

Comparaison de Jakarta Commons CLI et Args4j

par Gildas Cuisinier ([Hikage](#)) ([Blog](#))

Date de publication : 17/07/2007

Dernière mise à jour :

Afin d'éviter à tous les développeurs de gérer à la main le tableau d'arguments de la méthode *main*, il existe deux API dans le monde Open Source. La première est Jakarta Commons CLI, qui commence à avoir un certain âge, la seconde est Args4j qui est plus récente.

Cet article a pour but de vous montrer leur utilisation, ainsi que les avantages de chacune d'elles.

I - Introduction**II - Jakarta Commons CLI**

Téléchargement

Définition des options

Création via constructeur

Création grâce à OptionBuilder

Via une méthode de la classe Options

Analyse de la ligne de commandes

Utilisation

Utilisation avancée

Annexes

Listes des exceptions lancées par la méthode CommandLineParser.parse

Types de valeurs gérées pour un argument

III - Args4j

Téléchargement

Définitions des options

Traitement de la ligne de commandes

Utilisation des arguments

Affichage de l'usage

Création d'un Handler

Annexes : Handlers fourni par Args4j

IV - Comparaison des 2 API

Conclusion

V-A - Remerciements

I - Introduction

Afin de vous présenter les deux API, nous allons nous baser sur un exemple pratique.

Imaginons un serveur dont le port d'écoute doit être fourni, ainsi qu'un fichier optionnel dans lequel il écrirait ses traces d'exécution.

En plus, nous désirons pouvoir afficher l'usage de la commande, ainsi que le numéro de version du serveur.

L'usage de la commande donnerait ceci :

```
usage: MonServeur
-h                               Affiche l'aide
-l,--logfile <logfile>         Le fichier de log
-p,--port <port>               Le port sur lequel le serveur doit écouter
-v,--version                    Affiche la version du serveur
```

II - Jakarta Commons CLI

Téléchargement

Jakarta Commons CLI est disponible [ici](#), et nécessite Jakarta Commons Lang, disponible [ici](#), pour fonctionner.

Commons CLI est, bien sûr, sous licence Apache.

Dans le cas où vous utilisez Maven comme outil de gestion de projet, il vous suffit d'ajouter cette dépendance dans votre fichier *Pom* :

```
<dependencies>
  <dependency>
    <groupId>commons-cli</groupId>
    <artifactId>commons-cli</artifactId>
    <version>1.0</version>
  </dependency>
</dependencies>
```

Et dans le cas de Maven 1, la dépendance Commons Lang est nécessaire :

```
<dependencies>
  <dependency>
    <groupId>commons-lang</groupId>
    <artifactId>commons-lang</artifactId>
    <version>1.0</version>
  </dependency>
</dependencies>
```

Définition des options

La première phase dans Commons CLI est de définir les arguments possibles. Durant cette étape, nous allons aussi bien définir des arguments optionnels, qu'obligatoires.

Certains arguments nécessiteront d'avoir une valeur (le numéro de port, le chemin vers le fichier de log), et un type associé à celle-ci (ex : un entier pour le port).

Dans Commons CLI, un argument est représenté par une **Option**, elle-même stockée dans une classe **Options** qui représente la collection des **Option** que l'API devra pouvoir gérer.

Une **Option** possède plusieurs propriétés :

- Un nom court, par exemple **v** dans le cas de l'argument de version, qui sera représenté par **-v**
- Un nom long, par exemple **version** dans le cas de l'argument de version, qui sera représenté par **--version**
- Une description, qui sera affichée dans l'aide
- Un nom d'argument, qui sera affiché dans l'aide
- Un flag booléen qui définit si une Option est obligatoire
- Un flag booléen qui définit si une Option possèdera une valeur ou non
- Si oui, un champ pour stocker cette valeur
- Si oui, un champ pour définir le type d'objet que représente cette valeur

Afin de prendre en compte un ensemble d'**Option**, l'API fournit la classe **Options** :

```
Options options = new Options();
```

Maintenant que nous avons un conteneur pour nos classes *Option*, il faut les créer, et pour cela il existe trois possibilités :

- via le constructeur
- via la classe **OptionBuilder**
- via une méthode de la classe **Options**


