

Umber Core 0.3.5 User Guide

Chapter 1 - Introduction

Umber Core is a library of generic Java functions and wrappers for common third party libraries. The only required dependency for Umber Core is the Jakarta Commons Logging library. All other dependencies like Velocity, regular expression parsers, and JavaMail are optional, and will be detected automatically by the Core framework.

The Core is the lowest library in the Umber Tool set; all other Umber projects depend upon it.

Chapter 2 - Command-Line Arguments

Most command-line applications have to parse a series of command-line arguments to load their operating parameters. The `CommandArgs` class simplifies this by wrapping the `String[]` args and providing a Map-like interface to it. The `String[]` from a class' `main()` method can be passed directly into the `CommandArgs` constructor.

```
public CommandArgs (String[] args)
```

`CommandArgs` splits the arg list into named arguments and anonymous parameters. All named arguments consist of two elements in the `String[]`: an argument preceded by a "-" or "--", and the argument value. Parameters are values not associated with a named argument. Thus, given this series of arguments

```
-arg1 value1 --arg2 value2 value3 value4
```

`CommandArgs` will find two named arguments, "arg1" and "arg2", with the values "value1" and "value2" respectively, and the two parameters "value3" and "value4".

To access the values in the command line, you can use `getArgNames()` to get a list of all present argument names, and `getArgValue()` to retrieve the value of each individual argument. The `getParams()` method returns an array of all parameters.

```
public String[] getArgNames ()
public String getArgValue (String name)
public String[] getParams ()
```

In code, the above example would look like:

```
String[] input = new String[] { "-arg1", "value1", "--arg2",
    "value2", "value3", "value4" };

CommandArgs args = new CommandArgs (input);
String arg1 = args.getArgValue ("arg1"); // "value1"
String arg2 = args.getArgValue ("arg2"); // "value2"
String[] params = args.getParams ();      // { "value3", "value4" }
```

The parameters do not have to be at the end; the argument list below will provide the same results as above:

```
value3 -arg1 value1 value4 --arg2 value2
```

CommandArgs also recognizes when the command line contains the same named argument more than once. The getArgValue() method will still return only the first instance, but the getAllArgValues() will return an array of all values for a given argument name.

```
public String[] getAllArgValues (String name)
```

The code below demonstrates how each named argument exposes itself through the arg accessor methods.

```
String[] input = new String[] { "--arg1", "value1", "--arg2",
    "value2", "--arg2", "value3" };

CommandArgs args = new CommandArgs (input);
String arg1 = args.getArgValue ("arg1"); // "value1"
String arg2 = args.getArgValue ("arg2"); // "value2"
String[] args1 = args.getAllArgValues ("arg1"); // { "value1" }
String[] args2 = args.getAllArgValues ("arg2"); // { "value2", "value3" }
```

CommandArgs also offers a couple lower level methods to access the original String[] array, and a Map of the named arguments:

```
public String[] getRawArgs ()
public Map getArgMap ()
```

So far, CommandArgs will allow any argument name. However, using the ArgSpec (inner) class, you can define a set of required and optional named arguments. You create one spec object per possible named argument and pass them into the alternate constructor.

```
public CommandArgs (String[] args, ArgSpec[] specs)
```

The ArgSpec object has the following interface; all args have a name and a boolean flag to determine if they are required or not, plus an optional text description of the argument.

```
public ArgSpec (String name, boolean required)
public ArgSpec (String name, boolean required, String description)
public String getName ()
public boolean isRequired ()
public String getDescription ()
```

A CommandArgs object initialized with specs behaves a little differently than without. The isValid() method returns false if any required named arguments are missing, or if any unrecognized arguments are passed in. The usage() method will return a String description of all the parameters, useful for responding to invalid arguments. (If no specs are given, isValid() always returns true and usage() always returns null.)

```
public boolean isValid ()
public String usage ()
```

Thus, given the code here:

```
String[] input = ...;
ArgSpec[] specs = new ArgSpec[] {
    new ArgSpec ("arg1", true, "First argument"),
    new ArgSpec ("arg2", false, "Second argument")
};
CommandArgs args = new CommandArgs (input, specs);
```

The following command lines are valid:

```
--arg1 value1
--arg1 value1 --arg1 value2 value3 value4
--arg1 value1 --arg2 value2
```

And so on. As long as "arg1" is present and no unrecognized arguments appear, the arg array is valid. However, leaving off "arg1" or adding an unexpected named argument causes isValid() to return false. All of these command lines are invalid

```
value1
```

```
--arg2 value1
--arg1 value1 --unknown value2
```

Chapter 3 - Logging

Application logging is a critical and hotly argued issue. Many implementations exist, including Log4j, Sun's `java.util.logging` API, Jakarta's `commons-logging` package, and good old fashioned console `stdout`. Umber attempts to remain mostly agnostic on which framework to use itself, preferring the application to explicitly choose one. However, if the application does not choose a logging framework, Umber will search for them on the `CLASSPATH` and use the first one it finds. If it finds no logging framework, it will send all logging messages directly to the console.

This approach is similar to Jakarta `commons-logging`, except that it adds the feature of explicit choice. For example, if you have both `log4j` and the Sun logging API available, `commons-logging` will always choose `log4j`. Conversely, Umber Logging lets the application force usage of the Sun API if it so chooses. Also, Umber avoids the mischievous complexity of `ClassLoader` hacks used by `commons-logging`, and relies instead on a simple `Class.forName()` check for the frameworks within the current `ClassLoader`. (For a top-notch detailed account of these `commons-logging` hacks, see <http://www.qos.ch/logging/classloader.jsp>.)

The Umber logging API primarily consists of the `LogFactory`, the `ILog` interface, and `ILog` implementations. The `LogFactory` has only static methods, for controlling the logging wrapper and for instantiating `ILog` objects through which the application can write log messages.

```
public static ILog getLog (Class logClass)
public static ILog getLog (String logClassName)
```

The API for both of these classes is similar to the basic `commons-logging` API, so in most cases applications written for `commons-logging` can be converted to Umber logging by simply changing the import statements from this:

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
```

into this:

```
import org.umber.core.logging.Log;
import org.umber.core.logging.LogFactory;
```

In fact, to make migration from commons-logging to Umber logging simpler, Umber derives ILog from a Log interface which mirrors the commons-logging Log interface. Thus, the typical usage pattern for commons-logging initialization works as is.

