<http://commons.apache.org/proper/commons-cli/introduction.html>

# Introduction

There are three stages to command line processing. They are the definition, parsing and interrogation stages. The following sections will discuss each of these stages in turn, and discuss how to implement them with CLI.

## Definition Stage

Each command line must define the set of options that will be used to define the interface to the application.

CLI uses the [Options](http://commons.apache.org/proper/commons-cli/api-release/org/apache/commons/cli/Options.html) class, as a container for [Option](http://commons.apache.org/proper/commons-cli/api-release/org/apache/commons/cli/Option.html) instances. There are two ways to create Options in CLI. One of them is via the constructors, the other way is via the factory methods defined in Options.

The [Usage Scenarios](http://commons.apache.org/proper/commons-cli/usage.html) document provides examples how to create an Options object and also provides some real world examples.

The result of the definition stage is an Options instance.

## Parsing Stage

The parsing stage is where the text passed into the application via the command line is processed. The text is processed according to the rules defined by the parser implementation.

The parse method defined on [CommandLineParser](http://commons.apache.org/proper/commons-cli/api-release/org/apache/commons/cli/CommandLineParser.html) takes an Options instance and a String[] of arguments and returns a [CommandLine](http://commons.apache.org/proper/commons-cli/api-release/org/apache/commons/cli/CommandLine.html).

The result of the parsing stage is a CommandLine instance.

## Interrogation Stage

The interrogation stage is where the application queries the CommandLine to decide what execution branch to take depending on boolean options and uses the option values to provide the application data.

This stage is implemented in the user code. The accessor methods on CommandLine provide the interrogation capability to the user code.

The result of the interrogation stage is that the user code is fully informed of all the text that was supplied on the command line and processed according to the parser and Options rules.

<http://commons.apache.org/proper/commons-cli/usage.html>

# Usage Scenarios

The following sections describe some example scenarios on how to use CLI in applications.

### Using a boolean option

A boolean option is represented on a command line by the presence of the option, i.e. if the option is found then the option value is true, otherwise the value is false.

The DateApp utility prints the current date to standard output. If the -t option is present the current time is also printed.

### Create the Options

An [Options](http://commons.apache.org/proper/commons-cli/api-release/org/apache/commons/cli/Options.html) object must be created and the Option must be added to it.

// create Options object  
Options options = new Options();  
  
// add t option  
options.addOption("t", false, "display current time");

The addOption method has three parameters. The first parameter is a java.lang.String that represents the option. The second parameter is a boolean that specifies whether the option requires an argument or not. In the case of a boolean option (sometimes referred to as a flag) an argument value is not present so false is passed. The third parameter is the description of the option. This description will be used in the usage text of the application.

### Parsing the command line arguments

The parse methods of CommandLineParser are used to parse the command line arguments. There may be several implementations of the CommandLineParser interface, the recommended one is the DefaultParser.

CommandLineParser parser = new DefaultParser();  
CommandLine cmd = parser.parse( options, args);

Now we need to check if the t option is present. To do this we will interrogate the [CommandLine](http://commons.apache.org/proper/commons-cli/api-release/org/apache/commons/cli/CommandLine.html) object. The hasOption method takes a java.lang.String parameter and returns true if the option represented by the java.lang.String is present, otherwise it returns false.

if(cmd.hasOption("t")) {  
    // print the date and time  
}  
else {  
    // print the date  
}

### International Time

The InternationalDateApp utility extends the DateApp utility by providing the ability to print the date and time in any country in the world. To facilitate this a new command line option, c, has been introduced.

// add c option  
options.addOption("c", true, "country code");

The second parameter is true this time. This specifies that the c option requires an argument value. If the required option argument value is specified on the command line it is returned, otherwise null is returned.

### Retrieving the argument value

The getOptionValue methods of CommandLine are used to retrieve the argument values of options.

// get c option value  
String countryCode = cmd.getOptionValue("c");  
  
if(countryCode == null) {  
    // print default date  
}  
else {  
    // print date for country specified by countryCode  
}

## Ant Example

One of the most ubiquitous Java applications [Ant](http://ant.apache.org/) will be used here to illustrate how to create the Options required. The following is the help output for Ant.

ant [options] [target [target2 [target3] ...]]  
  Options:   
  -help                  print this message  
  -projecthelp           print project help information  
  -version               print the version information and exit  
  -quiet                 be extra quiet  
  -verbose               be extra verbose  
  -debug                 print debugging information  
  -emacs                 produce logging information without adornments  
  -logfile <file>        use given file for log  
  -logger <classname>    the class which is to perform logging  
  -listener <classname>  add an instance of class as a project listener  
  -buildfile <file>      use given buildfile  
  -D<property>=<value>   use value for given property  
  -find <file>           search for buildfile towards the root of the  
                         filesystem and use it

### Boolean Options

Lets create the boolean options for the application as they are the easiest to create. For clarity the constructors for Option are used here.

Option help = new Option( "help", "print this message" );  
Option projecthelp = new Option( "projecthelp", "print project help information" );  
Option version = new Option( "version", "print the version information and exit" );  
Option quiet = new Option( "quiet", "be extra quiet" );  
Option verbose = new Option( "verbose", "be extra verbose" );  
Option debug = new Option( "debug", "print debugging information" );  
Option emacs = new Option( "emacs",  
                           "produce logging information without adornments" );

### Argument Options

The argument options are created using the OptionBuilder.

Option logfile   = OptionBuilder.withArgName( "file" )  
                                .hasArg()  
                                .withDescription(  "use given file for log" )  
                                .create( "logfile" );  
  
Option logger    = OptionBuilder.withArgName( "classname" )  
                                .hasArg()  
                                .withDescription( "the class which it to perform "  
                                                  + "logging" )  
                                .create( "logger" );  
  
Option listener  = OptionBuilder.withArgName( "classname" )  
                                .hasArg()  
                                .withDescription( "add an instance of class as "  
                                                  + "a project listener" )  
                                .create( "listener");   
  
Option buildfile = OptionBuilder.withArgName( "file" )  
                                .hasArg()  
                                .withDescription(  "use given buildfile" )  
                                .create( "buildfile");  
  
Option find      = OptionBuilder.withArgName( "file" )  
                                .hasArg()  
                                .withDescription( "search for buildfile towards the "  
                                                  + "root of the filesystem and use it" )  
                                .create( "find" );

### Java Property Option

The last option to create is the Java property and it is also created using the OptionBuilder.

Option property  = OptionBuilder.withArgName( "property=value" )  
                                .hasArgs(2)  
                                .withValueSeparator()  
                                .withDescription( "use value for given property" )  
                                .create( "D" );

The map of properties specified by this option can later be retrieved by calling getOptionProperties("D") on the CommandLine.

### Create the Options

Now that we have created each [Option](http://commons.apache.org/proper/commons-cli/api-release/org/apache/commons/cli/Option.html) we need to create the [Options](http://commons.apache.org/proper/commons-cli/api-release/org/apache/commons/cli/Options.html) instance. This is achieved using the [addOption](http://commons.apache.org/proper/commons-cli/api-release/org/apache/commons/cli/CommandLine.html#addOptionorg.apache.commons.cli.Option) method of Options.

Options options = new Options();  
  
options.addOption( help );  
options.addOption( projecthelp );  
options.addOption( version );  
options.addOption( quiet );  
options.addOption( verbose );  
options.addOption( debug );  
options.addOption( emacs );  
options.addOption( logfile );  
options.addOption( logger );  
options.addOption( listener );  
options.addOption( buildfile );  
options.addOption( find );  
options.addOption( property );

All the preperation is now complete and we are now ready to parse the command line arguments.

### Create the Parser

We now need to create a CommandLineParser. This will parse the command line arguments, using the rules specified by the Options and return an instance of [CommandLine](http://commons.apache.org/proper/commons-cli/api-release/org/apache/commons/cli/CommandLine.html).

public static void main( String[] args ) {  
    // create the parser  
    CommandLineParser parser = new DefaultParser();  
    try {  
        // parse the command line arguments  
        CommandLine line = parser.parse( options, args );  
    }  
    catch( ParseException exp ) {  
        // oops, something went wrong  
        System.err.println( "Parsing failed.  Reason: " + exp.getMessage() );  
    }  
}

### Querying the commandline

To see if an option has been passed the hasOption method is used. The argument value can be retrieved using the getOptionValue method.

// has the buildfile argument been passed?  
if( line.hasOption( "buildfile" ) ) {  
    // initialise the member variable  
    this.buildfile = line.getOptionValue( "buildfile" );  
}

### Usage/Help

CLI also provides the means to automatically generate usage and help information. This is achieved with the [HelpFormatter](http://commons.apache.org/proper/commons-cli/api-release/org/apache/commons/cli/HelpFormatter.html) class.

// automatically generate the help statement  
HelpFormatter formatter = new HelpFormatter();  
formatter.printHelp( "ant", options );

When executed the following output is produced:

usage: ant  
-D <property=value>     use value for given property  
-buildfile <file>       use given buildfile  
-debug                  print debugging information  
-emacs                  produce logging information without adornments  
-file <file>            search for buildfile towards the root of the  
                        filesystem and use it  
-help                   print this message  
-listener <classname>   add an instance of class as a project listener  
-logger <classname>     the class which it to perform logging  
-projecthelp            print project help information  
-quiet                  be extra quiet  
-verbose                be extra verbose  
-version                print the version information and exit

If you also require to have a usage statement printed then calling formatter.printHelp( "ant", options, true ) will generate a usage statment as well as the help information.

## ls Example

One of the most widely used command line applications in the \*nix world is ls. Due to the large number of options required for ls this example will only cover a small proportion of the options. The following is a section of the help output.

Usage: ls [OPTION]... [FILE]...  
List information about the FILEs (the current directory by default).  
Sort entries alphabetically if none of -cftuSUX nor --sort.  
  
-a, --all                  do not hide entries starting with .  
-A, --almost-all           do not list implied . and ..  
-b, --escape               print octal escapes for nongraphic characters  
    --block-size=SIZE      use SIZE-byte blocks  
-B, --ignore-backups       do not list implied entries ending with ~  
-c                         with -lt: sort by, and show, ctime (time of last  
                           modification of file status information)  
                           with -l: show ctime and sort by name  
                           otherwise: sort by ctime  
-C                         list entries by columns

The following is the code that is used to create the [Options](http://commons.apache.org/proper/commons-cli/api-release/org/apache/commons/cli/Options.html) for this example.

