Commander.js

The complete solution for node.js command-line interfaces.

Read this in other languages: English | 简体中文

For information about terms used in this document see: terminology

Installation

```
npm install commander
```

Quick Start

You write code to describe your command line interface. Commander looks after parsing the arguments into options and command-arguments, displays usage errors for problems, and implements a help system.

Commander is strict and displays an error for unrecognised options. The two most used option types are a boolean option, and an option which takes its value from the following argument.

Example file: split.js

```
const { program } = require('commander');

program
  .option('--first')
  .option('-s, --separator <char>');

program.parse();

const options = program.opts();
const limit = options.first ? 1 : undefined;
console.log(program.args[0].split(options.separator, limit));
```

```
$ node split.js -s / --fits a/b/c
error: unknown option '--fits'
(Did you mean --first?)
$ node split.js -s / --first a/b/c
[ 'a' ]
```

Here is a more complete program using a subcommand and with descriptions for the help. In a multi-command program, you have an action handler for each command (or stand-alone executables for the commands).

Example file: string-util.js

```
const { Command } = require('commander');
const program = new Command();

program
  .name('string-util')
  .description('CLI to some JavaScript string utilities')
  .version('0.8.0');

program.command('split')
  .description('Split a string into substrings and display as an array')
  .argument('<string>', 'string to split')
  .option('--first', 'display just the first substring')
  .option('-s, --separator <char>', 'separator character', ',')
  .action((str, options) => {
    const limit = options.first ? 1 : undefined;
    console.log(str.split(options.separator, limit));
  });

program.parse();
```

```
$ node string-util.js help split
Usage: string-util split [options] <string>

Split a string into substrings and display as an array.

Arguments:
  string                  string to split

Options:
  --first                 display just the first substring
  -s, --separator <char>  separator character (default: ",")
  -h, --help              display help for command

$ node string-util.js split --separator=/ a/b/c
[ 'a', 'b', 'c' ]
```

More samples can be found in the examples directory.

Declaring *program* variable

Commander exports a global object which is convenient for quick programs. This is used in the examples in this README for brevity.

```
// CommonJS (.cjs)
const { program } = require('commander');
```

For larger programs which may use commander in multiple ways, including unit testing, it is better to create a local Command object to use.

```
// CommonJS (.cjs)
const { Command } = require('commander');
const program = new Command();
```

```
// ECMAScript (.mjs)
import { Command } from 'commander';
const program = new Command();
```

```
// TypeScript (.ts)
import { Command } from 'commander';
const program = new Command();
```

Options

Options are defined with the `.option()` method, also serving as documentation for the options. Each option can have a short flag (single character) and a long name, separated by a comma or space or vertical bar ('|').

The parsed options can be accessed by calling `.opts()` on a `Command` object, and are passed to the action handler.

Multi-word options such as "--template-engine" are camel-cased, becoming `program.opts().templateEngine` etc.

An option and its option-argument can be separated by a space, or combined into the same argument. The option-argument can follow the short option directly or follow an `=` for a long option.

```
serve -p 80
serve -p80
serve --port 80
serve --port=80
```

You can use `--` to indicate the end of the options, and any remaining arguments will be used without being interpreted.

By default options on the command line are not positional, and can be specified before or after other arguments.

There are additional related routines for when `.opts()` is not enough:

- `.optsWithGlobals()` returns merged local and global option values
- `.getOptionValue()` and `.setOptionValue()` work with a single option value
- `.getOptionValueSource()` and `.setOptionValueWithSource()` include where the option value came from

Common option types, boolean and value

The two most used option types are a boolean option, and an option which takes its value from the following argument (declared with angle brackets like `--expect <value>`). Both are `undefined` unless specified on command line.

Example file: [options-common.js]

```
program
  .option('-d, --debug', 'output extra debugging')
  .option('-s, --small', 'small pizza size')
  .option('-p, --pizza-type <type>', 'flavour of pizza');

program.parse(process.argv);

const options = program.opts();
if (options.debug) console.log(options);
console.log('pizza details:');
if (options.small) console.log('- small pizza size');
if (options.pizzaType) console.log(`- ${options.pizzaType}`);
```

```
$ pizza-options -p
error: option '-p, --pizza-type <type>' argument missing
$ pizza-options -d -s -p vegetarian
{ debug: true, small: true, pizzaType: 'vegetarian' }
pizza details:
- small pizza size
- vegetarian
$ pizza-options --pizza-type=cheese
pizza details:
- cheese
```

Multiple boolean short options may be combined together following the dash, and may be followed by a single short option taking a value. For example `-d -s -p cheese` may be written as `-ds -p cheese` or even `-dsp cheese`.

