jim.schmidt@dbexperts.com

August 8, 2010

Abstract

This article introduces a simple mechanism for declaratively creating a robust command line interface. This mechanism extends the functionality of the SourceForge jcmdline project available at http://jcmdline.sourceforge.net/ by providing declative arguments and options functionality using java annotations.

The jcmdline functionality is encapsulated and the underlying mechanism is subject to change.

The initial version this was written by one of my favorite programmers, Bryan Mason.

1 TODO

Support Multiple language error messages as in strings.properties

2 Introduction

The javautil-commandline contains classes used to parse and process command line options and arguments from a Java program. This package was written with the following goals:

- support option and argument processing described declarative fashion using java annotations in a java argument bean class
- support locale specific messages for command line processes using resource bundles
- Support the declaration, description and help displays for command line options in a declarative fashion using Java Annotations.
- Encourage uniformity of command line conventions and usage display throughout a project.
- Make it difficult to add a command line option without its being documented in the command usage.

2.1 Features

- Parses command line options and arguments.
- Supports frequently used parameter types - booleans, strings, integers, dates, file names...
- Performs common validation tasks - checks for number ranges, file/directory attributes, values that must be from a specified set, etc..
- Displays neatly formatted usage and error message in response to parameter specification errors.
- Provides support for commonly used command line options (such as -help and -version) and a means to standardize command line options across all executables of a project.
- Supports hidden options.
- Create Command Line Switch Processing with very little effort
  - Create an annotated class (a bean) for the arguments
  - Create a property file describing the arguments
  - Declare A CommandLineHandler with the bean as a constructor argument
  - Call the CommandLineHandler with the evaluateArguments method and the array of arguments
    - is this getting too deep?

3 Components

An emphargument bean that holds the arguments annotated with the rules.
An emphproperties file[s] that provide help messages and field descriptions.

4 The Argument Bean

Supported argument types:

Boolean	
Date	
File	
Integer	
String	

5 Annotations

DirectoryExists
DirectoryReadable
DirectoryWritable
Exclusive beginverbatim // not annotated should not be set private String parts; @Exclusive(property="parts") private String bits endverbatim
FileExists
FileReadable
FileWritable
Multivalue
Optional
Required
Requires
RequiredUnless

6 Terminology

Option

Argument

7 jcmdline

7.1 Overview

The jcmdline package contains classes used to parse and process command line options and arguments from a Java program. This package was written with the following goals:

- Simplify processing of command line options and parameters.
- Encourage uniformity of command line conventions and usage display throughout a project.
- Make it difficult to add a command line option without its being documented in the command usage.

7.2 Features

- Parses command line options and arguments.
- Supports frequently used parameter types - booleans, strings, integers, dates, file names...
- Performs common validation tasks - checks for number ranges, file/directory attributes, values that must be from a specified set, etc..
- Displays neatly formatted usage and error message in response to parameter specification errors.

- Provides support for commonly used command line options (such as -help and -version) and a means to standardize command line options across all executables of a project.
- Supports hidden options.

option A command line option is comprised of an identifying "tag", typically preceded by a dash or two, and optionally requires a value. A typical option might be "-outfile /tmp/myfile".

argument A command line argument is specified on the command line after all of the options and their values.

parameter A command line parameter is an option or an argument.

7.3 Parameter

7.3.1 tag

The tag is used to identify the parameter.

For options, the tag indicates how the option is specified, for instance, a tag of "outfile" indicates an option specified like "-outfile /tmp/myfile".

For command line arguments, the tag is typically used by the usage display formatter to identify the argument.

7.3.2 description

The description is used by the usage display formatter to describe a parameter. optional indicator. The optional indicator indicates whether a parameter is optional or required. If a parameter is configured to be required, the CmdLineHandler will reject the command line if the parameter is not specified.

7.3.3 multi-valued

A parameter that is multi-valued can be specified more than once. This is frequently used with the final command line argument when the command accepts multiple instances. The above example demonstrates this type of usage. acceptable values Most Parameter types allow the programmer to specify a set of values that the parameter will accept. The command line is rejected if a user attempts to set the parameter to any non-specified value.

