REPL

> *Stability: 2 - Stable*

The `repl` module provides a Read-Eval-Print-Loop (REPL) implementation that is available both as a standalone program or includible in other applications. It can be accessed using:

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

Design and features

The `repl` module exports the `repl.REPLServer` class. While running, instances of `repl.REPLServer` will accept individual lines of user input, evaluate those according to a user-defined evaluation function, then output the result. Input and output may be from `stdin` and `stdout`, respectively, or may be connected to any Node.js stream.

Instances of `repl.REPLServer` support automatic completion of inputs, completion preview, simplistic Emacs-style line editing, multi-line inputs, ZSH-like reverse-i-search, ZSH-like substring-based history search, ANSI-styled output, saving and restoring current REPL session state, error recovery, and customizable evaluation functions. Terminals that do not support ANSI styles and Emacs-style line editing automatically fall back to a limited feature set.

Commands and special keys

The following special commands are supported by all REPL instances:

- `.break` : When in the process of inputting a multi-line expression, enter the `.break` command (or press `Ctrl+C`) to abort further input or processing of that expression.
- `.clear` : Resets the REPL `context` to an empty object and clears any multi-line expression being input.
- `.exit` : Close the I/O stream, causing the REPL to exit.
- `.help` : Show this list of special commands.
- `.save` : Save the current REPL session to a file: `> .save ./file/to/save.js`
- `.load` : Load a file into the current REPL session. `> .load ./file/to/load.js`
- `.editor` : Enter editor mode (`Ctrl+D` to finish, `Ctrl+C` to cancel).

```
> .editor
// Entering editor mode (^D to finish, ^C to cancel)
function welcome(name) {
  return `Hello ${name}!`;
}

welcome('Node.js User');

// ^D
'Hello Node.js User!'
>
```

The following key combinations in the REPL have these special effects:

- `Ctrl+C`: When pressed once, has the same effect as the `.break` command. When pressed twice on a blank line, has the same effect as the `.exit` command.
- `Ctrl+D`: Has the same effect as the `.exit` command.
- `Tab`: When pressed on a blank line, displays global and local (scope) variables. When pressed while entering other input, displays relevant autocompletion options.

For key bindings related to the reverse-i-search, see `reverse-i-search`. For all other key bindings, see TTY keybindings.

Default evaluation

By default, all instances of `repl.REPLServer` use an evaluation function that evaluates JavaScript expressions and provides access to Node.js built-in modules. This default behavior can be overridden by passing in an alternative evaluation function when the `repl.REPLServer` instance is created.

JavaScript expressions

The default evaluator supports direct evaluation of JavaScript expressions:

```
> 1 + 1
2
> const m = 2
undefined
> m + 1
3
```

Unless otherwise scoped within blocks or functions, variables declared either implicitly or using the `const`, `let`, or `var` keywords are declared at the global scope.

Global and local scope

The default evaluator provides access to any variables that exist in the global scope. It is possible to expose a variable to the REPL explicitly by assigning it to the `context` object associated with each `REPLServer`:

```
const repl = require('repl');
const msg = 'message';

repl.start('> ').context.m = msg;
```

Properties in the `context` object appear as local within the REPL:

```
$ node repl_test.js
> m
'message'
```

Context properties are not read-only by default. To specify read-only globals, context properties must be defined using `Object.defineProperty()`:

```
const repl = require('repl');
const msg = 'message';
```

```
const r = repl.start('> ');
Object.defineProperty(r.context, 'm', {
  configurable: false,
  enumerable: true,
  value: msg
});
```

Accessing core Node.js modules

The default evaluator will automatically load Node.js core modules into the REPL environment when used. For instance, unless otherwise declared as a global or scoped variable, the input `fs` will be evaluated on-demand as `global.fs = require('fs')`.

```
> fs.createReadStream('./some/file');
```

Global uncaught exceptions

The REPL uses the `domain` module to catch all uncaught exceptions for that REPL session.

This use of the `domain` module in the REPL has these side effects:

- Uncaught exceptions only emit the `'uncaughtException'` event in the standalone REPL. Adding a listener for this event in a REPL within another Node.js program results in `ERR_INVALID_REPL_INPUT`.

  ```
  const r = repl.start();

  r.write('process.on("uncaughtException", () => console.log("Foobar"));\n');
  // Output stream includes:
  //   TypeError [ERR_INVALID_REPL_INPUT]: Listeners for `uncaughtException`
  //   cannot be used in the REPL

  r.close();
  ```

- Trying to use `process.setUncaughtExceptionCaptureCallback()` throws an `ERR_DOMAIN_CANNOT_SET_UNCAUGHT_EXCEPTION_CAPTURE` error.