// create the command line parser  
CommandLineParser parser = new DefaultParser();  
  
// create the Options  
Options options = new Options();  
options.addOption( "a", "all", false, "do not hide entries starting with ." );  
options.addOption( "A", "almost-all", false, "do not list implied . and .." );  
options.addOption( "b", "escape", false, "print octal escapes for nongraphic "  
                                         + "characters" );  
options.addOption( OptionBuilder.withLongOpt( "block-size" )  
                                .withDescription( "use SIZE-byte blocks" )  
                                .hasArg()  
                                .withArgName("SIZE")  
                                .create() );  
options.addOption( "B", "ignore-backups", false, "do not list implied entried "  
                                                 + "ending with ~");  
options.addOption( "c", false, "with -lt: sort by, and show, ctime (time of last "   
                               + "modification of file status information) with "  
                               + "-l:show ctime and sort by name otherwise: sort "  
                               + "by ctime" );  
options.addOption( "C", false, "list entries by columns" );  
  
String[] args = new String[]{ "--block-size=10" };  
  
try {  
    // parse the command line arguments  
    CommandLine line = parser.parse( options, args );  
  
    // validate that block-size has been set  
    if( line.hasOption( "block-size" ) ) {  
        // print the value of block-size  
        System.out.println( line.getOptionValue( "block-size" ) );  
    }  
}  
catch( ParseException exp ) {  
    System.out.println( "Unexpected exception:" + exp.getMessage() );  
}

<http://commons.apache.org/proper/commons-cli/properties.html>

# Option Properties

The following are the properties that each [Option](http://commons.apache.org/proper/commons-cli/api-release/org/apache/commons/cli/Option.html) has. All of these can be set using the accessors or using the methods defined in the [OptionBuilder](http://commons.apache.org/proper/commons-cli/api-release/org/apache/commons/cli/OptionBuilder.html).

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| opt | java.lang.String | the identification string of the Option. |
| longOpt | java.lang.String | an alias and more descriptive identification string |
| description | java.lang.String | a description of the function of the option |
| required | boolean | a flag to say whether the option **must** appear on the command line. |
| arg | boolean | a flag to say whether the option takes an argument |
| args | boolean | a flag to say whether the option takes more than one argument |
| optionalArg | boolean | a flag to say whether the option's argument is optional |
| argName | java.lang.String | the name of the argument value for the usage statement |
| valueSeparator | char | the character value used to split the argument string, that is used in conjunction with multipleArgs e.g. if the separator is ',' and the argument string is 'a,b,c' then there are three argument values, 'a', 'b' and 'c'. |
| type | java.lang.Object | the type of the argument |
| value | java.lang.String | the value of the option |
| values | java.lang.String[] | the values of the option |

<https://vageeshhoskere.wordpress.com/2011/02/26/command-line-argument-parsing-in-java/>

## [Command Line argument Parsing in Java](https://vageeshhoskere.wordpress.com/2011/02/26/command-line-argument-parsing-in-java/)

Posted on [February 26, 2011](https://vageeshhoskere.wordpress.com/2011/02/26/command-line-argument-parsing-in-java/) by [Vageesh Hoskere](https://vageeshhoskere.wordpress.com/author/vageeshhg/)

[Apache commons-cli](http://commons.apache.org/cli/) can be used for parsing command line arguments that might be supplied to the java program. The program below is a sample-program that uses commons-cli’s CommandLineParser utility to parse the arguments.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | import org.apache.commons.cli.CommandLine;  import org.apache.commons.cli.CommandLineParser;  import org.apache.commons.cli.BasicParser;  import org.apache.commons.cli.Option;  import org.apache.commons.cli.Options;    public class CommandParserUtility {        public static void main(String[] args) {          CommandLineParser parser =  new BasicParser();          Option helpOption = new Option("help",false," Program Help.");          Option authorOption = new Option("author",false," Program Author.");          Option userOption = new Option("user",true," Wish User.");          Options options = new Options();          options.addOption(authorOption);          options.addOption(userOption);          options.addOption(helpOption);          CommandLine cli = parser.parse(options, args);          if(cli.hasOption(helpOption.getOpt()))              System.out.println("CommandParserUtility Usage - \n \nCommandParserUtility -user <User Name> [-author (Show Author Name)][-help|h (Show Usage Help)]\n");          else if(cli.hasOption(userOption.getOpt()))              System.out.println("Hey "+cli.getOptionValue(userOption.getOpt())+". You are running the CommandParserUtility.");          else if(cli.hasOption(authorOption.getOpt()))              System.out.println("CommandParserUtility authored by Vageesh.");          else              System.out.println("CommandParserUtility Usage - \n \nCommandParserUtility -user <User Name> [-author (Show Author Name)][-help|h (Show Usage Help)]\n");          System.exit(0);      }    } |

In the above program, the command line options available are help, user and author.

|  |  |
| --- | --- |
| 1  2 | Option userOption = new Option("user",true," Wish User.");  Option helpOption = new Option("help",false," Program Help."); |

Specifies that the argument is user and the Boolean value true specifies that the option takes value when used as argument and in case of help/author, the Boolean value of false indicates that it does not take any arguments with it.

|  |  |
| --- | --- |
| 1 | cli.hasOption(helpOption.getOpt()) |

This code returns a Boolean indicating whether one of the specified options was passed as argument to the program. If you need a list of all arguments passed to the program that is not from the specified argument list, then the following code can be used which returns a list of all unprocessed arguments

|  |  |
| --- | --- |
| 1 | ArrayList<Object> res = cli.getArgList(); |

Finally the values specified for a particular argument can be accessed using the command –

|  |  |
| --- | --- |
| 1 | cli.getOptionValue (“user”); |

# <http://www.javaworld.com/article/2074849/core-java/processing-command-line-arguments-in-java--case-closed.html>

# Processing command line arguments in Java: Case closed

By Dr. Matthias Laux

JavaWorld | Aug 16, 2004 1:00 AM PT

Many Java applications started from the command line take arguments to control their behavior. These arguments are available in the string array argument passed into the application's static main() method. Typically, there are two types of arguments: options (or switches) and actual data arguments. A Java application must process these arguments and perform two basic tasks:

1. Check whether the syntax used is valid and supported
2. Retrieve the actual data required for the application to perform its operations

Often, the code that performs these tasks is custom-made for each application and thus requires substantial effort both to create and to maintain, especially if the requirements go beyond simple cases with only one or two options. The Options class described in this article implements a generic approach to easily handle the most complex situations. The class allows for a simple definition of the required options and data arguments, and provides thorough syntax checks and easy access to the results of these checks. New Java 5 features like generics and typesafe enums were also used for this project.

### Command line argument types

Over the years, I have written several Java tools that take command line arguments to control their behavior. Early on, I found it annoying to manually create and maintain the code for processing the various options. This led to the development of a prototype class to facilitate this task, but that class admittedly had its limitations since, on close inspection, the number of possible different varieties for command line arguments turned out to be significant. Eventually, I decided to develop a general solution to this problem.

In developing this solution, I had to solve two main problems:

1. Identify all varieties in which command line options can occur
2. Find a simple way to allow users to express these varieties when using the yet-to-be-developed class

Analysis of Problem 1 led to the following observations:

* Command line options contrary to command line data arguments—start with a prefix that uniquely identifies them. Prefix examples include a dash (-) on Unix platforms for options like -a or a slash (/) on Windows platforms.
* Options can either be simple switches (i.e., -a can be present or not) or take a value. An example is:

java MyTool -a -b logfile.inp

* Options that take a value can have different separators between the actual option key and the value. Such separators can be a blank space, a colon (:), or an equals sign (=):
* java MyTool -a -b logfile.inp
* java MyTool -a -b:logfile.inp

java MyTool -a -b=logfile.inp

* Options taking a value can add one more level of complexity. Consider the way Java supports the definition of environment properties as an example:

java -Djava.library.path=/usr/lib ...

* So, beyond the actual option key (D), the separator (=), and the option's actual value (/usr/lib), an additional parameter (java.library.path) can take on any number of values (in the above example, numerous environment properties can be specified using this syntax). In this article, this parameter is called "detail."
* Options also have a multiplicity property: they can be required or optional, and the number of times they are allowed can also vary (such as exactly once, once or more, or other possibilities).
* Data arguments are all command line arguments that do not start with a prefix. Here, the acceptable number of such data arguments can vary between a minimum and a maximum number (which are not necessarily the same). In addition, typically an application requires these data arguments to be last on the command line, but that doesn't always have to be the case. For example:

java MyTool -a -b=logfile.inp data1 data2 data3 // All data at the end

or

java MyTool -a data1 data2 -b=logfile.inp data3 // Might be acceptable to an application

* More complex applications can support more than one set of options:
* java MyTool -a -b datafile.inp
* java MyTool -k [-verbose] foo bar duh

java MyTool -check -verify logfile.out

* Finally, an application might elect to ignore any unknown options or might consider such options to be an error.

So, in devising a way to allow users to express all these varieties, I came up with the following general options form, which is used as the basis for this article:

<prefix><key>[[<detail>]<separator><value>]

This form must be combined with the multiplicity property as described above.

Within the constraints of the general form of an option described above, the Options class described in this article is designed to be the general solution for any command line processing needs that a Java application might have.

### The helper classes

The Options class, which is the core class for the solution described in this article, comes with two helper classes:

1. OptionData: This class holds all the information for one specific option
2. OptionSet: This class holds a set of options. Options itself can hold any number of such sets

Before describing the details of these classes, other important concepts of the Options class must be introduced.

#### Typesafe enums

The prefix, the separator, and the multiplicity property have been captured by enums, a feature provided for the first time by Java 5:

public enum Prefix {

DASH('-'),

SLASH('/');

private char c;

private Prefix(char c) {

this.c = c;

}

char getName() {

return c;

}

}

public enum Separator {

COLON(':'),

EQUALS('='),

BLANK(' '),

NONE('D');

private char c;

private Separator(char c) {

this.c = c;

}

char getName() {

return c;

}

}

public enum Multiplicity {

ONCE,

ONCE\_OR\_MORE,

ZERO\_OR\_ONE,

ZERO\_OR\_MORE;

}

Using enums has some advantages: increased type safety and tight, effortless control over the set of permissible values. Enums can also conveniently be used with genericized collections.

Note that the Prefix and Separator enums have their own constructors, allowing for the definition of an actual character representing this enum instance (versus the name used to refer to the particular enum instance). These characters can be retrieved using these enums' getName() methods, and the characters are used for the java.util.regex package's pattern syntax. This package is used to perform some of the syntax checks in the Options class, details of which will follow.

The Multiplicity enum currently supports four different values:

1. ONCE: The option has to occur exactly once
2. ONCE\_OR\_MORE: The option has to occur at least once
3. ZERO\_OR\_ONCE: The option can either be absent or present exactly once
4. ZERO\_OR\_MORE: The option can either be absent or present any number of times

More definitions can easily be added should the need arise.