Options with an expected option-argument are greedy and will consume the following argument whatever the value. So `--id -xyz` reads `-xyz` as the option-argument.

`program.parse(arguments)` processes the arguments, leaving any args not consumed by the program options in the `program.args` array. The parameter is optional and defaults to `process.argv`.

Default option value

You can specify a default value for an option.

Example file: [options-defaults.js]

```
program
  .option('-c, --cheese <type>', 'add the specified type of cheese', 'blue');

program.parse();

console.log(`cheese: ${program.opts().cheese}`);
```

```
$ pizza-options
cheese: blue
$ pizza-options --cheese stilton
cheese: stilton
```

Other option types, negatable boolean and boolean|value

You can define a boolean option long name with a leading `no-` to set the option value to false when used. Defined alone this also makes the option true by default.

If you define `--foo` first, adding `--no-foo` does not change the default value from what it would otherwise be.

Example file: options-negatable.js

```
program
  .option('--no-sauce', 'Remove sauce')
  .option('--cheese <flavour>', 'cheese flavour', 'mozzarella')
  .option('--no-cheese', 'plain with no cheese')
  .parse();

const options = program.opts();
const sauceStr = options.sauce ? 'sauce' : 'no sauce';
const cheeseStr = (options.cheese === false) ? 'no cheese' : `${options.cheese} cheese
console.log(`You ordered a pizza with ${sauceStr} and ${cheeseStr}`);
```

```
$ pizza-options
You ordered a pizza with sauce and mozzarella cheese
$ pizza-options --sauce
error: unknown option '--sauce'
$ pizza-options --cheese=blue
You ordered a pizza with sauce and blue cheese
$ pizza-options --no-sauce --no-cheese
You ordered a pizza with no sauce and no cheese
```

You can specify an option which may be used as a boolean option but may optionally take an option-argument (declared with square brackets like `--optional [value]`).

Example file: options-boolean-or-value.js

```
program
  .option('-c, --cheese [type]', 'Add cheese with optional type');

program.parse(process.argv);

const options = program.opts();
if (options.cheese === undefined) console.log('no cheese');
else if (options.cheese === true) console.log('add cheese');
else console.log(`add cheese type ${options.cheese}`);
```

```
$ pizza-options
no cheese
$ pizza-options --cheese
add cheese
$ pizza-options --cheese mozzarella
add cheese type mozzarella
```

Options with an optional option-argument are not greedy and will ignore arguments starting with a dash. So `id` behaves as a boolean option for `--id -5`, but you can use a combined form if needed like `--id=-5`.

For information about possible ambiguous cases, see options taking varying arguments.

Required option

You may specify a required (mandatory) option using `.requiredOption()`. The option must have a value after parsing, usually specified on the command line, or perhaps from a default value (say from environment). The method is otherwise the same as `.option()` in format, taking flags and description, and optional default value or custom processing.

Example file: options-required.js

```
program
  .requiredOption('-c, --cheese <type>', 'pizza must have cheese');

program.parse();
```

```
$ pizza
error: required option '-c, --cheese <type>' not specified
```

Variadic option

You may make an option variadic by appending `...` to the value placeholder when declaring the option. On the command line you can then specify multiple option-arguments, and the parsed option value will be an array. The extra arguments are read until the first argument starting with a dash. The special argument `--` stops option processing entirely. If a value is specified in the same argument as the option then no further values are read.

Example file: options-variadic.js

```
program
  .option('-n, --number <numbers...>', 'specify numbers')
  .option('-l, --letter [letters...]', 'specify letters');

program.parse();

console.log('Options: ', program.opts());
console.log('Remaining arguments: ', program.args);
```

```
$ collect -n 1 2 3 --letter a b c
Options:  { number: [ '1', '2', '3' ], letter: [ 'a', 'b', 'c' ] }
Remaining arguments:  []
$ collect --letter=A -n80 operand
Options:  { number: [ '80' ], letter: [ 'A' ] }
Remaining arguments:  [ 'operand' ]
$ collect --letter -n 1 -n 2 3 -- operand
Options:  { number: [ '1', '2', '3' ], letter: true }
Remaining arguments:  [ 'operand' ]
```

For information about possible ambiguous cases, see options taking varying arguments.