```
public class MyClass
{
    private static Log _logger = LogFactory.getLog (MyClass.class);
    ...
}
```

The Umber Log interface has the following methods for logging, with or without a thrown exception, in order of decreasing severity:

```
public void fatal (Object message, Throwable t)
public void fatal (Object message)
public void error (Object message, Throwable t)
public void error (Object message)
public void warn (Object message, Throwable t)
public void warn (Object message)
public void info (Object message, Throwable t)
public void info (Object message)
public void debug (Object message, Throwable t)
public void debug (Object message)
public void trace (Object message, Throwable t)
public void trace (Object message)
```

To keep logging efficient when log messages are more complex, Log has the log level query methods below. Applications can use them to completely skip cumbersome message-building code when logging is turned off for that log level.

```
public boolean isFatalEnabled ()
public boolean isErrorEnabled ()
public boolean isWarnEnabled ()
public boolean isInfoEnabled ()
public boolean isDebugEnabled ()
public boolean isTraceEnabled ()
```

By default, Umber Logging attempts to load bindings for log4j, then Sun's java.util.logging, then falls back on System.out.println() if neither are available. It does not bother automatically looking for commons-logging since

that library performs the same wrapper function as Umber Logging and can introduce ugly ClassLoader problems.

Furthermore, if Umber finds log4j, it will test for a valid configuration (by checking for appenders with `Logger.getAllAppenders()` on the root log4j logger). If no appenders exist -- which is strangely the case if no log4j config file is found -- Umber will preemptively assign a simple default log4j configuration. This avoids the unfortunate situation where log4j refuses to log anything without an application-assigned config file:

```
log4j:WARN No appenders could be found for logger (org.myorg.MyClass).
log4j:WARN Please initialize the log4j system properly.
```

Alternatively, the application can request a preferred logging framework by setting the system property "umber.logging.default" to the value "LOG4J", "SUN", "COMMONS", "STDOUT", or "OFF". This will attempt to load the Umber Logging implementation classes declared in the following static constants, respectively:

```
LogFactory.LOG4J_LOGGER    // Log4J wrapper
LogFactory.SUN_LOGGER      // java.util.logging
LogFactory.COMMONS_LOGGER  // Jakarta commons-logging
LogFactory.STDOUT_LOGGER   // System.out
LogFactory.NULL_LOGGER     // no logging
```

These classes can also be assigned in application code to `LogFactory.setLogClass()` to have the same effect. It will throw an exception if the log class does not exist or does not implement `ILog`.

```
public static String setLogClass (String logClassName)
    throws UmberClassException
```

If a logging framework is explicitly requested but the support libraries are missing, Umber will fall back on its default load order. Thus, if the application requests `LOG4J_LOGGER` but log4j is not on the `CLASSPATH`, Umber will attempt to load `SUN_LOGGER`; if that fails (which could happen in JVMs before 1.4), Umber will use `LOGGER_STDOUT`.

The `getLogClass()` method retrieves the current logger implementation.

```
public static Class getLogClass ()
```

Section 3.1 - Log Formatter

The default output of the builtin Sun Java logger (available in JVM 1.4 and later) tends to be a bit verbose; it puts the date and log message on separate lines. The `UmberLogFormatter` class is a drop-in replacement for the default logger that condenses the output to a single line per log event. Note, it only works with the `java.util.logging` package, not with other logging frameworks like `log4j`.

A quick and ugly way to use the Core logger is to edit the `logging.properties` file in your `#{JRE}/lib` directory to include a line like this:

```
java.util.logging.ConsoleHandler.formatter = org.umber.core.util.UmberLogFormatter
```

Unfortunately, that will affect all applications using that JRE installation. A much better way is to set the system property `"java.util.logging.config.file"` in your application, to point to a local logging properties file. More details are in the Javadocs for the `java.util.logging.LogManager` class.

If the Core log formatter doesn't meet your needs, you can use its source code as a starting point, and tweak it for your personal needs. Future versions of Umber may include a more modular version of this class.

Chapter 4 - Resource Loader

The `ResourceLoader` class provides some convenience methods for loading data content from resources on the `CLASSPATH`. All methods with a path argument behave similarly. Called with a `Class` object, the method will attempt to load the resource as a relative path from the package that owns that `Class`. Called without a `Class` (or with a null `Class`), it will treat the path as an absolute path, from the `CLASSPATH` root. In either case, a path beginning with `"/"` will always be loaded as an absolute path.

The `loadResourceAsStream()` methods simply wrap `Class.getResourceAsStream()` to return the resource as an `InputStream`.

```
public static InputStream loadResourceAsStream (String path)
public static InputStream loadResourceAsStream (String path,
        Class resClass)
```

The next set of methods load the resource and return it as a `String` object. The first is a helper method, and can be used with any `InputStream`; it does not have to be associated with a resource. The other two methods work like the stream methods above, essentially calling `loadResourceAsStream()` and feeding that

InputStream to loadInputStreamAsString().

```
public static String loadInputStreamAsString (InputStream in)
public static String loadResourceAsString (String path)
public static String loadResourceAsString (String path, Class
    resClass)
```

The final two methods load a properties file resource into a Properties object.

```
public static Properties loadResourceAsProperties (String path)
public static Properties loadResourceAsProperties (String path,
    Class resClass)
```

Chapter 5 - Property Name Helper

Umber Core offers a simple, useful helper class for converting between various property naming styles. Pass in a property name in any of the supported naming styles and PropertyName will convert it into any other naming style. The inner class PropertyName.PropertyStyle defines all styles; each valid style is declared as a static member of PropertyName. The styles and how they look are below.

```
PropertyName.LOWER_HYPHEN_STYLE      "property-name"
PropertyName.LOWER_UNDERSCORE_STYLE  "property_name"
PropertyName.UPPER_HYPHEN_STYLE      "PROPERTY-NAME"
PropertyName.UPPER_UNDERSCORE_STYLE  "PROPERTY_NAME"
PropertyName.CASE_DELIM_STYLE        "PropertyName"
PropertyName.JAVABEAN_STYLE          "propertyName"
PropertyName.DEFAULT_STYLE            "property-name"
```

The easiest way to use PropertyName is to invoke its static convenience method, applyStyle().