#### The OptionData class

The OptionData class is basically a data container: firstly, for the data describing the option itself, and secondly, for the actual data found on the command line for that option. This design is already reflected in the constructor:

OptionData(Options.Prefix prefix,

String key,

boolean detail,

Options.Separator separator,

boolean value,

Options.Multiplicity multiplicity)

The key is used as the unique identifier for this option. Note that these arguments directly reflect the findings described earlier: a full option description must have at least a prefix, a key, and multiplicity. Options taking a value also have a separator and might accept details. Note also that this constructor has package access, so applications cannot directly use it. Class OptionSet's addOption() method adds the options. This design principle has the advantage that we have much better control on the actual possible combinations of arguments used to create OptionData instances. For example, if this constructor were public, you could create an instance with detail set to true and value set to false, which is of course nonsense. Rather than having elaborate checks in the constructor itself, I decided to provide a controlled set of addOption() methods.

The constructor also creates an instance of java.util.regex.Pattern, which is used for this option's pattern-matching process. One example would be the pattern for an option taking a value, no details, and a nonblank separator:

pattern = java.util.regex.Pattern.compile(prefix.getName() + key + separator.getName() + "(.+)$");

The OptionData class, as already mentioned, also holds the results of the checks performed by the Options class. It provides the following public methods to access these results:

int getResultCount()

String getResultValue(int index)

String getResultDetail(int index)

The first method, getResultCount(), returns the number of times an option was found. This method design directly ties in with the multiplicity defined for the option. For options taking a value, this value can be retrieved using the getResultValue(int index) method, where the index can range between 0 and getResultCount() - 1. For value options that also accept details, these can be similarly accessed using the getResultDetail(int index) method.

#### The OptionSet class

The OptionSet class is basically a container for a set of OptionData instances and also the data arguments found on the command line.

The constructor has the form:

OptionSet(Options.Prefix prefix,

Options.Multiplicity defaultMultiplicity,

String setName,

int minData,

int maxData)

Again, this constructor has package access. Option sets can only be created through the Options class's different addSet() methods. The default multiplicity for the options specified here can be overridden when adding an option to the set. The set name specified here is a unique identifier used to refer to the set. minData and maxData are the minimum and maximum number of acceptable data arguments for this set.

The public API for OptionSet contains the following methods:

**General access methods:**

String getSetName()

int getMinData()

int getMaxData()

**Methods to add options:**

OptionSet addOption(String key)

OptionSet addOption(String key, Multiplicity multiplicity)

OptionSet addOption(String key, Separator separator)

OptionSet addOption(String key, Separator separator, Multiplicity multiplicity)

OptionSet addOption(String key, boolean details, Separator separator)

OptionSet addOption(String key, boolean details, Separator separator, Multiplicity multiplicity)

**Methods to access check result data:**

java.util.ArrayList<OptionData> getOptionData()

OptionData getOption(String key)

boolean isSet(String key)

java.util.ArrayList<String> getData()

java.util.ArrayList<String> getUnmatched()

Note that the methods for adding options that take a Separator argument create an OptionData instance accepting a value. The addOption() methods return the set instance itself, which allows invocation chaining:

Options options = new Options(args);

options.addSet("MySet").addOption("a").addOption("b");

After the checks have been performed, their results are available through the remaining methods. getOptionData() returns a list of all OptionData instances, while getOption() allows direct access to a specific option. isSet(String key) is a convenience method that checks whether an options was found at least once on the command line. getData() provides access to the data arguments found, while getUnmatched() lists all options found on the command line for which no matching OptionData instances were found.

### The Options class

Options is the core class with which applications will interact. It provides several constructors, all of which take the command line argument string array that the main() method provides as the first argument:

Options(String args[])

Options(String args[], int data)

Options(String args[], int defMinData, int defMaxData)

Options(String args[], Multiplicity defaultMultiplicity)

Options(String args[], Multiplicity defaultMultiplicity, int data)

Options(String args[], Multiplicity defaultMultiplicity, int defMinData, int defMaxData)

Options(String args[], Prefix prefix)

Options(String args[], Prefix prefix, int data)

Options(String args[], Prefix prefix, int defMinData, int defMaxData)

Options(String args[], Prefix prefix, Multiplicity defaultMultiplicity)

Options(String args[], Prefix prefix, Multiplicity defaultMultiplicity, int data)

Options(String args[], Prefix prefix, Multiplicity defaultMultiplicity, int defMinData, int defMaxData)

The first constructor in this list is the simplest one using all the default values, while the last one is the most generic.

**Table 1: Arguments for the Options() constructors and their meaning**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | Value | Description | Default | | prefix | This constructor argument is the only place where a prefix can be specified. This value is passed on to any option set and any option created subsequently. The idea behind this approach is that within a given application, it proves unlikely that different prefixes will need to be used. | Prefix.DASH | | defaultMultiplicity | This default multiplicity is passed to each option set and used as the default for options added to a set without specifying a multiplicity. Of course, this multiplicity can be overridden for each option added. | Multiplicity.ONCE | | defMinData | defMinData is the default minimum number of supported data arguments passed to each option set, but it can of course be overridden when adding a set. | 0 | | defMaxData | defMaxData is the default maximum number of supported data arguments passed to each option set, but it can of course be overridden when adding a set. | 0 | |

In the constructors above, where only one integer argument is present (data), this value is used to set both defMinData and defMaxData to the same value. This means that the number of acceptable data arguments is fixed to exactly that number, and there is no acceptable range for that number.

Adding an option set is possible through these methods:

OptionSet addSet(String setName)

OptionSet addSet(String setName, int data)

OptionSet addSet(String setName, int minData, int maxData)

Again, the newly created set is returned to allow for subsequent invocation chaining of the addOption() methods.

Option sets can be accessed through these methods:

OptionSet getSet()

OptionSet getSet(String setName)

Note one important concept here: one default OptionSet instance does not need to be explicitly created. This instance is available through the getSet() method and is useful for simpler applications that require only one set. In this case, setting up the Options instance could look like this:

Options options = new Options(args);

options.getSet().addOption("a").addOption("b");

Under the hood, this default set is of course based on a standard OptionSet instance with a name given by:

public final static String DEFAULT\_SET = "DEFAULT\_OPTION\_SET";

Some convenience methods have been added to the Options class to simplify the creation of the same option for all known sets at the same time:

void addOptionAllSets(String key)

void addOptionAllSets(String key, Multiplicity multiplicity)

void addOptionAllSets(String key, Separator separator)

void addOptionAllSets(String key, Separator separator, Multiplicity multiplicity)

void addOptionAllSets(String key, boolean details, Separator separator)

void addOptionAllSets(String key, boolean details, Separator separator, Multiplicity multiplicity)

These options correspond directly to the addOption() methods described earlier for the OptionSet class. One case where I have found using these methods useful was an optional verbosity option (-v), which had to be available for all sets of an application:

options.addOptionAllSets("v", Multiplicity.ZERO\_OR\_ONE);

#### Perform the checks

Performing the actual checks of the command line arguments against the specified options for all sets is obviously a core component of the Options class. The following check methods are available:

boolean check(String setName)

boolean check(String setName, boolean ignoreUnmatched, boolean requireDataLast)

boolean check()

boolean check(boolean ignoreUnmatched, boolean requireDataLast)

The first two methods check the specified option set, whereas the latter two check the default option set. The two Booleans have the following meanings.

**Table 2: Arguments to the check() methods and their meanings**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | Value | Description | Default | | ignoreUnmatched | Specifies whether command line options for which no corresponding OptionData instance was created are acceptable. Applications can choose to ignore such unmatched options or react with an error. | false | | requireDataLast | Specifies whether the actual data arguments need to be the last arguments on the command line or whether they can be interspersed within the options. | true | |

Again, the introduction of these methods is based on the observations made early in the project about the requirements for a class such as Options.

Two more convenience methods are provided:

OptionSet getMatchingSet()

OptionSet getMatchingSet(boolean ignoreUnmatched, boolean requireDataLast)

These methods run the checks for each known OptionSet and return the first one, which is successfully checked.

The last public method in the list is:

String getCheckErrors()

During the checks, the check() methods write all observed problems into a StringBuffer, the value of which can then be accessed through the getCheckErrors() method. This method proves useful for debugging purposes, but applications can also use it to tell its users about the problem with the provided input.

The actual check process consists of the following steps:

1. Some trivial cases are caught. No options have been defined for the set to check, or no command line arguments have been provided.
2. All command line arguments are processed in a loop. Using java.util.regex's pattern-matching capabilities, these arguments are compared with the known options, and, if a match is found, the value and the detail information are retrieved for options expecting such information. All this information is stored in the OptionData instance that matched the option.
3. Any unmatched options are identified and stored in a list. In addition, the data arguments are identified and stored in another list.
4. The multiplicity is checked for all the options based on the number of matches found for each one.
5. The range of the data arguments is checked against the defined boundaries.
6. If desired, data arguments can be checked to verify whether they are last on the command line.
7. If desired, the presence of unmatched options are checked.

If all checks are successful, true returns. If, at any of the stages above, a check failure results, false returns immediately, and a comment explaining the problem is written to the error log (which is accessible through the getCheckErrors() method).

### Examples

The following examples are designed to demonstrate the use of the Options class, ranging from a simple case of an application requiring just one option set to a complex case, with many different option sets and multiplicities for the options.

#### Example 1: A simple case

The first example is a simple case that demonstrates how quickly a tool can leverage the capabilities of the Options class.

The command line syntax for this example looks like this:

java Example1 [-a] [-log=<logfile>] <inpfile> <outfile>

I used the standard syntax here, which denotes optional data (like [a]) with square brackets.

The code to handle these options can look like this:

Options opt = new Options(args, 2);

opt.getSet().addOption("a", Multiplicity.ZERO\_OR\_ONE);

opt.getSet().addOption("log", Separator.EQUALS, Multiplicity.ZERO\_OR\_ONE);

if (!opt.check()) {

// Print usage hints

System.exit(1);

}

// Normal processing

if (opt.getSet().isSet("a")) {

// React to option -a

}

if (opt.getSet().isSet("log")) {

// React to option -log

String logfile = opt.getSet().getOption("log").getResultValue(0);

}

...

String inpfile = opt.getSet().getData().get(0);

String outfile = opt.getSet().getData().get(1);

...

The Options instance is created, specifying that exactly two data arguments are required. After that, the two options are added with the multiplicity of ZERO\_OR\_ONE, which corresponds to the angle brackets. The checks are run by invoking check(), and if the checks are not successful, a usage description can be written.

Using Options.getSet().isSet(), you can easily check whether the options in square brackets have been specified, and the program can react accordingly. If -log was specified, that option's value is available from the OptionData instance's getResultValue() method.

The data arguments can be accessed using the getData() method on the default option set.

