Mango User Guide

GengoAI

Version v1.1

Table of Contents

1. Overview

A set of utilities and data structures that make java more convenient to use (in particular for Natural Language Processing, Text Mining, and Machine Learning). Mango is Apache 2.0 licensed allowing it to be used for whatever purpose.

2. Installation

Mango requires Java 11 and is available via the maven central repository at:

```
<dependency>
     <groupId>com.gengoai</groupId>
     <artifactId>mango</artifactId>
     <version>1.1</version>
</dependency>
```

Special Mango Annotations like Preload:

```
<dependency>
     <groupId>com.gengoai</groupId>
     <artifactId>mango-annoations</artifactId>
     <version>1.1</version>
</dependency>
```

Swing Applications and Helpers:

```
<dependency>
     <groupId>com.gengoai</groupId>
     <artifactId>mango-swing-ui</artifactId>
     <version>1.1</version>
</dependency>
```

SQL framework:

```
<dependency>
     <groupId>com.gengoai</groupId>
     <artifactId>mango-sql</artifactId>
     <version>1.1</version>
</dependency>
```

3. Collections

3.1. Maps

3.1.1. Multimaps

3.2. Trees

3.3. Indices

3.4. Counters

3.5. Disk-backed Collections

3.6. Tuples

4. Graphs

5. Key-Value Stores

6. Input / Output

6.1. Resource Framework

7. Json

8. Concurrency

9. Object Conversion and Casting

10. Reflection

11. Dynamic Enumerations

Dynamic enumerations are an enum-like objects that can have elements defined at runtime. Elements on a dynamic enumeration are singleton objects. In most cases it is acceptable to use the `==` operator for checking equality. There are two types of dynamic enumerations:

1. Flat enums - act in the same manner as Java enums
2. Hierarchical enums - each value is capable of having a single parent forming a tree structure with a single ROOT.

Both flat and hierarchical enums are uniquely defined by the label used to make them. Labels are restricted to only containing letters, digits, and underscores. Further, all labels are normalized to uppercase. Note that all labels should be unique within the dynamic enumeration.

Dynamic enumeration elements implement the `Tag` interface, which defines the `name()`, `label()`, and `isInstance(Tag)` methods. For flat enum elements these methods are all based on its normalized label, i.e. `name()` and `label()` return the normalized label and `isInstance(Tag)` checks that the given tag is of the same class and then checks for label name equality. However, hierarchical enum elements are defined with a label and a parent. Therefore, the `name()` method of hierarchical enum elements returns the full path from the ROOT (but not including the ROOT), e.g. if we have an element with label `ScienceTeacher` whose parent is `Teacher` which has ROOT as the parent, the name would be `Teacher$ScienceTeacher`. The `isInstance(Tag)` method will travese the hierarchy, such that the method would return true if we ask if `Teacher$ScienceTeacher` is an instance of `Teacher`.

11.1. Generating Dynamic Enumerations

The main method of the `EnumValue` class provides cli interface for bootstraping the creation of a dynamic enumeration. Usage is as follows:

```
 java EnumValue --className=<Name of Enum> --packageName=<Package to put the Class in> --src=<Source
directory>
```

The generated class will be placed in the provided source folder under the given package name. Optionally, a `-t` parameter can be passed to the command line to generate a hierarchical enum.

Core to the definition of both flat and hierarchical enumerations are:

1. **Registry** - The registry stores the defined elements.
2. **public static Collection<Colors> values()** - Acts the same as the `values()` method on a Java enum.
3. **public static Colors valueOf(String name)** - Acts the same as the `valueOf(String)` method on a Java enum.

In addition, the following make method is defined for flat enumerations: `public static TYPE` `make(String name)` The following make method is defined for hierarchical enumerations: `public static TYPE make(TYPE parent, String name)`

The supplied methods should not be removed. It is possible to update the logic to suit your needs, but removing the methods all together can result in problems.

11.2. Defining Elements

We can define elements by adding static final variables like the following for flat enumerations:

```
public static final Colors RED = make("RED");
public static final Colors BLUE = make("BLUE");
```

and the following for hierarchical enumerations:

```
public static final Entity ANIMAL = make(ROOT,"ANIMAL");
public static final Entity CANINE = make(ANIMAL,"CANINE");
```

In the case of hierarchical dynamic enumerations or flat enumerations that require other information, it is useful to use the Preload annotation on the class defining the elements. This will ensure that the elements are initialized at startup when using the Mango application.

12. Parameter Maps

Parameter maps are specialized maps that have predefined set of keys (parameters) where each key has an associated type and default value. They are useful to simulate "named and default parameters" found in other languages like Python. However, parameters defined in a parameter map are typed and will validate valeus of the correct type are being assigned. Parameter maps are implemented using the `ParamMap` class.

