Ada Utility Library Programmer's Guide

Stephane Carrez

2024-09-29

Contents

1 Introduction **4**

2 Installation **5**
2.1 Using Alire . . . 5
2.2 Without Alire . . . 5
2.3 Using . . . 6

3 Files **7**
3.1 Reading and writing . . . 7
3.2 Searching files . . . 7
3.3 Rolling file manager . . . 8
3.4 Path filters . . . 9
3.5 Directory tree walk . . . 10

4 Logging **12**
4.1 Using the log framework . . . 12
4.2 Logger Declaration . . . 12
4.3 Logger Messages . . . 13
4.4 Log Configuration . . . 13
4.4.1 Console appender . . . 15
4.4.2 File appender . . . 15
4.4.3 Rolling file appender . . . 16
4.5 Custom appender . . . 18
4.6 Custom formatter . . . 18

5 Property Files **21**
5.1 File formats . . . 21
5.2 Using property files . . . 21
5.3 Reading JSON property files . . . 22
5.4 Property bundles . . . 23
5.5 Advance usage of properties . . . 23

6 Date Utilities **25**
6.1 Date Operations . . . 25
6.2 RFC7231 Dates . . . 25
6.3 ISO8601 Dates . . . 26
6.4 Localized date formatting . . . 26

7 Ada Beans **30**
7.1 Objects . . . 30
7.2 Object maps . . . 32
7.3 Object vectors . . . 32
7.4 Datasets . . . 33
7.5 Object iterator . . . 34
7.6 Bean Interface . . . 34

8 Command Line Utilities **35**
8.1 Command arguments . . . 35
8.2 Command line driver . . . 35
8.3 Command line parsers . . . 36
8.4 Example . . . 36

9 Serialization of data structures in CSV/JSON/XML **38**
9.1 Introduction . . . 38
9.2 Record Mapping . . . 38
9.3 Mapping Definition . . . 39
9.4 De-serialization . . . 40
9.5 Parser Specificities . . . 41
9.5.1 XML . . . 41
9.5.2 CSV . . . 41

10 HTTP **42**
10.1 Client . . . 42
10.1.1 GET request . . . 42

11 Streams **43**
11.1 Buffered Streams . . . 43
11.2 Texts . . . 44
11.3 File streams . . . 44
11.4 Pipes . . . 44
11.5 Sockets . . . 46
11.6 Raw files . . . 46
11.7 Part streams . . . 46
11.8 Encoder Streams . . . 47
11.9 Base16 Encoding Streams . . . 48
11.10 Base64 Encoding Streams . . . 48
11.11 AES Encoding Streams . . . 48

12 Encoders **50**
12.1 URI Encoder and Decoder . . . 50
12.2 Error Correction Code . . . 50

13 Other utilities **52**
13.0.1 Nullable types . . . 52
13.1 Text Builders . . . 52
13.2 Listeners . . . 53
13.2.1 Creating the listener list . . . 53
13.2.2 Creating the observers . . . 53
13.2.3 Implementing the observer . . . 53
13.2.4 Registering the observer . . . 53
13.2.5 Publishing . . . 54
13.3 Timer Management . . . 54
13.3.1 Timer Creation . . . 54
13.3.2 Timer Main Loop . . . 54
13.4 Executors . . . 55

14 Performance Measurements **56**
14.1 Create the measure set . . . 56
14.2 Measure the implementation . . . 56
14.3 Reporting results . . . 57
14.4 Measure Overhead . . . 57
14.5 What must be measured . . . 57

1 Introduction

The Ada Utility Library provides a collection of utility packages which includes:

- A logging framework close to Java log4j framework,
- A support for properties,
- A serialization/deserialization framework for XML, JSON, CSV,
- Ada beans framework,
- Encoding/decoding framework (Base16, Base32, Base64, SHA, HMAC-SHA, AES256, PBKDF2, ECC),
- A composing stream framework (raw, files, buffers, pipes, sockets, compress, encryption),
- Several concurrency tools (reference counters, counters, pools, fifos, arrays, sequences, executors),
- Process creation and pipes,
- Support for loading shared libraries (on Windows or Unix),
- HTTP client library on top of CURL or AWS.