Actually, the code above can be further simplified by specifying a different default multiplicity directly in Options's constructor and by using invocation chaining for the options definition:

Options opt = new Options(args, Multiplicity.ZERO\_OR\_ONE, 2);

opt.getSet().addOption("a").addOption("log", Separator.EQUALS);

#### Example 2: A more complex case

This more complex example demonstrates using several OptionSet instances, different option multiplicities, and option details.

The command line syntax looks like this:

java Example2 -c [-v] [-D<detail>=<value> [...]] data1 data2

java Example2 -a [-v] [-check] data1 [data2] [data3]

java Example2 -d [-v] -k <kval> -t <tval> data1 data2 [data3] [data4]

So this tool has three main modes of operation, which are chosen by a (mandatory) option (either -c, -a, or -d).

The code could look like this:

Options opt = new Options(args, 2);

opt.addSet("cset").addOption("c").addOption("D", true, Separator.EQUALS,

Multiplicity.ZERO\_OR\_MORE);

opt.addSet("aset", 1, 3).addOption("a").addOption("check",

Multiplicity.ZERO\_OR\_ONE);

opt.addSet("dset", 2, 4).addOption("d").addOption("k",

Separator.BLANK).addOption("t", Separator.BLANK);

opt.addOptionAllSets("v", Multiplicity.ZERO\_OR\_ONE);

OptionSet set = opt.getMatchingSet();

if (set == null) {

// Print usage hints

System.exit(1);

}

Note how simple it is to capture this complex set of options!

The evaluation section could look like this (where System.out.println() calls have been inserted for clarity):

// This can be used for ALL sets since we added it using addOptionAllSets()

if (set.isSet("v")) {

System.out.println("v is set");

}

// Evaluate the different option sets

if (set.getSetName().equals("cset")) {

for (String d : set.getData())

System.out.println(d);

OptionData d = set.getOption("D");

for (int i = 0; i < d.getResultCount(); i++) {

System.out.println("D detail " + i + " : " + d.getResultDetail(i));

System.out.println("D value " + i + " : " + d.getResultValue(i));

}

} else if (set.getSetName().equals("aset")) {

for (String d : set.getData())

System.out.println(d);

if (set.isSet("check"))

System.out.println("check is set");

} else { // We \_know\_ it has to be the third set now

for (String d : set.getData())

System.out.println(d);

System.out.println(set.getOption("k").getResultValue(0));

System.out.println(set.getOption("t").getResultValue(0));

}

Even this relatively complex example can be handled easily with the Options class, and one particular benefit becomes clear here: no check code is required at the application level, since the Options class handles it. All relevant result data is accessible through a simple and convenient set of methods.

#### Example 3: A really complex case

For the third example, I decided to retrofit the Options class into the [URLManager](http://www.javaworld.com/article/2074849/core-java/processing-command-line-arguments-in-java--case-closed.html?page=2#resources) package. This package contains the three Java command line tools URLManage, URLCheck, and URLPublish, each of which takes a large set of options. The most complex case is URLManage, whose usage description looks like this:

Create a new entry in the DB: java URLManage [-v] -c <dbprop> <url> <desc> <context>

java URLManage [-v] -bc <dbprop> <urlfile>

Update the description of an entry in the DB: java URLManage [-v] -u <dbprop> <url> <desc>

Delete an entry from the DB: java URLManage [-v] -d <dbprop> <url>

Select URL entries from the DB: java URLManage [-v] -s <dbprop> <pattern>

java URLManage [-v] -sa <dbprop>

Select contexts from the DB: java URLManage [-v] -con <dbprop>

Init the tables in the DB: java URLManage [-v] -init <dbprop>

Delete the tables from the DB: java URLManage [-v] -drop <dbprop>

Add the URL to a specific context: java URLManage [-v] -ac <dbprop> <url> <context>

java URLManage [-v] -bac <dbprop> <confile>

Remove the URL from a specific context: java URLManage [-v] -rc <dbprop> <url> <context>

It turns out that the Options class can be used to handle these option sets with limited coding effort; the code resembles Example 2:

...

ml.options.Options options = new ml.options.Options(args, 1);

options.addSet("create", 4).addOption("c");

options.addSet("createBatch", 2).addOption("bc");

options.addSet("update", 3).addOption("u");

options.addSet("delete", 2).addOption("d");

options.addSet("select", 2).addOption("s");

options.addSet("addURL", 3).addOption("ac");

options.addSet("addURLBatch", 2).addOption("bac");

options.addSet("removeURL", 3).addOption("rc");

options.addSet("selectAll").addOption("sa");

options.addSet("contexts").addOption("con");

options.addSet("initTables").addOption("init");

options.addSet("deleteTables").addOption("drop");

options.addOptionAllSets("v", ml.options.Options.Multiplicity.ZERO\_OR\_ONE);

ml.options.OptionSet optionSet = options.getMatchingSet();

...

### Conclusion

This article describes a Java class that allows for the convenient processing of command line options for Java programs. The structure is flexible enough to handle even complex situations, while at the same time offering an API that allows for the definition of acceptable command line syntax with limited coding effort. The Options class provides all the checking algorithms required to ensure that acceptable sets of command line arguments are identified, which relieves application programmers of having to hand-code the same algorithms time and again. This class can add a lot of value to every Java application requiring command line options. If some capability is missing, I'd of course appreciate [feedback](http://www.javaworld.com/feedback).

Dr. Matthias Laux is a senior engineer for Sun Microsystems working in the Global SAP-Sun Competence Center in Walldorf, Germany. His main interests are Java and J2EE technology, architecture, and programming, as well as Web services and XML technology in general, databases, and performance and benchmarking. Although he also has a background in aerospace engineering and HPC/parallel programming, today his languages of choice are Java and Perl. He is a certified Solaris Administrator, Java Programmer, and Java Enterprise Architect.

### Learn more about this topic

* Download the source code that accompanies this article  
  <http://images.techhive.com/downloads/idge/imported/article/jvw/2004/08/jw-0816-command.zip>
* The URLManager package"Keeping Those Links Up-to-Date," Dr. Matthias Laux (developers.sun.com, September 2003)  
  <http://java.sun.com/developer/technicalArticles/Programming/linkupdate>
* For more articles on Java development tools, browse the **Development Tools** section of JavaWorld's Topical Index  
  <http://www.javaworld.com/channel_content/jw-tools-index.shtml>

<http://www.javaworld.com/article/2072482/command-line-parsing-with-apache-commons-cli.html>

# Command-line Parsing with Apache Commons CLI

By Dustin Marx JavaWorld | Nov 10, 2008 4:35 PM PT

From time to time, I find myself needing to handle [command-line arguments in Java](http://java.sun.com/docs/books/tutorial/essential/environment/cmdLineArgs.html) either for Java-based applications or for main() function implementations that provide a simple testing mechanism directly within the class being tested. The Java developer has many choices for [command-line parsing](http://www2.sys-con.com/ITSG/virtualcd/Java/archives/0404/kougiouris/index.html). When there is only one, two, or a small number of command-line arguments (especially if the presence or absence of a flag is all that is needed rather than an accompanying value), write a few lines of code to process these command-line options is not a big deal. When there are more options and/or some options have values, it is nice to access more sophisticated support for command-line parsing.

In this blog entry, I will look at using the [Apache Commons CLI](http://commons.apache.org/cli/) library, but there are numerous other choices such as [args4j](https://args4j.dev.java.net/), [TE-Code](http://te-code.sourceforge.net/) command line parsing, [CLAJR](http://clajr.sourceforge.net/) ([Command-Line Arguments with Java Reflection](http://sourceforge.net/projects/clajr)), [JArgs](http://jargs.sourceforge.net/), [JSAP](http://www.martiansoftware.com/jsap/) ([Java Simple Argument Processor](http://sourceforge.net/projects/jsap)), and [several others](http://java-source.net/open-source/command-line) ([even more here](http://javathink.blogspot.com/2008/04/why-does-parsing-arguments-in-java-suck.html)).

Although [Apache Commons CLI](http://commons.apache.org/cli/introduction.html) library is part of [Apache Commons](http://commons.apache.org/), it is a separate [(JAR) download](http://commons.apache.org/downloads/download_cli.cgi) from the JAR download for [Apache Commons Modeler](http://commons.apache.org/modeler/) and from the JAR download for [Apache Commons Lang](http://commons.apache.org/lang/) that I talked about in previous blog entries available [here](http://marxsoftware.blogspot.com/2008/07/jmx-model-mbeans-with-apache-commons.html) and [here](http://marxsoftware.blogspot.com/2008/11/apache-commons-tostringbuilder.html). For this blog entry, I am using CLI 1.1 because there is no anticipated release for CLI 2.0 (more details on this at the end of this entry).

I will demonstrate some very simple examples of Apache Common CLI and include some links to other resources on use of this library.

Two important classes in use of Apache Common CLI are the [org.apache.commons.cli.Option](http://commons.apache.org/cli/api-release/org/apache/commons/cli/Option.html) class and the closely related [org.apache.commons.cli.Options](http://commons.apache.org/cli/api-release/org/apache/commons/cli/Options.html) (contains multiple instances of the Option class). These classes are used to represent the expected command-line [options](http://commons.apache.org/cli/properties.html). The following two code snippets demonstrate setting up of an Options class for [Posix-style options](http://www.iam.ubc.ca/guides/javatut99/essential/attributes/_posix.html) and [GNU-style options](http://www.gnu.org/manual/gawk/html_node/Options.html).

**Using the Options Class with Multiple Option Instances**

/\*\*

\* Construct and provide Posix-compatible Options.

\*

\* @return Options expected from command-line of Posix form.

\*/

public static Options constructPosixOptions()

{

final Options posixOptions = new Options();

posixOptions.addOption("display", false, "Display the state.");

return posixOptions;

}

/\*\*

\* Construct and provide GNU-compatible Options.

\*

\* @return Options expected from command-line of GNU form.

\*/

public static Options constructGnuOptions()

{

final Options gnuOptions = new Options();

gnuOptions.addOption("p", "print", false, "Option for printing")

.addOption("g", "gui", false, "HMI option")

.addOption("n", true, "Number of copies");

return gnuOptions;

}

Note in the examples of setting up Options that there is no difference yet in the handling of [Posix-style versus GNU-style options](http://markmail.org/message/ftuptfdi2dct3id4). So far, the options can be treated the same.

Before moving onto demonstrating CLI's parsing of command-line arguments based on these anticipated options, it is worth noting CLI's support for usage information and help information via the [org.apache.commons.cli.HelpFormatter](http://commons.apache.org/cli/api-release/org/apache/commons/cli/HelpFormatter.html) class. This useful utility class contains methods such as overloaded versions of [printHelp](http://commons.apache.org/cli/api-release/org/apache/commons/cli/HelpFormatter.html#printHelp%28int,%20java.lang.String,%20java.lang.String,%20org.apache.commons.cli.Options,%20java.lang.String%29), overloaded versions of [printUsage](http://commons.apache.org/cli/api-release/org/apache/commons/cli/HelpFormatter.html#printUsage%28java.io.PrintWriter,%20int,%20java.lang.String%29), and several other output and related methods.

