Mango User Guide

GengoAI

Version v1.1, ifndef: :sourcedir[:sourcedir: ../../main/java

Table of Contents

1. Overview

Mango is a roughly 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). It is Apache 2.0 licensed allowing it to be used for whatever purpose. Mango is similar to Guava, Apache Commons Lang, and Apache Commons Collections. The main reasons for creating Mango and not relying on existing libraries are:

1. These libaries (especially Guava) are widely used and often cause version conflicts. They (Guava) often have breaking changes, which can cause can code to not work at all or as intended. Shading jars is a possibility, but is not optimal.
2. Better control and integration into functionality not provided by this libaries.
3. Use of Java 8's built-in functional interfaces without the need for unused classes.

In addition, Mango provides helpful concepts, such as a generic Resources and easily distributed Mango Streams, utilities for converting from one type to another, and an easier to use wrapper around Java's reflection capabilities. Finally, Mango has an intuitive Configuration and Application framework that makes it simple to get command line and gui applications up and running.

2. Installation

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

Core Mango:

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

Swing Applications and Helpers:

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

2.1. Dependencies

2.1.1. Runtime

Library	License	URL
sqlite-jdbc (mango-sql)	Apache 2	https://github.com/xerial/sqlite-jdbc
Jackson	Apache 2	https://github.com/FasterXML/jackson
MapDB (1.0.9)	Apache 2	http://www.mapdb.org/
juniversalchardet	MPL	https://code.google.com/archive/p/juniversalchardet/
Apache Commons-Compress	Apache 2	https://commons.apache.org/proper/commons-compress/
flatlaf (mang-swing-ui)	Apache 2	https://www.formdev.com/flatlaf/

2.1.2. Compile

Library	License	URL
metainf-services	MIT	https://github.com/kohsuke/metainf-services
lombok	MIT	https://projectlombok.org/

2.1.3. Optional

Library	License	URL
Apache Spark	Apache 2	https://spark.apache.org/

3. Collections

Mango provides a variety of useful custom collections and convenience methods for working with collections. Some of the custom collections will be familiar to those who have used Guava or Apache Common Collections. Mango provides custom implementations as to not rely on these 3rd party libraries that are often used and result in version conflicts.

3.1. Utility Classes

Mango provides utility classes for accessing and manipulating the base Java collections. The following table lists the utility class for the given Java type.

Java Type	Mango Utility Class
`Iterable`	`Iterables`
`Iterator`	`Iterators`

Java Type	Mango Utility Class
`Collection`	`Collect`
`Stream`	`Streams`
`List`	`Lists`
`Set`	`Sets`
`Map`	`Maps`
`Array`	`Arrays2`

The utility classes provide methods for creating new instances and manipulating existing instances. Methods used to create an instance of a given type are defined as follows:

```
asTYPE(...) ①
TYPEOf(...) ②
```

① Converts an existing Java util type into the defined `TYPE` of collection, e.g. `asArrayList(Iterable<?>)` will convert an `Iterable` into an `ArrayList`.

② Creates a new instance of the defined `TYPE`, e.g. `hashSetOf(T⋯)` will create a `HashSet` containing the given items.

Methods for manipulating collections varying based on type. We refer the reader to the JavaDoc.

3.2. Counters

A counter is mapping from an `Object` to a `Double` where the double value represents a count. A double value is used in case the value represents a normalized count. Mango provides the `Counters` utility class for constructing new Counters. Currently, Mango provides a `HashMapCounter` and a `ConcurrentHashMapCounter` implementation.

In addition to the standard Counters, Mango provides a `MultiCounter` which maps a tuple of objects to a value. Mango provides the `MutliCounters` utility class for constructing new MultiCounters. Currently, Mango provides a `HashMapMultiCounter` and a `ConcurrentHashMapMultiCounter` implementation.

Both Counters and MultiCounters provide numerous methods to manipulate and query the counts of the objects. These methods include finding the top or bottom N items, filtering by key or value, and determining the minimum and maximum values and their associated objects.

3.3. Multimaps and Tables

A `Multimap` maps keys to multiple values. They act as a `Map<K, Collection<V>` where individual implementations specify the type of collection, e.g. List, Set, etc. Mango currently provides Multimap implementations wrapping the following collection types:

1. ArrayList
2. LinkedList
3. Set
4. LinkedHashSet
5. TreeSet

In all cases the backing map used is a `HashMap`.

Multimaps provide views over the keys and values the same as a Java Map. These views update the underlying Multimap when changed (e.g. items are deleted). Please see the JavaDoc for the full list of methods available on Multimaps.

A table is a two-dimensional structure that associates a value with two keys (i.e. a row and column key). A table maybe sparse, meaning not all cells contain values. Methods on a table that work with rows and columns return Map views that when updated will be reflected in table. Currently, Mango provides a single table implementation, `HashBasedTable`, which wraps Java's HashMap.

3.4. Trees

Mango provides two tree-based datastructures. The first is an `IntervalTree` which facilitates fast lookup of ranges including overlapping ranges. Interval Trees are a Set-like object which take subclasses of `Span` as their values. A `Span` defines a start and end range. Interval tress provide similar methods to those on a `NaviagableSet` with an additional method `overlapping(Span)` which provides fast lookup of all Spans in the tree that overlap with the given span.

Mango also provides a basic `Trie` implementation that facilitates fast prefix lookups in strings. The Trie implements the Map interface where the key is a String and the value can be defined per use. The Trie class provides some useful methods for suggesting the most similar strings given a maximum edit distance and finding all matches of the keys in the Trie in a given String.

