Commander.js

Build Status Install Size

The complete solution for node.js command-line interfaces, inspired by Ruby's commander.

Read this in other languages: English | [?]体中文

Installation

```
npm install commander
```

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.

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.

```
const { program } =
        require('commander');
program.version('0.0.1');
```

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

```
const { Command } =
        require('commander');
const program = new Command();
program.version('0.0.1');
```

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 options can be accessed as properties on the Command object. Multi-word options such as “–template-engine” are camel-

cased, becoming `program.templateEngine` etc. See also optional new behaviour to avoid name clashes.

Multiple short flags may optionally be combined in a single argument following the dash: boolean flags, the last flag may take a value, and the value. For example `-a -b -p 80` may be written as `-ab -p80` or even `-abp80`.

You can use `--` to indicate the end of the options, and any remaining arguments will be used without being interpreted. This is particularly useful for passing options through to another command, like: `do -- git --version`.

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

Common option types, boolean and value

The two most used option types are a boolean flag, and an option which takes a value (declared using angle brackets). Both are `undefined` unless specified on command line.

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

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



```
    console.log('Examples:');
    console.log('');
    console.log('  $ deploy exec
        sequential');
    console.log('  $ deploy exec
        async');
  });

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

More Demos can be found in the examples directory.

License

MIT

Support

Commander 5.x is fully supported on Long Term Support versions of Node, and is likely to work with Node 6 but not tested. (For versions of Node below Node 6, use Commander 3.x or 2.x.)

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

```
  .option('-C, --chdir <path>', 'change
       the working directory')
  .option('-c, --config <path>', 'set
       config path. defaults to ./
       deploy.conf')
  .option('-T, --no-tests',
       'ignore test hook');

program
  .command('setup [env]')
  .description('run setup commands for
       all envs')
  .option("-s, --setup_mode [mode]",
       "Which setup mode to use")
  .action(function(env, options){
    const mode = options.setup_mode ||
       "normal";
    env = env || 'all';
    console.log('setup for %s env(s)
       with %s mode', env, mode);
  });

program
  .command('exec <cmd>')
  .alias('ex')
  .description('execute the given remote
       cmd')
  .option("-e, --exec_mode <mode>",
       "Which exec mode to use")
  .action(function(cmd, options){
    console.log('exec "%s" using %s
       mode', cmd, options.exec_mode);
  }).on('--help', function() {
    console.log('');
```



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

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

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

`program.parse(arguments)` processes the arguments, leaving any args not consumed by the program options in the `program.args` array.

Default option value

You can specify a default value for an option which takes a value.

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

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

program.parse(process.argv);

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

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

Other option types, negatable boolean and flag|value

You can specify 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.

Override exit 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...
}
```

Examples

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

program
  .version('0.1.0')
```

and you may override it to customise the new subcommand (examples using subclass and function).

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.

If you define `--foo` first, adding `--no-foo` does not change the default value from what it would otherwise be. You can specify a default boolean value for a boolean flag and it can be overridden on command line.

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

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

const sauceStr = program.sauce ?
        'sauce' : 'no sauce';
const cheeseStr = (program.cheese ===
        false) ? 'no cheese' : `$
        {program.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 functions as a flag but may also take a value (declared using square brackets).

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

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

program.parse(process.argv);

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

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



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

const programOptions = program.opts();
console.log(programOptions.name);
```

TypeScript

The Commander package includes its TypeScript Definition file.

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()`,

Avoiding option name clashes

The original and default behaviour is that the option values are stored as properties on the program, and the action handler is passed a command object with the options values stored as properties. This is very convenient to code, but the downside is possible clashes with existing properties of Command.

There are two new routines to change the behaviour, and the default behaviour may change in the future:

- `storeOptionsAsProperties`: whether to store option values as properties on command object, or store separately (specify false) and access using `.opts()`
- `passCommandToAction`: whether to pass command to action handler, or just the options (specify false)

(example)

```
program
  .storeOptionsAsProperties(false)
  .passCommandToAction(false);

program
  .name('my-program-name')
  .option('-n,--name <name>');

program
  .command('show')
  .option('-a,--action <action>')
  .action((options) => {
    console.log(options.action);
  });
```


Custom option processing

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

This allows you to coerce the option value 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.