7.4 hidden indicator

The hidden indicator is used to indicate whether a parameter is hidden or public. By default, all parameters are public, meaning that they will be described in a regular usage statement.

Optionally, a parameter may be designated hidden, in which case it may still be specified by the user, but is not displayed in a normal usage statement. Who has not shipped code supporting "debug" or "trace" options that might be performance impacting and thus not be suitable for publication to the casual user? This indicator is for that type of option.

Both the DefaultCmdLineHandler and HelpCmdLineHandler support (hidden) command line options that will cause hidden options and help to be displayed.

7.5 Help

8 Annotations

8.1 DirectoryExists

The argument represented by the field must specify an existing directory.

8.2 DirectoryReadable

The argument represented by the field must specify an existing readable directory.

8.3 DirectoryWritable

The argument represented by the field must specify an existing readable directory.

8.4 Exclusive

The argument represented by the specified field name is mutually exclusive with respect to the annotated field.

8.5 FieldValue

TODO this is completely unnecessary.

8.6 FileExists

Applied to a File object, the file specified file must exist.

8.7 FileReadable

Applied to a File object, the file must exist and be readable.

8.8 FileWritable

Applied to a File object, the file must exist and be writable.

8.9 MultiValue

Applied to a collection of objects when the argument may be specified more than once assumuning a collection of values.

ParamType must be one of FILE,INTEGER,STRING,DATE,BOOLEAN

8.10 Optional

Every argument must be emphOptional or emphRequired

8.11 RequiredBy

If the specified argument is on the command line, the member annoted with this value must be specified.

8.12 Required

Every argument must be *Optional* or *Required*

8.13 RequiredUnless

8.14 ResourceTag

8.15 StringSet

An enumeration of the assumable values of this tag.

8.16 Option Types

8.16.1 boolean

boolean (true / false) switch. Defaults to false; If specified on the commandline is true. Of course this argument type should be specified as optional.

8.16.2 File

Handles filename parameters

8.16.3 Int

Numeric Integer Parameters

8.16.4 Date

8.16.5 Time

8.16.6 DirectoryExists

8.16.7 DirectoryWritable

8.17 MultiValued

If an option may be specified multiple times use the emphMultiValue tag to annotate a Collection:

```
@Optional
@MultiValue
private Collection<String>words;
```

9 Cookbook

9.1 The Argument Bean

An ArgumentBean is any class that contains the variables to be set based on command line arguments augmented by annotations that specify the properties (see 8.16 on page 4)

10 Features

10.1 Multi Language Support

Different Resource bundles per language.

10.2 Messages and Help

11 Notes

Every field in the argument class must have an annotation. Why, I have no idea. but this isn't true for booleans?

Every argument should be specified in resource bundle.

The entry is of the form

Must be one of optional or mandatory.

11.1 ArgumentBean

Write a bean with objects and accessors.

Annotate each argument with either @Optional or @Required.

