Commander.js

Build Status Install Size

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

API documentation

Installation

```
$ npm install commander
```

Option parsing

Options with commander are defined with the `.option()` method, also serving as documentation for the options. The example below parses args and options from `process.argv`, leaving remaining args as the `program.args` array which were not consumed by options.

```
 * Module dependencies.
 */

var program = require('commander');

program
  .version('0.1.0')
  .option('-p, --peppers', 'Add
        peppers')
  .option('-P, --pineapple', 'Add
        pineapple')
  .option('-b, --bbq-sauce', 'Add bbq
        sauce')
  .option('-c, --cheese [type]', 'Add
        the specified type of cheese
        [marble]', 'marble')
  .parse(process.argv);

console.log('you ordered a pizza
        with:');
if (program.peppers) console.log('  -
        peppers');
if (program.pineapple) console.log('  -
        pineapple');
if (program.bbqSauce) console.log('  -
        bbq');
console.log('  - %s cheese',
        program.cheese);
```

Short flags may be passed as a single arg, for example `-abc` is equivalent to `-a -b -c`. Multi-word options such as "–template-engine" are camel-cased, becoming `program.templateEngine` etc.

Note that multi-word options starting with `--no` prefix negate the boolean value of the following word. For example, `--no-sauce` sets the value of `program.sauce` to false.

```
#!/usr/bin/env node

/**
 * Module dependencies.
 */

var program = require('commander');

program
  .option('--no-sauce', 'Remove sauce')
  .parse(process.argv);

console.log('you ordered a pizza');
if (program.sauce) console.log('  with
        sauce');
else console.log(' without sauce');
```

To get string arguments from options you will need to use angle brackets <> for required inputs or square brackets [] for optional inputs.

e.g. `.option('-m --myarg [myVar]', 'my super cool description')`

Then to access the input if it was passed in.

e.g. `var myInput = program.myarg`

NOTE: If you pass a argument without using brackets the example above will return true and not the value passed in.

Version option

Calling the `version` implicitly adds the `-V` and `--version` options to the command. When either of these options is present, the command prints the version number and exits.

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

If you want your program to respond to the `-v` option instead of the `-V` option, simply pass custom flags to the `version` method using the same syntax as the `option` method.

```
program
  .version('0.0.1', '-v, --version')
```

The version flags can be named anything, but the long option is required.

Command-specific options

You can attach options to a command.

```
#!/usr/bin/env node

var program = require('commander');

program
  .command('rm <dir>')
  .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('');
    console.log('Examples:');
    console.log('');
    console.log('  $ deploy exec sequential');
    console.log('  $ deploy exec async');
  });

program
  .command('*')
  .action(function(env){
    console.log('deploying "%s"', env);
  });

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

More Demos can be found in the examples directory.

License

MIT

Examples

```
var program = require('commander');

program
  .version('0.1.0')
  .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){
    var 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')
```



```
  .option('-r, --recursive', 'Remove
       recursively')
  .action(function (dir, cmd) {
    console.log('remove ' + dir +
       (cmd.recursive ? '
       recursively' : ''))
  })

program.parse(process.argv)
```

A command's options are validated when the command is used. Any unknown options will be reported as an error. However, if an action-based command does not define an action, then the options are not validated.

Coercion

```
function range(val) {
  return val.split('..').map(Number);
}

function list(val) {
  return val.split(',');
}

function collect(val, memo) {
  memo.push(val);
  return memo;
}
```

```
function increaseVerbosity(v, total) {
  return total + 1;
}

program
  .version('0.1.0')
  .usage('[options] <file ...>')
  .option('-i, --integer <n>', 'An
        integer argument', parseInt)
  .option('-f, --float <n>', 'A float
        argument', parseFloat)
  .option('-r, --range <a>..<b>', 'A
        range', range)
  .option('-l, --list <items>', 'A
        list', list)
  .option('-o, --optional [value]', 'An
        optional value')
  .option('-c, --collect [value]', 'A
        repeatable value', collect, [])
  .option('-v, --verbose',
        'A value that can be
        increased', increaseVerbosity,
        0)
  .parse(process.argv);

console.log(' int: %j',
        program.integer);
console.log(' float: %j',
        program.float);
console.log(' optional: %j',
        program.optional);
program.range = program.range || [];
```


.help(cb)

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

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;
});

// error on unknown commands
program.on('command:*', function () {
  console.error('Invalid command:
        %s\nSee --help for a list of
        available commands.',
        program.args.join(' '));
  process.exit(1);
});
```