3.5. Graphs

Mango provides a basic graph data structure which is currently has one implementation backed by a `Table`. Mango graphs can be defined as being directed or undirected by defining the `EdgeFactory` used by the graph. A number of graph algorithms and traversal strategies are implemented including, breadth-fist and depth-first search, Dijkstra's shortest path, and random walks. Additionally, Mango provides implementations of connected components and Chinese Whispers for clustering. Vertices can be scored by degree, Page Rank, and random walks using one the implementations of `VertexScorer`. Finally, graphs can be written to json or GraphViz dot format and rendered using GraphViz.

3.6. Caches

Mango provides a basic set of in-memory Caches to speed up IO intensive processes. The `Cache`

interface is inspired by Guava's Cache and has methods for getting, putting, and invalidating entries. There are currently two implemenations of `Cache`: `LRUCache` which keeps the last `N` most recently used items and `AutoCalculatingLRUCache` which extends `LRUCache` to auto-calculate missing values.

3.7. Key-Value Stores

Mango provides a generic interface for key-value stores and provides in-memory and disk-backed versions. A `KeyValueStore` defines a mapping from keys to values and extends the Java `Map` interface. Additionally, a `NavigableKeyValueStore` is defined in Mango that extends the Java `NavigableMap` interface.

Connections to key-value stores are done through a `KeyValueStoreConnection` as follows:

```
KeyValueStoreConnection connection = KeyValueStoreConnection.parse("kv:mem:people"); ①
KeyValueStore<String, String> kvStore = connection.connect(); ②
```

① Connections are defined by parsing a **specification** string defining the store.

② The `connect` method of the `KeyValueStoreConnection` provides an instance of a `KeyValueStore` or `NavigableKeyValueStore`.

The key-value store specification is defined as follows:

```
kv:(mem|disk):namespace::<PATH>?readOnly=(true|false)
```

Where `mem` creates an in-memory key-value store and `disk` a disk-based key-value store. The namespace defines the store name an allows multiple stores to be associated with a single file. The path is only required for disk-based key-value stores and the `readOnly` parameter is optional denoting if the store is read only (this is false by default).

4. Mango Streams

Mango streams provide a common interface for working with and manipulating streams regardless of their backend implementation. Currently, there are implementations that wrap Java's `Stream` and Spark's `RDD` classes. Mango supports 3 types of streams:

Stream Class	Description
`MStream<T>`	A stream of generic objects `T`.
`MPairStream<K,V>`	A stream of key-value pairs.
`MDoubleStream`	A stream of double values.

Mango provides `Serializable` versions of the Java functional interfaces which are used in calls to Mango streams in order to allow a common interface between Java streams and Spark streams.

4.1. Creating a Stream

Streams are created through a `StreamingContext`. A local streaming context is generated using `StreamingContext.local()` or through `StreamingContext.get(false)` whereas a distributed (Spark) contetext is retrieved by `StreamingContext.distributed()` or `StreamingContext.get(false)`. Streaming contexts provide a variety of ways for creating an MStream, including the following:

Method	Description
`empty()`	Creates an empty MStream
`doubleStream(double…)`	Creates an MDoubleStream over the given values.
`doubleStream(DoubleStream)`	Creates an MDoubleStream from the given Java double stream.
`stream(T…)`	Creates an MStream by converting the array into a List.
`stream(Iterator<T>)`	Creates an MStream over the given Iterator by treating the iterator as an Iterable. Note that local MStreams are not reusable.
`stream(Iterable<T>)`	Creates an MStream over the given Iterable. Note that if the Iterable can be iterated over multiple times, local MStreams will be reusable.
`stream(Stream<T>)`	Creates an MStream over the given Stream. Note that if the Stream can be iterated over multiple times, local MStreams will be reusable.
`textFile(String)`	Creates a new MStream where each element is a line in the resources (recursive) at the given location.
`textFile(Resource)`	Creates a new MStream where each element is a line in the resources (recursive) at the given location.
`textFile(Resource, boolean)`	Creates a new MStream where each element is the entire content of a resource (wholeFile = true) or a single line of the resource (wholeFile = False) and resources are gathered recursively from the given location.
`textFile(Resource, String)`	Creates a new MStream where each element is a line in the resources (recursive) at the given location only reading files matching the given pattern.
`pairStream(Collection<Entry<K,V>>)`	Creates an MPairStream over the collection of key-value pairs.

Method	Description
`pairStream(Map<K,V>)`	Creates an MPairStream over the key-value pairs in the map.
`pairStream(Tuple2<K,V>…)`	Creates an MPairStream over the array of key-value pairs.

Note that Mango also implements a reusable versions of Java's Stream classes (Stream, IntStream, DoubleStream, and LongStream). The reusable streams use a `Supplier` to provide the underlying stream. Individual methods, i.e. `map` then create a new supplier where the return stream applies the given method. Reusable streams can be created outside of MStreams using the `Streams` utility class.

4.2. Accumulators

An accumulator is a variable that can be used for aggregating values in a stream. As with streams, accumulators are created using a streaming context, e.g. `StreamingContext.local().counterAccumulator()`. Mango provides the following accumulator implementations:

Accumulator Class	Description
`MCounterAccumulator<T>`	Accumulator wrapping a Mango `Counter`.
`MDoubleAccumulator`	Accumulator wrapping a double value.
`MLongAccumulator`	Accumulator wrapping a long value.
`MMapAccumulator<K,V>`	Accumulator wrapping a Java Map.
`MStatisticsAccumulator`	Accumulator wrapping a Mango `EnhancedDoubleStatistics` for recording a series of double values and calculating descriptive statistics.
`MAccumulator<List<T>>`	Accumulator wrapping a Java list.
`MAccumulator<Set<T>>`	Accumulator wrapping a Java set.