```
public static String applyStyle (String name, PropertyStyle style)
```

Pass in the original property name and the style you want. For example, to convert "my-prop-name" into "myPropName":

```
String beanName = PropertyName.applyStyle ("my-prop-name",
    PropertyName.JAVABEAN_STYLE);
```

The static method creates and discards a PropertyName object each time it's called. If you have a property name that you need to re-use, create a

PropertyName instance and call its conversion accessors:

```
public String getPropertyName ()
public String getCaseDelimitedName ()
public String getBeanPropertyName ()
public String getLowerHyphenatedName ()
public String getLowerUnderscoredName ()
public String getUpperHyphenatedName ()
public String getUpperUnderscoredName ()
```

To convert "my-prop-name" into upper- and lower-case underscored names:

```
PropertyName name = new PropertyName ("my-prop-name");
String upper = name.getUpperUnderscoredName(); // "MY_PROP_NAME"
String lower = name.getLowerUnderscoredName(); // "my_prop_name"
```

Chapter 6 - Text Parsing

The GenericParser class is a simple convenience class for efficiently and flexibly parsing text content. It takes a String object as input, in the loadText() method, and grabs chunks of text by searching ahead for one or more String token. The loadText() method will discard any state info from previous parses.

```
public GenericParser ()
public void loadText (String text)
public String parseUntil (String token, boolean includeToken)
public String parseUntil (String[] tokens, boolean includeToken)
public String parseRemaining ()
```

The parseUntil() methods search from the current text position for any of the supplied tokens. If it finds a token, it will return all text between the current position and the token. If 'includeToken' is true, it will concatenate the token text onto the returned text. If the tokens do not exist after the current position, parseUntil() will leave the current position untouched and return null.

The parseRemaining() method returns all text between the current position and the end, and moves the current position to the end. If the parser is already at the end, it will return a blank String (not null).

To search for a token without affecting the current position, call the peek()

method. It will return how many characters ahead the token is, or -1 if the token is not found.

```
public int peek (String token)
```

To roll the parser's current position back to the beginning of the text, call the `reset()` method. To find the parser's current position in the content, you can call `getPosition()`. The `getIterator()` method retrieves the underlying `CharacterIterator`; you should only use this for low level manipulation.

```
public void reset ()
public int getPosition ()
public CharacterIterator getIterator ()
```

Chapter 7 - Text Utilities

Umber Core comes with a collection of simple text processing classes which each perform a single discrete task. Each text class implements one of four Umber interfaces with a single method that translates between various combinations of `String` and `String[]`.

```
ITextSplitter: String[] splitText (String text)
ITextJoiner:   String joinText (String[] text)
ITextFilter:   String filterText (String text)
ITextProcessor: String[] processorText (String[] text)
```

Splitters convert from `String` to `String[]`. The following classes implement `ITextSplitter`:

```
LineSplitter
ParagraphSplitter
TokenSplitter
RegexExtractor
RegexLineSplitter
RegexSplitter
BraceMatchSplitter
```

The first two splitters divide text content by linefeeds. The `LineSplitter` divides the text at every linefeed; it respects UNIX (`/n`), Windows (`/r/n`), and old Macintosh (`/r`). If `keepNewlines` is set to true, the line splitter will include the linefeeds in the output array. The `ParagraphSplitter` is similar, except it breaks text wherever it sees blank lines, with consecutive linefeeds.

```
public LineSplitter (boolean keepNewlines)
public ParagraphSplitter ()
```

The `TokenSplitter` class requires an array of tokens; each time one of the tokens occurs in the text, the splitter will chop it into a separate array element. If `keepDelimiters` is true, the splitter will return the tokens as separate array element in the output array.

```
public TokenSplitter (String[] tokens, boolean keepDelimiters)
```

For more sophisticated splitting, you can define split points with a regular expression.

```
public RegexpSplitter (String regexp, boolean keepDelimiters)
```

The other regular expression splitter pulls data from within parenthetical regexp groups and puts that in the output array. If `isPerLine` is true, the splitter applies the regular expression separately to each line in the input text; if false, it applies it once to the entire text content.

```
public RegexpExtractor (String regexp, boolean isPerLine)
```

For example, given the regular expression `"[a-z]*(((0-9)+[d-f]+).*" and the input text "abc123defghi", the splitter would extract two Strings: "123def" to match the outer parens, and "123" to match the inner parens.`

The final splitter class, `BraceMatchSplitter`, breaks text input apart according to matching pairs of braces. For example, given the text document "one (two (three) four) five)", and the default braces of "(" and ")", the brace matcher would split the text into the fragments "one ", "(two (three) four)", and " five)". It would not match the nested closing brace after "three", nor the unpaired brace after "five". Braces can be arbitrary strings, but should not be identical to each other. The default constructor uses the braces "(" and ")".

```
public BraceMatchSplitter ()
public BraceMatchSplitter (String openBrace, String closeBrace)
```

The only Core class to implement `ITextJoiner` is `Concatenator`. By default, the concatenator connects all input array elements into a single String output, without changing any content. The alternate constructor tells the concatenator to insert a chunk of static text between each element, for example a linefeed.

```
public Concatenator ()
```

```
public Concatenator (String delimiter)
```

The filter classes go from String in to String out. The simplest filter is TextReplacer, which performs search and replace on text content. All instances of oldText are replaced with newText. If maxCount is specified, the filter will only replace that number of occurrences.

```
public TextReplacer (String oldText, String newText)
public TextReplacer (String oldText, String newText, int maxCount)
```

The TextResolver filter is similar, but more sophisticated. It accepts a group of search/replace pairs, in the form of a Map. Each key in the Map is a variable name in the text; by default, variables are delimited by "\${" and "}", as in UNIX shell scripts, but those delimiters can be customized with startToken and endToken. If swallowUndefined is false, the filter will leave variables in the text that don't appear in the Map untouched. If true, the filter will delete the undefined variable reference.