The following code snippet demonstrates a method that makes use of one of HelpFormatter's printUsage methods and one of that class's printHelp methods.

**printUsage() and printHelp()**

/\*\*

\* Print usage information to provided OutputStream.

\*

\* @param applicationName Name of application to list in usage.

\* @param options Command-line options to be part of usage.

\* @param out OutputStream to which to write the usage information.

\*/

public static void printUsage(

final String applicationName,

final Options options,

final OutputStream out)

{

final PrintWriter writer = new PrintWriter(out);

final HelpFormatter usageFormatter = new HelpFormatter();

usageFormatter.printUsage(writer, 80, applicationName, options);

writer.close();

}

/\*\*

\* Write "help" to the provided OutputStream.

\*/

public static void printHelp(

final Options options,

final int printedRowWidth,

final String header,

final String footer,

final int spacesBeforeOption,

final int spacesBeforeOptionDescription,

final boolean displayUsage,

final OutputStream out)

{

final String commandLineSyntax = "java -cp ApacheCommonsCLI.jar";

final PrintWriter writer = new PrintWriter(out);

final HelpFormatter helpFormatter = new HelpFormatter();

helpFormatter.printHelp(

writer,

printedRowWidth,

commandLineSyntax,

header,

options,

spacesBeforeOption,

spacesBeforeOptionDescription,

footer,

displayUsage);

writer.close();

}

The next code snippet shows some calls to the printHelp() and printUsage() methods shown above and is followed by a screen snapshot showing the output from running those.

System.out.println("-- USAGE --");

printUsage(applicationName + " (Posix)", constructPosixOptions(), System.out);

displayBlankLines(1, System.out);

printUsage(applicationName + " (Gnu)", constructGnuOptions(), System.out);

displayBlankLines(4, System.out);

System.out.println("-- HELP --");

printHelp(

constructPosixOptions(), 80, "POSIX HELP", "End of POSIX Help",

3, 5, true, System.out);

displayBlankLines(1, System.out);

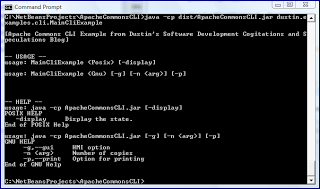
printHelp(

constructGnuOptions(), 80, "GNU HELP", "End of GNU Help",

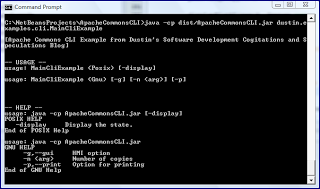
5, 3, true, System.out);

The first screen snapshot shows the results when the code above is executed exactly as shown (with true passed to both uses of the printHelp method to indicate that options should be included in the usage portion). The second screen snapshot shows what happens when the second call to printHelp has false passed to it so that the options are not displayed.

**printUsage and printHelp**

[](http://1.bp.blogspot.com/_sDOe5HxTdMk/SRkdxEbVPhI/AAAAAAAAAsA/nR1tfFgz_iM/s1600-h/helpAndUsageForCLIExampleBothOptionsUsage.png)

**printUsage and printHelp with One printHelp Not Displaying Options**

[](http://3.bp.blogspot.com/_sDOe5HxTdMk/SRkd8ueRMuI/AAAAAAAAAsI/SbyQAdoamDs/s1600-h/helpAndUsageForCLIExample.png)

While the usage and help information about the options is, as their names imply, helpful and useful, the real reason for using command-line arguments is usually to control the behavior of the application. The next code listing shows two methods for parsing GNU-style and Posix-style command-line arguments. While the setting up of the Options did not care about the specific style other than specifying the options themselves, the type of option is important now for determining the appropriate [parser](http://commons.apache.org/cli/api-release/org/apache/commons/cli/CommandLineParser.html) to use.

**usePosixParser() and useGnuParser()**

/\*\*

\* Apply Apache Commons CLI PosixParser to command-line arguments.

\*

\* @param commandLineArguments Command-line arguments to be processed with

\* Posix-style parser.

\*/

public static void usePosixParser(final String[] commandLineArguments)

{

final CommandLineParser cmdLinePosixParser = new PosixParser();

final Options posixOptions = constructPosixOptions();

CommandLine commandLine;

try

{

commandLine = cmdLinePosixParser.parse(posixOptions, commandLineArguments);

if ( commandLine.hasOption("display") )

{

System.out.println("You want a display!");

}

}

catch (ParseException parseException) // checked exception

{

System.err.println(

"Encountered exception while parsing using PosixParser:\n"

+ parseException.getMessage() );

}

}

/\*\*

\* Apply Apache Commons CLI GnuParser to command-line arguments.

\*

\* @param commandLineArguments Command-line arguments to be processed with

\* Gnu-style parser.

\*/

public static void useGnuParser(final String[] commandLineArguments)

{

final CommandLineParser cmdLineGnuParser = new GnuParser();

final Options gnuOptions = constructGnuOptions();

CommandLine commandLine;

try

{

commandLine = cmdLineGnuParser.parse(gnuOptions, commandLineArguments);

if ( commandLine.hasOption("p") )

{

System.out.println("You want to print (p chosen)!");

}

if ( commandLine.hasOption("print") )

{

System.out.println("You want to print (print chosen)!");

}

if ( commandLine.hasOption('g') )

{

System.out.println("You want a GUI!");

}

if ( commandLine.hasOption("n") )

{

System.out.println(

"You selected the number " + commandLine.getOptionValue("n"));

}

}

catch (ParseException parseException) // checked exception

{

System.err.println(

"Encountered exception while parsing using GnuParser:\n"

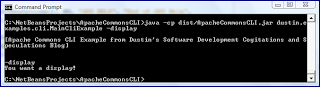
+ parseException.getMessage() );

}

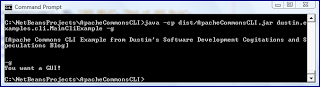
}

When the above code is executed, its output looks like that shown in the next two screen snapshots:

**PosixParser Results**

[](http://1.bp.blogspot.com/_sDOe5HxTdMk/SRke94kmB7I/AAAAAAAAAsQ/9NlTmlAgsMU/s1600-h/cliOutputPosixOption.png)

**GNU Parser Results**

[](http://1.bp.blogspot.com/_sDOe5HxTdMk/SRkfIqNDw0I/AAAAAAAAAsY/M0xgkvnU1KI/s1600-h/cliOutputGnuOption.png)

**The Complete Example**

The complete code for the example application from which portions were shown above is now listed for convenience.

package dustin.examples.cli;

import java.io.IOException;

import java.io.OutputStream;

import java.io.PrintWriter;

import org.apache.commons.cli.CommandLine;

import org.apache.commons.cli.CommandLineParser;

import org.apache.commons.cli.GnuParser;

import org.apache.commons.cli.HelpFormatter;

import org.apache.commons.cli.Options;

import org.apache.commons.cli.ParseException;

import org.apache.commons.cli.PosixParser;

/\*\*

\* Main example demonstrating Apache Commons CLI. Apache Commons CLI and more

\* details on it are available at <http://commons.apache.org/cli/>.

\*

\* @author Dustin

\*/

public class MainCliExample

{

private static Options options = new Options();

/\*\*

\* Apply Apache Commons CLI PosixParser to command-line arguments.

\*

\* @param commandLineArguments Command-line arguments to be processed with

\* Posix-style parser.

\*/

public static void usePosixParser(final String[] commandLineArguments)

{

final CommandLineParser cmdLinePosixParser = new PosixParser();

final Options posixOptions = constructPosixOptions();

CommandLine commandLine;

try

{

commandLine = cmdLinePosixParser.parse(posixOptions, commandLineArguments);

if ( commandLine.hasOption("display") )

{

System.out.println("You want a display!");

}

}

catch (ParseException parseException) // checked exception

{

System.err.println(

"Encountered exception while parsing using PosixParser:\n"

+ parseException.getMessage() );

}

}

/\*\*

\* Apply Apache Commons CLI GnuParser to command-line arguments.

\*

\* @param commandLineArguments Command-line arguments to be processed with

\* Gnu-style parser.

\*/

public static void useGnuParser(final String[] commandLineArguments)

{

final CommandLineParser cmdLineGnuParser = new GnuParser();

final Options gnuOptions = constructGnuOptions();

CommandLine commandLine;

try

{

commandLine = cmdLineGnuParser.parse(gnuOptions, commandLineArguments);

if ( commandLine.hasOption("p") )

{

System.out.println("You want to print (p chosen)!");

}

if ( commandLine.hasOption("print") )

{

System.out.println("You want to print (print chosen)!");

}

if ( commandLine.hasOption('g') )

{

System.out.println("You want a GUI!");

}

if ( commandLine.hasOption("n") )

{

System.out.println(

"You selected the number " + commandLine.getOptionValue("n"));

}

}

catch (ParseException parseException) // checked exception

{

System.err.println(

"Encountered exception while parsing using GnuParser:\n"

+ parseException.getMessage() );

}

}

/\*\*

\* Construct and provide Posix-compatible Options.

\*

\* @return Options expected from command-line of Posix form.

\*/

public static Options constructPosixOptions()

{

final Options posixOptions = new Options();

posixOptions.addOption("display", false, "Display the state.");

return posixOptions;

}

/\*\*

\* Construct and provide GNU-compatible Options.

\*

\* @return Options expected from command-line of GNU form.

\*/

public static Options constructGnuOptions()

{

final Options gnuOptions = new Options();

gnuOptions.addOption("p", "print", false, "Option for printing")

.addOption("g", "gui", false, "HMI option")

.addOption("n", true, "Number of copies");

return gnuOptions;

}

/\*\*

\* Display command-line arguments without processing them in any further way.

\*

\* @param commandLineArguments Command-line arguments to be displayed.

\*/

public static void displayProvidedCommandLineArguments(

final String[] commandLineArguments,

final OutputStream out)

{

final StringBuffer buffer = new StringBuffer();

for ( final String argument : commandLineArguments )

{

buffer.append(argument).append(" ");

}

try

{

out.write((buffer.toString() + "\n").getBytes());

}

catch (IOException ioEx)

{

System.err.println(

"WARNING: Exception encountered trying to write to OutputStream:\n"

+ ioEx.getMessage() );

System.out.println(buffer.toString());

}

}

/\*\*

\* Display example application header.

\*

\* @out OutputStream to which header should be written.