In order to define a `ParamMap`, you must first define the parameters. The first step is to construct a parameter definition (`ParameterDef`) that maps a parameter name to a type. Parameter definitions can be used by multiple `ParamMap`s. To construct a `ParameterDef`, we use one of the static methods as such:

```
public static final ParameterDef<String> STRING_PARAMETER = ParameterDef.strParam("stringParameter");
public static final ParameterDef<Boolean> BOOLEAN_PARAMETER = ParameterDef.boolParam("booleanParameter
");
```

With the parameters defined, we can now create a parameter map. Typically, you will want to subclass the `ParamMap` class setting its generic type to the class you are creating. You will want to define a set of public final variables of type `Parameter` that will map a parameter definition to a value. Each of the parameters has a default value associated with it, such that whenever the parameter map is used the calling method can be assured that a reasonable value for a parameter will be set. The following example illustrates the definition of a `MyParameters` parameter map with two parameters.

```
public class MyParameters extends ParamMap<MyParameters> {
 public final Parameter<String> stringParameter = parameter(STRING_PARAMETER, "DEFAULT");
 public final Parameter<Boolean> booleanParameter = parameter(BOOLEAN_PARAMETER, true);
}
```

Now we can define methods that utilize our `MyParameters` class. We can define the method to take a `MyParameters` object or to take a `Consumer`. Examples of this are as follows:

```
public void myMethod(MyParameters parameters) {
    System.out.println(parameters.<String>get(STRING_PARAMETER));
    System.out.println(parameters.<Boolean>get(BOOLEAN_PARAMETER));
}

public void myMethod2(Consumer<MyParameters> consumer) {
    myMethod(new MyParameters().update(consumer));
}
```

`ParamMap` have fluent accessors, so that we when using them as the argument to `myMethod`, we can do the following:

```
myMethod(new MyParameters().set(STRING_PARAMETER, "Set")
                           .set(BOOLEAN_PARAMETER, false));
```

We can also use the public fields directly:

```
myMethod(new MyParameters().stringParameter.set("SET")
                           .booleanParameter.set(false));
```

The `myMethod2` illustrates how we can mimic named parameters using `Consumer`s. Whe can call the method in the following manner:

```
myMethod2($ -> {
  $.stringParameter.set("Now is the time");
  $.booleanParameter.set(true);
});

//Or via fluent accessors
myMethod2($ -> $.stringParameter.set("Now is the time")
              .booleanParameter.set(true));
```

In addition to using the public variable, we can also set a parameter’s value using its name as follows:

```
myMethod2(p -> {
  p.set("stringParameter", "Now is the time");
  p.set("booleanParameter", true);
});
```

You can use inheritance to specialize your parameter maps, for example:

```
public abstract class BaseParameters<V extends BaseParameters<V> extends ParamMap<V> {
    public final Parameter<Integer> iterations = parameter(ITERATIONS, 100);
}

public class ClusterParameters extends BaseParameters<ClusterParameters> {
    public final Parameter<Integer> K = parameter(K, 2);
}

public class ClassifierParameters extends BaseParameters<ClassifierParameters> {
    public final Parameter<Integer> labelSize = parameter(LABEL_SIZE, 2);
}
```

Creates an abstract base parameter class (`BaseParameters`) which defines common parameters (`iterations`). Child classes (`ClusterParameters` and `ClassifierParameters`) then can add parameters specific to their use case. We can then construct a method which takes the `BaseParameters`, e.g. `train(BaseParameters<?> parameters)` which we during invocation we can send the correct set of parameters.

```
//Option 1 use the as method
public void train(BaseParameters<?> parameters) {
    ClassifierParameters cParameters = parameters.as(ClassifierParameters.class);
    int iterations = cParameters.get(ITERATIONS);
    int labelSize = cParameters.get(LABEL_SIZE);
}

//Option 2 use the getOrDefault methods
public void train(BaseParameters<?> parameters) {
    int iterations = parameters.get(ITERATIONS);
    int labelSize = parameters.getOrDefault(LABEL_SIZE,2);
}
```

When using the `BaseParameters` class we can cast the class to the correct instance type (e.g. `ClassifierParameters`) as shown in option 1 or use the `getOrDefault` methods on the `ParamMap` as shown in option2.

13. Tags

14. Parsing Framework

15. Specification Framework

16. Mango Streaming Framework

17. Logging

18. String Utilities

19. Application Framework

The application framework takes away much of the boilerplate in creating a command line or gui application, such as initializing configuration and command line parsing. Application has three abstract implementations: `CommandLineApplication` and `SwingApplication` (mango-swing). While Similar there are small differences in the use of these classes.

