except that if path is a symbolic link, then the link itself is stat-ed, not the file that
it refers to.
fs.lstatSync(path);                     // Synchronous lstat. Returns an instance of
fs.Stats.
fs.fstat(fd, callback);                 // Asynchronous fstat. The callback gets two
arguments (err, stats) where stats is a fs.Stats object. fstat() is identical to stat(),
except that the file to be stat-ed is specified by the file descriptor fd.
fs.fstatSync(fd);                       // Synchronous fstat. Returns an instance of
fs.Stats.

fs.link(srcpath, dstpath, callback);            // Asynchronous link. No arguments other
than a possible exception are given to the completion callback.
fs.linkSync(srcpath, dstpath);                  // Synchronous link.
fs.symlink(srcpath, dstpath, [type], callback);  // Asynchronous symlink. No arguments
other than a possible exception are given to the completion callback. The type argument
can be set to 'dir', 'file', or 'junction' (default is 'file') and is only available on
Windows (ignored on other platforms)
fs.symlinkSync(srcpath, dstpath, [type]);       // Synchronous symlink.
fs.readlink(path, callback);                    // Asynchronous readlink. The callback
gets two arguments (err, linkString).
fs.readlinkSync(path);                          // Synchronous readlink. Returns the
symbolic link's string value.
fs.unlink(path, callback);                      // Asynchronous unlink. No arguments
other than a possible exception are given to the completion callback.
fs.unlinkSync(path);                            // Synchronous unlink.

fs.realpath(path, [cache], callback);     // Asynchronous realpath. The callback gets two
arguments (err, resolvedPath).
fs.realpathSync(path, [cache]);           // Synchronous realpath. Returns the resolved
path.

fs.rmdir(path, callback);                 // Asynchronous rmdir. No arguments other than a
possible exception are given to the completion callback.
fs.rmdirSync(path);                       // Synchronous rmdir.
fs.mkdir(path, [mode], callback);         // Asynchronous mkdir. No arguments other than a
possible exception are given to the completion callback. mode defaults to 0777.
fs.mkdirSync(path, [mode]);               // Synchronous mkdir.
fs.readdir(path, callback);               // Asynchronous readdir. Reads the contents of a
directory. The callback gets two arguments (err, files) where files is an array of the
names of the files in the directory excluding '.' and '..'.
fs.readdirSync(path);                     // Synchronous readdir. Returns an array of
filenames excluding '.' and '..'.
fs.close(fd, callback);                   // Asynchronous close. No arguments other than a
possible exception are given to the completion callback.
fs.closeSync(fd);                         // Synchronous close.
fs.open(path, flags, [mode], callback);   // Asynchronous file open.
fs.openSync(path, flags, [mode]);         // Synchronous version of fs.open().
fs.utimes(path, atime, mtime, callback);  // Change file timestamps of the file referenced
by the supplied path.
fs.utimesSync(path, atime, mtime);        // Synchronous version of fs.utimes().
fs.futimes(fd, atime, mtime, callback);   // Change the file timestamps of a file
referenced by the supplied file descriptor.
fs.futimesSync(fd, atime, mtime);         // Synchronous version of fs.futimes().
fs.fsync(fd, callback);                   // Asynchronous fsync. No arguments other than a
possible exception are given to the completion callback.
fs.fsyncSync(fd);                         // Synchronous fsync.

fs.write(fd, buffer, offset, length, position, callback);  // Write buffer to the file
specified by fd.
fs.writeSync(fd, buffer, offset, length, position);        // Synchronous version of
fs.write(). Returns the number of bytes written.
fs.read(fd, buffer, offset, length, position, callback);   // Read data from the file
specified by fd.
fs.readSync(fd, buffer, offset, length, position);         // Synchronous version of
```

```
fs.read. Returns the number of bytesRead.
fs.readFile(filename, [options], callback);                // Asynchronously reads the
entire contents of a file.
fs.readFileSync(filename, [options]);                      // Synchronous version of
fs.readFile. Returns the contents of the filename. If the encoding option is specified
then this function returns a string. Otherwise it returns a buffer.