Version option

The optional `version` method adds handling for displaying the command version. The default option flags are `-V` and `--version`, and when present the command prints the version number and exits.

```
program.version('0.0.1');
```

```
$ ./examples/pizza -V
0.0.1
```

You may change the flags and description by passing additional parameters to the `version` method, using the same syntax for flags as the `option` method.

```
program.version('0.0.1', '-v, --vers', 'output the current version');
```

More configuration

You can add most options using the `.option()` method, but there are some additional features available by constructing an `Option` explicitly for less common cases.

Example files: options-extra.js, options-env.js, options-conflicts.js, options-implies.js

```
program
  .addOption(new Option('-s, --secret').hideHelp())
  .addOption(new Option('-t, --timeout <delay>', 'timeout in seconds').default(60, 'or
  .addOption(new Option('-d, --drink <size>', 'drink size').choices(['small', 'medium'
  .addOption(new Option('-p, --port <number>', 'port number').env('PORT'))
  .addOption(new Option('--donate [amount]', 'optional donation in dollars').preset('2
  .addOption(new Option('--disable-server', 'disables the server').conflicts('port'))
  .addOption(new Option('--free-drink', 'small drink included free ').implies({ drink:
```

```
$ extra --help
Usage: help [options]

Options:
  -t, --timeout <delay>  timeout in seconds (default: one minute)
  -d, --drink <size>     drink cup size (choices: "small", "medium", "large")
  -p, --port <number>    port number (env: PORT)
  --donate [amount]      optional donation in dollars (preset: "20")
  --disable-server       disables the server
  --free-drink           small drink included free
  -h, --help             display help for command

$ extra --drink huge
error: option '-d, --drink <size>' argument 'huge' is invalid. Allowed choices are sma

$ PORT=80 extra --donate --free-drink
Options:  { timeout: 60, donate: 20, port: '80', freeDrink: true, drink: 'small' }

$ extra --disable-server --port 8000
error: option '--disable-server' cannot be used with option '-p, --port <number>'
```

Specify a required (mandatory) option using the `Option` method `.makeOptionMandatory()`. This matches the `Command` method .requiredOption().

Custom option processing

You may specify a function to do custom processing of option-arguments. The callback function receives two parameters, the user specified option-argument and the previous value for the option. It returns the new value for the option.

This allows you to coerce the option-argument to the desired type, or accumulate values, or do entirely custom processing.

You can optionally specify the default/starting value for the option after the function parameter.

Example file: [options-custom-processing.js]

```
function myParseInt(value, dummyPrevious) {
  // parseInt takes a string and a radix
  const parsedValue = parseInt(value, 10);
  if (isNaN(parsedValue)) {
    throw new commander.InvalidArgumentError('Not a number.');
  }
  return parsedValue;
}

function increaseVerbosity(dummyValue, previous) {
  return previous + 1;
}

function collect(value, previous) {
  return previous.concat([value]);
}

function commaSeparatedList(value, dummyPrevious) {
  return value.split(',');
}

program
  .option('-f, --float <number>', 'float argument', parseFloat)
  .option('-i, --integer <number>', 'integer argument', myParseInt)
  .option('-v, --verbose', 'verbosity that can be increased', increaseVerbosity, 0)
  .option('-c, --collect <value>', 'repeatable value', collect, [])
  .option('-l, --list <items>', 'comma separated list', commaSeparatedList)
;

program.parse();

const options = program.opts();
if (options.float !== undefined) console.log(`float: ${options.float}`);
if (options.integer !== undefined) console.log(`integer: ${options.integer}`);
if (options.verbose > 0) console.log(`verbosity: ${options.verbose}`);
if (options.collect.length > 0) console.log(options.collect);
if (options.list !== undefined) console.log(options.list);
```

```
$ custom -f 1e2
float: 100
$ custom --integer 2
integer: 2
$ custom -v -v -v
verbose: 3
$ custom -c a -c b -c c
[ 'a', 'b', 'c' ]
$ custom --list x,y,z
[ 'x', 'y', 'z' ]
```

Commands

You can specify (sub)commands using `.command()` or `.addCommand()`. There are two ways these can be implemented: using an action handler attached to the command, or as a stand-alone executable file (described in more detail later). The subcommands may be nested (example).