Création via constructeur

La première est d'utiliser simplement les différents constructeurs fournis par la classe :

```
Option(String nomCourt, boolean possedeArgument, String description)  
Option(String nomCourt, String description)  
Option(String nomCourt, String nomLong, boolean possedeArgument, String description);
```

Ceux-ci créent une **Option** simple, qui est optionnelle, avec ou sans valeur. Il est ensuite possible d'utiliser les accesseurs pour paramétrer avec plus de détail l'**Option** :

```
Option port = new Option("p", "port", true, "Le port sur lequel le serveur doit écouter");  
// Spécification du nom du paramètre de l'argument qui sera affiché dans l'aide  
port.setArgName("port");  
// Spécification du type d'object que retournera getValue, ici un Number vu que c'est un port  
port.setType(Number.class);  
// Oblige le paramètre d'être présent lors de l'exécution  
port.setRequired(true);  
  
// On ajout ensuite l'Option dans la collection  
  
options.add(port);
```

 Dans le cas de l'option du port, vous vous demandez peut-être pourquoi utiliser un **Number** et non pas directement un **Integer**. Simplement que l'**Integer** n'est pas géré directement, mais Commons CLI va utiliser la classe **NumberUtils** de Commons Lang.

Cette classe va, selon le contenu de la variable, générer soit un **Integer**, soit un **Float**.

Création grâce à OptionBuilder

La deuxième méthode pour créer une option est d'utiliser **OptionBuilder**, celle-ci aide à la création d'**Option**. Toutes les méthodes sont statiques, et permettent de paramétrer une option, qui sera réellement instanciée lors de l'appel de la méthode statique **create(String nomOpt)**.

```
// Spécification du nom du paramètre de l'argument  
OptionBuilder.withArgName("file");  
// Spécification du nom long
```

```
OptionBuilder.withLongOpt("logfile");  
// Spécification du type de retour de la valeur du paramètre, ici c'est un fichier  
OptionBuilder.withType(File.class);  
// Spécification de la description  
OptionBuilder.withDescription("Le fichier de log");  
// Spécification de l'existence d'un paramètre à l'argument  
OptionBuilder.hasArg();  
// Crée l'argument en lui passant un nom court  
Option logfile = OptionBuilder.create("l");  
  
// Ajout de l'Option dans la collection  
options.add(logfile);
```

Via une méthode de la classe Options

Dans ces cas simples, il existe deux méthodes directement dans la classe **Options**.

```
public Options addOption(String opt, boolean hasArg, String description)  
public Options addOption(String opt, String longOpt, boolean hasArg, String description)
```

C'est pratique dans le cas d'une option simple booléenne (j'existe ou pas), telle que l'option d'affichage de l'aide ou de la version :

```
options.addOption("v", "version", false, "Affiche la version du serveur");  
options.addOption("h", false, "Affiche l'aide");
```

Analyse de la ligne de commandes

La deuxième étape dans l'utilisation de l'API CLI est l'analyse de la ligne de commandes, et cela se fait grâce à une implémentation de l'interface `CommandLineParser`. Commons CLI fournit deux implémentations de celle-ci :

- `PosixParser` : qui ne gère que des options à un seul caractère (**-v** ou **-h**)
- `GnuParser` : qui gère en plus les options à plusieurs caractères (**--version**, **--help**)

L'interface `CommandLineParser` possède deux méthodes :

```
public CommandLine parse( Options options, String[] arguments )  
    throws ParseException;  
  
public CommandLine parse( Options options, String[] arguments, boolean stopAtNonOption )  
    throws ParseException
```

Le premier paramètre de ces méthodes est la collection d'option qui a été créée précédemment, le second est le tableau d'arguments qui est fourni par la *méthode public static void main(String args)*. Le troisième argument spécifie comment doit réagir le parseur lorsqu'il rencontre une option non définie.

Si l'on passe la valeur true, alors il arrêtera le traitement de la ligne, mais les arguments qui ont déjà été traités sont tout de même gardés. Dans le cas contraire, une `UnrecognizedOptionException` sera lancée.

Cette méthode peut aussi lancer une exception `MissingOptionException` lorsqu'une option obligatoire n'est pas présente, ou une `MissingArgumentException` dans le cas où une option nécessitant une valeur et que celle-ci n'est pas présente.