fs.writeFile(filename, data, [options], callback);   // Asynchronously writes data to a
file, replacing the file if it already exists. data can be a string or a buffer.
fs.writeFileSync(filename, data, [options]);         // The synchronous version of
fs.writeFile.
fs.appendFile(filename, data, [options], callback);  // Asynchronously append data to a
file, creating the file if it not yet exists. data can be a string or a buffer.
fs.appendFileSync(filename, data, [options]);        // The synchronous version of
fs.appendFile.
fs.watch(filename, [options], [listener]);           // Watch for changes on filename,
where filename is either a file or a directory. The returned object is a fs.FSWatcher. The
listener callback gets two arguments (event, filename). event is either 'rename' or
'change', and filename is the name of the file which triggered the event.
fs.exists(path, callback);                           // Test whether or not the given path
exists by checking with the file system. Then call the callback argument with either true
or false. (should not be used)
fs.existsSync(path);                                 // Synchronous version of fs.exists.
(should not be used)

// fs.Stats: objects returned from fs.stat(), fs.lstat() and fs.fstat() and their
synchronous counterparts are of this type.
stats.isFile();
stats.isDirectory()
stats.isBlockDevice()
stats.isCharacterDevice()
stats.isSymbolicLink()  // (only valid with fs.lstat())
stats.isFIFO()
stats.isSocket()

fs.createReadStream(path, [options]);   // Returns a new ReadStream object.
fs.createWriteStream(path, [options]);  // Returns a new WriteStream object.


// 11. Path.
// Use require('path') to use this module.
// This module contains utilities for handling and transforming file paths.
// Almost all these methods perform only string transformations.
// The file system is not consulted to check whether paths are valid.
// http://nodejs.org/api/fs.html


path.normalize(p);                    // Normalize a string path, taking care of '..' and
'.' parts.
path.join([path1], [path2], [...]);   // Join all arguments together and normalize the
resulting path.
path.resolve([from ...], to);         // Resolves 'to' to an absolute path.
path.relative(from, to);              // Solve the relative path from 'from' to 'to'.
path.dirname(p);                      // Return the directory name of a path. Similar to
the Unix dirname command.
path.basename(p, [ext]);              // Return the last portion of a path. Similar to the
Unix basename command.
path.extname(p);                      // Return the extension of the path, from the last
'.' to end of string in the last portion of the path.

path.sep;                             // The platform-specific file separator. '\\' or
'/'.
path.delimiter;                       // The platform-specific path delimiter, ';' or ':'.


// 12. HTTP.
// To use the HTTP server and client one must require('http').
// http://nodejs.org/api/http.html
```

```
http.STATUS_CODES;                                          // A collection of all the
standard HTTP response status codes, and the short description of each.
http.request(options, [callback]);                          // This function allows one
to transparently issue requests.
http.get(options, [callback]);                              // Set the method to GET
and calls req.end() automatically.

server = http.createServer([requestListener]);              // Returns a new web server
object. The requestListener is a function which is automatically added to the 'request'
event.
server.listen(port, [hostname], [backlog], [callback]);     // Begin accepting
connections on the specified port and hostname.
server.listen(path, [callback]);                            // Start a UNIX socket
server listening for connections on the given path.
server.listen(handle, [callback]);                          // The handle object can be
set to either a server or socket (anything with an underlying _handle member), or a {fd:
<n>} object.
server.close([callback]);                                   // Stops the server from
accepting new connections.
server.setTimeout(msecs, callback);                         // Sets the timeout value
for sockets, and emits a 'timeout' event on the Server object, passing the socket as an
argument, if a timeout occurs.

server.maxHeadersCount;  // Limits maximum incoming headers count, equal to 1000 by
default. If set to 0 - no limit will be applied.
server.timeout;          // The number of milliseconds of inactivity before a socket is
presumed to have timed out.

server.on('request', function (request, response) { });     // Emitted each time there
is a request.
server.on('connection', function (socket) { });             // When a new TCP stream is
established.
server.on('close', function () { });                        // Emitted when the server
closes.
server.on('checkContinue', function (request, response) { });  // Emitted each time a
request with an http Expect: 100-continue is received.
server.on('connect', function (request, socket, head) { });    // Emitted each time a
client requests a http CONNECT method.
server.on('upgrade', function (request, socket, head) { });    // Emitted each time a
client requests a http upgrade.
server.on('clientError', function (exception, socket) { });    // If a client connection
emits an 'error' event - it will forwarded here.

request.write(chunk, [encoding]);                           // Sends a chunk of the
body.
request.end([data], [encoding]);                            // Finishes sending the
request. If any parts of the body are unsent, it will flush them to the stream.
request.abort();                                            // Aborts a request.
request.setTimeout(timeout, [callback]);                    // Once a socket is
assigned to this request and is connected socket.setTimeout() will be called.
request.setNoDelay([noDelay]);                              // Once a socket is
assigned to this request and is connected socket.setNoDelay() will be called.
request.setSocketKeepAlive([enable], [initialDelay]);       // Once a socket is
assigned to this request and is connected socket.setKeepAlive() will be called.