Assignment of the `_` (underscore) variable

The default evaluator will, by default, assign the result of the most recently evaluated expression to the special variable `_` (underscore). Explicitly setting `_` to a value will disable this behavior.

```
> [ 'a', 'b', 'c' ]
[ 'a', 'b', 'c' ]
> _.length
3
> _ += 1
Expression assignment to _ now disabled.
4
> 1 + 1
2
```

```
> _
4
```

Similarly, `_error` will refer to the last seen error, if there was any. Explicitly setting `_error` to a value will disable this behavior.

```
> throw new Error('foo');
Error: foo
> _error.message
'foo'
```

`await` keyword

Support for the `await` keyword is enabled at the top level.

```
> await Promise.resolve(123)
123
> await Promise.reject(new Error('REPL await'))
Error: REPL await
    at repl:1:45
> const timeout = util.promisify(setTimeout);
undefined
> const old = Date.now(); await timeout(1000); console.log(Date.now() - old);
1002
undefined
```

One known limitation of using the `await` keyword in the REPL is that it will invalidate the lexical scoping of the `const` and `let` keywords.

For example:

```
> const m = await Promise.resolve(123)
undefined
> m
123
> const m = await Promise.resolve(234)
undefined
> m
234
```

`--no-experimental-repl-await` shall disable top-level await in REPL.

Reverse-i-search

The REPL supports bi-directional reverse-i-search similar to ZSH. It is triggered with `Ctrl+R` to search backward and `Ctrl+S` to search forwards.

Duplicated history entries will be skipped.

Entries are accepted as soon as any key is pressed that doesn't correspond with the reverse search. Cancelling is possible by pressing `Esc` or `Ctrl+C`.

Changing the direction immediately searches for the next entry in the expected direction from the current position on.

Custom evaluation functions

When a new `repl.REPLServer` is created, a custom evaluation function may be provided. This can be used, for instance, to implement fully customized REPL applications.

The following illustrates a hypothetical example of a REPL that performs translation of text from one language to another:

```
const repl = require('repl');
const { Translator } = require('translator');

const myTranslator = new Translator('en', 'fr');

function myEval(cmd, context, filename, callback) {
  callback(null, myTranslator.translate(cmd));
}

repl.start({ prompt: '> ', eval: myEval });
```

Recoverable errors

At the REPL prompt, pressing `Enter` sends the current line of input to the `eval` function. In order to support multi-line input, the `eval` function can return an instance of `repl.Recoverable` to the provided callback function:

```
function myEval(cmd, context, filename, callback) {
  let result;
  try {
    result = vm.runInThisContext(cmd);
  } catch (e) {
    if (isRecoverableError(e)) {
      return callback(new repl.Recoverable(e));
    }
  }
  callback(null, result);
}

function isRecoverableError(error) {
  if (error.name === 'SyntaxError') {
    return /^(Unexpected end of input|Unexpected token)/.test(error.message);
  }
  return false;
}
```

Customizing REPL output

By default, `repl.REPLServer` instances format output using the `util.inspect()` method before writing the output to the provided `Writable` stream (`process.stdout` by default). The `showProxy` inspection option is set to true by default and the `colors` option is set to true depending on the REPL's `useColors` option.

The `useColors` boolean option can be specified at construction to instruct the default writer to use ANSI style codes to colorize the output from the `util.inspect()` method.

If the REPL is run as standalone program, it is also possible to change the REPL's inspection defaults from inside the REPL by using the `inspect.replDefaults` property which mirrors the `defaultOptions` from `util.inspect()` .

```
> util.inspect.replDefaults.compact = false;
false
> [1]
[
  1
]
>
```

To fully customize the output of a `repl.REPLServer` instance pass in a new function for the `writer` option on construction. The following example, for instance, simply converts any input text to upper case:

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

const r = repl.start({ prompt: '> ', eval: myEval, writer: myWriter });

function myEval(cmd, context, filename, callback) {
  callback(null, cmd);
}

function myWriter(output) {
  return output.toUpperCase();
}
```

Class: `REPLServer`

- `options` {Object|string} See `repl.start()`
- Extends: {readline.Interface}

Instances of `repl.REPLServer` are created using the `repl.start()` method or directly using the JavaScript `new` keyword.

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

const options = { useColors: true };

const firstInstance = repl.start(options);
const secondInstance = new repl.REPLServer(options);
```

Event: `'exit'`

The `'exit'` event is emitted when the REPL is exited either by receiving the `.exit` command as input, the user pressing `Ctrl+C` twice to signal `SIGINT` , or by pressing `Ctrl+D` to signal `'end'` on the input stream. The listener

callback is invoked without any arguments.

```
replServer.on('exit', () => {
  console.log('Received "exit" event from repl!');
  process.exit();
});
```

Event: `'reset'`

The `'reset'` event is emitted when the REPL's context is reset. This occurs whenever the `.clear` command is received as input *unless* the REPL is using the default evaluator and the `repl.REPLServer` instance was created with the `useGlobal` option set to `true`. The listener callback will be called with a reference to the `context` object as the only argument.