This document describes how to build the library and how you can use the different features to simplify and help you in your Ada application.

2 Installation

This chapter explains how to build and install the library.

2.1 Using Alire

The Ada Utility Library is available as several Alire crates to simplify the installation and setup your project. Run the following commands to setup your Alire project to use the library:

```
alr index --update-all
alr with utilada
alr with utilada_xml
alr with utilada_unit
alr with utilada_curl
alr with utilada_aws
alr with utilada_lzma
```

2.2 Without Alire

The library needs Alire to compile but you can use it in your project if you build the library with the same compiler and then install it.

The support for AWS, Curl, LZMA and XML/Ada are enabled only when a `HAVE_XXX=yes` configuration variable has defined. Run the setup command that records in the `Makefile.conf` the configuration you want to build:

```
make setup BUILD=debug PREFIX=/build/install HAVE_XML_ADA=yes HAVE_CURL
   =yes HAVE_LZMA=yes HAVE_AWS=yes
```

Then, build:

```
   make
```

After building, it is good practice to run the unit tests before installing the library. The unit tests are built and executed using:

```
make test
```

And unit tests are executed by running the `bin/util_harness` test program.

The installation is done by running the `install` target:

```
make install
```

To use the installed libraries, make sure your `ADA_PROJECT_PATH` contains the directory where you installed the libraries (configured by the `PREFIX=<path>` option in the setup phase). The installed GNAT projects are the same as those used when using Alire.

If you want to install on a specific place, you can change the `prefix` and indicate the installation direction as follows:

```
1 make install PREFIX=/opt
```

2.3 Using

To use the library in an Ada project, add the following line at the beginning of your GNAT project file:

```
1 with "utilada";
```

If you use only a subset of the library, you may use the following GNAT projects:

GNAT project	Description
utilada_core	Provides: Util.Concurrent, Util.Strings, Util.Texts, Util.Locales, Util.Refs, Util.Stacks, Util.Listeners Util.Executors
utilada_base	Provides: Util.Beans, Util.Commands, Util.Dates, Util.Events, Util.Files, Util.Log, Util.Properties, Util.Systems
utilada_sys	Provides: Util.Encoders, Util.Measures, Util.Processes, Util.Serialize, Util.Streams
utilada_lzma	Provides: Util.Encoders.Lzma, Util.Streams.Buffered.Lzma
utilada_aws	Provides HTTP client support using AWS
utilada_curl	Provides HTTP client support using CURL
utilada_http	Provides Util.Http
utilada	Uses all utilada GNAT projects except the unit test library
utilada_unit	Support to write unit tests on top of Ahven or AUnit

3 Files

The `Util.Files` package provides various utility operations around files to help in reading, writing, searching for files in a path. To use the operations described here, use the following GNAT project:

```
1 with "utilada_base";
```

3.1 Reading and writing

To easily get the full content of a file, the `Read_File` procedure can be used. A first form exists that populates a `Unbounded_String` or a vector of strings. A second form exists with a procedure that is called with each line while the file is read. These different forms simplify the reading of files as it is possible to write:

```
1 Content : Ada.Strings.Unbounded.Unbounded_String;
2 Util.Files.Read_File ("config.txt", Content);
```

or

```
1 List : Util.Strings.Vectors.Vector;
2 Util.Files.Read_File ("config.txt", List);
```

or

```
1 procedure Read_Line (Line : in String) is ...
2 Util.Files.Read_File ("config.txt", Read_Line'Access);
```

Similarly, writing a file when you have a string or an `Unbounded_String` is easily written by using `Write_File` as follows:

```
1 Util.Files.Write_File ("config.txt", "full content");
```

3.2 Searching files

Searching for a file in a list of directories can be accomplished by using the `Iterate_Path`, `Iterate_Files_Path` or `Find_File_Path`.