request.on('response', function(response) { });             // Emitted when a response
is received to this request. This event is emitted only once.
request.on('socket', function(socket) { });                 // Emitted after a socket
is assigned to this request.
request.on('connect', function(response, socket, head) { });   // Emitted each time a
server responds to a request with a CONNECT method. If this event isn't being listened
for, clients receiving a CONNECT method will have their connections closed.
request.on('upgrade', function(response, socket, head) { });   // Emitted each time a
server responds to a request with an upgrade. If this event isn't being listened for,
clients receiving an upgrade header will have their connections closed.
request.on('continue', function() { });                     // Emitted when the server
sends a '100 Continue' HTTP response, usually because the request contained 'Expect: 100-
continue'. This is an instruction that the client should send the request body.
```

```
response.write(chunk, [encoding]);                                // This sends a chunk of
the response body. If this merthod is called and response.writeHead() has not been called,
it will switch to implicit header mode and flush the implicit headers.
response.writeContinue();                                         // Sends a HTTP/1.1 100
Continue message to the client, indicating that the request body should be sent.
response.writeHead(statusCode, [reasonPhrase], [headers]);     // Sends a response header
to the request.
response.setTimeout(msecs, callback);                             // Sets the Socket's
timeout value to msecs. If a callback is provided, then it is added as a listener on the
'timeout' event on the response object.
response.setHeader(name, value);                                  // Sets a single header
value for implicit headers. If this header already exists in the to-be-sent headers, its
value will be replaced. Use an array of strings here if you need to send multiple headers
with the same name.
response.getHeader(name);                                         // Reads out a header
that's already been queued but not sent to the client. Note that the name is case
insensitive.
response.removeHeader(name);                                      // Removes a header that's
queued for implicit sending.
response.addTrailers(headers);                                    // This method adds HTTP
trailing headers (a header but at the end of the message) to the response.
response.end([data], [encoding]);                                 // This method signals to
the server that all of the response headers and body have been sent; that server should
consider this message complete. The method, response.end(), MUST be called on each
response.

response.statusCode;                                              // When using implicit
headers (not calling response.writeHead() explicitly), this property controls the status
code that will be sent to the client when the headers get flushed.
response.headersSent;                                             // Boolean (read-only).
True if headers were sent, false otherwise.
response.sendDate;                                                // When true, the Date
header will be automatically generated and sent in the response if it is not already
present in the headers. Defaults to true.

response.on('close', function () { });  // Indicates that the underlying connection was
terminated before response.end() was called or able to flush.
response.on('finish', function() { });  // Emitted when the response has been sent.

message.httpVersion;                    // In case of server request, the HTTP version
sent by the client. In the case of client response, the HTTP version of the connected-to
server.
message.headers;                        // The request/response headers object.
message.trailers;                       // The request/response trailers object. Only
populated after the 'end' event.
message.method;                         // The request method as a string. Read only.
Example: 'GET', 'DELETE'.
message.url;                            // Request URL string. This contains only the URL
that is present in the actual HTTP request.
message.statusCode;                     // The 3-digit HTTP response status code. E.G.
404.
message.socket;                         // The net.Socket object associated with the
connection.

message.setTimeout(msecs, callback);    // Calls message.connection.setTimeout(msecs,
callback).


// 13. URL.
// This module has utilities for URL resolution and parsing. Call require('url') to use
it.
// http://nodejs.org/api/url.html


url.parse(urlStr, [parseQueryString], [slashesDenoteHost]);  // Take a URL string, and
return an object.
url.format(urlObj);                                           // Take a parsed URL object,
and return a formatted URL string.
```

```
url.resolve(from, to);                                  // Take a base URL, and a
href URL, and resolve them as a browser would for an anchor tag.


// 14. Query String.
// This module provides utilities for dealing with query strings. Call
require('querystring') to use it.
// http://nodejs.org/api/querystring.html


querystring.stringify(obj, [sep], [eq]);         // Serialize an object to a query string.
Optionally override the default separator ('&') and assignment ('=') characters.
querystring.parse(str, [sep], [eq], [options]);  // Deserialize a query string to an
object. Optionally override the default separator ('&') and assignment ('=') characters.