```
public TextResolver (Map map, boolean swallowUndefined)
public TextResolver (Map map, String startToken, String endToken,
    boolean swallowUndefined)
```

Consider the code example below:

```
HashMap map = new HashMap ();
map.put ("pet", "dog");
map.put ("stuff", "fleas");

TextReplacer filter = new TextReplacer (props, false);
String result = filter.filterText ("My ${pet} has ${desc} ${stuff}.");
// "My dog has ${desc} fleas."
```

The 'desc' variable is not defined, and since swallowUndefined is false, that variable is left in the output. Changing it to true would cause the filter to drop the undefined \${desc} reference.

```
TextReplacer filter = new TextReplacer (props, true);
String result = filter.filterText ("My ${pet} has ${desc} ${stuff}.");
// "My dog has fleas."
```

Customizing the delimiters to Ant-style variables would look like this:

```
TextReplacer filter = new TextReplacer (props, "@", "@", false);
```

```
String result = filter.filterText ("My @pet@ has @desc@ @stuff@.");  
// "My dog has @desc@ fleas."
```

The TextResolver also lets you specify variable delimiters and legal content with regular expressions.

```
public TextResolver (Map map, String regexp, boolean swallowUndefined)
```

The variable name is defined in a single parenthetical region; the start and end tokens are the expression before and after the parens. Thus, a regexp of "[x]+([0-9]+)[y]+" declares that variables begin with an arbitrary number of "x" characters and end with an arbitrary number of "y" characters, and that all variable names must contain only numbers. If a variable reference fails any of the regexp constraints, it is completely ignored (which means swallowUndefined will have no effect on it).

In the example above, these variables are valid:

```
"x0y"           ==> "0"  
"xxxxx345345yy" ==> "345345"
```

And these are not:

```
"xx-235yyy"  
"xxxx1234"  
"xxxpetyyy"
```

The default Normalizer filter collapses spans of whitespace into single space characters.

```
public Normalizer ()
```

For example, the input text

```
" My   dog   has   fleas.  "
```

would become " My dog has fleas. "

The Normalizer can be customized to spans of text other than whitespace, and can be told to ignore spans of literal text that are delimited by special exclusion tokens, like quote marks. This customized filter will scan the input text; when it finds one of its normalizable tokens, it will continue scanning as long as it keeps finding adjacent tokens. When it hits a non-token, it will replace the entire scan of consecutive tokens with a single copy of normalText.

```
public Normalizer (String[] tokens, String normalText,
    String[][] exclusionTokens)
```

This can lead to all sorts of fruity normalizations. This one replaces all spans of commas and periods with a single "#" character, but excludes all text delimited by [] and @@.

```
String[] tokens = new String[] { ".", ", " };
String[][] exclusions = new String[][] { { "[", "]" }, { "@", "@" } };
Normalizer filter = new Normalizer (tokens, "#", exclusions);

String input = "...Text.[.,with..custom,.]whitespace...@,.....@,....";
String result = filter.filterText (input);
// "#Text#[.,with..custom,.]#whitespace#@,.....@#"
```

The CaseFilter filter normalizes capitalization in the input text.

```
public CaseFilter (CaseStyle style)
```

The CaseStyle parameter can take the following values:

```
CaseFilter.UPPER
CaseFilter.LOWER
CaseFilter.TITLE
CaseFilter.SENTENCE
```

UPPER converts all alpha characters into uppercase. LOWER converts to all lowercase. TITLE capitalizes the first letter of every word. SENTENCE capitalizes the first letter of each sentence. The TITLE and SENTENCE styles will also lowercase all alpha characters that are not explicitly uppercased, so the text "thIS iS NOT. a seNTENCE." would change to "This Is Not. A Sentence." with TITLE style, and "This is not. A sentence." with SENTENCE style.

The final two filters are mainly useful for processing source code. They strip out single line and multi-line comment text, respectively.

```
public SingleLineCommentStripper (String startToken)
public MultiLineCommentStripper (String startToken, String endToken)
```

For example, to strip out shell script comments:

```
SingleLineCommentStripper filter =
    new SingleLineCommentStripper ("#");
```

To strip out C-style multiline comments:

```
MultiLineCommentStripper filter =  
    new MultiLineCommentStripper ("/**", "**/");
```

There are currently no implementations yet for `ITextProcessor`. Future versions of Umber Core may add them.

Chapter 8 - Java Class Helper

Java offers a rich, powerful, and sometimes complex metadata API for managing objects and implementations generically, at a high level. Nearly all these methods throw exceptions which you must explicitly catch and react to in your application code, and some methods throw quite a few different exceptions.

The Umber `ClassHelper` class offers a simple, concise API for the most common Java reflection actions you'll need in day to day programming. The static methods either catch and handle the thrown exceptions, or wrap them in `UmberClassException` to simplify application exception handling code. In all cases, `ClassHelper` will set a comment in the exception explaining what happened.

The most generally useful method in `ClassHelper` is `classExists()`, which tests for the presence of a Java class, using the `Class.forName()` method. It swallows all exceptions, since the purpose of the method is to test for its existence, not necessarily to load the class (which is when you'd be more interested in those exceptions).

```
static public boolean classExists(String className)
```

It's also possible to perform bulk checks for Java classes, typically if you have a collection of third-party libraries your application relies on. The `checkDependencies()` method takes an array of two-element `String` arrays. Each two-element array should contain a text description and the fully qualified Java class. If any dependency is not found, the method throws an exception with a text comment including the description you gave of that library.

```
static public void checkDependencies(String[][] deps)  
    throws UmberClassException
```

For example, to test for commons-logging and `iText`, you might invoke `checkDependencies()` like this:

```
String[][] deps = new String[][] {
    { "Jakarta Commons Logging", "org.apache.commons.logging.Log" },
    { "iText PDF", "com.lowagie.text.Document" } };
ClassHelper.checkDependencies(deps);
```

Another querying method is `implementsClass()`, which checks to see if one class extends or implements a base class or interface. It can take a `Class` object, or a fully qualified class name. If that class name is not resolvable, the method will return false. The second parameter is the `Class` object for the base class or interface. These methods essentially wrap the `Class.isAssignableFrom()` method.

```
static public boolean implementsClass (String objClassName, Class baseClass)
static public boolean implementsClass(Class objClass, Class baseClass)
```

The remaining methods in `ClassHelper` are useful for dynamically instantiating Java objects. They wrap methods in the `Class` class, adding some flexibility and simplicity not present in the `Class` API. The `constructorExists()` method determines if the given Java class contains an accessible constructor that takes the given arguments. The `newInstance()` methods instantiate a new object of the requested type, passing the given args into the appropriate constructor. If the object class cannot be found, or if no constructor matches the arguments, it will throw an exception.