Utilisation

Le résultat du traitement est un objet `CommandLine`. L'utilisation de celui-ci se fait via quelques méthodes :

Method	Description
<code>public boolean hasOption(String opt)</code>	Renvoie <i>vrai</i> si l'argument opt existe dans la ligne de commande
<code>public String getOptionValue(String opt)</code>	Renvoie la valeur de l'argument sous forme de String
<code>public Object getOptionObject(String opt) {</code>	Renvoie la valeur de l'argument sous forme d'instance d'un objet du type spécifié dans l'Option correspondante (ex : un File dans le cas du fichier de log) Si cela n'est pas possible, renvoie null

Le lanceur de `MonServeur` pourrait ressembler à ceci :

```
try {
    CommandLine cmd = parser.parse(options, args, false);

    if(cmd.hasOption("help")){
        // Affiche l'aide
    }
    if(cmd.hasOption("version")){
        System.out.println("MonServeur, version 0.1a");
    }

    Integer port = (Integer) cmd.getObject("port");
    File logfile = (File) cmd.getObject("logfile");

    MonServer server = new MonServeur(port, logfile);

} catch (ParseException e) {
    // Affichage de l'aide
}
```

Reste maintenant à afficher l'aide, ou l'usage du lanceur, et pour cela l'API fournit une méthode très simple :

```
HelpFormatter formatter = new HelpFormatter();
formatter.printHelp("MonServeur", options);
```

La classe `HelpFormatter` fournit la méthode `printHelp`, dont le premier paramètre est le nom de l'exécutable en lui-même (qui pourrait être remplacé par `"MonServeur.exe"` si l'on utilisait un outil tel que `exe4j`). Le second est tout simplement les options définies.

Utilisation avancée

En plus des arguments booléens simples, ou même avec valeurs, l'API fournit un moyen de spécifier que certaines options ne peuvent pas être présentes en même temps que d'autres.

Par exemple, il ne devrait pas être possible d'afficher à la fois la version, l'aide, et de lancer le programme. Ces trois options doivent être exclusives, si l'une est présente, aucune des deux autres ne doivent l'être.

Cette responsabilité est gérée par la classe `OptionGroup`, qui est aussi un conteneur d'`Option`, mais est aussi contenu dans le conteneur `Options`.

```
// Abandon de la creation des options par la méthode d'Options au profit de la version via
// constructeurs
Option help = new Option(ARGUMENT_HELP_SHORT, false, "Affiche l'aide");
Option version = new Option(ARGUMENT_VERSION_SHORT, false, "Affiche la version");

// Creation du groupe d'option
OptionGroup group = new OptionGroup();
// Ajout des options exclusives
group.addOption(help) ;
group.addOption(version);
group.addOption(port);

// Possibilite de rendre un groupe obligatoire
group.setRequired(true);

// Ajout du groupe dans le conteneur Options
options.addOptionGroup(group);
```

Annexes

Listes des exceptions lancées par la méthode `CommandLineParser.parse`

Exception	Cause
<code>MissingOptionException</code>	Un argument obligatoire n'a pas été fourni dans la ligne de commande
<code>MissingArgumentException</code>	Une valeur obligatoire pour un argument n'a pas été fourni
<code>AlreadySelectedException</code>	Plus d'une option d'un même groupe d'options exclusives a été fourni
<code>UnrecognizedOptionException</code>	Un argument qui n'a pas été déclaré dans les options possibles a été fourni

Types de valeurs gérées pour un argument

Classe	Description
<code>File.class</code>	Renvoie un objet <code>File</code> (<valeur de l'argument>)
<code>Number.class</code>	Renvoie un <code>Integer</code> si il n'y a pas de virgule dans la valeur de l'argument. Sinon renvoie un <code>Float</code>
<code>Class.class</code>	Renvoie une classe via <code>Class.forName(<valeur de l'argument>)</code>
<code>Object.class</code>	Tente d'instancier un objet de la classe <valeur de l'argument>
<code>Url.class</code>	Renvoie un <code>URL</code> (<valeur de l'argument>)
<code>Date.class</code>	Devrait renvoyer une <code>Date</code> , mais n'est pas implémenté dans la version stable

III - Args4j

Téléchargement

Args4j est une API plus récente, et nécessite d'ailleurs un JRE 1.5 au minimum. Cela est dû au fait qu'elle se base sur des annotations pour la définition des arguments.