Accumulators can have names associated with them, which will show up in the Spark interface. Additional accumulators can be created by implementing the base `MAccumulator` interface.

Streams should only update / modify the value of an accumulator and not try to read the value. While reading the value will work in local streams, distributed streams do not support reading. Thus, to make your logic reusable across stream types it is best to not read the values.

4.3. Working with Streams

Working with Mango streams is very similar to working with Java streams. The core operations are:

Operation	Description
`map(SerializableFunction<IN,OUT>)`	Transforms the items in the stream using the supplied function.
`mapToDouble(SerializableToDoubleFunction<IN>)`	Maps objects in this stream to double values
`mapToPair(SerializableFunction<IN,Map.Entry<K,V>)`	Transforms the MStream into a MPairStream by transforming individual items into tuples.
`flatMap(SerializableFunction<IN,Stream<OUT>>)`	Transforms the items in the stream to multiple items using the supplied function.
`flatMapToPair(SerializableFunction<IN,Stream<Map.Entry<K,V>>)`	Transforms the MStream into a MPairStream by transforming individual items into multiple tuples.
`filter(SerializablePredicate<IN>)`	Filters the item in the MStream to only those for which the given Predicate evaluates to `true`.
`distinct()`	Remove duplicate items from the stream.
`limit(long)`	Limits the stream to the first given number of items.
`skip(long)`	Skips the first given number of items in the stream.
`take(long)`	Takes the first given number of items in the stream.
`intersection(MStream<T>)`	Returns a new MStream containing the intersection of elements in this stream and the argument stream.
`union(MStream<T>)`	Returns a new MStream containing the union of elements in this stream and the argument stream.

Aggregation can be performed over streams using one of the following methods:

Operation	Description
`count()`	Gets the total number of items in the stream.
`countByValue()`	Provides a count per item in the stream by performing a group by.
`fold(T, SerializableBinaryOperator<T>)`	Performs a reduction on the elements of this stream using the given binary operator and given initial value.
`groupBy(SerializableFunction<IN,OUT>)`	Groups the items in the stream using the given function that maps objects to key values.
`reduce(SerializableBinaryOperator<T>)`	Performs a reduction on the elements of this stream using the given binary operator.
`max()`	Returns the max item in the stream requiring that the items be comparable.

Operation	Description
`max(SerializableComparator<T>)`	Returns the max item in the stream using the given comparator to compare items.
`min()`	Returns the min item in the stream requiring that the items be comparable.
`min(SerializableComparator<T>)`	Returns the min item in the stream using the given comparator to compare items.

Mango Streams can be converted to distrubted Spark streams by simplying calling `toDistributedStream`. Similarly, a Java stream can be created using the method `javaStream`.

4.3.1. Distributed Streams and Configuration

Mango will automatically distribute the current configuration to the Spark worker nodes when using a distributed stream. If for some reason the configuration is updated after the stream is created, you should call `updateConfig()` on the distributed stream to ensure it receives the changes in configuration.

5. Reflection, Casting, and Conversion

Mango provides a number of classes and utilities for performing reflection on objects and classes and casting or converting objects from type to another. Mango can provide a programming style seen in less strict languages, such as Python, at the cost of extra overhead. While not best practices, this programming style can be useful for prototyping, dealing with generics, or cases where the return type is unknown. This programming style is encapsulated in the `Val` object, which represents an immutable value of possibly unknown (to us) type. Take the following code snippet as example:

```
//If for some reason we do not know the return type (e.g. it returns Object) we can wrap it in a Val
Val v = Val.of(someRemoteCallThatCanReturnDifferentValues(....));

//A method that can return mulitple different types can wrap their return value in a Val
public Val computeValue(double in, boolean returnArray){
    if( returnArray ){
      return Val.of(new double[]{in});
    }
    return Val.of(in);
}
```

The val class provides "is" methods for determining type, e.g. `isArray`, `isMap`, and `isPrimitiveArray`. Additionally, you can the class information of the wrapped value using `getWrappedClass`. Val, provides convenience methods for getting the wrapped value as a number of different types, e.g. `asString()`, `asInteger()`, and `asDoubleArray()`. Additionally, a default value can be given to these methods in case the wrapped value is `null` or cannot be converted into the given type. Convenience methods also exist for converting into collections and maps, e.g. `asSet(Type)` and `asMap(Class, Class)` where the supplied

type information is used to convert the elements of the collection / map. Each of these methods make a call to `as(Type)`, which attempts to convert the wrapped value into the target type returning `null` if the conversion fails.

While Val can be very useful (e.g. as the value of Map) it can be hard to debug if something goes wrong and as such should be used only when needed or prototyping code.

5.1. Reflection

Mango aims to make using reflection a little easier. Reflection in Mango starts with the `RBase` class which defines the base set of functionality for performing reflection on an object, class, method, field, or parameter. This base set of functionality comes in the form of querying the annotations on the reflected item and conditionally performing some action based on the presence of an annotation. The following figure illustrates the hierarchy of classes that defines the Java reflection wrappers.