This can be used primarily to re-initialize REPL context to some pre-defined state:

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

function initializeContext(context) {
  context.m = 'test';
}

const r = repl.start({ prompt: '> ' });
initializeContext(r.context);

r.on('reset', initializeContext);
```

When this code is executed, the global `'m'` variable can be modified but then reset to its initial value using the `.clear` command:

```
$ ./node example.js
> m
'test'
> m = 1
1
> m
1
> .clear
Clearing context...
> m
'test'
>
```

`replServer.defineCommand(keyword, cmd)`

- `keyword` {string} The command keyword (*without* a leading `.` character).
- `cmd` {Object|Function} The function to invoke when the command is processed.

The `replServer.defineCommand()` method is used to add new `.`-prefixed commands to the REPL instance. Such commands are invoked by typing a `.` followed by the `keyword`. The `cmd` is either a `Function` or an

`Object` with the following properties:

- `help` {string} Help text to be displayed when `.help` is entered (Optional).
- `action` {Function} The function to execute, optionally accepting a single string argument.

The following example shows two new commands added to the REPL instance:

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

const replServer = repl.start({ prompt: '> ' });
replServer.defineCommand('sayhello', {
  help: 'Say hello',
  action(name) {
    this.clearBufferedCommand();
    console.log(`Hello, ${name}!`);
    this.displayPrompt();
  }
});
replServer.defineCommand('saybye', function saybye() {
  console.log('Goodbye!');
  this.close();
});
```

The new commands can then be used from within the REPL instance:

```
> .sayhello Node.js User
Hello, Node.js User!
> .saybye
Goodbye!
```

`replServer.displayPrompt([preserveCursor])`

- `preserveCursor` {boolean}

The `replServer.displayPrompt()` method readies the REPL instance for input from the user, printing the configured `prompt` to a new line in the `output` and resuming the `input` to accept new input.

When multi-line input is being entered, an ellipsis is printed rather than the 'prompt'.

When `preserveCursor` is `true`, the cursor placement will not be reset to `0`.

The `replServer.displayPrompt` method is primarily intended to be called from within the action function for commands registered using the `replServer.defineCommand()` method.

`replServer.clearBufferedCommand()`

The `replServer.clearBufferedCommand()` method clears any command that has been buffered but not yet executed. This method is primarily intended to be called from within the action function for commands registered using the `replServer.defineCommand()` method.

`replServer.parseREPLKeyword(keyword[, rest])`

> *Stability: 0 - Deprecated.*

- `keyword` {string} the potential keyword to parse and execute
- `rest` {any} any parameters to the keyword command
- Returns: {boolean}

An internal method used to parse and execute `REPLServer` keywords. Returns `true` if `keyword` is a valid keyword, otherwise `false`.

`replServer.setupHistory(historyPath, callback)`

- `historyPath` {string} the path to the history file
- `callback` {Function} called when history writes are ready or upon error
  - `err` {Error}
  - `repl` {repl.REPLServer}

Initializes a history log file for the REPL instance. When executing the Node.js binary and using the command-line REPL, a history file is initialized by default. However, this is not the case when creating a REPL programmatically. Use this method to initialize a history log file when working with REPL instances programmatically.

`repl.builtinModules`

- {string[]}

A list of the names of all Node.js modules, e.g., `'http'`.

`repl.start([options])`

- `options` {Object|string}
  - `prompt` {string} The input prompt to display. **Default:** `'> '` (with a trailing space).
  - `input` {stream.Readable} The `Readable` stream from which REPL input will be read. **Default:** `process.stdin`.
  - `output` {stream.Writable} The `Writable` stream to which REPL output will be written. **Default:** `process.stdout`.
  - `terminal` {boolean} If `true`, specifies that the `output` should be treated as a TTY terminal. **Default:** checking the value of the `isTTY` property on the `output` stream upon instantiation.
  - `eval` {Function} The function to be used when evaluating each given line of input. **Default:** an async wrapper for the JavaScript `eval()` function. An `eval` function can error with `repl.Recoverable` to indicate the input was incomplete and prompt for additional lines.
  - `useColors` {boolean} If `true`, specifies that the default `writer` function should include ANSI color styling to REPL output. If a custom `writer` function is provided then this has no effect. **Default:** checking color support on the `output` stream if the REPL instance's `terminal` value is `true`.
  - `useGlobal` {boolean} If `true`, specifies that the default evaluation function will use the JavaScript `global` as the context as opposed to creating a new separate context for the REPL instance. The node CLI REPL sets this value to `true`. **Default:** `false`.
  - `ignoreUndefined` {boolean} If `true`, specifies that the default writer will not output the return value of a command if it evaluates to `undefined`. **Default:** `false`.
  - `writer` {Function} The function to invoke to format the output of each command before writing to `output`. **Default:** `util.inspect()`.