Elle est disponible [ici](#), sous licence MIT.

Il est aussi possible de l'utiliser via Maven, mais cela nécessite l'ajout du repository *dev.java.net*, pour ce faire, il suffit de suivre [ces indications](#).

Ensuite, il suffit d'ajouter la dépendance Maven :

```
<dependencies>
  <dependency>
    <groupId>args4j</groupId>
    <artifactId>args4j</artifactId>
    <version>2.0.8</version>
  </dependency>
</dependencies>
```

Définitions des options

La première étape est aussi de définir les **Option**, mais à la différence de CLI, on se base sur des annotations.

```
public class MesOptions {
    @Option(name = "-h", aliases = {"--help"}, usage = "Affiche l'aide")
    private boolean help;

    @Option(name = "-v", aliases = {"--version"}, usage = "Affiche la version")
    private boolean version;

    @Option(name = "-l", aliases = {"--logfile"}, metaVar = "FILE", usage = "Specifie le fichier de trace")
    private File logfile;

    @Option(name = "-p", aliases = {"--port"}, metaVar = "PORT", required = true)
    private int port;

    @Argument
    private List<String> argument;
}
```

Chaque argument est donc annoté par **@Option**, qui prend divers paramètres tels qu'un nom, divers alias, un message d'usage.

L'annotation **@Argument** sert à spécifier une liste dans laquelle tout argument qui ne serait pas considéré comme une option sera inséré.

Traitement de la ligne de commandes

Il faut ensuite traiter la ligne de commandes en elle-même :

```

public static void main(String ... args){

    MesOptions options = new MesOptions();

    CmdLineParser parser = new CmdLineParser(options);

    try {
        parser.parseArgument(args);

        // Utilisation des arguments

    } catch (CmdLineException e) {
        // Affichage de l'aide
    }
}
    
```

Il suffit de gérer les exceptions lancées dans le cas où une exception obligatoire n'existe pas, quand un argument avec valeur n'en possède pas ou qu'une option non précisée a été passée.

Il est cependant intéressant de savoir comment les exceptions sont gérées, autrement dit, quelles sont les options qui seront traitées lorsqu'une exception aura été lancée ?

La réponse est : toutes celles qui se trouvaient avant la cause de l'exception.

Ligne de commandes	Descriptions du problème	Etat des options	Raisons
-p 42 -l c:\serveur.log --mauvaiseoption	Une mauvaise option est insérée à la fin	-p : Initialisée -l Initialisée	Le traitement a été correctement effectué afin d'atteindre le problème
--mauvaiseoption -p 42 -l c:\server.log	Une mauvaise option est insérée en tout premier argument	-p : non initialisée -l : non initialisée	L'exception a été lancée avant d'avoir pu traiter les bons arguments
-v -h -l c:\serveur.log	L'option -p obligatoire n'est pas fournie	-l : initialisée -v : initialisée -h : initialisée	La vérification des arguments obligatoires se fait après traitement de la ligne complète
-p TEST -l c:\serveur.log	La valeur de l'option -p n'est pas dans un format correct	-p : non initialisée -l : non initialisée	La traitement de -p a échoué, donc le traitement s'arrête

Utilisation des arguments

Ici, il n'y a pas de traitement spécial : en effet lors du traitement par Args4j, les valeurs des arguments ont été placées dans les champs associés, il suffit donc de travailler directement sur ceux-ci.

```

public static void main(String ... args){

    MesOptions options = new MesOptions();

    CmdLineParser parser = new CmdLineParser(options);

    try {
        parser.parseArgument(args);
    }
}
    
```

```
        if(options.isHelp()){
            // Affichage de l'aide
        }
        if(options.isVersion()){
            // Affichage de la version
        }

        MonServeur server = new MonServeur(options.getPort(), options.getLogfile() );

    } catch (CmdLineException e) {
        // Affichage de l'aide
    }
}
```

Affichage de l'usage

```
public static void main(String ... args){

    MesOptions options = new MesOptions();

    CmdLineParser parser = new CmdLineParser(options);

    try {
        parser.parseArgument(args);

        if(options.isHelp()){
            // Affichage de l'aide
        }
        if(options.isVersion()){
            // Affichage de la version
        }

        MonServeur server = new MonServeur(options.getPort(), options.getLogfile() );

    } catch (CmdLineException e) {

        parser.setUsageWidth(80);

        parser.printUsage(System.out);

    }
}
```

L'affichage de l'aide se fait via la méthode *printUsage(OutputStream)* de l'objet **CmdLineParser**:

Il est possible d'internationaliser l'aide, en passant un deuxième paramètre de type **RessourceBundle** à la méthode *printUsage*. Dès lors, s'il existe une entrée pour l'usage, celui-ci sera affiché dans la langue voulue.

