OPTIONSMANAGER

Version 1.0.2

Table of Contents

Introduction....3
- Terms used by this document....3
  - Options Model....3
  - Options path....3
  - Top level model....3
- Options Models....3
  - Supported types....4
  - Arrays and Maps....4
  - Collections....5
  - The Annotations....5
    - @Option(name="…") / @Name("…")....6
    - @Option(description="...") / @Description("...")....6
    - @Option(required=true) / @Required....7
    - @Optional....7
    - @Option(type=MyType.class) / @Type(MyType.class)....7
    - @Option(validate="") / @Validate("expression=""")....7
    - @Option(validValues={"value", ...}) / @ValidValues({"value", ...})....7
- The options managers....7
  - CommandLineOptionsManager....7
  - PropertiesOptionsManager....8
  - XMLElementOptionsManager....8
  - XMLAttributeOptionsManager....8
  - Producing help text....9
- Exceptions (OptionsException, OptionsModelException, IOException)....9
- Maven Usage....9
- Writing your own OptionsManager subclass for other inputs....10
- Author / Contact....12
- License (Apache 2.0)....12
  - License text....12

Introduction

The idea for OptionsManager came (as usual) from a need. When working with another project I needed to have one set of options models and be able to populate them from maven, ant, and a command line tool. Maven and ant being as nice as they are didn't fight me on this. But when it came to command line arguments I had to manually parse the args string array and populate the model. That really annoyed me. I started thinking, what if you could load models automatically from command line arguments also ? The idea of OptionsManager was born.

OptionsManager supports instantiating and populating annotated (decided to require annotations to offer more flexibility) models from different forms of input. There are 4 variants provided with the package (so far): Command line, properties, XML elements only (like maven), XML elements and attributes (like ant). It is quite simple to create new variants for other type of input if needed.

These are very simple to use for handling options/config of both simple and complex structure. Each option can be documented using the annotations and a help text can be automatically generated from that.

Terms used by this document

Options Model

This refers to all annotated JavaBean models whose top level model Class object is passed to the constructor of each OptionsManager subclass and that will be instantiated and loaded with the option values.

Options path

This refers to a path from the top level options model down to a JavaBean property value. Internally each part of the path is separated by a '.'. This can be translated to something else externally by the OptionsManager subclass (or the user). For example the CommandLineOptionsManager sets it to "-". This allows arguments to be paths like "--thirdparty-license-type" where "Thirdparty", and "License" are options models, and "type" is a JavaBean property in the "License" model.

All exception messages that are result of bad indata when loading includes the options path to the item that failed, separated by the external path separator for the specific OptionsManager.

All key values in the input, no matter what type it is must match in some way the options path to a value. For OptionsManager subclasses that handles input in random order the options path is always a full path. For example CommandlineOptionsManager. For those that takes input in a sequential structured order the options path can be relative. For example the 2 XML OptionsManager variants where each child element can be seen as a relative path to its parent element.

Top level model

This is the options model that is specified as T in the generics declaration and also passed to the OptionsManager subclass constructor. This model can in turn have submodels. To differentiate between submodels and this model the term "top level (options) model" is used.

Options Models

The options models must be java beans with some supported extensions. Either the private field of the bean property or the public setter method can be annotated. The setter usually offers more flexibility, but in some cases a field is more useful. In general it is a matter of personal taste. It is always the setter that is used to populate a bean property with a value. You can change the name of a property either using @Option(name="name") or using @Name("name"). The setter must however match the annotated name so this is only useful if you annotate fields.

Supported types

The following bean property types are supported.

•All Java primitives + thier object equivalents.

•java.lang.String

•java.util.Date

> When it comes to Date OptionsManager needs to know the format of the date string to be able to parse it into a Date object. By default the following formats are supported:
>
> •YYYY-MM-DD HH:MM
>
> •YYYY-MM-DD
>
> •HH:MM
>
> If you need to support other Date formats, use the addSimpleDateFormat("…") or the addDateFormat(DateFormat format) method on the OptionsManager instance before calling loadOptions(…). OptionsManager will take each format in order and try until it succeeds or fails all.

•java.net.URL

•java.io.File

•Any enum

When loading a value for an enum property the value is case insensitive. Enum properties automatically validates input. An OptionsException will be thrown if the loaded value does not match one of the enum constants.

Arrays and Maps

Currently arrays and Map types are not supported.

Collections are supported, so use those instead of arrays.