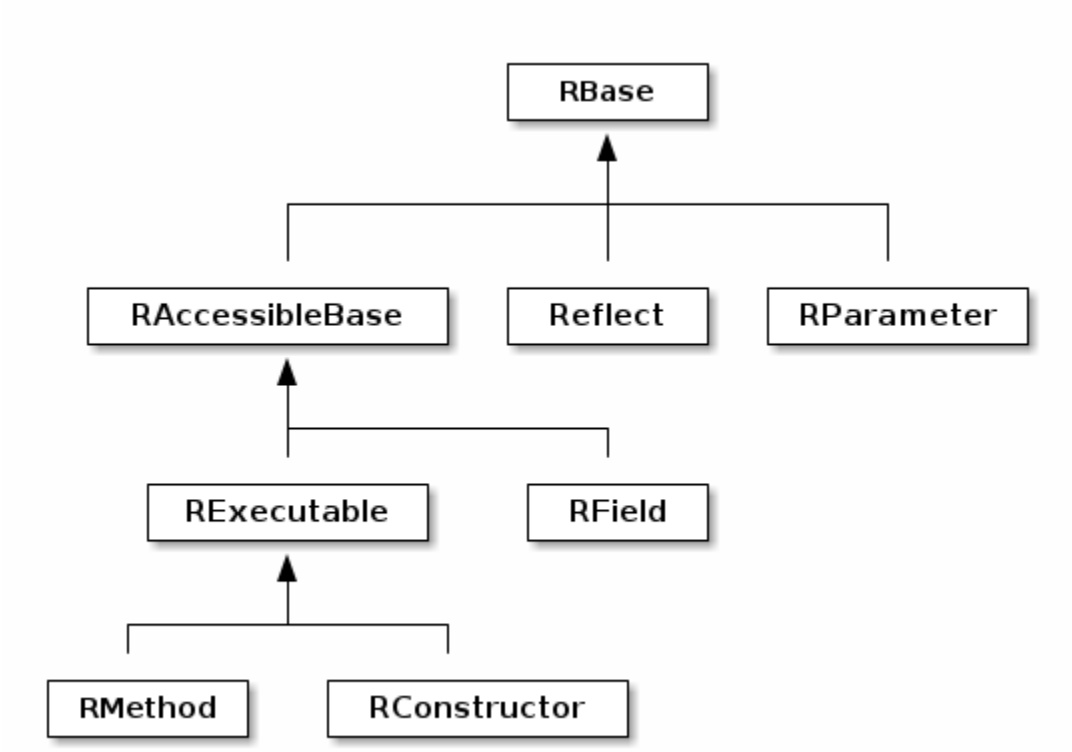

Figure 1. Hierarchy of classes for reflection.

The main entry point for working with reflection in Mango is the `Reflect` class. Reflect is a wrapper around an object or class providing easy ways to access the methods, fields, constructors, and annotations on the object/class. An instance is created using on of the static methods as follows:

```
// Reflecting on a class (we can only set / get / invoke static methods or create an instance)
Reflect rc = Reflect.onClass(MyClass.class);

// Reflecting on an object allows us to modify its fields and call its methods.
Reflect ro = Reflect.onObject(myClassInstance);
```

The Reflect instance respects scope by default. This can be changed by allowing privileged access as follows:

```
//Allow privileged access
ro.allowPrivilegedAccess();

//Go back to normal non-privileged access
ro.setIsPrivileged(false);
```

Privileged access will allow the protected and private elments of the object/class to be manipulated.

Once we have an instance of reflect we can query the object/class for its accessible fields as follows:

```
//Attempts to retrieve the value of a field called "name", if it cannot it will check for a getter
"getName()"
String name = ro.get("name");

//We can easily set the value of a field using the set method
ro.set("name", name + "-Smith");

//Retrieves the RField for the given field name if it is accessible.
RField ageField = ro.get("age");

//Gets all accessible fields on the object
List<RField> allFields = ro.getFields();

//Gets all accessible fields that are ints
List<RField> intFields = getFieldsWhere(f -> f.getType().equals(int.class));

//Gets all accessible fields that have an annotation of MyAnnotation on them
List<RField> annotatedFields = getFieldsWithAnnotation(MyAnnotation.class);
```

Similarly, we can query the object/class for its accessible methods as follows:

```
//Retrieves the RMethod for the given method name if it is accessible.
//Note we are looking for a no-argument method
RMethod calculateAge = ro.getMethod("calculateAge");

//Retrieves the RMethod for the given method name with given parameter types if it is accessible.
//Note we are looking for a method named "doSomethingImportant" whose first argument is a String and
second argument is an Integer.
RMethod doSomethingImportant = ro.getMethod("doSomethingImportant", String.class, Integer.class);

//Gets all accessible methods
List<RMethod> allMethods = ro.getMethods();

//Gets all accessible methods named "update"
List<RMethod> updateMethods = ro.getMethods("update");

//Gets all accessible methods named "update" that matching the given criteria (have 2 parameters)
List<RMethod> updateStringMethods = ro.getFieldsWhere("update", m -> m.getParameterCount()==2);

//Gets all accessible methods matching the given criteria (have 1 parameter)
List<RMethod> pseudoSetters = ro.getFieldsWhere(m -> m.getParameterCount()==1);

//Gets all accessible methods that have an annotation of MyAnnotation on them
List<RMethod> annotatedMethods = ro.getMethodsWithAnnotation(MyAnnotation.class);
```

Similarly, we can query the object/class for its accessible constructors as follows:

```
//Retrieves the RConstructor whose first parameter is an Integer and second parameter is a String.
RConstructor constructor = ro.getConstructor(Integer.class, String.class);
```

5.1.1. Fields, Methods, and Constructors

The `RAccessibleBase` base class wrapps `AccessibleObject` providing a `process(CheckedFunction)` and `with(CheckedConsumer)` method which automatically take care of setting the privileges of the object. `RField` implements the `RAccessibleBase` and wraps a Java `Field`. The `RExecutable` is child class of `RAccessibleBase` that serves as a base class for reflected objects that can be "executed", e.g. Methods and Constructors. The `RExecutable` also provides ways to examine the parameters (wrapped as `RParameter`) of the executable. The `RMethod` and `RConstructor` classes provide methods for calling (i.e. invoking) the underlying method / constructor.

