# jackspeak

A very strict and proper argument parser.

Validate string, boolean, and number options, from the command line and the environment.

Call the jack method with a config object, and then chain methods off of it.

At the end, call the .parse() method, and you'll get an object with positionals and values members.

Any unrecognized configs or invalid values will throw an error.

As long as you define configs using object literals, types will be properly inferred and TypeScript will know what kinds of things you got.

If you give it a prefix for environment variables, then defaults will be read from the environment, and parsed values written back to it, so you can easily pass configs through to child processes.

Automatically generates a usage/help banner by calling the .usage() method.

Unless otherwise noted, all methods return the object itself.

## USAGE

import { jack } from 'jackspeak'

// this works too:

// const { jack } = require('jackspeak')

const { positionals, values } = jack({ envPrefix: 'FOO' })

.flag({

asdf: { description: 'sets the asfd flag', short: 'a', default: true },

'no-asdf': { description: 'unsets the asdf flag', short: 'A' },

foo: { description: 'another boolean', short: 'f' },

})

.optList({

'ip-addrs': {

description: 'addresses to ip things',

delim: ',', // defaults to '\n'

default: ['127.0.0.1'],

},

})

.parse([

'some',

'positional',

'--ip-addrs',

'192.168.0.1',

'--ip-addrs',

'1.1.1.1',

'args',

'--foo', // sets the foo flag

'-A', // short for --no-asdf, sets asdf flag to false

])

console.log(process.env.FOO\_ASDF) // '0'

console.log(process.env.FOO\_FOO) // '1'

console.log(values) // {

// 'ip-addrs': ['192.168.0.1', '1.1.1.1'],

// foo: true,

// asdf: false,

// }

console.log(process.env.FOO\_IP\_ADDRS) // '192.168.0.1,1.1.1.1'

console.log(positionals) // ['some', 'positional', 'args']

## jack(options: JackOptions = {}) => Jack

Returns a Jack object that can be used to chain and add field definitions. The other methods (apart from validate(), parse(), and usage() obviously) return the same Jack object, updated with the new types, so they can be chained together as shown in the code examples.

Options:

* allowPositionals Defaults to true. Set to false to not allow any positional arguments.
* envPrefix Set to a string to write configs to and read configs from the environment. For example, if set to MY\_APP then the foo-bar config will default based on the value of env.MY\_APP\_FOO\_BAR and will write back to that when parsed.

Boolean values are written as '1' and '0', and will be treated as true if they're '1' or false otherwise.

Number values are written with their toString() representation.

Strings are just strings.

Any value with multiple: true will be represented in the environment split by a delimiter, which defaults to \n.

* env The place to read/write environment variables. Defaults to process.env.
* usage A short usage string to print at the top of the help banner.
* stopAtPositional Boolean, default false. Stop parsing opts and flags at the first positional argument. This is useful if you want to pass certain options to subcommands, like some programs do, so you can stop parsing and pass the positionals to the subcommand to parse.
* stopAtPositionalTest Conditional stopAtPositional. Provide a function that takes a positional argument string and returns boolean. If it returns true, then parsing will stop. Useful when *some* subcommands should parse the rest of the command line options, and others should not.

### Jack.heading(text: string, level?: 1 | 2 | 3 | 4 | 5 | 6)

Define a short string heading, used in the usage() output.

Indentation of the heading and subsequent description/config usage entries (up until the next heading) is set by the heading level.

If the first usage item defined is a heading, it is always treated as level 1, regardless of the argument provided.

Headings level 1 and 2 will have a line of padding underneath them. Headings level 3 through 6 will not.

### Jack.description(text: string, { pre?: boolean } = {})

Define a long string description, used in the usage() output.

If the pre option is set to true, then whitespace will not be normalized. However, if any line is too long for the width allotted, it will still be wrapped.

## Option Definitions

Configs are defined by calling the appropriate field definition method with an object where the keys are the long option name, and the value defines the config.

Options:

* type Only needed for the addFields method, as the others set it implicitly. Can be 'string', 'boolean', or 'number'.
* multiple Only needed for the addFields method, as the others set it implicitly. Set to true to define an array type. This means that it can be set on the CLI multiple times, set as an array in the values and it is represented in the environment as a delimited string.
* short A one-character shorthand for the option.
* description Some words to describe what this option is and why you'd set it.
* hint (Only relevant for non-boolean types) The thing to show in the usage output, like --option=<hint>
* validate A function that returns false (or throws) if an option value is invalid.
* validOptions An array of strings or numbers that define the valid values that can be set. This is not allowed on boolean (flag) options. May be used along with a validate() method.
* default A default value for the field. Note that this may be overridden by an environment variable, if present.