For Map there is a possible workaround. Make a JavaBean model that internally sits on a Map, and has a key and value setter. When a key is received it should be temporarily saved, and when a value is received the key and value should be added to the map. This will only work with OptionsManager subclasses handling sequential input like XML. In this case annotate the key and value setter methods rather than the Map field.

Collections

Both value bean properties and submodel bean properties can be of a 'java.util.Collection' type for multiple entries. In this case the type of the object in the 'Collection' must be annotated since OptionsManager have no way of knowing what to put in there otherwise. (Generic declarations does unfortunately not reflect).

There are 3 ways to provide setters for 'Collection' type properties:

1.A setter that takes the whole 'Collection'. This is what you would expect when following the JavaBean standard.

In the Collection case, if the field name or name annotation is 'picture' then a setter method name of 'setPictures' will still be found.

For this type of setter there are 2 variants for specifying input:

1.A parent entry of name 'list' and entrys matching the annotated collection content type.

2.A parent entry name that matches the JavaBean property name and setter name and that must end with 's' with entries matching the annotated collection content type. If the content type matches the parent name but without the 's' it will be compatible with mavens way of loading collections.

2.A setter that takes just the type of the objects in the collection (that is a setter for a non 'Collection' property) but thats adds each set object to the internal 'Collection'. In this case the type annotation is not needed since OptionsManager gets it from the setter. This is still JavaBean legal as far as I know since the JavaBean specification only specifies setters and getters the internal storage is up to the bean, again as far as I know.

One reason for doing this is to make the input structure smaller. If we take the 'Collection' setter described above as an example we would have something like this:

```
<data>
        <pictures>
                <picture .../>
                <picture .../>
                ...
        </pictures>
</data>
```

But using the method described here would give you:

```
<data>
        <picture .../>
        <picture .../>
        ...
</data>
```

3.The same as the previous alternative, but instead of setMyProp(…) you have an addMyProp(…) which makes the code clearer than the previous setter. This of course completely breaks the JavaBean standard.

The Annotations

With the exception of @Option all other annotations are optional and most are alternatives for attributes of @option. All @Option attributes are optional (though some are required in certain situations).

•Option(…) - This is required on each field/setter that are supposed to be loaded by OptionsManager.

•OptionsModel(name="…", description="…") - This is used on a options model class and is optional. This can be used on submodels, but is of most use on the top level model.

•name="…" - Sets the name of the options model. The default will be the name of the property that holds the model in the parent or if this is the top level options model the name of the model class with the first character decapitalized.

You want to set the name="" when you dont want the top level options model name to be part of the option path. For example, when using the CommandLineOptionsManager you dont want to specify "--optionmodelname-optionpropertyname". It is much nicer for the user to have to specify "--optionpropertyname". Setting the top level options model name to "" accomplishes that.

The @Name("name") annotation can be used as an alternative to this.

•description="…" - Supplies a description of the model. This is only used for the top level options model and will be displayed in the generated help text before any options help texts. If the options model is not the top level model this will simply be ignored.

Please note that when you use the @OptionsModel on the top level model and only provide a description you will get a default name of "" which will still override the default non annotated name! This might not be what you intended.

The @Description("description") annotation can be used as an alternative to this, and will not affect the name.

@Option(name="…") / @Name("…")

This specifies the name of the option. If name is not set the name of the field is used if it is a field that is annotated. If it is a setter that is annotated the setter name without "set" and decapitalized will be used. Please note that if you annotate a field a different name than the fieldname can be specified for name="...". The setter must however follow the annotated name. Due to that if it is a setter that is annotated the name cannot differ from the setter name, and thus this is not very useful on a setter.

@Option(description="...") / @Description("...")

This provides a description of the option and are used when generating a help text.

It is possible to influence the formatting of the description on help output. The following are available:

•\\ or Description.NL

Forces a line break.

•\\ \\ or Description.NLNL

Forces 2 consective line breaks. This creates one empty line.

•\\m\\ or Description.ManualLineBreaks

Here after no automatic line breaks will done when line is larger than specified length.

•\\a\\ or Description.AutoLineBreaks

Here after automatic line breaks will be done again when line is larger than the specified length.

No more than 2 consecutive line breaks are allowed. Any more than that will simply be ignored.

Example:

```
@Description("This is my first paragraph." + Description.NLNL +
            "This is my second paragraph." +
            Description.ManualLineBreaks +
            "My very long and unbroken line until I break it.")
```

@Option(required=true) / @Required

This says that the option is required, and validateLoadedOptions() after a loadOptions(...) will throw an OptionsException if a required option has not been set.