The `Find_File_Path` function is helpful to find a file in some `PATH` search list. The function looks in each search directory for the given file name and it builds and returns the computed path of the first file found in the search list. For example:

```
1 Path : String := Util.Files.Find_File_Path ("ls",
2                                             "/bin:/usr/bin",
```

```
3                                                    ':');
```

This will return `/usr/bin/ls` on most Unix systems.

3.3 Rolling file manager

The `Util.Files.Rolling` package provides a simple support to roll a file based on some rolling policy. Such rolling is traditionally used for file logs to move files to another place when they reach some size limit or when some date conditions are met (such as a day change). The file manager uses a file path and a pattern. The file path is used to define the default or initial file. The pattern is used when rolling occurs to decide how to reorganize files.

The file manager defines a triggering policy represented by `Policy_Type`. It controls when the file rolling must be made.

- `No_Policy`: no policy, the rolling must be triggered manually.
- `Size_Policy`: size policy, the rolling is triggered when the file reaches a given size.
- `Time_Policy`: time policy, the rolling is made when the date/time pattern no longer applies to the active file; the `Interval` configuration defines the period to check for time changes,
- `Size_Time_Policy`: combines the size and time policy, the rolling is triggered when either the file reaches a given size or the date/time pattern no longer applies to the active file.

To control how the rolling is made, the `Strategy_Type` defines the behavior of the rolling.

- `Rollover_Strategy`:
- `Direct_Strategy`:

To use the file manager, the first step is to create an instance and configure the default file, pattern, choose the triggering policy and strategy:

```
1 Manager : Util.Files.Rolling.File_Manager;
2 Manager.Initialize ("dynamo.log", "dynamo-%i.log",
3                     Policy => (Size_Policy, 100_000),
4                     Strategy => (Rollover_Strategy, 1, 10));
```

After the initialization, the current file is retrieved by using the `Get_Current_Path` function and you should call `Is_Rollover_Necessary` before writing content on the file. When it returns `True`, it means you should call the `Rollover` procedure that will perform roll over according to the rolling strategy.

3.4 Path filters

The generic package `Util.Files.Filters` implements a path filter mechanism that emulates the classical `.gitignore` path filter. It defines the `Filter_Type` tagged type to represent and control a set of path patterns associated with values represented by the `Element_Type` generic type. The package must be instantiated with the type representing values associated to path patterns. A typical instantation for an inclusion, exclusion filter such as `.gitignore` could be:

```
1  type Filter_Mode is (Not_Found, Included, Excluded);
2  package Path_Filter is
3     new Util.Files.Filters (Filter_Mode);
```

The `Filter_Type` provides one operation to add a pattern in the filter and associate it with a value. A pattern can contain fixed paths, wildcards or regular expressions. When inserting a filter, you can indicate whether the filter is to be applied locally on recursively (this is similar to the `.gitignore` rules, with a pattern that starts with a `/` or without). Example of pattern setup:

```
1  Filter : Path_Filter.Filter_Type;
2  ...
3  Filter.Insert ("*.o", Recursive => True, Value => Excluded);
4  Filter.Insert ("alire/", Recursive => False, Value => Excluded);
5  Filter.Insert ("docs/*", Recursive => False, Value => Included);
```

The `Match` function looks in the filter for a match and it indicates either that there is a match with a value (`Found`), a match witout a value (`No_Value`) or no match at all (`Not_Found`). When there is a match `Found`, the associated value is retrieved by using `Get_Value`. The `Match` operation is called as follows:

```
1  Result : Path_Filter.Filter_Result := Filter.Match ("test.o");
2  ...
3  if Result.Match = Path_Filter.Found then
4     ...
5  end if;
```

The table below gives results found for several paths and with the filters defined above:

Operation	Result.Match	Path_Filter.Get_Value
Filter.Match ("test.o")	Found	Excluded
Filter.Match ("test.a")	Not_Found	
Filter.Match ("docs/test.o")	Found	Included
Filter.Match ("alire/")	Found	Included

Operation	Result.Match	Path_Filter.Get_Value
Filter.Match ("test/alire")	Not_Found	

It is also possible to use the generic package as a mapping framework to implement a mapping of a path to a string, for example:

```
1  package Language_Mappers is
2     new Util.Files.Filters (Element_Type => String);
```

And filters could be populated as follows:

```
1  Filter : Language_Mappers.Filter_Type;
2  ...
3  Filter.Add ("*.c", True, "C");
4  Filter.Add ("*.adb", True, "Ada");
5  Filter.Add ("*.ads", True, "Ada");
6  Filter.Add ("Makefile", True, "Make");
```

3.5 Directory tree walk

It is sometimes necessary to walk a directory tree while taking into account some inclusion or exclusion patterns or more complex ignore lists. The `Util.Files.Walk` package provides a support to walk such directory tree while taking into account some possible ignore lists such as the `.gitignore` file. The package defines the `Filter_Type` tagged type to represent and control the exclusion or inclusion filters and a second tagged type `Walker_Type` to walk the directory tree.

The `Filter_Type` provides two operations to add patterns in the filter and one operation to check against a path whether it matches a pattern. A pattern can contain fixed paths, wildcards or regular expressions. Similar to `.gitignore` rules, a pattern which starts with a `/` will define a pattern that must match the complete path. Otherwise, the pattern is a recursive pattern. Example of pattern setup:

```
1  Filter : Util.Files.Walk.Filter_Type;
2  ...
3  Filter.Exclude ("*.o");
4  Filter.Exclude ("/alire/");
5  Filter.Include ("/docs/*");
```

The `Match` function looks in the filter for a match. The path could be included, excluded or not found. For example, the following paths will match:

Operation	Result
Filter.Match ("test.o")	Walk.Excluded
Filter.Match ("test.a")	Walk.Not_Found
Filter.Match ("docs/test.o")	Walk.Included
Filter.Match ("alire/")	Walk.Included
Filter.Match ("test/alire")	Walk.Not_Found

To scan a directory tree, the `Walker_Type` must have some of its operations overriden:

- The `Scan_File` should be overriden to be notified when a file is found and handle it.
- The `Scan_Directory` should be overriden to be notified when a directory is entered.
- The `Get_Ignore_Path` is called when entering a new directory. It can be overriden to indicate a path of a file which contains some patterns to be ignored (ex: the `.gitignore` file).

See the `tree.adb` example.

4 Logging

The `Util.Log` package and children provide a simple logging framework inspired from the Java Log4j library. It is intended to provide a subset of logging features available in other languages, be flexible, extensible, small and efficient. Having log messages in large applications is very helpful to understand, track and fix complex issues, some of them being related to configuration issues or interaction with other systems. The overhead of calling a log operation is negligible when the log is disabled as it is in the order of 30ns and reasonable for a file appender has it is in the order of 5us. To use the packages described here, use the following GNAT project:

```
1 with "utilada_base";
```

4.1 Using the log framework

A bit of terminology:

- A *logger* is the abstraction that provides operations to emit a message. The message is composed of a text, optional formatting parameters, a log level and a timestamp.
- A *formatter* is the abstraction that takes the information about the log and its parameters to create the formatted message.
- An *appender* is the abstraction that writes the message either to a console, a file or some other final mechanism. A same log can be sent to several appenders at the same time.
- A *layout* describes how the formatted message, log level, date are used to form the final message. Each appender can be configured with its own layout.