```
static public boolean constructorExists(Class objClass, Object[] args)
static public Object newInstance(Class objClass, Object[] args)
    throws UmberClassException
static public Object newInstance(String objClassName, Object[] args)
    throws UmberClassException
```

The `constructorExists()` method wraps the `Class.getConstructor()` and `Class.getConstructors()` methods, except that it can accept a `Class[]` for the args like those methods, but also the array of objects you might later pass to `newInstance()`. Also, unlike the original Java methods, `constructorExists()` is flexible about primitive parameter types. If it fails to find an exact parameter match, it will re-try with primitive (or non-primitive) types.

Thus, given a constructor of `MyObject(int i)`, the following invocations would return true:

```
// Primitive.
ClassHelper.constructorExists(MyObject.class, new Class[] { Integer.TYPE });
```

```
// Integer class.  
ClassHelper.constructorExists(MyObject.class, new Class[] { Integer.class });  
// Integer object  
ClassHelper.constructorExists(MyObject.class, new Object[] { new Integer (123) });
```

The `newInstance()` methods also use the same expanded rules for dynamically finding class constructors, except you must pass in the actual objects. This makes it possible to implicitly map, for example, an Integer object to a primitive int parameter:

```
// Invoke MyObject(int) with value of 123.  
Object myObj = ClassHelper.newInstance(MyObject.class,  
    new Object[] { new Integer (123) });
```

Chapter 9 - Email

The JavaMail API provides a platform independent way of sending and retrieving email, and does a decent job of covering the subtlety and complexity of email standards like SMTP, POP3, IMAP, and MIME. However, it does require some fairly verbose and confusing code in places, especially when reading and writing MIME emails.

The Umber email API encapsulates most of that complexity by wrapping the JavaMail API in a single simple interface, `IEmailManager`, and by introducing a easy-to-use JavaBean email object, `UmberEmail`, to abstract out the complexity of the MIME API. Umber also introduces a boolean toggle method for switching JavaMail on and off.

The `EmailFactory` class is the entry point to the Umber email API. Its two main methods retrieve an implementation of the `IEmailManager` interface, which by default wraps the JavaMail API. One method accepts a `Map` of JavaMail-style configuration properties (you can also pass a `Properties` object into this method). The other method takes a resource path to a properties file, which is loaded by `ResourceLoader.loadResourceAsProperties()`. See the JavaMail documentation and the `EmailFactory` javadocs for details on the properties.

```
public static IEmailManager getEmail (Map properties)  
    throws UmberEmailException  
public static IEmailManager getEmail (String path)  
    throws UmberEmailException
```

If you invoke `getEmail()` without a valid JavaMail implementation on the

CLASSPATH, it will throw an exception. Alternatively, you can first call `isJavaMailAvailable()`, which will safely check for the JavaMail classes.

```
public static boolean isJavaMailAvailable ()
```

To test your application without accidentally sending real emails, you can call the global `setDebug()` method with `true`. This will cause `getEmail()` to return an alternate implementation that stores emails in memory instead of sending them via JavaMail.

```
public static boolean isDebug ()
public static void setDebug (boolean debug)
```

You can also use a custom implementation of `IMailManager` by setting the "mail.manager.class" property to the fully qualified path of your implementation. The `getEmail()` method will throw an `UmberEmailException` if it fails to load the custom manager

The `IMailManager` interface provides a high level wrapper around the JavaMail API, including methods for sending (SMTP) and receiving (POP3, IMAP) emails. The email manager determines which servers it should use, and how it should connect, from the properties passed into the factory's `getEmail()` method. These properties are available after the fact from the `IMailManager` object:

```
public void setEmailProperties (Map settings)
public Map getEmailProperties ()
```

The retrieval API is very simple, with a method to find how many emails are on the server, and two methods to retrieve emails one at a time, or in batches. The retrieve methods return null or an empty array, respectively, if no mail is on the server. The email manager implicitly connects to the server the first time you call one of these methods.

```
public int getEmailCount ()
    throws UmberEmailException
public UmberEmail retrieveNextEmail ()
    throws UmberEmailException
public UmberEmail[] retrieveNextEmails (int count)
    throws UmberEmailException
```

Sending an email is as easy as creating an `UmberEmail` object and calling `sendEmail()` on it.

```
public void sendEmail (UmberEmail email)
    throws UmberEmailException
```

Although there is no explicit connect method, you must still call the close() method when you are done. This releases the connection to the server and releases any system resources the JavaMail API might have allocated and releases locks on the remote server. Failure to call close() after retrieving email can prevent other applications from accessing the mailbox until the remote lock expires.

```
public void close ()
    throws UmberEmailException
```

It is more important to call close() after using the three retrieval methods. Umber email leaves the connection open for these, to improve performance. Otherwise, it would have to reconnect to the server each time you fetched another message. Since email retrieval is often in an iterative loop, the connection is maintained across successive fetches, and must be released explicitly when the application is finished.

On the other hand, sending an email is a single atomic action, and does not leave a server connection open. Calling close() after sendEmail() is always safe, but is not required.

The UmberEmail class encapsulates a single email message. JavaBean properties define to/from addressing, email headers, and content. The from, to, and cc properties take a String argument, which can be a single email address, a comma-separated list of addresses, or any legal RFC822 address format.

```
public String getFrom ()
public void setFrom (String from)
public String getTo ()
public void setTo (String to)
public String getCc ()
public void setCc (String cc)
public String getSubject ()
public void setSubject (String subject)
```

The headers are stored in a Map, which you can retrieve and modify directly.

```
public Map getHeaders ()
```

The email body is stored in the 'content' property, an Object[] array which can

hold arbitrary Java objects. By convention, a content array with a single element will go out in non-MIME format, while an array with two or more elements will automatically go out as MIME attachments.