\*/

public static void displayHeader(final OutputStream out)

{

final String header =

"[Apache Commons CLI Example from Dustin's Software Development "

+ "Cogitations and Speculations Blog]\n";

try

{

out.write(header.getBytes());

}

catch (IOException ioEx)

{

System.out.println(header);

}

}

/\*\*

\* Write the provided number of blank lines to the provided OutputStream.

\*

\* @param numberBlankLines Number of blank lines to write.

\* @param out OutputStream to which to write the blank lines.

\*/

public static void displayBlankLines(

final int numberBlankLines,

final OutputStream out)

{

try

{

for (int i=0; i<numberBlankLines; ++i)

{

out.write("\n".getBytes());

}

}

catch (IOException ioEx)

{

for (int i=0; i<numberBlankLines; ++i)

{

System.out.println();

}

}

}

/\*\*

\* Print usage information to provided OutputStream.

\*

\* @param applicationName Name of application to list in usage.

\* @param options Command-line options to be part of usage.

\* @param out OutputStream to which to write the usage information.

\*/

public static void printUsage(

final String applicationName,

final Options options,

final OutputStream out)

{

final PrintWriter writer = new PrintWriter(out);

final HelpFormatter usageFormatter = new HelpFormatter();

usageFormatter.printUsage(writer, 80, applicationName, options);

writer.flush();

}

/\*\*

\* Write "help" to the provided OutputStream.

\*/

public static void printHelp(

final Options options,

final int printedRowWidth,

final String header,

final String footer,

final int spacesBeforeOption,

final int spacesBeforeOptionDescription,

final boolean displayUsage,

final OutputStream out)

{

final String commandLineSyntax = "java -cp ApacheCommonsCLI.jar";

final PrintWriter writer = new PrintWriter(out);

final HelpFormatter helpFormatter = new HelpFormatter();

helpFormatter.printHelp(

writer,

printedRowWidth,

commandLineSyntax,

header,

options,

spacesBeforeOption,

spacesBeforeOptionDescription,

footer,

displayUsage);

writer.flush();

}

/\*\*

\* Main executable method used to demonstrate Apache Commons CLI.

\*

\* @param commandLineArguments Commmand-line arguments.

\*/

public static void main(final String[] commandLineArguments)

{

final String applicationName = "MainCliExample";

displayBlankLines(1, System.out);

displayHeader(System.out);

displayBlankLines(2, System.out);

if (commandLineArguments.length < 1)

{

System.out.println("-- USAGE --");

printUsage(applicationName + " (Posix)", constructPosixOptions(), System.out);

displayBlankLines(1, System.out);

printUsage(applicationName + " (Gnu)", constructGnuOptions(), System.out);

displayBlankLines(4, System.out);

System.out.println("-- HELP --");

printHelp(

constructPosixOptions(), 80, "POSIX HELP", "End of POSIX Help",

3, 5, true, System.out);

displayBlankLines(1, System.out);

printHelp(

constructGnuOptions(), 80, "GNU HELP", "End of GNU Help",

5, 3, true, System.out);

}

displayProvidedCommandLineArguments(commandLineArguments, System.out);

usePosixParser(commandLineArguments);

//useGnuParser(commandLineArguments);

}

}

**Drawback of CLI: Version Issues**

One of the most significant drawbacks of Apache Commons CLI is the [CLI version paradox](http://journal.dedasys.com/2008/03/04/apache-commons-cli-and-the-paradox-of-choice) advertised on the [CLI's main page](http://commons.apache.org/cli/#a1.x_vs_2.x). This [main CLI page](http://commons.apache.org/cli/) points out that "the 2.x design is generally preferred" while also pointing out that, because there is no planned 2.0 release, "the 1.1 release is recommended to most users." I used CLI 1.1 for the examples in this blog entry.

**Conclusion**

The Apache Commons CLI is one of [many Java-based command-line argument parsing libraries](http://furiouspurpose.blogspot.com/2008/07/command-line-parsing-libraries-for-java.html) that is available to make writing text-based and command-line based Java applications and tests easier. This example has shown how to use CLI to implement command-line parsing along with output of help and usage information. However, CLI has many more options and uses than shown here. Some of these are demonstrated in the easy-to-read [CLI Usage Scenarios](http://commons.apache.org/cli/usage.html) document. Other useful introductions to Apache Commons CLI include [Parsing Simple Command Line Arguments in Java Using the Commons CLI Library](http://snippets.dzone.com/posts/show/3504), [Process the Command Line with CLI in Java](http://articles.techrepublic.com.com/5100-10878_11-5813561.html), and [Using the Jakarta Commons, Part 1](http://www.onjava.com/pub/a/onjava/2003/06/25/commons.html).

<http://www.programcreek.com/java-api-examples/index.php?api=org.apache.commons.cli.CommandLineParser>

# Java Code Examples for org.apache.commons.cli.CommandLineParser

The following code examples are extracted from open source projects. You can click http://www.programcreek.com/java-api-examples/includes/images/like.pngto vote up the examples you like. Your votes will be used in an intelligent system to get more and better code examples. If you want to read more examples, check out [javased](http://www.javased.com).

**Code Example 1:**

From project *anarchyape*, under directory */src/main/java/ape/*.

Source *Main.java*

/\*\*

\* This method would parse the array that store all arguments read from StdIn and return a CommandLine object

\*/

public static CommandLine getCommand(String[] args) throws ParseException {

if (args == null || args.length < 1 || args[0] == null) {

printHelp();

return null;

}

CommandLineParser parser=new PosixParser();

return parser.parse(opts,args);

}

**Code Example 2:**

From project *ADFS*, under directory */adfs-hdfs-project/adfs-hdfs/src/test/java/org/apache/hadoop/hdfs/*.

Source *TestFileAppend2.java*

@SuppressWarnings("static-access") public static void main(String[] args) throws Throwable {

Options options=new Options();

options.addOption(OptionBuilder.withLongOpt(OPT\_NUM\_DNS).hasArg().withDescription("Number of DNs to start").create());

options.addOption(OptionBuilder.withLongOpt(OPT\_NUM\_THREADS).hasArg().withDescription("number of threads to append from").create());

options.addOption(OptionBuilder.withLongOpt(OPT\_NUM\_FILES).hasArg().withDescription("number of files to append to").create());

options.addOption(OptionBuilder.withLongOpt(OPT\_NUM\_APPENDS).hasArg().withDescription("number of appends per thread").create());

CommandLineParser parser=new GnuParser();

CommandLine line;

try {

line=parser.parse(options,args);

if (line.getArgs().length != 0) {

throw new ParseException("Unexpected options");

}

}

catch ( ParseException pe) {

HelpFormatter formatter=new HelpFormatter();

formatter.printHelp("TestFileAppend2",options);

throw pe;

}

TestFileAppend2 tfa2=new TestFileAppend2();

tfa2.numDatanodes=Integer.parseInt(line.getOptionValue(OPT\_NUM\_DNS,"1"));

tfa2.numThreads=Integer.parseInt(line.getOptionValue(OPT\_NUM\_THREADS,"30"));

tfa2.numberOfFiles=Integer.parseInt(line.getOptionValue(OPT\_NUM\_FILES,"1"));

tfa2.numAppendsPerThread=Integer.parseInt(line.getOptionValue(OPT\_NUM\_APPENDS,"1000"));

tfa2.sleepBetweenSizeChecks=10;

try {

tfa2.testComplexAppend();

}

catch ( Throwable t) {

LOG.error("FAILED",t);

System.exit(1);

}

System.exit(0);

}

**Code Example 3:**

From project *android-rindirect*, under directory */rindirect/src/main/java/de/akquinet/android/rindirect/*.

Source *Main.java*

/\*\*

\* Main method. This methods defines the arguments, parse them and launch the R indirection generation.

\* @param args the arguments.

\* @throws ParseException

\* @throws IOException

\*/

public static void main(String[] args) throws ParseException, Exception {

LOGGER.setLevel(Level.WARNING);

Options options=new Options();

options.addOption("P","package",true,"destination package (mandatory)").addOption("R","classname",true,"generated java file [R]").addOption("D","destination",true,"the root of the destination [src]").addOption("I","input",true,"R file [searched in the 'gen' folder]").addOption("V","verbose",false,"Enable verbose mode");

CommandLineParser parser=new PosixParser();

CommandLine cmd=parser.parse(options,args);

RIndirect rindirect=configure(cmd);

rindirect.generate();

}

**Code Example 4:**

From project *asakusafw*, under directory */dsl-project/ashigel-compiler-bootstrap/src/main/java/com/asakusafw/compiler/bootstrap/*.

Source *OperatorCompilerDriver.java*

private static void start(String[] args) throws Exception {

CommandLineParser parser=new BasicParser();

CommandLine cmd=parser.parse(OPTIONS,args);

String sourcePath=cmd.getOptionValue(OPT\_SOURCEPATH.getOpt());

String output=cmd.getOptionValue(OPT\_OUTPUT.getOpt());

String encoding=cmd.getOptionValue(OPT\_ENCODING.getOpt(),"UTF-8");

String[] classes=cmd.getOptionValues(OPT\_CLASSES.getOpt());

List<Class<?>> operatorClasses=Lists.create();

for ( String className : classes) {

Class<?> oc=Class.forName(className);

operatorClasses.add(oc);

}

compile(new File(sourcePath),new File(output),Charset.forName(encoding),operatorClasses);

}

**Code Example 5:**

From project *ADFS*, under directory */adfs-common-project/adfs-common/src/main/java/org/apache/hadoop/util/*.

Source *GenericOptionsParser.java*

/\*\*

\* Parse the user-specified options, get the generic options, and modify configuration accordingly

\* @param conf Configuration to be modified

\* @param args User-specified arguments

\* @return Command-specific arguments

\*/

private String[] parseGeneralOptions(Options opts,Configuration conf,String[] args) throws IOException {

opts=buildGeneralOptions(opts);

CommandLineParser parser=new GnuParser();

try {

commandLine=parser.parse(opts,args,true);

processGeneralOptions(conf,commandLine);

return commandLine.getArgs();

}

catch ( ParseException e) {

LOG.warn("options parsing failed: " + e.getMessage());

HelpFormatter formatter=new HelpFormatter();

formatter.printHelp("general options are: ",opts);

}

return args;

}

**Code Example 6:**

From project *ADFS*, under directory */adfs-hdfs-project/adfs-hdfs/src/test/java/org/apache/hadoop/io/file/tfile/*.

Source *TestTFileSeek.java*

public MyOptions(String[] args){

seed=System.nanoTime();

try {

Options opts=buildOptions();

CommandLineParser parser=new GnuParser();

CommandLine line=parser.parse(opts,args,true);

processOptions(line,opts);

validateOptions();

}

catch ( ParseException e) {

System.out.println(e.getMessage());

System.out.println("Try \"--help\" option for details.");

setStopProceed();

}

}

**Code Example 7:**

From project *ADFS*, under directory */adfs-hdfs-project/adfs-hdfs/src/test/java/org/apache/hadoop/io/file/tfile/*.