### Jack.flag({ [option: string]: definition, ... })

Define one or more boolean fields.

Boolean options may be set to false by using a --no-${optionName} argument, which will be implicitly created if it's not defined to be something else.

If a boolean option named no-${optionName} with the same multiple setting is in the configuration, then that will be treated as a negating flag.

### Jack.flagList({ [option: string]: definition, ... })

Define one or more boolean array fields.

### Jack.num({ [option: string]: definition, ... })

Define one or more number fields. These will be set in the environment as a stringified number, and included in the values object as a number.

### Jack.numList({ [option: string]: definition, ... })

Define one or more number list fields. These will be set in the environment as a delimited set of stringified numbers, and included in the values as a number array.

### Jack.opt({ [option: string]: definition, ... })

Define one or more string option fields.

### Jack.optList({ [option: string]: definition, ... })

Define one or more string list fields.

### Jack.addFields({ [option: string]: definition, ... })

Define one or more fields of any type. Note that type and multiple must be set explicitly on each definition when using this method.

## Actions

Use these methods on a Jack object that's already had its config fields defined.

### Jack.parse(args: string[] = process.argv): { positionals: string[], values: OptionsResults }

Parse the arguments list, write to the environment if envPrefix is set, and returned the parsed values and remaining positional arguments.

### Jack.validate(o: any): asserts o is OptionsResults

Throws an error if the object provided is not a valid result set, for the configurations defined thusfar.

### Jack.usage(): string

Returns the compiled usage string, with all option descriptions and heading/description text, wrapped to the appropriate width for the terminal.

### Jack.setConfigValues(options: OptionsResults, src?: string)

Validate the options argument, and set the default value for each field that appears in the options.

Values provided will be overridden by environment variables or command line arguments.

### Jack.usageMarkdown(): string

Returns the compiled usage string, with all option descriptions and heading/description text, but as markdown instead of formatted for a terminal, for generating HTML documentation for your CLI.

## Some Example Code

Also see [the examples folder](https://github.com/isaacs/jackspeak/tree/master/examples)

import { jack } from 'jackspeak'

const j = jack({

// Optional

// This will be auto-generated from the descriptions if not supplied

// top level usage line, printed by -h

// will be auto-generated if not specified

usage: 'foo [options] <files>',

})

.heading('The best Foo that ever Fooed')

.description(

`

Executes all the files and interprets their output as

TAP formatted test result data.

To parse TAP data from stdin, specify "-" as a filename.

`,

)

// flags don't take a value, they're boolean on or off, and can be

// turned off by prefixing with `--no-`

// so this adds support for -b to mean --bail, or -B to mean --no-bail

.flag({

flag: {

// specify a short value if you like. this must be a single char

short: 'f',

// description is optional as well.

description: `Make the flags wave`,

// default value for flags is 'false', unless you change it

default: true,

},

'no-flag': {

// you can can always negate a flag with `--no-flag`

// specifying a negate option will let you define a short

// single-char option for negation.

short: 'F',

description: `Do not wave the flags`,

},

})

// Options that take a value are specified with `opt()`

.opt({

reporter: {

short: 'R',

description: 'the style of report to display',

},

})

// if you want a number, say so, and jackspeak will enforce it

.num({

jobs: {

short: 'j',

description: 'how many jobs to run in parallel',

default: 1,

},

})

// A list is an option that can be specified multiple times,

// to expand into an array of all the settings. Normal opts

// will just give you the last value specified.

.optList({

'node-arg': {},

})

// a flagList is an array of booleans, so `-ddd` is [true, true, true]

// count the `true` values to treat it as a counter.

.flagList({

debug: { short: 'd' },

})

// opts take a value, and is set to the string in the results

// you can combine multiple short-form flags together, but

// an opt will end the combine chain, posix-style. So,

// -bofilename would be like --bail --output-file=filename

.opt({

'output-file': {

short: 'o',

// optional: make it -o<file> in the help output insead of -o<value>

hint: 'file',

description: `Send the raw output to the specified file.`,

},

})

// now we can parse argv like this:

const { values, positionals } = j.parse(process.argv)

// or decide to show the usage banner

console.log(j.usage())

// or validate an object config we got from somewhere else

try {

j.validate(someConfig)

} catch (er) {

console.error('someConfig is not valid!', er)

}

## Name

The inspiration for this module is [yargs](http://npm.im/yargs), which is pirate talk themed. Yargs has all the features, and is infinitely flexible. "Jackspeak" is the slang of the royal navy. This module does not have all the features. It is declarative and rigid by design.