@Optional

This is the opposite of @Required. All options are however optional unless specified as required, so this does not supply any functionallity, it only has retention SOURCE and are completely ignored. It is only documentative / cosmetic. It has no equivalent attribute in @Option.

@Option(type=MyType.class) / @Type(MyType.class)

If you have a JavaBean property whose type is a java.util.Collection then this must be specified to indicate what type is put into the collection.

@Option(validate="") / @Validate("expression=""")

This allows you to constrain values for options. When options are loaded the value will be matched against the reqular expression and an OptionsException thrown if it does not match. This can be used for any type of option since all values are provided as strings when loaded and this match is done before the value is converted to the real type.

@Option(validValues={"value", ...}) / @ValidValues({"value", ...})

This is an alternative to validate, and simply provides a list of valid values.

The options managers

CommandLineOptionsManager

This is intended for handling command line options, but it really only takes a String array of "key, value, ..., key, value" where the key is an otions path possibly prefixed with custom specified prefix (usually "-" or "--").

Multiple values for a single key is currently not supported, but something like "value,value,value" can be passed as a single value and your options model can parse it into separate values as a workaround.

Example usage:

```
  public void main(String[] args) {
    CommandLineOptionsManager<MyOptModel> clom =
      new CommandLineOptionsManager(MyOptModel.class);

    MyOptModel myOptModel =
      clom.loadOptions("--", "-", args);

    if (myOptModel.isHelp()) {
      clom.printHelpText("--","-",System.out);
    }
    else {
        ...
    }
  }
```

PropertiesOptionsManager

This loads an option model from a properties file or the properties XML format. Each key in the properties file is an options path separated by '.'.

Example usage:

```
  PropertiesOptionsManager<MyOptModel> pom =
    new PropertiesOptionsManager(MyOptModel.class);

  MyOptModel myOptModel =
    pom.loadOptions(myPropertiesFile);
```

XMLElementOptionsManager

This loads an option model from an XML file containing only elements (like the maven pom). This should actualy be more or less compatible with the section of a maven plugin configuration. That is you should be able to copy the content of thesection, stick it in its own xml file, and then load it with this class into the same model as a maven plugin does. That said this class has nothing to do with maven, it just uses the same XML format.

The XML file does not need to be DTD och XML Schema validated at all. It can be the simplest form of XML. It gets validated against the model when loaded, and model options can specify regular expressions that loaded values must match.

Using this or XMLAttributeOptionsManager is just a matter of personal taste in how you like to have your XML.

Example usage:

```
  XMLElementOptionsManager<MyOptmodel> xeom =
    new XMLElementOptionsManager(MyOptModel.class);

  MyOptModel myOptModel =
    xeom.loadOptions(myXMLStream);
```

XMLAttributeOptionsManager

This loads an option model from an XML file containing elements representing models and attributes representing value JavaBean proerties, and sub elements representing sub models. This is more or less the same format that Ant tasks use.

The XML file does not need to be DTD och XML Schema validated at all. It can be the simplest form of XML. It gets validated against the model when loaded, and model options can specify regular expressions that loaded values must match.

Using this or XMLElementOptionsManager is just a matter of personal taste in how you like to have your XML.

Example usage:

```
  XMLAttributeOptionsManager<MyOptmodel> xaom =
    new XMLAttributeOptionsManager(MyOptModel.class);

  MyOptModel myOptModel =
    xeom.loadOptions(myXMLStream);
```

Producing help text

There are 2 variants of help text generators. All take a prefix, path separator, and a PrintStream or a PrintWriter to produce output on.

•printHelpText(...)

This variant produces help text for only leafs of the model tree.

•printHelpTextFull(...)

This variant produces help text for all branches and leafs of the model tree.

Annotating the top level model with @OptionsModel and providing a description using the description attribute or @Description("...") will cause this to be displayed before all other properties descriptions. This is useful for a general help description.

Exceptions (OptionsException, OptionsModelException, IOException)

All public APIs throws OptionsException. You can get this on bad input when loading, problems parsing model when instantiating, and problems with model when loading.

There is however a subclass to OptionsException called OptionsModelException and this is thrown when there are any problems with the model class specified, like not finding a setter, not being allowed to instantiate a model, etc. Both the constructor and loadOptions(...) will throw this on model problems.

Model problems means that your model(s) are not entirely correct! A plain OptionsException means the end user has supplied bad input. You should concider handling these 2 cases separately in your code!

OptionsManager subclasses that do IO operations when loading options will also throw IOException.