Source *TestTFileSeqFileComparison.java*

public MyOptions(String[] args){

seed=System.nanoTime();

try {

Options opts=buildOptions();

CommandLineParser parser=new GnuParser();

CommandLine line=parser.parse(opts,args,true);

processOptions(line,opts);

validateOptions();

}

catch ( ParseException e) {

System.out.println(e.getMessage());

System.out.println("Try \"--help\" option for details.");

setStopProceed();

}

}

**Code Example 8:**

From project *AQuA*, under directory */kcl-content-apraisal/src/main/java/uk/bl/dpt/aqua/*.

Source *InventoryWorkflow.java*

/\*\*

\* @param args the command line args to the workflow

\*/

public static void main(String[] args){

CommandLineParser cmdParser=new GnuParser();

try {

CommandLine cmd=cmdParser.parse(OPTIONS,args);

if ((cmd.hasOption(HELP\_OPT)) || (cmd.getOptions().length < 1) || (!cmd.hasOption(DIR\_OPT))) {

HelpFormatter formatter=new HelpFormatter();

formatter.printHelp("inventory",OPTIONS);

System.exit(0);

}

File root=new File(cmd.getOptionValue(DIR\_OPT));

if ((!root.exists()) || (!root.isDirectory())) {

System.out.println("-dir value " + cmd.getOptionValue(DIR\_OPT) + " should be an existing directory.");

System.exit(0);

}

System.out.println("Started:" + new Date().toString());

InventoryWorkflow wrkflw=new InventoryWorkflow(root,true);

wrkflw.outputReport();

}

catch ( ParseException e) {

System.err.println("Command parsing failed. Reason: " + e.getMessage());

}

}

**Code Example 9:**

From project *atlas-lb\_1*, under directory */core-tools/core-crypto-tool/src/main/java/org/openstack/atlas/common/crypto/*.

Source *CryptoTool.java*

public static void main(String[] args) throws Exception {

final CommandLineParser parser=new PosixParser();

final Options options=getOptions();

CommandLine cmd=null;

try {

cmd=parser.parse(options,args);

}

catch ( ParseException e) {

System.err.println("Failed to parse command line: " + e.getMessage());

showUsage(options);

System.exit(1);

}

if (cmd.hasOption(HELP) || args.length == 0) {

showUsage(options);

System.exit(0);

}

if (cmd.hasOption(ENC)) {

final String valueToEncrypt=cmd.getOptionValue(ENC);

final String encryptedValue=CryptoUtil.encrypt(valueToEncrypt);

System.out.println("Encrypted value:" + encryptedValue);

System.exit(0);

}

if (cmd.hasOption(DEC)) {

final String valueToDecrypt=cmd.getOptionValue(DEC);

final String decypteddValue=CryptoUtil.decrypt(valueToDecrypt);

System.out.println("Decrypted value:" + decypteddValue);

System.exit(0);

}

System.exit(0);

}

**Code Example 10:**

From project *axis2-java*, under directory */modules/adb-codegen/src/org/apache/axis2/schema/*.

Source *XSD2Java.java*

/\*\*

\* for now the arguments this main method accepts is the source schema and the output location

\* @param args

\*/

@SuppressWarnings("static-access") public static void main(String[] args) throws Exception {

options=new Options();

options.addOption(OptionBuilder.withArgName(getMessage("schema.ns2p.argname")).hasArgs(2).withValueSeparator().withDescription(getMessage("schema.ns2p.description")).create("ns2p"));

options.addOption(OptionBuilder.withArgName(getMessage("schema.mp.argname")).hasArg().withDescription(getMessage("schema.mp.description")).create("mp"));

options.addOption(OptionBuilder.withArgName(getMessage("schema.dp.argname")).hasArg().withDescription(getMessage("schema.dp.description")).create("dp"));

options.addOption(OptionBuilder.withDescription(getMessage("schema.h.description")).create("h"));

options.addOption(OptionBuilder.withArgName(getMessage("schema.p.argname")).hasArg().withDescription(getMessage("schema.p.description")).create("p"));

CommandLineParser parser=new GnuParser();

try {

line=parser.parse(options,args);

}

catch ( ParseException ex) {

System.out.println(ex.getLocalizedMessage());

System.out.println();

printUsage();

System.out.println(ex);

System.exit(1);

}

args=line.getArgs();

if (args.length < 2) {

printUsage();

System.exit(1);

}

else {

File outputFolder=new File(args[args.length - 1]);

for (int i=0; i < args.length - 1; i++) {

File xsdFile=new File(args[i]);

if (args.length > 2) {

System.out.println(getMessage("schema.compiling",xsdFile.getName()));

}

compile(xsdFile,outputFolder);

}

}

}

**Code Example 11:**

From project *bagheera*, under directory */src/main/java/com/mozilla/bagheera/consumer/*.

Source *KafkaHBaseConsumer.java*

public static void main(String[] args){

OptionFactory optFactory=OptionFactory.getInstance();

Options options=KafkaConsumer.getOptions();

options.addOption(optFactory.create("tbl","table",true,"HBase table name.").required());

options.addOption(optFactory.create("f","family",true,"Column family."));

options.addOption(optFactory.create("q","qualifier",true,"Column qualifier."));

options.addOption(optFactory.create("pd","prefixdate",false,"Prefix key with salted date."));

CommandLineParser parser=new GnuParser();

ShutdownHook sh=ShutdownHook.getInstance();

try {

CommandLine cmd=parser.parse(options,args);

final KafkaConsumer consumer=KafkaConsumer.fromOptions(cmd);

sh.addFirst(consumer);

final KeyValueSink sink=new HBaseSink(cmd.getOptionValue("table"),cmd.getOptionValue("family","data"),cmd.getOptionValue("qualifier","json"),Boolean.parseBoolean(cmd.getOptionValue("prefixdate","true")));

sh.addLast(sink);

consumer.setSink(sink);

MetricsManager.getInstance();

consumer.poll();

}

catch ( ParseException e) {

LOG.error("Error parsing command line options",e);

HelpFormatter formatter=new HelpFormatter();

formatter.printHelp(KafkaHBaseConsumer.class.getName(),options);

}

}

**Code Example 12:**

From project *bagheera*, under directory */src/main/java/com/mozilla/bagheera/consumer/*.

Source *KafkaLoggerConsumer.java*

public static void main(String[] args){

OptionFactory optFactory=OptionFactory.getInstance();

Options options=KafkaConsumer.getOptions();

options.addOption(optFactory.create("lv","logvalues",false,"Log values."));

CommandLineParser parser=new GnuParser();

ShutdownHook sh=ShutdownHook.getInstance();

try {

CommandLine cmd=parser.parse(options,args);

final KafkaConsumer consumer=KafkaConsumer.fromOptions(cmd);

sh.addFirst(consumer);

final KeyValueSink sink=new LoggerSink(Boolean.parseBoolean(cmd.getOptionValue("logvalues","false")));

sh.addLast(sink);

consumer.setSink(sink);

MetricsManager.getInstance();

consumer.poll();

}

catch ( ParseException e) {

LOG.error("Error parsing command line options",e);

HelpFormatter formatter=new HelpFormatter();

formatter.printHelp(KafkaHBaseConsumer.class.getName(),options);

}

}

**Code Example 13:**

From project *bagheera*, under directory */src/main/java/com/mozilla/bagheera/consumer/*.

Source *KafkaSequenceFileConsumer.java*

public static void main(String[] args){

OptionFactory optFactory=OptionFactory.getInstance();

Options options=KafkaConsumer.getOptions();

options.addOption(optFactory.create("o","output",true,"HDFS base path for output."));

options.addOption(optFactory.create("df","dateformat",true,"Date format for the date subdirectories."));

options.addOption(optFactory.create("fs","filesize",true,"Max file size for output files."));

options.addOption(optFactory.create("b","usebytes",false,"Use BytesWritable for value rather than Text."));

CommandLineParser parser=new GnuParser();

ShutdownHook sh=ShutdownHook.getInstance();

try {

CommandLine cmd=parser.parse(options,args);

final KafkaConsumer consumer=KafkaConsumer.fromOptions(cmd);

sh.addFirst(consumer);

final KeyValueSink sink=new SequenceFileSink(cmd.getOptionValue("topic"),cmd.getOptionValue("output","/bagheera"),cmd.getOptionValue("dateformat","yyyy-MM-dd"),Long.parseLong(cmd.getOptionValue("filesize","536870912")),Boolean.parseBoolean(cmd.getOptionValue("usebytes","false")));

sh.addLast(sink);

consumer.setSink(sink);

MetricsManager.getInstance();

consumer.poll();

}

catch ( ParseException e) {

LOG.error("Error parsing command line options",e);

HelpFormatter formatter=new HelpFormatter();

formatter.printHelp(KafkaSequenceFileConsumer.class.getName(),options);

}

catch ( NumberFormatException e) {

LOG.error("Failed to parse filesize option",e);

}

catch ( IOException e) {

LOG.error("Error creating data sink",e);

}

}

**Code Example 14:**

From project *behemoth*, under directory */core/src/main/java/com/digitalpebble/behemoth/util/*.

Source *ContentExtractor.java*

public int run(String[] args) throws Exception {

Options options=new Options();

HelpFormatter formatter=new HelpFormatter();

CommandLineParser parser=new GnuParser();

options.addOption("h","help",false,"print this message");

options.addOption("i","input",true,"Behemoth corpus");

options.addOption("o","output",true,"local corpus dir");

options.addOption("b","binary",false,"dumps binary content, text otherwise");

options.addOption("n","filenaming",true,"whether to name files based on URL, UUID (default) or NUM");

try {

CommandLine line=parser.parse(options,args);

String input=line.getOptionValue("i");

String output=line.getOptionValue("o");

if (line.hasOption("help")) {

formatter.printHelp("ContentExtractor",options);

return 0;

}

if (input == null || output == null) {

formatter.printHelp("ContentExtractor",options);

return -1;

}

dumpBinary=line.hasOption("binary");

if (line.hasOption("filenaming")) {

String naming=line.getOptionValue("n");

mode=FileNamingMode.toMode(naming);

}

return generateDocs(input,output);

}

catch ( ParseException e) {

formatter.printHelp("ContentExtractor",options);

return -1;

}

}

**Code Example 15:**

From project *behemoth*, under directory */gate/src/main/java/com/digitalpebble/behemoth/gate/*.