In the first parameter to `.command()` you specify the command name. You may append the command-arguments after the command name, or specify them separately using `.argument()`. The arguments may be `<required>` or `[optional]`, and the last argument may also be `variadic...`.

You can use `.addCommand()` to add an already configured subcommand to the program.

For example:

```
// Command implemented using action handler (description is supplied separately to `.c
// Returns new command for configuring.
program
  .command('clone <source> [destination]')
  .description('clone a repository into a newly created directory')
  .action((source, destination) => {
    console.log('clone command called');
  });

// Command implemented using stand-alone executable file, indicated by adding descript
// Returns `this` for adding more commands.
program
  .command('start <service>', 'start named service')
  .command('stop [service]', 'stop named service, or all if no name supplied');

// Command prepared separately.
// Returns `this` for adding more commands.
program
  .addCommand(build.makeBuildCommand());
```

Configuration options can be passed with the call to `.command()` and `.addCommand()`. Specifying `hidden: true` will remove the command from the generated help output. Specifying `isDefault: true` will run the subcommand if no other subcommand is specified (example).

You can add alternative names for a command with `.alias()`. (example)

For safety, `.addCommand()` does not automatically copy the inherited settings from the parent command. There is a helper routine `.copyInheritedSettings()` for copying the settings when they are wanted.

Command-arguments

For subcommands, you can specify the argument syntax in the call to `.command()` (as shown above). This is the only method usable for subcommands implemented using a stand-alone executable, but for other subcommands you can instead use the following method.

To configure a command, you can use `.argument()` to specify each expected command-argument. You supply the argument name and an optional description. The argument may be `<required>` or `[optional]`. You can specify a default value for an optional command-argument.

Example file: argument.js

```
program
  .version('0.1.0')
  .argument('<username>', 'user to login')
  .argument('[password]', 'password for user, if required', 'no password given')
  .action((username, password) => {
    console.log('username:', username);
    console.log('password:', password);
  });
```

The last argument of a command can be variadic, and only the last argument. To make an argument variadic you append `...` to the argument name. A variadic argument is passed to the action handler as an array. For example:

```
program
  .version('0.1.0')
  .command('rmdir')
  .argument('<dirs...>')
  .action(function (dirs) {
    dirs.forEach((dir) => {
      console.log('rmdir %s', dir);
    });
  });
```

There is a convenience method to add multiple arguments at once, but without descriptions:

```
program
  .arguments('<username> <password>');
```

More configuration

There are some additional features available by constructing an `Argument` explicitly for less common cases.

Example file: arguments-extra.js

```
program
  .addArgument(new commander.Argument('<drink-size>', 'drink cup size').choices(['smal
  .addArgument(new commander.Argument('[timeout]', 'timeout in seconds').default(60, '
```

Custom argument processing

You may specify a function to do custom processing of command-arguments (like for option-arguments). The callback function receives two parameters, the user specified command-argument and the previous value for the argument. It returns the new value for the argument.

The processed argument values are passed to the action handler, and saved as `.processedArgs`.

You can optionally specify the default/starting value for the argument after the function parameter.

Example file: arguments-custom-processing.js

```
program
  .command('add')
  .argument('<first>', 'integer argument', myParseInt)
  .argument('[second]', 'integer argument', myParseInt, 1000)
  .action((first, second) => {
    console.log(`${first} + ${second} = ${first + second}`);
  })
;
```

Action handler

The action handler gets passed a parameter for each command-argument you declared, and two additional parameters which are the parsed options and the command object itself.

Example file: thank.js

```
program
  .argument('<name>')
  .option('-t, --title <honorific>', 'title to use before name')
  .option('-d, --debug', 'display some debugging')
  .action((name, options, command) => {
    if (options.debug) {
      console.error('Called %s with options %o', command.name(), options);
    }
    const title = options.title ? `${options.title} ` : '';
    console.log(`Thank-you ${title}${name}`);
  });
```

If you prefer, you can work with the command directly and skip declaring the parameters for the action handler. The `this` keyword is set to the running command and can be used from a function expression (but not from an arrow function).

Example file: action-this.js

```
program
  .command('serve')
  .argument('<script>')
  .option('-p, --port <number>', 'port number', 80)
  .action(function() {
    console.error('Run script %s on port %s', this.args[0], this.opts().port);
  });
```

You may supply an `async` action handler, in which case you call `.parseAsync` rather than `.parse`.

```
async function run() { /* code goes here */ }

async function main() {
  program
    .command('run')
    .action(run);
  await program.parseAsync(process.argv);
}
```

A command's options and arguments on the command line are validated when the command is used. Any unknown options or missing arguments will be reported as an error. You can suppress the unknown option checks with `.allowUnknownOption()`. By default it is not an error to pass more arguments than declared, but you can make this an error with `.allowExcessArguments(false)`.