The following is an example of a command line application:

```
@Application.Description("My application example") public class MyApplication extends
CommandLineApplication {

  @Option(description = "The user name", required = true, aliases={"n"} )
  String userName

  @Option(name="age", description="The user age", required=true, aliases={"a"})
  int userAge

  @Override
  protected void programLogic() throws Exception {
    System.out.println("Hello " + userName + "! You are " + userAge + " years old!");
  }

  public static void main(String[] args){
    new MyApplication.run(args);
  }
}
```

The sample MyApplication class extends the `CommandLineApplication` class. Command line applications implement their logic in the programLogic method and should have the `run(args[])` method called in the main method. The super class takes care of converting command line arguments into local fields on MyApplication using the `@Option` annotation (for information on the specification see Command

Line Parsing). @Option annotations that do not have a name set use the field name as the command line option (e.g. --userName in the example above). In addition, the global "Config" (see Configuration for more information) instance is initialized using default configuration file associated with the package of the application. By default the application name is set to the class name. Note: the application name and associated default config package can be specified via a constructor by calling super.

A simple Swing application is defined as follows:

```
@Application.Description("My application example")
public class MySwingApplication extends SwingApplication {

  @Option(description = "The user name", required = true, aliases={"n"} )
  String userName

  @Option(name="age", description="The user age", required=true aliases={"a"})
  int userAge

  @Override
  public void setup() {
    //prepare your GUI
  }

  public static void main(String[] args){
    new MySwingApplication.run(args);
  }
}
```

Swing applications require the mango-swing libreary.

19.1. Preloading Static Elements

20. Configuration

The configuration format is a mix between json and java properties format. The need to know features are:

- The global Config object accesses properties from config files, the command line, and environment variables
- Comments with #
- Property names can be a combination of letters, digits, ".", and "_"
- Properties and their values are separated using = or :
- Property values can be referenced using ${propertyName}
- Beans can be referenced using @{beanName}

- Properties can be appended to using `+=`
- The `\` is used to escape characters in property value (especially useful for whitespace at the beginning of a value)
- The `\` at the end of a line with no spaces after it indicates a multiline property value (Same as java properties)
- Other config files can be imported using @import for example `@import com/mycompany/myapp/myconf.conf` by default the resource is considered to be a classpath resource

20.1. Sections

Sections avoid the need to retype the same prefix multiple times. For example:

```
remote {
   apis {
      search = google
      translate = bing
    }
   storage {
     text = s3
     search = solr
   }
}
```

would equate to the following individual properties being set:

```
tools.api.search = google
tools.api.translate = bing
tools.storage.text = s3
tools.storage.search = solr
```

20.2. Beans

Beans can be defined as follows:

```
ParentJohn {
    singleton=true
    class=com.mycompany.app.Parent
    constructor {
       param1 {
         type  = String
         value = John
       }
       param2 {
         type = String[]
         value = Same,Ryan,Billy
       }
    }
}
```

21. Command Line Parsing

Mango provides a posix-like command line parser that is capable of handling non-specified arguments. Command line arguments can be specified manually adding by adding a `NamedOption` via the `addOption(NamedOption)` method or automatically based on fields with `@Option` annotations by setting the parser's `owner` object via the constructor. The parser accepts long (e.g. `--longOption`) and short (e.g. `-s`) arguments. Multiple short (e.g. single character) arguments can be specified at one time (e.g. `-xzf` would set the x, z, and f options to true). Short arguments may have values (e.g. `-f FILENAME`). Long arguments whose values are not defined as being boolean require their value to be set. Boolean valued long arguments can specified without the true/false value. All parsers will have help (`-h` or `--help`), config (`--config`), and explain config (`--config-explain`) options added automatically.>

Values for options will be specified on the corresponding `NamedOption` instance. The value can be retrieved either directly from the `NamedOption` or by using the `get(String)` method. Argument names need not specify the `--` or `-` prefix.

An example of manually building a CommandLineParser is listed below:

```
CommandLineParser parser = new CommandLineParser();
parser.addOption(NamedOption.builder()
                               .name("arg1")
                               .description("dummy")
                               .required(true)
                               .type(String.class)
                               .build()
                   );
String[] notParsed = parser.parse(args)
```

An example of using fields to define your command line arguments is as follows:

```java
public class MyMain {

    @Option(description="The input file", required=true, aliases={"i"})
    String input;

    @Option(name ="l", description="Convert input to lowercase", default="false")
    boolean lowerCase;

    public static void main(String[] args){
        MyMain app = new MyMain();
        CommandLineParser parser = new CommandLineParser(app);
    }

}
```

22. Helpful Utilities, Classes, and Interfaces

`Copyable`	The Copyable interface defines a method for returning a copy of an object. Individual implementations are left to determine if the copy is deep or shallow. However, a preference is for deep copies.
`EncryptionMethod`	Convenience methods for encryption with common algorithms.
`Language`	Enumeration of world languages with helpful information on whether or not the language is Whitespace delimited or if language is read right to left (May not be complete)
`Stopwatch`	Tracks start and ending times to determine total time taken. (Not Thread Safe)
`MultithreadedStopwatch`	Tracks start and ending times to determine total time taken. (Thread Safe)
`Interner`	Mimics `String.intern()` with any object using heap memory. Uses weak references so that objects no longer in memory can be reclaimed.
`Lazy`	Lazily create a value in a thread safe manner.
`Validation`	Convenience methods for validating method arguments.