Maven Usage

If you are using maven, add the following to your pom:

```
<project>
    ...
    <dependencies>
        ...
        <dependency>
```

```
            <groupId>se.natusoft.tools.optionsmgr</groupId>
            <artifactId>OptionsManager</artifactId>
            <version>1.0</version>
        </dependency>
    </dependencies>
    ...
    <repositories>
        ...
        <repository>
            <id>maven-natusoft-se</id>
            <name>Natusoft maven repository</name>
            <url>http://maven.natusoft.se/</url>
        </repository>
    </repositories>
    ...
</project>
```

Writing your own OptionsManager subclass for other inputs

Please note that the javadoc on the project site only shows the public user APIs, not the internals needed when doing your own subclass!

Declare a class extending OptionsManager.

```
public class MyOptionsManager<T> extends OptionsManager<T> {

  public MyOptionsManager(Class optionsModelClass)
      throws OptionsException {
    super(optionsModelClass);
  }

  ...
}
```

Then you need to do an inner class holding all the arguments passed to your loadOptions(...) methods. This class must implement the empty marker interface "Arguments".

Please note that this is a private inner class used nowhere else, so I consider it completely OK to not use setters and getters on this, but reference the fields directly, which can be done even if they are private.

```
private static class MyArguments implements Arguments {

  /** The prefix used to indicate an option. */
  private String argPrefix = "--";

  /** The command line args. */
  private String[] args;

  /** The delimeter of a component argument name that maps
      to a model structure. */
  private String modelSeparator = "-";

}
```

Provide a public loadOptions(...) user method for loading optons, create an instance of your Arguments, copy arg data to it, and then pass it to loadOptions(Arguments) in base class. If the loading of options from your input does not throw any IOException you can call the loadOptionsNoIO(Arguments) instead.

```
public T loadOptions(String argPrefix, String modSep,
    String[] args) throws OptionsException {

  MyArguments arguments = new MyArguments();
  arguments.argPrefix = argPrefix;
  arguments.modelSeparator = modelSeparator;
  arguments.args = args;

  return loadOptions(arguments);
}
```

Now you have to override one of the following 2 methods and provide your loading implmentation in it.

```
protected void loadOptions(OptionInfos optionInfos, Arguments arguments) throws
Exception;

protected void loadOptions(OptionModelInfo optionModelInfo, Arguments arguments) throws
Exception;
```

The first variant receives an OptionInfos object which contains all OptionInfo and OptionModelInfo created from parsing the model class. These are also used for instantiating and populating the model.

The OptionInfos class contains several methods for accessing the OptionInfo objects:

```
OptionInfo getOptionInfoByName(String name);

OptionInfo getOptionInfoByPublicPath(Path publicPath);

List<OptionInfo> getRequiredOptionInfos();

List<OptionInfo> getAllValueOptionInfos();

List<OptionInfo> getAllOptionInfos();
```

This is very useful when you are loding input of random paths. In this case you should also call

```
void assureModelInstance() ;
```

on an OptionInfo before trying to set a value on it (see further down).

The other variant that receives an OptionModelInfo is more useful for loding input of sequential paths. The OptionModelInfo received is the one representing the top level options model. It has 2 useful methods in addition to those inherited from OptionInfo:

```
List<OptionInfo> getChildren();

OptionInfo getChildByName(String name);
```

These either return all immediate children or a specific child by name, where the name can be seen as a relative options path.

OptionsModelInfo also contains the following method:

```
Object instantiateModelAndAddToParent();
```

This will create an instance of the model represented by the OptionsModelInfo and add that instance to its parent model (which is expected to have already been instantiated).

No matter if you have random or sequential input the setting of an option value is the same. The OptionInfo class (which OptionModelInfo extends) contains the following method:

```
void setValueAsString(String value) throws OptionsException;
```

This will validate the value if a @Option(validate="...") or @Validate("...") have been specified for the value, and then try to convert the string into the type of the JavaBean property, and finally call the property setter with the value.

The easiest way is to download the sourcecode and look at the existing OptionManager subclasses.

Author / Contact

My name is Tommy Svensson and I can be reached at tommy@natusoft.se.

License (Apache 2.0)

This code is open source and released under the Apache 2.0 software license available at http://www.apache.org/licenses/LICENSE-2.0.html and below.

License text

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2.**Grant of Copyright License.** Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3.**Grant of Patent License.** Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4.**Redistribution.** You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

You must give any other recipients of the Work or Derivative Works a copy of this License; and

You must cause any modified files to carry prominent notices stating that You changed the files; and

You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License. You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5.**Submission of Contributions.** Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.