```
public Object[] getContent ()
public void setContent (Object content)
public void setContent (Object[] content)
```

By default, Umber email will convert all MIME content elements it doesn't recognize to String form (with `String.valueOf()`) and add it as "text/plain".

If the Umber Bellows package is available, it will automatically convert incoming "text/xml" and "text/html" content to a Bellows Datum tree. If the third-party TagSoup HTML parsing library is also available, the MIME loader will use it to better parse poorly formed HTML; if TagSoup is not on the CLASSPATH, the loader can only handle HTML which is a valid XML document. If Bellows cannot parse the text/xml content, the loader will pass the content through in its original String form.

The `MimeManager` class provides a hook for applications to customize Umber's MIME conversion rules. You can access it from the `IEmailManager` object:

```
public MimeManager getMimeManager ()
```

Each customization is encapsulated in an `IMimeConverter` object. The `MimeManager` will pass each incoming and outgoing MIME attachment through its chain of converters in the reverse order they were added. The most recently added converters take precedence over earlier converters. The built-in converters (for "text/plain", "text/xml", and "text/html") always run last.

The `MimeManager` has two methods for accessing MIME customizations. The first one adds new converters; the second retrieves an array of previously added converters (not including built-ins). The array is a copy of the internal list, so re-ordering it will have no effect on the invocation order.

```
public void addCustomConverter(IMimeConverter converter)
public IMimeConverter[] getCustomConverters()
```

The `IMimeConverter` interface has three methods; all methods should return null if the converter implementation does not handle conversions for the given object.

```
public String getMimeType (Object appObject)
public String convertOutgoing (Object appObject)
```

```
throws UmberEmailException
public Object convertIncoming (String mimeType, Object mimeObject)
throws UmberEmailException
```

The `getMimeType()` and `convertOutgoing()` methods are used to convert application objects into outgoing MIME attachments for `IEmailManager.sendEmail()`. The `getMimeType()` method finds a MIME type for the object; this MIME type is set for the MIME attachment in the outgoing email. The `convertOutgoing()` method performs the actual conversion of the object into an attachment body.

The `convertIncoming()` method handles a MIME attachment from an incoming email. The `mimeType` is pulled from the email attachment, and the `mimeObject` is the body of that attachment.

Chapter 10 - Regular Expressions

Regular expressions are text patterns which can be tested against various text content. Regular expressions can be used for simple true/false matching, or for extracting selected content from a larger string.

Umber Core supports two different regular expression parsers, Jakarta ORO, and the native Jdk1.4 parser. The `RegularExpressionFactory` will load specific parsers, or dynamically search for one:

```
public static IRegularExpression getOROParser ()
public static IRegularExpression getNativeParser ()
public static IRegularExpression getDefaultParser ()
```

The `getDefaultParser()` method first looks for ORO, then falls back on the Jdk1.4 parser if the ORO parser isn't available. Each of these methods will return null if it can't find its desired parser(s).

The factory returns an object which implements the `IRegularExpression` interface in terms of one of the two parser implementations. The basic API consists of a `loadPattern()` method to load the regular expression into the parser and check its validity, and a `matches()` method to test data content against the expression:

```
boolean loadPattern (String pattern)
boolean matches (String text)
```

The code below instantiates a parser, loads it with the regular expression "abc?[def]", and tests a few Strings against it. Normally, you would want to do

something with the return values of the matches() method calls. See the ORO and Jdk1.4 documentation for the final word on valid patterns. For the most part, the two implementations use similar syntax, but the advanced features tend to vary.

```
IRegularExpression parser =
    RegularExpressionFactory.getDefaultParser ();
if (parser != null && parser.loadPattern ("abc?[def]"))
{
    parser.matches ("abcd");    // true
    parser.matches ("abf");    // true
    parser.matches ("abcdef"); // false
    parser.matches ("abc");    // false
}
```

The RegularExpression parser can also extract chunks, or "groups", of data from the text you match against. You can delimit each group with parentheses inside the expression pattern; groups can be nested to arbitrary depths. After calling the matches() method on your input text, you can use the group API to pull out the content, if any, that the parser found inside those group delimiters.

The getGroupCount() method returns the total number of groups within a given pattern. The expression in loadPattern() determines the number of groups. The group count is equivalent to the number of matched parentheses in the pattern. The group() method returns the matched text inside the given group; start() and end() return the starting and ending index offset, respectively, of the given group, relative to the beginning of the entire matched text.

```
int getGroupCount ()
String group (int group)
int start (int group)
int end (int group)
```

As demonstrated below, group zero is always the entire matched expression. The remaining groups, from index 1 to the value of group count, contain the contents inside each pair of parentheses.

```
String regexp = "a(b*)c?((d+)[efg])";

parser.loadPattern (regexp);    // true
parser.matches ("abbdddf");    // true
```

```

int count = parser.getGroupCount () // 3
String group0 = parser.group (0); // "abbdddf"
String group1 = parser.group (1); // "bb"
String group2 = parser.group (2); // "dddf"
String group3 = parser.group (3); // "ddd"

```

Chapter 11 - Velocity Runner

To facilitate code generation and more generic templating, Umber Core provides an easy-to-use wrapper around the Jakarta Velocity template engine.

VelocityFactory insulates your application from direct dependencies to the Velocity jar. As long as you instantiate the VelocityRunner through the factory, it will never throw a ClassNotFoundException, even if Velocity is not on the CLASSPATH. The isVelocityAvailable() method checks for the Velocity jar, returning true if found. The getRunner() method instantiates a VelocityRunner object if it finds Velocity, or null if no Velocity.

```

public static boolean isVelocityAvailable ()
public static VelocityRunner getRunner ()

```

VelocityRunner performs the actual template rendering. The runner's single method is processTemplate(), which takes a Velocity template and a Map of properties to pass into the template.

```

public String processTemplate (String template, Map properties)

```

Here is a code example that passes in two properties, "prop1" and "prop2" (see the Velocity documentation for a full explanation of template rules).