.outputHelp(cb)

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

If you want to display help by default (e.g. if no command was provided), you can use something like:

```
var program = require('commander');
var colors = require('colors');

program
  .version('0.1.0')
  .command('getstream [url]', 'get
        stream URL')
  .parse(process.argv);

if (!process.argv.slice(2).length) {
  program.outputHelp(make_red);
}

function make_red(txt) {
  return colors.red(txt); //display the
        help text in red on the console
}
```



```
console.log(' range: %j..%j',
        program.range[0],
        program.range[1]);
console.log(' list: %j', program.list);
console.log(' collect: %j',
        program.collect);
console.log(' verbosity: %j',
        program.verbose);
console.log(' args: %j', program.args);
```

Regular Expression

```
program
  .version('0.1.0')
  .option('-s --size <size>', 'Pizza
        size', /^(large|medium|small)$/
        i, 'medium')
  .option('-d --drink [drink]',
        'Drink', /^(coke|pepsi|izze)$/i)
  .parse(process.argv);

console.log(' size: %j', program.size);
console.log(' drink: %j',
        program.drink);
```

Variadic arguments

The last argument of a command can be variadic, and only the last argument. To make an argument variadic you have to append `...` to the argument name. Here is an example:

```
#!/usr/bin/env node

/**
 * Module dependencies.
 */

var 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);
```

An `Array` is used for the value of a variadic argument. This applies to `program.args` as well as the argument passed to your action as demonstrated above.


```
program.on('--help', function(){
  console.log('')
  console.log('Examples:');
  console.log('  $ custom-help --help');
  console.log('  $ custom-help -h');
});

program.parse(process.argv);

console.log('stuff');
```

Yields the following help output when `node script-name.js -h` or `node script-name.js --help` are run:

```
Usage: custom-help [options]

Options:
  -h, --help     output usage
information
  -V, --version  output the version
number
  -f, --foo      enable some foo
  -b, --bar      enable some bar
  -B, --baz      enable some baz

Examples:
  $ custom-help --help
  $ custom-help -h
```

```
  -C, --no-cheese       You do not want
any cheese
```

Custom help

You can display arbitrary `-h, --help` information by listening for "–help". Commander will automatically exit once you are done so that the remainder of your program does not execute causing undesired behaviors, for example in the following executable "stuff" will not output when `--help` is used.

```
#!/usr/bin/env node

/**
 * Module dependencies.
 */

var program = require('commander');

program
  .version('0.1.0')
  .option('-f, --foo', 'enable some
        foo')
  .option('-b, --bar', 'enable some
        bar')
  .option('-B, --baz', 'enable some
        baz');

// must be before .parse() since
// node's emit() is immediate
```


Specify the argument syntax

```
#!/usr/bin/env node

var program = require('commander');

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

Angled brackets (e.g. `<cmd>`) indicate required input. Square brackets (e.g. `[env]`) indicate optional input.

Git-style sub-commands

```
// file: ./examples/pm
var program = require('commander');

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

When `.command()` is invoked with a description argument, no `.action(callback)` should be called to handle sub-commands, otherwise there will be an error. This tells commander that you're going to use separate executables for sub-commands, much like `git(1)` and other popular tools.

The commander will try to search the executables in the directory of the entry script (like `./examples/pm`) with the name `program-command`, like `pm-install`, `pm-search`.

Options can be passed with the call to `.command()`. Specifying `true` for `opts.noHelp` will remove the subcommand from the generated help output. Specifying `true` for `opts.isDefault` will run the subcommand if no other subcommand is specified.

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

`--harmony`

You can enable `--harmony` option in two ways: * Use `#! /usr/bin/env node --harmony` in the sub-commands scripts. Note some os version don't 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 sub-command process.

Automated –help

The help information is auto-generated based on the information commander already knows about your program, so the following `--help` info is for free:

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

An application for pizzas ordering

Options:
  -h, --help           output usage
information
  -V, --version        output the
version number
  -p, --peppers        Add peppers
  -P, --pineapple      Add pineapple
  -b, --bbq            Add bbq sauce
  -c, --cheese <type>  Add the specified
type of cheese [marble]
```