5.1.2. Getting a class from its name

The Reflect class provides two convenience static methods for getting a `Class` for the name represented in a String. The methods support the detection of arrays where the name ends with `[]` or starts with `[L` or just `[`. Moreover, all classes in `java.lang`, `java.util`, and `com.gengoai` can be accessed by their simple name (e.g. ArrayList). The following code example shows the usage:

```
//Note this method will throw an Exception if the class is not found
Class<?> listClass = Reflect.getClassForName("ArrayList");

//This method will end up finding the class in com.gengoai.collection.counter.HashMapCounter
//Note that the "Silently" means it will return a null value if the class is not found instead of
throwing an
//exception.
Class<?> counterClass = Reflect.getClassForNameSilently("collection.counter.HashMapCounter")

//int[].class
Class<?> tClass = Reflect.getClassForName("int[]");
```

5.1.3. Type vs Class

In many places of Mango you will find methods that can type a Java `Type` or Java `Class`. A `Class` is a `Type` in Java, but other type implementations can be useful for dealing with generics, i.e. `ParameterizedType`. Thus, in most cases Mango will provide the option to use either a `Type` or `Class`. Note that when calling a method using a `Type` you either need to add the parameter type to the method call or capture the return value in a variable.

```
public static <T> T fromClass(Class<T> type) { ... }
public static <T> T fromType(Type type) { ... }

// OK - the compiler can guess the return type
System.out.println(fromClass(Double.class));
// NOT OK - the compiler cannot guess the return type
System.out.println(fromType(...));
```

Mango provides the `TypeUtils` class that contains useful methods for learning more about `Type` and converting them into `Class`. Additionally, Mango provides the method:

```
public static Type parameterizedType(Type rawType, Type... typeArguments)
```

to create `ParameterizedType` instances so that you can store / pass with generic information. As a convienenc, there is a `parse(String)` method on `TypeUtils` that will parse a string representation of a parameterized or non-parameterized type, e.g. `List<String>` will be parsed into a `ParameterizedType` with the raw class of `List` and the type argument of `String`. This allows you to specify generic types in your configuration files when defining types.

5.2. Type Conversion

The core component of Mango's `Val` class and configuration framework is the ability to convert any arbitrary type to another. This conversion is done using the Mango `Converter` class, which utilizes a number of `TypeConverter` registered using Java's Service Loader. A `TypeConverter` defines the following:

```
Object convert(Object source, Type... parameters) throws TypeConversionException; ①
Class[] getConversionType(); ②
```

① Defines the methodology to convert a *source* of any type with the following `Type` parameters (used for generics).

② Defines the classes the converter implementation can convert into.

When converting a source object we can us the following methods:

```
Converter.convert(source, TARGET_TYPE); ①
Converter.convertSilently(source, TARGET_TYPE); ②
```

① A `TypeConversionException` will be thrown if the source object cannot be converted into the target type (e.g. trying to convert an int into a Map).

② Returns a `null` value if the conversion fails.

Mango provides many type converters out of the box which cover core Java, java.util, and Mango types. New type converters can be registered using Java's Service loader. We recommend using the `org.kohsuke.metainf-services` package to ease this process by only needing to add a `@MetaInfServices(value = TypeConverter.class)` annotation to your type converter class.

5.3. Casting

Mango provides the `Cast` class to easily cast an object. It provides the following basic methods:

```
static <T> T as(Object o) ①
static <T> T as(Object o, Class<T> clazz)  ②
```

① Casts an object to the desired return type throwing a `java.lang.ClassCastException` if the given object cannot be cast as the desired type. This method uses an "unchecked" conversion.

② Casts an object to a given type throwing a `java.lang.ClassCastException` if the given object cannot be cast as the desired type. This method uses `Class.cast`.

Additionally, there are methods for casting the elements of collections, iterables, iterators, and maps: sahi2001

```
static <T> Iterator<T> cast(Iterator<?> iterator)
static <T> Iterable<T> cast(Iterable<?> iterable);
static <T> Collection<T> cast(Collection<?> collection)
static <T> Set<T> cast(Set<?> set);
static <T> List<T> cast(List<?> list)
static <K, V> Map<K, V> cast(Map<?, ?> map)
```

The methods listed above perform casting in lazy manner and do not change the underlying data. These methods are most useful when passing an item as method parameter.

6. Input / Output

Mango provides classes for working with archive files, csv and json encoded streams, asynchronous and multi-file writers, and an abstraction around a "resource" which could be a file, url, string, etc.

6.1. Resources

A resource represents a source or destination of/for data. It is similar in idea to Spring's Resource class. At the core a `Resource` object facilities opening the data source for reading (`InputStream` or `Reader`) and/or writing (`OutputStream` or `Writer`) and traversing and manipulating the structure of the data source (e.g. retrieving all children of a folder or the folder for a file, deleting an element, or adding new folders).