```

VelocityRunner runner = VelocityFactory.getRunner ();
if (runner != null)
{
    String template = "($prop1); ($prop2, $prop2.length())";
    Map props = new HashMap ();
    props.put ("prop1", "value1");
    props.put ("prop2", "value2");

    String result = runner.processTemplate (template, props);
    // "(value1); (value2, 6)"
}

```

To load a template as a resource (from the CLASSPATH or inside a jar), use ResourceLoader:

```
String template = ResourceLoader.loadResourceAsString ("tmpl/template.vm");
```

Umber offers a special Velocity-friendly wrapper for XML content in the Bellows package, VelocityDatumAdapter. By default, a Bellows Datum XML object passed to Velocity will behave like a normal Java object which you can invoke methods on inside the template, just like any Velocity parameter. However, this does not make it easy, or even possible, to use the Bellows query engine. Also, if you are generating Java source code from an XML source document, there is often no way to convert XML content into valid Java identifiers, especially when hyphens are present.

The VelocityDatumAdapter wrapper class makes all of this easier. A Datum inside the wrapper is exported through a set of methods that greatly simplifies template actions on its XML content. The basic wrapper API exposes the Datum.getType() and Datum.getProperty() methods.

```
public VelocityDatumAdapter (Datum root)
public PropertyName getType ()
public PropertyName getProperty (String property)
```

The first getPcdata() method grabs all PCDATA from the current element and any of its children, leaving all whitespace intact. The second getPcdata() method allows you to specify whether you want the whitespace collapsed or not.

The final method, getSafePcdata(), returns a quotable version of the PCDATA, suitable for including in quoted strings in Java. It strips out all line breaks and extra whitespace and escapes all raw quote (") characters with backslashes.

```
public String getPcdata ()
public String getPcdata (boolean collapseWhiteSpace)
public String getSafePcdata ()
```

The getRoot() method provides access to the internal Datum object.

```
public Datum getRoot ()
```

Another critical Bellows feature that DatumAdapter wraps is the Bellows query engine. This lets you select and filter XML data inside a Bellows tree with a simple text query string. For example, the query "root/child" would select all <child> elements who are immediate children of the top-level <root> element. The

query "root/child[@id=child1]" would select only those <child> elements that have an "id" attribute set to the value "child1".

The wrapper query methods run a Bellows query on the embedded Datum object (see Bellows user guide for query rules) and return the results in various formats.

The query() method runs the Bellows query and wraps each returned Datum inside another VelocityDatumAdapter object. This ensures that running a query does not drop the wrapper API. The queryPcdata() method runs the query, then extracts the PCDATA of each result Datum object. The queryProperties() method runs the query, then calls Datum.getStringProperty() on each result and wraps it in a PropertyName.

```
public VelocityDatumAdapter[] query (String query)
public String[] queryPcdata (String query)
public PropertyName[] queryProperties (String query, String attribute)
```

For example, consider this XML document:

```
<my-root name="the-root">
  <child name="child-prop1"/>
  <child name="child-prop2"/>
  <child name="child-prop3"/>
</my-root>
```

and the setup code:

```
Datum xml = ...;
String template = ...;

Map props = new HashMap ();
props.put ("xml", xml);

String result = runner.processTemplate (template, props);
```

The Velocity template can access the XML through the \$xml parameter. Consider this template:

```
#foreach ($child in $xml.query("my-root/child"))
  $child.getProperty("name").BeanPropertyName
#end
```

The #foreach loop iterates through the array of VelocityDatumAdapter wrappers returned from the "my-root/child" query, in this case all three <child> elements.

Each iteration through the loop calls `getProperty()` on the wrapper, which is "child-prop1", etc. wrapped in a `PropertyName` object. Velocity then calls the `PropertyName.getBeanPropertyName()` method and sends that to the output. The final result is:

```
childProp1
childProp2
childProp3
```

If we change the `BeanPropertyName` to `UpperUnderscoredName` like this:

```
#foreach ($child in $xml.query("my-root/child"))
    $child.getProperty("name").BeanPropertyName
#end
```

We get this result instead:

```
CHILD_PROP1
CHILD_PROP2
CHILD_PROP3
```

Again, for more details about the Velocity templating language, see the Velocity documentation at <http://jakarta.apache.org/velocity/>.

Finally, let's have a quick look at a more complicated example, generating Java code from an XML source file. Suppose we wanted to create JavaBean accessor methods for each `<child>` element in the XML sample above. The template for the entire Java class file might look like this:

```
public class $xmlroot.Type.CaseDelimitedName
{
#foreach ($child in $xmlroot.query("my-root/child"))
    private String _$child.getProperty("id").BeanPropertyName;
#end

#foreach ($child in $xmlroot.query("my-root/child"))
#set ( $prop = $child.getProperty("id") )
    public String get$prop.CaseDelimitedName ()
    {
        return _$prop.BeanPropertyName;
    }

public void set$prop.CaseDelimitedName (String $prop.BeanPropertyName)
```

```
{
    _$prop.BeanPropertyName = $prop.BeanPropertyName;
}

#end
```

The generated Java source would look like this:

```
public class MyRoot
{
    private String _childProp1;
    private String _childProp2;
    private String _childProp3;

    public String getChildProp1 ()
    {
        return _childProp1;
    }

    public void setChildProp1 (String childProp1)
    {
        _childProp1 = childProp1;
    }

    public String getChildProp2 ()
    {
        return _childProp2;
    }

    public void setChildProp2 (String childProp2)
    {
        _childProp2 = childProp2;
    }

    public String getChildProp3 ()
    {
        return _childProp3;
    }

    public void setChildProp3 (String childProp3)
    {
        _childProp3 = childProp3;
    }
}
```

Chapter 12 - ASCII Field Parsing

To facilitate the extraction and creation of fixed-length ASCII or binary data, Umber Core offers the `AsciiFieldManager` class. This class converts simple Java objects into a packed ASCII data string according to a field specification described below. It can also take a String of ASCII and explode it into Java objects, to go the reverse direction.

The ASCII field specification is a series of field type and size declarations. Each field contains a numerical length and field type, plus an optional array count, e.g., "4i" or "4i[3]". The field length determines how many ASCII characters of data correspond to a single element of that field data. The array count determines how many consecutive blocks of data to load for that field.

Whitespace in the field spec is completely ignored by the parser, and can be used freely to make the field specs more human legible.

The field type must be one of six categories:

Type	Description	Java Type
----	-----	-----
x	padding	N/A
b	byte	<code>java.lang.Byte</code>
c	character	<code>java.lang.Character</code>
i	integer	<code>java.lang.Integer</code>
s	string	<code>java.lang.String</code>
f	float	<code>java.lang.Double</code>

Non-array field specs are simply the field length and the type. Thus, the field spec "4s 3i" declares seven characters of ASCII data; the first four characters make up a String object and the final three characters are converted into an Integer object. Thus, the ASCII data "1234567" would yield a String value of "1234" and an Integer value of 567. This association works both ways, so conversely a String field of "nope" and an Integer of 34 would pack into ASCII data "nope34".

Array field specs provide an easy way to load large chunks of uniformly sized data fields into a single Java array object. To declare an array spec, append the array size to the field's spec, surrounded by square brackets. For example, a field spec of "2s 2i[4]" declares one two-character String and four two-character Integer objects. With this field spec, the ASCII data "0123456789"

would unpack into a String of "01" and an Integer[] array of [23, 45, 67, 89].

The field type of 'x' represents padding characters, which are ignored by the field manager. Thus, a field spec of "2s 16x 2s" represents a string 20 characters long, with a pair of 2-char fields; the first field is the first two characters and the second field is the last two characters. The 16 characters in the middle are skipped.

The class constructor takes the field spec as its lone argument. A field manager object can be used to read or write many different chunks of data that conform to the same field spec.