// 15. Assert.
// This module is used for writing unit tests for your applications, you can access it
with require('assert').
// http://nodejs.org/api/assert.html


assert.fail(actual, expected, message, operator);     // Throws an exception that displays
the values for actual and expected separated by the provided operator.
assert(value, message); assert.ok(value, [message]);  // Tests if value is truthy, it is
equivalent to assert.equal(true, !!value, message);
assert.equal(actual, expected, [message]);            // Tests shallow, coercive equality
with the equal comparison operator ( == ).
assert.notEqual(actual, expected, [message]);         // Tests shallow, coercive non-
equality with the not equal comparison operator ( != ).
assert.deepEqual(actual, expected, [message]);        // Tests for deep equality.
assert.notDeepEqual(actual, expected, [message]);     // Tests for any deep inequality.
assert.strictEqual(actual, expected, [message]);      // Tests strict equality, as
determined by the strict equality operator ( === )
assert.notStrictEqual(actual, expected, [message]);   // Tests strict non-equality, as
determined by the strict not equal operator ( !== )
assert.throws(block, [error], [message]);             // Expects block to throw an error.
error can be constructor, RegExp or validation function.
assert.doesNotThrow(block, [message]);                // Expects block not to throw an
error, see assert.throws for details.
assert.ifError(value);                                // Tests if value is not a false
value, throws if it is a true value. Useful when testing the first argument, error in
callbacks.


// 16. OS.
// Provides a few basic operating-system related utility functions.
// Use require('os') to access this module.
// http://nodejs.org/api/os.html


os.tmpdir();             // Returns the operating system's default directory for temp
files.
os.endianness();         // Returns the endianness of the CPU. Possible values are "BE" or
"LE".
os.hostname();           // Returns the hostname of the operating system.
os.type();               // Returns the operating system name.
os.platform();           // Returns the operating system platform.
os.arch();               // Returns the operating system CPU architecture.
os.release();            // Returns the operating system release.
os.uptime();             // Returns the system uptime in seconds.
os.loadavg();            // Returns an array containing the 1, 5, and 15 minute load
averages.
os.totalmem();           // Returns the total amount of system memory in bytes.
os.freemem();            // Returns the amount of free system memory in bytes.
os.cpus();               // Returns an array of objects containing information about each
CPU/core installed: model, speed (in MHz), and times (an object containing the number of
milliseconds the CPU/core spent in: user, nice, sys, idle, and irq).
os.networkInterfaces();  // Get a list of network interfaces.
```

```
os.EOL;                  // A constant defining the appropriate End-of-line marker for the
operating system.


// 17. Buffer.
// Buffer is used to dealing with binary data
// Buffer is similar to an array of integers but corresponds to a raw memory allocation
outside the V8 heap
// http://nodejs.org/api/buffer.html


new Buffer(size);                                                  // Allocates a new
buffer of size octets.
new Buffer(array);                                                 // Allocates a new
buffer using an array of octets.
new Buffer(str, [encoding]);                                       // Allocates a new
buffer containing the given str. encoding defaults to 'utf8'.

Buffer.isEncoding(encoding);                                       // Returns true if the
encoding is a valid encoding argument, or false otherwise.
Buffer.isBuffer(obj);                                              // Tests if obj is a
Buffer
Buffer.concat(list, [totalLength]);                                // Returns a buffer
which is the result of concatenating all the buffers in the list together.
Buffer.byteLength(string, [encoding]);                             // Gives the actual
byte length of a string.

buf.write(string, [offset], [length], [encoding]);                 // Writes string to
the buffer at offset using the given encoding
buf.toString([encoding], [start], [end]);                          // Decodes and returns
a string from buffer data encoded with encoding (defaults to 'utf8') beginning at start
(defaults to 0) and ending at end (defaults to buffer.length).
buf.toJSON();                                                      // Returns a JSON-
representation of the Buffer instance, which is identical to the output for JSON Arrays
buf.copy(targetBuffer, [targetStart], [sourceStart], [sourceEnd]);  // Does copy between
buffers. The source and target regions can be overlapped
buf.slice([start], [end]);                                         // Returns a new
buffer which references the same memory as the old, but offset and cropped by the start
(defaults to 0) and end (defaults to buffer.length) indexes. Negative indexes start from
the end of the buffer.
buf.fill(value, [offset], [end]);                                  // Fills the buffer
with the specified value
buf[index];                                                        // Get and set the
octet at index
buf.length;                                                        // The size of the
buffer in bytes, Note that this is not necessarily the size of the contents

buffer.INSPECT_MAX_BYTES;                                          // How many bytes will
be returned when buffer.inspect() is called. This can be overridden by user modules.
```