The following is a list of the supported resource types and whether they support being read from, written to, or traversed.

Resource Type	Readable	Writeable	Traversable	Description
`ByteArrayResource`	✔	✔		Wraps an expandable array of bytes for reading/writing.
`ClasspathResource`	✔	✔	✔	Points to a stream resource on the classpath.
`EmptyResource`				Special resource representing no content.
`FileResource`	✔	✔	✔	Wraps a Java File.
`InputStreamResource`	✔			Wraps a Java InputStream.
`OutputStreamResource`		✔		Wraps a Java OutputStream.
`ReaderResource`		✔		Wraps a Java Reader.
`StdinResource`	✔			Wraps System.in.
`StdoutResource`		✔		Wraps System.out
`StringResource`	✔	✔		Wraps a Java StringBuilder allowing reading and writing (by overwriting the value).
`URIResource`	✔	✔	✔	Wraps a Java URI.
`URLResource`	✔	✔	✔	Wraps a Java URL.
`WriterResource`		✔		Wraps a Java Writer.

Resource Type	Readable	Writeable	Traversable	Description
`ZipResource`	✔		✔	Wraps a Java ZipFile and ZipEntry allowing reading and traversal of a zip archive.

Resources are created by either using the constructor of one of the implementations or by using the `Resources` utlility class. The main way of creating a resource is using `Resources.from(String)` where the given string defines the resource scheme. Each scheme has an associated `ResourceProvider` which takes care of parsing the scheme and creating a Resource instance. The scheme is in the following format: `PROTOCOL(?OPTIONS):PATH` where `OPTIONS` is optional. For example a `FileResource` can specified using `file:/home/user/file.text`. Common options include, the charset and compression technique and are set as follows: `file?compression=GZIP,charset=SJIS:/home/user/file.text`. Note that options are set using a BeanMap thus the valid options for each resource type are the setters on that type.

The following is a list of Schemes, the generated resource type, and what the path represents for those types accessible via `Resources.from(String)`:

Scheme	ResourceType	Path
`bytes`	`ByteArrayResource`	A string which will be converted into bytes (empty is ok).
`classpath`	`ClasspathResource`	the location of the resource found within in the default Classloader.
`file`	`FileResource`	the location of the resource on a local disk.
`stdin`	`StdinResource`	empty.
`stdout`	`StdoutResource`	empty.
`string`	`StringResource`	the String representing the content of the resource (empty is ok).
`http(s)`	`URLResource`	path of the url.
`zip`	`ZipResource`	the location on local disk where the zip file is located.

Note that for convenience you can leave off the scheme for file resources, e.g. `Resources.from("/home/user/test.csv")` will assume the given String is a `FileResource`.

6.1.1. Reading

A `Resource` implementation provides the following methods for reading:

Return Value	Method	Description
`boolean`	`canRead`	Returns *true* if the resource is readable, *false* if not.
`InputStream`	`inputStream`	Opens an input stream over this resource.

Return Value	Method	Description
`MStream<String>`	`lines`	Creates an MStream (see Mang Streams) over the lines in the resource.
`byte[]`	`readBytes`	Reads the resource into an array of bytes.
`Reader`	`reader`	OOpens a reader using guessing the encoding and falling back to the default on the resource.
`List<String>`	`readLines`	Reads the complete resource in as text breaking it into lines based on the newline character.
`T`	`readObject`	Deserializes an object from a resource.
`String`	`readToString`	Reads the entire resource as a String.

One of the advantages of using a `Resource` is it will automatically determine the character set of the data source when reading (except `inputStream` and `readBytes`). The default charset can set using the `setCharset` method. Moreover, the resource will automatically determine if the underlying data is compressed in gzip or bzip2 format and handling it accordingly.

6.1.2. Writing

A `Resource` implementation provides the following methods for writing:

Return Value	Method	Description
`Resource`	`append(String)`	Appends the given string content to the resource.
`Resource`	`append(byte[])`	Appends the given byte array content to the resource.
`boolean`	`canWrite`	Returns *true* if the resource is writable, *false* if not.
`OutputStream`	`outputStream`	Opens an output stream over this resource.
`Resource`	`write(byte[])`	Writes the given byte array to the resource overwriting any existing content.
`Resource`	`write(String)`	Writes the given string to the resource overwriting any existing content.
`Resource`	`writeObject(Object)`	Serializes an object to the resource using Java Serialization.
`Writer`	`writer()`	Opens a writer for writing to the resource.

6.2. Resource Monitoring

A common pitfall in Java is not properly closing resources. This can become especially tricky when dealing with concurrency and the new Java stream framework. Mango provides a `ResourceMonitor` which tracks `MonitoredObjects` and automatically closes (frees) them when they are no longer referenced. The `ResourceMonitor` is basically a garbage collector for resources!