Source *GATECorpusGenerator.java*

public int run(String[] args) throws Exception {

Options options=new Options();

HelpFormatter formatter=new HelpFormatter();

CommandLineParser parser=new GnuParser();

options.addOption("h","help",false,"print this message");

options.addOption("i","input",true,"Behemoth corpus");

options.addOption("o","output",true,"local GATE XML corpus dir");

try {

CommandLine line=parser.parse(options,args);

String input=line.getOptionValue("i");

String output=line.getOptionValue("o");

if (line.hasOption("help")) {

formatter.printHelp("GATECorpusGenerator",options);

return 0;

}

if (input == null || output == null) {

formatter.printHelp("GATECorpusGenerator",options);

return -1;

}

generateXMLdocs(input,output);

}

catch ( ParseException e) {

formatter.printHelp("GATECorpusGenerator",options);

}

return 0;

}

**Code Example 16:**

From project *behemoth*, under directory */mahout/src/main/java/com/digitalpebble/behemoth/mahout/util/*.

Source *ClusterDocIDDumper.java*

public int run(String[] args) throws Exception {

Options options=new Options();

HelpFormatter formatter=new HelpFormatter();

CommandLineParser parser=new GnuParser();

options.addOption("h","help",false,"print this message");

options.addOption("i","input",true,"input clusteredPoints");

options.addOption("o","output",true,"output doc cluster IDs");

CommandLine line=null;

try {

line=parser.parse(options,args);

if (line.hasOption("help")) {

formatter.printHelp("ClusterDocIDDumper",options);

return 0;

}

if (!line.hasOption("o") | !line.hasOption("i")) {

formatter.printHelp("ClusterDocIDDumper",options);

return -1;

}

}

catch ( ParseException e) {

formatter.printHelp("ClusterDocIDDumper",options);

}

Path inPath=new Path(line.getOptionValue("i"));

Path outPath=new Path(line.getOptionValue("o"));

int retVal=extract(inPath,outPath);

if (retVal != 0) {

HadoopUtil.delete(getConf(),outPath);

return retVal;

}

return 0;

}

**Code Example 17:**

From project *BitMate*, under directory */uis/src/org/gudy/azureus2/ui/common/*.

Source *Main.java*

private static CommandLine parseCommands(String[] args,boolean constart){

if (args == null) return null;

CommandLineParser parser=new PosixParser();

Options options=new Options();

options.addOption("h","help",false,"Show this help.");

OptionBuilder.withLongOpt("exec");

OptionBuilder.hasArg();

OptionBuilder.withArgName("file");

OptionBuilder.withDescription("Execute script file. The file should end with 'logout', otherwise the parser thread doesn't stop.");

options.addOption(OptionBuilder.create('e'));

OptionBuilder.withLongOpt("command");

OptionBuilder.hasArg();

OptionBuilder.withArgName("command");

OptionBuilder.withDescription("Execute single script command. Try '-c help' for help on commands.");

options.addOption(OptionBuilder.create('c'));

OptionBuilder.withLongOpt("ui");

OptionBuilder.withDescription("Run <uis>. ',' separated list of user interfaces to run. The first one given will respond to requests without determinable source UI (e.g. further torrents added via command line).");

OptionBuilder.withArgName("uis");

OptionBuilder.hasArg();

options.addOption(OptionBuilder.create('u'));

CommandLine commands=null;

try {

commands=parser.parse(options,args,true);

}

catch ( ParseException exp) {

Logger.getLogger("azureus2").error("Parsing failed. Reason: " + exp.getMessage(),exp);

if (constart) System.exit(2);

}

if (commands.hasOption('h')) {

if (constart) {

HelpFormatter hf=new HelpFormatter();

hf.printHelp("java org.gudy.azureus2.ui.common.Main","Optionally you can put torrent files to add to the end of the command line.\r\n",options,"Available User Interfaces: swt (default), web, console\r\nThe default interface is not started if you give either the '-e' or '-c' option (But you can start it by hand with '-u').",true);

System.exit(0);

}

}

return commands;

}

**Code Example 18:**

From project *BitMate*, under directory */uis/src/org/gudy/azureus2/ui/console/commands/*.

Source *OptionsConsoleCommand.java*

/\*\*

\* take the args and try and create a command line object

\*/

public void execute(String commandName,ConsoleInput console,List arguments){

CommandLineParser parser=getParser();

try {

String[] args=new String[arguments.size()];

int i=0;

for (Iterator iter=arguments.iterator(); iter.hasNext(); ) {

String arg=(String)iter.next();

args[i++]=arg;

}

CommandLine line=parser.parse(getOptions(),args);

execute(commandName,console,line);

}

catch ( ParseException e) {

console.out.println(">> Invalid arguments: " + e.getMessage());

printHelp(console.out,arguments);

}

}

**Code Example 19:**

From project *bonecp*, under directory */bonecp-benchmark/src/main/java/com/jolbox/benchmark/*.

Source *BenchmarkMain.java*

/\*\*

\* @param args

\* @throws ClassNotFoundException

\* @throws PropertyVetoException

\* @throws SQLException

\* @throws NoSuchMethodException

\* @throws InvocationTargetException

\* @throws IllegalAccessException

\* @throws InterruptedException

\* @throws SecurityException

\* @throws IllegalArgumentException

\* @throws NamingException

\* @throws ParseException

\*/

public static void main(String[] args) throws ClassNotFoundException, SQLException, PropertyVetoException, IllegalArgumentException, SecurityException, InterruptedException, IllegalAccessException, InvocationTargetException, NoSuchMethodException, NamingException, ParseException {

Options options=new Options();

options.addOption("t","threads",true,"Max number of threads");

options.addOption("s","stepping",true,"Stepping of threads");

options.addOption("p","poolsize",true,"Pool size");

options.addOption("h","help",false,"Help");

CommandLineParser parser=new PosixParser();

CommandLine cmd=parser.parse(options,args);

if (cmd.hasOption("h")) {

HelpFormatter formatter=new HelpFormatter();

formatter.printHelp("benchmark.jar",options);

System.exit(1);

}

Class.forName("com.jolbox.bonecp.MockJDBCDriver");

new MockJDBCDriver();

BenchmarkTests tests=new BenchmarkTests();

BenchmarkTests.threads=200;

BenchmarkTests.stepping=20;

BenchmarkTests.pool\_size=200;

System.out.println("JIT warm up");

tests.testMultiThreadedConstantDelay(0);

BenchmarkTests.threads=200;

BenchmarkTests.stepping=5;

BenchmarkTests.pool\_size=100;

if (cmd.hasOption("t")) {

BenchmarkTests.threads=Integer.parseInt(cmd.getOptionValue("t","400"));

}

if (cmd.hasOption("s")) {

BenchmarkTests.stepping=Integer.parseInt(cmd.getOptionValue("s","20"));

}

if (cmd.hasOption("p")) {

BenchmarkTests.pool\_size=Integer.parseInt(cmd.getOptionValue("p","200"));

}

System.out.println("Starting benchmark tests with " + BenchmarkTests.threads + " threads (stepping "+ BenchmarkTests.stepping+ ") using pool size of "+ BenchmarkTests.pool\_size+ " connections");

System.out.println("Starting tests");

plotLineGraph(tests.testMultiThreadedConstantDelay(0),0,false);

plotBarGraph("Single Thread","bonecp-singlethread-poolsize-" + BenchmarkTests.pool\_size + "-threads-"+ BenchmarkTests.threads+ ".png",tests.testSingleThread());

plotBarGraph("Prepared Statement\nSingle Threaded","bonecp-preparedstatement-single-poolsize-" + BenchmarkTests.pool\_size + "-threads-"+ BenchmarkTests.threads+ ".png",tests.testPreparedStatementSingleThread());

}

**Code Example 20:**

From project *bookkeeper*, under directory */hedwig-client/src/main/java/org/apache/hedwig/client/benchmark/*.

Source *HedwigBenchmark.java*

public static void main(String[] args) throws Exception {

Options options=new Options();

options.addOption("mode",true,"sub, recv, or pub");

options.addOption("nTopics",true,"Number of topics, default 50");

options.addOption("nMsgs",true,"Number of messages, default 1000");

options.addOption("nRegions",true,"Number of regsions, default 1");

options.addOption("startTopicLabel",true,"Prefix of topic labels. Must be numeric. Default 0");

options.addOption("partitionIndex",true,"If partitioning, the partition index for this client");

options.addOption("nPartitions",true,"Number of partitions, default 1");

options.addOption("replicaIndex",true,"default 0");

options.addOption("rate",true,"default 0");

options.addOption("npar",true,"default 100");

options.addOption("msgSize",true,"Size of messages, default 1024");

options.addOption("nwarmups",true,"Number of warmup messages, default 1000");

options.addOption("defaultHub",true,"Default hedwig hub to connect to, default localhost:4080");

CommandLineParser parser=new PosixParser();

final CommandLine cmd=parser.parse(options,args);

if (cmd.hasOption("help")) {

HelpFormatter formatter=new HelpFormatter();

formatter.printHelp("HedwigBenchmark <options>",options);

System.exit(-1);

}

ClientConfiguration cfg=new ClientConfiguration(){

public HedwigSocketAddress getDefaultServerHedwigSocketAddress(){

return new HedwigSocketAddress(cmd.getOptionValue("defaultHub","localhost:4080"));

}

public boolean isSSLEnabled(){

return false;

}

}

;

InternalLoggerFactory.setDefaultFactory(new Log4JLoggerFactory());

HedwigBenchmark app=new HedwigBenchmark(cfg,cmd);

app.call();

System.exit(0);

}

<http://stackoverflow.com/questions/11704338/java-cli-commandlineparser>

|  |
| --- |
| I am trying to use Java cli commanlineparser to parse the follwing arguments,  java -OC:\mydirectory -NMyfile  Option -O is for directory and -N is for the name of file.  I have been looking online but couldnt find a good example and this is what I am trying to do,  Option option = new Option()  option.addOpton("O",true, "output directory)  option.addOpton("N",true, "file name)  ...  CommandLineParser parser = new BasicParser();  ...  if (cmd.hasOption("O")  ...  Basically, I am trying to add multiple options and be able to parse them. Is this correct way to run the program with above options? |

## 1 Answer 28/07/2012

|  |
| --- |
| Try the following:  Option opt1 = OptionBuilder.hasArgs(1).withArgName("output directory")  .withDescription("This is the output directory").isRequired(true)  .withLongOpt("output").create("O");  Option opt2 = OptionBuilder.hasArgs(1).withArgName("file name")  .withDescription("This is the file name").isRequired(true)  .withLongOpt("name").create("N")  Options o = new Options();  o.addOption(opt1);  o.addOption(opt2);  CommandLineParser parser = new BasicParser();  try {  CommandLine line = parser.parse(o, args); // args are the arguments passed to the the application via the main method  if (line.hasOption("output") {  //do something  } else if(line.hasOption("name") {  // do something else  }  } catch(Exception e) {  e.printStackTrace();  }  ...  Also, you should leave a blank space between the argument and the value in the command line. |