    - `completer` {Function} An optional function used for custom Tab auto completion. See `readline.InterfaceCompleter` for an example.
    - `replMode` {symbol} A flag that specifies whether the default evaluator executes all JavaScript commands in strict mode or default (sloppy) mode. Acceptable values are:
        - `repl.REPL_MODE_SLOPPY` to evaluate expressions in sloppy mode.
        - `repl.REPL_MODE_STRICT` to evaluate expressions in strict mode. This is equivalent to prefacing every repl statement with `'use strict'`.
    - `breakEvalOnSigint` {boolean} Stop evaluating the current piece of code when `SIGINT` is received, such as when `Ctrl+C` is pressed. This cannot be used together with a custom `eval` function. **Default:** `false`.
    - `preview` {boolean} Defines if the repl prints autocomplete and output previews or not. **Default:** `true` with the default eval function and `false` in case a custom eval function is used. If `terminal` is falsy, then there are no previews and the value of `preview` has no effect.
- Returns: {repl.REPLServer}

The `repl.start()` method creates and starts a `repl.REPLServer` instance.

If `options` is a string, then it specifies the input prompt:

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

// a Unix style prompt
repl.start('$ ');
```

The Node.js REPL

Node.js itself uses the `repl` module to provide its own interactive interface for executing JavaScript. This can be used by executing the Node.js binary without passing any arguments (or by passing the `-i` argument):

```
$ node
> const a = [1, 2, 3];
undefined
> a
[ 1, 2, 3 ]
> a.forEach((v) => {
...   console.log(v);
...   });
1
2
3
```

Environment variable options

Various behaviors of the Node.js REPL can be customized using the following environment variables:

- `NODE_REPL_HISTORY`: When a valid path is given, persistent REPL history will be saved to the specified file rather than `.node_repl_history` in the user's home directory. Setting this value to `''` (an empty string) will disable persistent REPL history. Whitespace will be trimmed from the value. On Windows

platforms environment variables with empty values are invalid so set this variable to one or more spaces to disable persistent REPL history.

- `NODE_REPL_HISTORY_SIZE`: Controls how many lines of history will be persisted if history is available. Must be a positive number. **Default:** `1000`.
- `NODE_REPL_MODE`: May be either `'sloppy'` or `'strict'`. **Default:** `'sloppy'`, which will allow non-strict mode code to be run.

Persistent history

By default, the Node.js REPL will persist history between `node` REPL sessions by saving inputs to a `.node_repl_history` file located in the user's home directory. This can be disabled by setting the environment variable `NODE_REPL_HISTORY=''`.

Using the Node.js REPL with advanced line-editors

For advanced line-editors, start Node.js with the environment variable `NODE_NO_READLINE=1`. This will start the main and debugger REPL in canonical terminal settings, which will allow use with `rlwrap`.

For example, the following can be added to a `.bashrc` file:

```
alias node="env NODE_NO_READLINE=1 rlwrap node"
```

Starting multiple REPL instances against a single running instance

It is possible to create and run multiple REPL instances against a single running instance of Node.js that share a single `global` object but have separate I/O interfaces.

The following example, for instance, provides separate REPLs on `stdin`, a Unix socket, and a TCP socket:

```
const net = require('net');
const repl = require('repl');
let connections = 0;

repl.start({
  prompt: 'Node.js via stdin> ',
  input: process.stdin,
  output: process.stdout
});

net.createServer((socket) => {
  connections += 1;
  repl.start({
    prompt: 'Node.js via Unix socket> ',
    input: socket,
    output: socket
  }).on('exit', () => {
    socket.end();
  });
}).listen('/tmp/node-repl-sock');

net.createServer((socket) => {
  connections += 1;
```

```
  repl.start({
    prompt: 'Node.js via TCP socket> ',
    input: socket,
    output: socket
  }).on('exit', () => {
    socket.end();
  });
}).listen(5001);
```

Running this application from the command line will start a REPL on stdin. Other REPL clients may connect through the Unix socket or TCP socket. `telnet`, for instance, is useful for connecting to TCP sockets, while `socat` can be used to connect to both Unix and TCP sockets.

By starting a REPL from a Unix socket-based server instead of stdin, it is possible to connect to a long-running Node.js process without restarting it.

For an example of running a "full-featured" (`terminal`) REPL over a `net.Server` and `net.Socket` instance, see: https://gist.github.com/TooTallNate/2209310.

For an example of running a REPL instance over `curl(1)`, see: https://gist.github.com/TooTallNate/2053342.