The `ResourceMonitor` class provides convenience methods for monitoring the most common types:

SQL Connection	InputStream	OutputStream
Reader	Writer	Stream<T>
MStream<T>	DoubleStream	IntStream
LongStream		

Additionally, it provides a generic `monitor` method that takes an `Object` and returns a `MonitoredObject` wrapping the given object. For generic Objects you can also specify a custom procedure to run when the resource is "closed" by passing in `Consumer<T>` that will free resources. An example of creating a Monitored object from a custom class is as follows:

```
MonitoredObject<MyClass> m = ResourceMonitor.monitor(new MyClass(), mc -> {
 //special on-close stuff here
});
System.out.println(m.object.getValue());
```

The object is wrapped in a `MonitoredObject` which is tracked by the resource monitor. We have specified a custom on-close operation, which will be called when there are no other references to the wrapped `MyClass` object. The wrapped object is accessed via the public field `object`

6.3. CSV

Delimited Separated Value (DSV) files, where the delimiter is most commonly a comma or tab, are widely used data format for everything from finance to to-do lists. Mango provides a reader, writer, and formatter for DSV which is configurable to match most standards. The following code snippet illustrates how easy it is to read in a CSV file:

```
try( CSVReader reader = CSV.csv().reader(Resources.from("/data/people.csv")) ){
  List<String> row;
  while( (row = reader.nextRow()) != null ){
    System.out.println(row);
  }
}
```

The CSV format can be specified using fluent accessors on the `CSV` class. A generic CSV and TSV format are accessible via `csv()` and `tsv()` respectively, but one can also call `builder()` which will use default values. You can instruct the reader that the first line of a CSV file is the header by calling `hasHeader()` on your CSV object. Alternatively, you can specify the header if one is not given in the file using `header(String…)` or `header(List<String>)`. When a header is specified you can iterate over the file using: `CSV.rowMapStream()` which will provide Java stream of `Map<String,String>`.

6.4. JSON

Mango uses Jackson for handling JSON. Mango provides a utility class `Json` to perform basic operations, such as serializing an object to a JSon string or to a resource and deserializing Json into an Object. Additionally, Mango provides a `JsonEntry` class that wraps the Jackson Json classes allowing easy builder style approaches to constructing Json.

6.5. Specifications

The builder pattern is an excellent way to create Objects which have multiple parameters. However, the builder pattern doesn't help when we want to specify parameters via configuration or in a concise manner. That is where Mango Specificatons come in handy. A specification is a URI-like object defining a `Schema`, `Protocol`, `SubProtocols`, `Path`, and `Query Parameters` that define a resource, connection, etc. The specification form is as follows:

```
SCHEMA:(PROTOCOL(:SUB-PROTOCOL)*)?(::PATH)?(;query=value)
```

An example is `kv:mem:people` which defines an in-memory ke-value store with the namespace people. The specifciation `kv:disk:people::~/people.db;readOnly=true` defines a disk-based key-value store with the namespace *people* stored at `~/people.db` and being accessed as read only. Note that the Path and Query Arguments can will be resolved against the current Config allowing for dynamic paths like `${BASE_DIR}/myFile` for paths and `parameter=${parameter.defaultValue}` where `${BASE_DIR}` and `${parameter.defaultValue}` will be set via the Config.

7. Pseudo-Language Extensions

Mango provides a number of classes and utilities that act as extensions / enhancements to Java concepts. We call these enhancements "pseudo-language extensions" as Java does not not provide an easy way of extending the language. In many cases these extensions were created for specific use cases in the Hermes and Apollo libraries.

7.1. 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`.

7.1.1. 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.

7.1.2. 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.

7.2. 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`. 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.

7.3. Tuples

A tuple is a finite sequence of items. Mango provides specific implementations for degree 0-4 tuples, which all each element's type to be defined via generics. For tuples with degree 5 or more, a generic `NTuple` is provided.

8. Parsing Framework

9. 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.

9.1. 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:

```
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);
    }

}
```

Note: All command line arugments specified using as long or short options are automatically put into the Configuration object.

9.2. Configuration

Mango provides a convenient to use Configuration system that allows for the definition of properties and beans. Configuration files have a `.conf` extension and packages can define a `default.conf` for its configuration. Mango's application framework will take care of initializing the configuration for you.

However, if you are not using the application framework you can initialize the Configuration yourself using one of the `initialize` methods. The initialize methods take some combination of the name of the program being ran, the array of command line arguments, the command line parser, and an optional array of other packages whose configuration settings we need to load. The signature of the full initialize method is as follows:

```
Config.initialize(String programName,
                  String[] commandLineArgs,
                  CommandLineParser commandLineParser,
                  String... otherPackages)
```

Mango also provides an initialize method which construct a command line parser (this is useful if you do not care about the command line arguments). Finally, for unit testing there is an `initializeTest()` method.

Mango will attempt to load a package configuration based on the call class. For all package configurations, Mango will traverse the package tree up until a `default.conf` is found or it has reaced the root. For example, if a `default.conf` is defined in `com.mycompany` and our Config is initialized in a class in the package `com.mycompany.apps`, it will attempt to resolve the package configuration in the following order:

2 `com.mycompany` *: Found will load the* `default.conf`

You can also pass a custom configuration file via the `--config` command line option. This configuration file will take precedence overriding any values in the `default.conf` files.

The Config object can be accessed via its static methods. To retrieve values you use a `get` method which returns a `Val` object. The following `get` methods are defined:

```
get(Class<?> clazz, Object... propertyComponents) ①
get(String propertyPrefix, Object... propertyComponents) ②
```

① Generates a property name which is the form `className.component1.component2. ⋯ .componentN`

② Generates a property name which is the form `propertyPrefix.component1.component2. ⋯ .componentN`

The `get` method calls the `findKey` method to determine the best matching property name for the given prefix / class name and components.