Stand-alone executable (sub)commands

When `.command()` is invoked with a description argument, this tells Commander that you're going to use stand-alone executables for subcommands. Commander will search the files in the directory of the entry script for a file with the name combination `command-subcommand`, like `pm-install` or `pm-search` in the example below. The search includes trying common file extensions, like `.js`. You may specify a custom name (and path) with the `executableFile` configuration option. You may specify a custom search directory for subcommands with `.executableDir()`.

You handle the options for an executable (sub)command in the executable, and don't declare them at the top-level.

Example file: pm

```
program
  .name('pm')
  .version('0.1.0')
  .command('install [name]', 'install one or more packages')
  .command('search [query]', 'search with optional query')
  .command('update', 'update installed packages', { executableFile: 'myUpdateSubCommar
  .command('list', 'list packages installed', { isDefault: true });

program.parse(process.argv);
```

If the program is designed to be installed globally, make sure the executables have proper modes, like `755`.

Life cycle hooks

You can add callback hooks to a command for life cycle events.

Example file: hook.js

```
program
  .option('-t, --trace', 'display trace statements for commands')
  .hook('preAction', (thisCommand, actionCommand) => {
    if (thisCommand.opts().trace) {
      console.log(`About to call action handler for subcommand: ${actionCommand.name()
      console.log('arguments: %O', actionCommand.args);
      console.log('options: %o', actionCommand.opts());
    }
  });
```

The callback hook can be `async`, in which case you call `.parseAsync` rather than `.parse`. You can add multiple hooks per event.

The supported events are:

event name	when hook called	callback parameters
`preAction`, `postAction`	before/after action handler for this command and its nested subcommands	`(thisCommand, actionCommand)`
`preSubcommand`	before parsing direct subcommand	`(thisCommand, subcommand)`

For an overview of the life cycle events see parsing life cycle and hooks.

Automated help

The help information is auto-generated based on the information commander already knows about your program. The default help option is `-h,--help`.

Example file: pizza

```
$ node ./examples/pizza --help
Usage: pizza [options]

An application for pizza ordering

Options:
  -p, --peppers        Add peppers
  -c, --cheese <type>  Add the specified type of cheese (default: "marble")
  -C, --no-cheese      You do not want any cheese
  -h, --help           display help for command
```

A `help` command is added by default if your command has subcommands. It can be used alone, or with a subcommand name to show further help for the subcommand. These are effectively the same if the `shell` program has implicit help:

```
shell help
shell --help

shell help spawn
shell spawn --help
```

Long descriptions are wrapped to fit the available width. (However, a description that includes a line-break followed by whitespace is assumed to be pre-formatted and not wrapped.)

Custom help

You can add extra text to be displayed along with the built-in help.

Example file: custom-help

```
program
  .option('-f, --foo', 'enable some foo');

program.addHelpText('after', `

Example call:
  $ custom-help --help`);
```

Yields the following help output:

```
Usage: custom-help [options]

Options:
  -f, --foo   enable some foo
  -h, --help  display help for command

Example call:
  $ custom-help --help
```

The positions in order displayed are:

- `beforeAll`: add to the program for a global banner or header
- `before`: display extra information before built-in help
- `after`: display extra information after built-in help
- `afterAll`: add to the program for a global footer (epilog)

The positions "beforeAll" and "afterAll" apply to the command and all its subcommands.

The second parameter can be a string, or a function returning a string. The function is passed a context object for your convenience. The properties are:

- error: a boolean for whether the help is being displayed due to a usage error
- command: the Command which is displaying the help

Display help after errors

The default behaviour for usage errors is to just display a short error message. You can change the behaviour to show the full help or a custom help message after an error.

```
program.showHelpAfterError();
// or
program.showHelpAfterError('(add --help for additional information)');
```

```
$ pizza --unknown
error: unknown option '--unknown'
(add --help for additional information)
```

The default behaviour is to suggest correct spelling after an error for an unknown command or option. You can disable this.

```
program.showSuggestionAfterError(false);
```

```
$ pizza --hepl
error: unknown option '--hepl'
(Did you mean --help?)
```

Display help from code

`.help()`: display help information and exit immediately. You can optionally pass `{ error: true }` to display on stderr and exit with an error status.