11.1.1 Example Argument Bean

```
package org.javautil.commandline;

import java.io.File;
import java.util.Collection;
import java.util.Date;

import org.javautil.commandline.annotations.MultiValue;
import org.javautil.commandline.annotations.Optional;
import org.javautil.commandline.annotations.Required;
import org.javautil.commandline.annotations.RequiredUnless;

public class ArgumentBean extends BaseArgumentBean {

        @Required
        private File file;

        @Optional
        @MultiValue(type = ParamType.DATE)
        private Collection<Date> postingDates;

        @MultiValue(type = ParamType.FILE)
        @Optional
        private Collection<File> files;

        @MultiValue(type = ParamType.INTEGER)
        @Optional
        private Collection<Integer> integers;

        public Collection<Integer> getIntegers() {
                return integers;
        }
```

```
public void setIntegers(Collection<Integer> integers) {
        this.integers = integers;
}

public Collection<String> getSecretWords() {
        return secretWords;
}

public void setSecretWords(Collection<String> secretWords) {
        this.secretWords = secretWords;
}

@Optional
@MultiValue(type = ParamType.STRING)
private Collection<String> words;

@RequiredUnless(property = "words")
private Collection<String> secretWords;

@Optional
private Date date;

/**
 * @return the file
 */
public File getFile() {
        return file;
}

/**
 * @param file
 *            the file to set
 */
public void setFile(File file) {
        this.file = file;
}

/**
 * @return the files
 */
public Collection<File> getFiles() {
        return files;
}

/**
 * @param files
 *            the files to set
 */
public void setFiles(Collection<File> files) {
        this.files = files;
}

/**
 * @return the words
 */
public Collection<String> getWords() {
        return words;
}
```

```
        /**
         * @param words
         *            the words to set
         */
        public void setWords(Collection<String> words) {
                this.words = words;
        }

        /**
         * @return the date
         */
        public Date getDate() {
                return date;
        }

        /**
         * @param date
         *            the date to set
         */
        public void setDate(Date date) {
                this.date = date;
        }

        public Collection<Date> getPostingDates() {
                return postingDates;
        }

        public void setPostingDates(Collection<Date> postingDates) {
                this.postingDates = postingDates;
        }

}
```

11.2 Properties File

Each object used should have an entry in a properties file with the name of the object followed by ".description"

Any argument or option not defined in the properties file will result in a message of the formatted *No description for 'argumentName' found*

Special words are

- application.description
- application.name
- application.helpText

12 Argument Bean

An argument bean is a class of Objects that hold the arguments used by the program.

12.0.1 Properties File

The argument bean should be described in a properties file.

By convention the file should be named the name of the bean followed by .properties.

13 Invocation

```
package org.javautil.commandline;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.util.Collection;
import java.util.Date;
import java.util.ResourceBundle;
import java.util.TreeSet;

import org.apache.log4j.BasicConfigurator;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;

/**
 * TODO this needs to go away, this has been highly simplified. TODO or show how
 * to use it the simple way an annotated command line example
 *
 * provides utility to escape a file's contents and echo it to System.out or
 * another file with specified characters escaped
 *
 * @author jjs
 */
public class CommandLineMain {

        private static String revision = "$Revision 1.0$";

        private Logger logger = Logger.getLogger(getClass());

        /**
         * stub for getting revision from build
         */
        public static String getBuildIdentifier() {
                String[] words = revision.split(" ");
                return words[1];
        }

        private Collection<String> getReverseSortedWords(
                        Collection<String> unsortedWords) {
                TreeSet<String> words = new TreeSet<String>();
                words.addAll(unsortedWords);
                return words.descendingSet();
        }

        private void doRun(ArgumentBean arguments) throws IOException {
                //File file = arguments.getFile();
                BufferedReader reader = new BufferedReader(new FileReader(arguments
                                .getFile()));

                Collection<File> files = arguments.getFiles();
                File file1 = files.iterator().next();
                logger.debug(file1 + ", " + file1.getClass().getName());

                Collection<Date> dates = arguments.getPostingDates();
                Date date = dates.iterator().next();
                logger.debug(date + ", " + date.getClass().getName());
```

```
                Collection<String> words = arguments.getWords();
                logger.debug(getReverseSortedWords(words));

                reader.close();

        }

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

                BasicConfigurator.configure();
                Logger logger = Logger.getRootLogger();
                logger.setLevel(Level.DEBUG);
                CommandLineMain invocation = new CommandLineMain();
                ArgumentBean arguments = new ArgumentBean();

                // CommandLineHandler cmd = new CommandLineHandler();
                // todo jjs should arguments and resource bundle be part of the
                // constructor?
                ResourceBundle resources = ResourceBundle
                                .getBundle("org/javautil/commandline/CommandLineArguments");
                CommandLineHandler cmd = new CommandLineHandler(resources, arguments);
                cmd.setArguments(arguments);
                cmd.setApplicationVersion(getBuildIdentifier());

                cmd.evaluateArguments(javaArgs);

                invocation.doRun(arguments);
        }
}
```

13.1 Optional

14 Exceptions

15 More

setDieOnParseError
    ignoreUnrecognizedOptions