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

function myParseInt(value,
        dummyPrevious) {
  // parseInt takes a string and an
        optional radix
  return parseInt(value);
}

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(process.argv);

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



```
  mySuggestBestMatch(operands[0],
        availableCommands);
  process.exitCode = 1;
});
```

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' });
```

.addHelpCommand()

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');
```

Custom event listeners

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

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

program.on('command:*', function
        (operands) {
  console.error(`error: unknown command
        '${operands[0]}'`);
  const availableCommands =
        program.commands.map(cmd =>
        cmd.name());
```



```
$ 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' ]
```

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.

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

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

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

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

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. The version flags can be named anything, but a long name is required.

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

Commands

You can specify (sub)commands using `.command()` or `.addCommand()`. There are two ways these can be

.help(cb)

Output help information and exit immediately. Optional callback cb allows post-processing of help text before it is displayed.

.outputHelp(cb)

Output help information without exiting. Optional callback cb allows post-processing of help text before it is displayed.

.helpInformation()

Get the command help information as a string for processing or displaying yourself. (The text does not include the custom help from `--help` listeners.)

.helpOption(flags, description)

Override the default help flags and description.

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

```
  console.log('');
  console.log('Example call:');
  console.log('  $ 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
```

.usage and .name

These allow you to customise the usage description in the first line of the help. The name is otherwise deduced from the (full) program arguments. Given:

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

The help will start with:

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


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 and any command arguments. 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 `.command`)
// 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 (description is
        second parameter to `.command`)
// 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 `true` for `opts.hidden` will remove the command from the generated help output. Specifying `true` for `opts.isDefault` will run the subcommand if no other subcommand is specified (example).

Specify the argument syntax

You use `.arguments` to specify the arguments for the top-level command, and for subcommands they are usually included in the `.command` call. Angled brackets (e.g. `<required>`) indicate required input. Square brackets (e.g. `[optional]`) indicate optional input.

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

program
  .version('0.1.0')
  .arguments('<cmd> [env]')
```



```
  -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
```

Custom help

You can display extra information by listening for "–help". (example)

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

// must be before .parse()
program.on('--help', () => {
```

```
program
  .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:
        'myUpdateSubCommand'})
  .command('list', 'list packages
        installed', {isDefault: true})
  .parse(process.argv);
```

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

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)

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

An application for pizzas ordering

Options:
  -V, --version        output the
        version number
```



```
  .action(function (cmd, env) {
    cmdValue = cmd;
    envValue = env;
  });

program.parse(process.argv);

if (typeof cmdValue === 'undefined') {
  console.error('no command given!');
  process.exit(1);
}
console.log('command:', cmdValue);
console.log('environment:', envValue ||
        "no environment given");
```

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. For example:

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

program
  .version('0.1.0')
  .command('rmdir <dir> [otherDirs...]')
  .action(function (dir, otherDirs) {
    console.log('rmdir %s', dir);
    if (otherDirs) {
      otherDirs.forEach(function (oDir)
        {
        console.log('rmdir %s', oDir);
      });
    }
  });
```

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

The variadic argument is passed to the action handler as an array.

Action handler (sub)commands

You can add options to a command that uses an action handler. The action handler gets passed a parameter for each argument you declared, and one additional argument which is the command object itself. This command argument has the values for the command-specific options added as properties.

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

program
  .command('rm <dir>')
  .option('-r, --recursive', 'Remove
        recursively')
  .action(function (dir, cmdObj) {
    console.log('remove ' + dir +
        (cmdObj.recursive ? '
        recursively' : ''))
  })

program.parse(process.argv)
```

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 on the command line are validated when the command is used. Any unknown options will be reported as an error.

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 executables in the directory of the entry script (like `./examples/pm`) with the name `program-subcommand`, like `pm-install`, `pm-search`. You can specify a custom name with the `executableFile` configuration option.

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

```
// file: ./examples/pm
const { program } =
        require('commander');
```