If the first property component is a `Language`, it will be used to find language dependent values, if available. The language will be checked using its enum name (e.g. `ENGLISH`), its lower-cased name (e.g. `english`), its two-letter language code (e.g. `EN`), and the lower-cased version of its two-letter language code (e.g. `en`). When only a language is given a property component it will check for each language variation and if none are found fallback to the just the prefix or class name. For example, given the following configuration:

```
timeout=30
timeout.en=60
```

if we call `get("timeout", Language.SPANISH)`, its call to `findKey` will return `timeout` as the prefix exists, but there is not a Spanish specific version. In contrast calling `get("timeout", Language.ENGLISH)` would

result in `timeout.en` as an English specific value is present. Similarly, when one or more non-Language components are specified the search will be performed looking for `prefix.language.components`, `prefix.components.language`, and fallback to `prefix.components`

Once a property name is found its value will be searched for in the Config object, then in the system properties, and finally as an environment variable.

9.2.1. Configuration File Format

Mango's configuration format is a mix between json and java properties format. One configuration file can import another using the `@import` keyword which takes as its argument a package name or file location as follows:

```
@import com.mycompany.mypackage ①
@import file:/home/me/myconf.conf ②
```

① Imports the `default.conf` associated with the `com.mycompany.mypackage` package.

② Imports `myconf.conf` located in `/home/me/`

Mango configuration files use `#` to denote comments, e.g.:

```
####################################
# My Package Configuration
####################################

# Use strict mode for parsing ?
strict.parsing = false
```

Defining properties and their values can be done in one of the following ways:

```
propertyName = propertyValue ①
propertyName += propertyValue ②
```

① Simple assignment where the property with given name is assigned the given value

② Append assignment where the given value is appended to the property of the given name treating it as a list

Property names must start with an alphabetic or underscore (_) character and can be followed by zero or more alphanumeric, period, or underscore characters. For example, `abc_def` is a valid property name, but `abc$def` is not. Property values can be:

- Strings represented using `"string value"` (use the backslash character to escape quotes in the string)
- Safe Strings which with an alphabetic or underscore (_) character and can be followed by zero or

more alphanumeric, period, or underscore characters

- Numbers e.g. `1`, `23.4`, or `1e-5`
- Boolean values, i.e. `true` or `false`
- Arrays with values enclosed in brackets, e.g. `[value1, value2, value3, ..., valueN]'
- Maps defined as `{ key : value}`, e.g. `{"size": 34.5, "age" : 21}`
- Null value represnted as `null`

Mango configuration support sections. Sections are prepended to the names of the properties contained within. For example, the following configuration snippet has a "remote" section:

```
remote {
   storage {
     text = s3
     search = solr
   }
}
```

which would be transformed into the following effective properties:

```
remote.storage.text = s3
remote.storage.search = solr
```

Beans

Bean objects can be defined in the configuration as follows:

```
beanName {
 # bean definition
}
```

Type information for the bean can be defined using the `@type` property name as follows:

```
beanName {
  @type = com.mycompany.Person
}
```

A constructor can be defined using the `@constructor` property. The constructor values can be defined as an array:

```
    @constructor = [
        {"String" : "John"},
        {"Int" : 36}
    ]
}
```

where the values are single entry maps whose key is the type and value is parameter value. Or as a map as follows:

```
    @constructor = {
        "String" : "John",
        "Int" : 36
    }
}
```

Note that when a constructor requires two or more parameter values of the same type, you must use the array version.

You may also define properties of the object that will be set via setters (which will include `add` methods which only take 1 argument) An example is as follows which will set the `income` and `address` of the person:

```
beanName {
    @type = com.mycompany.Person
    @constructor = {
        "String" : "John",
        "Int" : 36
    }
    income = 75000.00
    address = "666 Elm St."
}
```

Beans can be defined as singletons, by adding a property `singleton = true` to the bean definition.

You can reference beans as property values using `@{BEAN NAME}`. For example, assuming we have defined a bean `MyService` we can the specify it as property value as follows:

```
defaultService = @{MyService}
```

Object Parameterization

Similar to the ability to construct beans, Mango provides the ability to parameterize objects (i.e. inject values) using configuration. Parameterization is automatically performed as part of bean creation and object conversion (see [conversion|Conversion]). Parameterization is done by defining a property

named as `fully_qualified_class_name.property`. For example, if we have an interface `com.mycompany.ABCService` which defines a `apiKey` property (i.e. `setApiKey` and `getApiKey`), we can parameterize this across all implementations by defining:

```
com.mycompany.ABCService.apiKey {
    @type = String
    _ = "ABE123F562FDE"
}
```

Note that we use `to say that we want to set com.mycompany.ABCService.apiKey to the value of` meaning we will generate the following effective property names and values:

```
com.mycompany.ABCService.apiKey.@type = String
com.mycompany.ABCService.apiKey= "ABE123F562FDE"
```

With this defined any bean created from a Config object or through conversion will automatically have the api key property set. To manually force parameterization, you can call `BeanUtils.parameterizeObject(T)`. For example, if we have defined a `GeoLocationService` which is of type `ABCService` we can parameterize as follows:

```
var geoService = BeanUtils.parameterizeObject(new GeoLocationService());
```

You may also overide values for more specific classes. In our previous example, we could specify a different api key for the GeoLocationService as follows:

```
com.mycompany.GeoLocationService.apiKey {
    @type = String
    _ = "FFEE123FF99"
}
```

By combining object parameterization and bean definition via configuration you can construct a minimal dependency injection framework. Coupling this with overrides via the command line can make this highly configurable and easier to use. For example, if you only need to override the GeoLocationService's api key for certain runs of your program you could omit the configuration setting and pass it via the command line within a bash script.

9.3. Preloading Static Elements

Mango provides a `Preload` annotation for classes which will force the static fields to be initialized. This is automatically done as part of Config initialization.

10. 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.