```
public AsciiFieldManager (String fieldFormat)
```

These four methods provide information about the field spec, including the number of fields in the spec, and the type, length, and array size of each field.

Non-array fields have an array size of one.

```
public int getFieldCount ()
public Class getFieldTypes (int field)
public int getFieldLength (int field)
public int getFieldArraySize (int field)
```

To extract Java objects from a packed ASCII String, call `extractValue()` with a field index and input data.

```
public Object extractValue (int fieldIndex, String asciiData)
```

In the example below, the "2x" field spec is completely ignored. Notice that it does not affect the field index at all.

```
String data = "1234567890";
AsciiFieldManager mgr = new AsciiFieldManager ("4i 2x 2s");
Integer i = (Integer)mgr.extractValue (0, data); // 1234
String s = (String)mgr.extractValue (1, data); // "78"
```

To load Java objects into packed String data, use the `insertValue()` method. The `fieldIndex` is the field in the field spec you want to populate, the `asciiData` is where the output is written, and `newValue` is the Java object you want written to `asciiData`. The field manager will do its best to coerce the Java Object into an appropriate ASCII form.

```
public void insertValue (int fieldIndex, StringBuffer asciiData,
```

```
Object newValue)
```

The following code uses the previous field spec to populate a StringBuffer.

```
AsciiFieldManager mgr = new AsciiFieldManager ("4i 2x 2s");
StringBuffer buff = new StringBuffer ();
mgr.insertValue (0, new Integer (1234));
mgr.insertValue (1, "78");
String out = buff.toString (); // "1234 78"
```

By default, padding is filled with empty space characters. You can customize that with `setPadding()` and `normalize()`.

```
public void setPadding (String padding)
public void normalize (StringBuffer data)
```

The `setPadding()` method assigns a repeating pattern to fill the padded areas; `normalize()` will write padding to a StringBuffer to match the field spec, but leave data areas untouched.

```
AsciiFieldManager mgr = new AsciiFieldManager ("2i 7x 2s 3x");

// Empty buffer.
StringBuffer buff = new StringBuffer ();
mgr.setPadding ("123");
mgr.normalize (buff); // " 1231231 231"

// Existing content.
buff = new StringBuffer ("aabbccddeeffgg");
mgr.setPadding ("123");
mgr.normalize (buff); // "aa1231231ef231"
```

Chapter 13 - Umber Socket Server

Java threading can be a sticky feature to implement, especially for newer programmers. Add socket programming into the mix, and you're quickly dealing with blocking I/O, socket timeouts, and thread synchronization at the same time.

Umber Core provides the `UmberSocketServer` class to simplify development of client/server socket applications. The socket server class runs a separate thread with the `java.net.SocketServer` instance, polling for incoming client socket connections. As each client connects, the socket server gathers that

connection into a simple pool which the server application can cycle through.

The default constructor takes the socket port number that client applications should connect to.

```
public UmberSocketServer (int port)
```

The alternate constructor takes two millisecond timeout parameters. The first timeout determines how long the server socket will block waiting for an incoming client connection before bailing out and cycling through another polling loop. If this is set too high, the socket server might take a long time to shut down, since it will potentially be stuck in blocking I/O until that timeout lapses. The other timeout is for blocking I/O when reading from each client socket connection. Both timeouts default to 5 second intervals.

```
public UmberSocketServer (int port, int acceptTimeout, int readTimeout)
```

The socket server's port number can be retrieved with `getPort()`:

```
public int getPort ()
```

The socket server does not immediately start polling for incoming connections. You must call `start()` to kick off the thread, and `stop()` to shut it down when you're done. The `isRunning()` method reports the current status of the polling thread.

```
public boolean isRunning ()
public void start ()
    throws UmberIOException
public void stop ()
    throws UmberIOException
```

Once the socket server is running and accepting incoming connections, you can start cycling through them. The easiest way is to call `getNextClient()`, which will cycle through all live connections one by one, or return null if no connections are available.

```
synchronized public ClientConnection getNextClient ()
```

The `UmberSocketServer.ClientConnection` object holds the server end of the socket connection to the client, and also an open `InputStream` and `OutputStream` to read from and write to the client application. The socket server opens these as soon as the client connects, and wraps them in `BufferedInput/OutputStream` for you.

The socket server will also take care of closing the streams and sockets when it shuts down in the stop() method. You can also access the raw Socket.

```
public Socket getSocket ()
public InputStream getInput ()
public OutputStream getOutput ()
```

For a little more control, you can get an array of all current client connections with getAllClients(). It will not reflect new connections, so you should call it periodically to get an up-to-date list. The getClientCount() method gives the number of current client connections.

```
synchronized public ClientConnection[] getAllClients ()
synchronized public int getClientCount ()
```

By default, the socket server will manage all client connections itself. However, you can also explicitly add or remove connections yourself. If you get ahold of a Socket object that you want to add into the pool, wrap it in a ClientConnection object and call addClient() on it.

```
public ClientConnection (Socket socket)
    throws IOException
synchronized public void addClient (ClientConnection client)
```

If you want to close down or get rid of a connection prematurely, call removeClient(), which will close the streams and the sockets for you, and take it out of the pool.

```
synchronized public void removeClient (ClientConnection client)
```