`.outputHelp()`: output help information without exiting. You can optionally pass `{ error: true }` to display on stderr.

`.helpInformation()`: get the built-in command help information as a string for processing or displaying yourself.

.name

The command name appears in the help, and is also used for locating stand-alone executable subcommands.

You may specify the program name using `.name()` or in the Command constructor. For the program, Commander will fallback to using the script name from the full arguments passed into `.parse()`. However, the script name varies depending on how your program is launched so you may wish to specify it explicitly.

```
program.name('pizza');
const pm = new Command('pm');
```

Subcommands get a name when specified using `.command()`. If you create the subcommand yourself to use with `.addCommand()`, then set the name using `.name()` or in the Command constructor.

.usage

This allows you to customise the usage description in the first line of the help. Given:

```
program
  .name("my-command")
  .usage("[global options] command")
```

The help will start with:

```
Usage: my-command [global options] command
```

.description and .summary

The description appears in the help for the command. You can optionally supply a shorter summary to use when listed as a subcommand of the program.

```
program
  .command("duplicate")
  .summary("make a copy")
  .description(`Make a copy of the current project.
This may require additional disk space.
  `);
```

.helpOption(flags, description)

By default every command has a help option. You may change the default help flags and description. Pass false to disable the built-in help option.

```
program
  .helpOption('-e, --HELP', 'read more information');
```

.addHelpCommand()

A help command is added by default if your command has subcommands. You can explicitly turn on or off the implicit help command with `.addHelpCommand()` and `.addHelpCommand(false)`.

You can both turn on and customise the help command by supplying the name and description:

```
program.addHelpCommand('assist [command]', 'show assistance');
```

More configuration

The built-in help is formatted using the Help class. You can configure the Help behaviour by modifying data properties and methods using `.configureHelp()`, or by subclassing using `.createHelp()` if you prefer.

The data properties are:

- `helpWidth`: specify the wrap width, useful for unit tests
- `sortSubcommands`: sort the subcommands alphabetically
- `sortOptions`: sort the options alphabetically
- `showGlobalOptions`: show a section with the global options from the parent command(s)

You can override any method on the Help class. There are methods getting the visible lists of arguments, options, and subcommands. There are methods for formatting the items in the lists, with each item having a *term* and *description*. Take a look at `.formatHelp()` to see how they are used.

Example file: configure-help.js

```
program.configureHelp({
  sortSubcommands: true,
  subcommandTerm: (cmd) => cmd.name() // Just show the name, instead of short usage.
});
```

Custom event listeners

You can execute custom actions by listening to command and option events.

```
program.on('option:verbose', function () {
  process.env.VERBOSE = this.opts().verbose;
});
```

Bits and pieces

.parse() and .parseAsync()

The first argument to `.parse` is the array of strings to parse. You may omit the parameter to implicitly use `process.argv`.

If the arguments follow different conventions than node you can pass a `from` option in the second parameter:

- 'node': default, `argv[0]` is the application and `argv[1]` is the script being run, with user parameters after that
- 'electron': `argv[1]` varies depending on whether the electron application is packaged
- 'user': all of the arguments from the user

For example:

```
program.parse(process.argv); // Explicit, node conventions
program.parse(); // Implicit, and auto-detect electron
program.parse(['-f', 'filename'], { from: 'user' });
```

Parsing Configuration

If the default parsing does not suit your needs, there are some behaviours to support other usage patterns.

By default program options are recognised before and after subcommands. To only look for program options before subcommands, use `.enablePositionalOptions()`. This lets you use an option for a different purpose in subcommands.

Example file: positional-options.js

With positional options, the `-b` is a program option in the first line and a subcommand option in the second line:

```
program -b subcommand
program subcommand -b
```

By default options are recognised before and after command-arguments. To only process options that come before the command-arguments, use `.passThroughOptions()`. This lets you pass the arguments and following options through to another program without needing to use `--` to end the option processing. To use pass through options in a subcommand, the program needs to enable positional options.

Example file: pass-through-options.js

With pass through options, the `--port=80` is a program option in the first line and passed through as a command-argument in the second line:

```
program --port=80 arg
program arg --port=80
```

By default the option processing shows an error for an unknown option. To have an unknown option treated as an ordinary command-argument and continue looking for options, use `.allowUnknownOption()`. This lets you mix known and unknown options.