Création d'un Handler

Une des valeurs ajoutées de Args4j par rapport à CLI est qu'elle permet de créer des handlers pour des types particuliers.

Il n'y a par exemple pas d'Handler pour les URL ou les Date, mais il est assez simple de les créer :

Handler d'URL

```
public class URLOptionHandler extends OptionHandler {

    public URLOptionHandler(CmdLineParser cmdLineParser, OptionDef optionDef, Setter setter) {
        super(cmdLineParser, optionDef, setter);
    }

    public int parseArguments(Parameters parameters) throws CommandLineException {
        try {

            // Récupération de la première valeur de l'option
            // Création d'une URL et ajout dans la liste des valeurs
            String value = parameters.getParameter(0);
            setter.addValue(new URL(value));
            // Renvoie le nombre de paramètre utilisé
            return 1;
        } catch (MalformedURLException e) {
            e.printStackTrace(); //To change body of catch statement use File | Settings | File
            Templates.
                throw new CommandLineException(e);
        }
    }

    public String getDefaultMetaVariable() {
        // Spécifie la valeur par défaut qui sera utilisée dans l'usage pour le paramètre de l'option
        return "URL";
    }
}
```

Handler de date

```
public class DateOptionHandler extends OptionHandler {

    public DateOptionHandler(CmdLineParser cmdLineParser, OptionDef optionDef, Setter setter) {
        super(cmdLineParser, optionDef, setter);
    }

    public int parseArguments(Parameters parameters) throws CommandLineException {

        try {
            String value = parameters.getParameter(0);
            SimpleDateFormat sdf = new SimpleDateFormat("yyyyMMdd");
            setter.addValue(sdf.parse(value));
            return 1;
        } catch (ParseException e) {
            throw new CommandLineException("Bad date format, use date like 20070625", e);
        }
    }

    public String getDefaultMetaVariable() {
        return "YYYYMMDD";
    }
}
```

Leur utilisation est tout aussi simple :

```
@Option(name="-d", handler = DateOptionHandler.class, usage = "Spécifie une date")
private Date date;
```

```
@Option(name="-u", handler = URLOptionHandler.class)  
URL url;
```

Annexes : Handlers fourni par Args4j

Classe / type	Handler
File	FileOptionHandler
String	StringOptionHandler
int	IntOptionHandler
double	DoubleOptionHandler
Enum	EnumOptionHandler

IV - Comparaison des 2 API

Voici un petit tableau récapitulatif des deux API :

Critère	Commons CLI	Args4j
Licence	Apache	MIT
Dépendance	Commons Lang	Java 5
Définition des options	Via des classes	Via des annotations
Gestion d'argument avec une valeur	Oui	Oui
Gestion de multivaleurs	Oui	Oui
Gestion de types personnels	Non	Oui
Gestion d'arguments exclusifs	Non	Oui
Gestion d'internationalisation de l'usage	Oui	Non

Conclusion

Comme on peut le voir, chacune des API a son lot d'avantages et d'inconvénients. Si CLI est utilisable dans la majorité des projets, elle n'est pas aussi complète dans certains cas.

De son côté, args4j utilise la force des annotations, ce qui nécessite un JRE 1.5 obligatoirement, ce qui peut être contraignant dans certains cas.

Il est bon à savoir aussi que Commons CLI est toujours en développement, et devrait un jour sortir une version 2, mais pour l'instant celle-ci n'est pas encore sortie. Les sources de celle-ci sont tout de même disponible via le serveur Subversion du projet (<http://svn.apache.org/viewvc/jakarta/commons/proper/cli/trunk/>).

Cela permet de tester les futures nouveautés de cette version.

V-A - Remerciements

Je tiens à remercier ma compagne de l'aide qu'elle m'apporte en relisant mes articles, ainsi que Bulbo pour sa relecture technique et Ricky81 pour sa relecture orthographique.