By default the argument processing does not display an error for more command-arguments than expected. To display an error for excess arguments, use `.allowExcessArguments(false)`.

Legacy options as properties

Before Commander 7, the option values were stored as properties on the command. This was convenient to code but the downside was possible clashes with existing properties of `Command`. You can revert to the old behaviour to run unmodified legacy code by using `.storeOptionsAsProperties()`.

```
program
  .storeOptionsAsProperties()
  .option('-d, --debug')
  .action((commandAndOptions) => {
    if (commandAndOptions.debug) {
      console.error(`Called ${commandAndOptions.name()}`);
    }
  });
```

TypeScript

extra-typings: There is an optional project to infer extra type information from the option and argument definitions. This adds strong typing to the options returned by `.opts()` and the parameters to `.action()`. See commander-js/extra-typings for more.

```
import { Command } from '@commander-js/extra-typings';
```

ts-node: If you use `ts-node` and stand-alone executable subcommands written as `.ts` files, you need to call your program through node to get the subcommands called correctly. e.g.

```
node -r ts-node/register pm.ts
```

createCommand()

This factory function creates a new command. It is exported and may be used instead of using `new`, like:

```
const { createCommand } = require('commander');
const program = createCommand();
```

`createCommand` is also a method of the Command object, and creates a new command rather than a subcommand. This gets used internally when creating subcommands using `.command()`, and you may override it to customise the new subcommand (example file custom-command-class.js).

Node options such as `--harmony`

You can enable `--harmony` option in two ways:

- Use `#! /usr/bin/env node --harmony` in the subcommands scripts. (Note Windows does not support this pattern.)
- Use the `--harmony` option when call the command, like `node --harmony examples/pm publish`. The `--harmony` option will be preserved when spawning subcommand process.

Debugging stand-alone executable subcommands

An executable subcommand is launched as a separate child process.

If you are using the node inspector for debugging executable subcommands using `node --inspect` et al, the inspector port is incremented by 1 for the spawned subcommand.

If you are using VSCode to debug executable subcommands you need to set the `"autoAttachChildProcesses": true` flag in your launch.json configuration.

Display error

This routine is available to invoke the Commander error handling for your own error conditions. (See also the next section about exit handling.)

As well as the error message, you can optionally specify the `exitCode` (used with `process.exit`) and `code` (used with `CommanderError`).

```
program.error('Password must be longer than four characters');
program.error('Custom processing has failed', { exitCode: 2, code: 'my.custom.error' }
```

Override exit and output handling

By default Commander calls `process.exit` when it detects errors, or after displaying the help or version. You can override this behaviour and optionally supply a callback. The default override throws a `CommanderError`.

The override callback is passed a `CommanderError` with properties `exitCode` number, `code` string, and `message`. The default override behaviour is to throw the error, except for async handling of executable subcommand completion which carries on. The normal display of error messages or version or help is not affected by the override which is called after the display.

```
program.exitOverride();

try {
  program.parse(process.argv);
} catch (err) {
  // custom processing...
}
```

By default Commander is configured for a command-line application and writes to stdout and stderr. You can modify this behaviour for custom applications. In addition, you can modify the display of error messages.

Example file: configure-output.js

```
function errorColor(str) {
  // Add ANSI escape codes to display text in red.
  return `\x1b[31m${str}\x1b[0m`;
}

program
  .configureOutput({
    // Visibly override write routines as example!
    writeOut: (str) => process.stdout.write(`[OUT] ${str}`),
    writeErr: (str) => process.stdout.write(`[ERR] ${str}`),
    // Highlight errors in color.
    outputError: (str, write) => write(errorColor(str))
  });
```

Additional documentation

There is more information available about:

- deprecated features still supported for backwards compatibility
- options taking varying arguments
- parsing life cycle and hooks

Support

The current version of Commander is fully supported on Long Term Support versions of Node.js, and requires at least v14. (For older versions of Node.js, use an older version of Commander.)

The main forum for free and community support is the project Issues on GitHub.

Commander for enterprise

Available as part of the Tidelift Subscription

The maintainers of Commander and thousands of other packages are working with Tidelift to deliver commercial support and maintenance for the open source dependencies you use to build your applications. Save time, reduce risk, and improve code health, while paying the maintainers of the exact dependencies you use. Learn more.