4.2 Logger Declaration

Similar to other logging framework such as Java Log4j and Log4cxx, it is necessary to have an instance of a logger to write a log message. The logger instance holds the configuration for the log to enable, disable and control the format and the appender that will receive the message. The logger instance is associated with a name that is used for the configuration. A good practice is to declare a `Log` instance in the package body or the package private part to make available the log instance to all the package operations. The instance is created by using the `Create` function. The name used for the configuration is free but using the full package name is helpful to control precisely the logs.

```
1 with Util.Log.Loggers;
2 package body X.Y is
3   Log : constant Util.Log.Loggers.Logger := Util.Log.Loggers.Create ("X
      .Y");
4 end X.Y;
```

4.3 Logger Messages

A log message is associated with a log level which is used by the logger instance to decide to emit or drop the log message. To keep the logging API simple and make it easily usable in the application, several operations are provided to write a message with different log level.

A log message is a string that contains optional formatting markers that follow more or less the Java `MessageFormat` class. A parameter is represented by a number enclosed by `{}`. The first parameter is represented by `{0}`, the second by `{1}` and so on. Parameters are replaced in the final message only when the message is enabled by the log configuration. The use of parameters allows to avoid formatting the log message when the log is not used. The log formatter is responsible for creating the message from the format string and the parameters.

The example below shows several calls to emit a log message with different levels:

```
1  Log.Error ("Cannot open file {0}: {1}", Path, "File does not exist");
2  Log.Warn ("The file {0} is empty", Path);
3  Log.Info ("Opening file {0}", Path);
4  Log.Debug ("Reading line {0}", Line);
```

The logger also provides a special `Error` procedure that accepts an Ada exception occurrence as parameter. The exception name and message are printed together with the error message. It is also possible to activate a complete traceback of the exception and report it in the error message. With this mechanism, an exception can be handled and reported easily:

```
1  begin
2     ...
3  exception
4     when E : others =>
5        Log.Error ("Something bad occurred", E, Trace => True);
6  end;
```

4.4 Log Configuration

The log configuration uses property files close to the Apache Log4j and to the Apache Log4cxx configuration files. The configuration file contains several parts to configure the logging framework:

- First, the *appender* configuration indicates the appender that exists and can receive a log message.
- Second, a root configuration allows to control the default behavior of the logging framework. The root configuration controls the default log level as well as the appenders that can be used.
- Last, a logger configuration is defined to control the logging level more precisely for each logger.

The log configuration is loaded with `Initialize` either from a file or from a `Properties` object that has been loaded or populated programatically. The procedure takes two arguments. The first argument must be either a path to the file that must be loaded or the `Properties` object. The second argument is a prefix string which indicates the prefix of configuration properties. Historically, the default prefix used is the string `"log4j."`. Each application can use its own prefix.

```
1  Util.Log.Loggers.Initialize ("config.properties", "log4j.");
```

Here is a simple log configuration that creates a file appender where log messages are written. The file appender is given the name `result` and is configured to write the messages in the file `my-log-file.log`. The file appender will use the `level-message` format for the layout of messages. Last is the configuration of the `X.Y` logger that will enable only messages starting from the `WARN` level.

```
1 log4j.rootCategory=DEBUG,result
2 log4j.appender.result=File
3 log4j.appender.result.File=my-log-file.log
4 log4j.appender.result.layout=level-message
5 log4j.logger.X.Y=WARN
```

By default when the `layout` is not set or has an invalid value, the full message is reported and the generated log messages will look as follows:

```
1 [2018-02-07 20:39:51] ERROR - X.Y - Cannot open file test.txt: File
     does not exist
2 [2018-02-07 20:39:51] WARN  - X.Y - The file test.txt is empty
3 [2018-02-07 20:39:51] INFO  - X.Y - Opening file test.txt
4 [2018-02-07 20:39:51] DEBUG - X.Y - Reading line ......
```

When the `layout` configuration is set to `data-level-message`, the message is printed with the date and message level.

```
1 [2018-02-07 20:39:51] ERROR: Cannot open file test.txt: File does not
     exist
2 [2018-02-07 20:39:51] WARN : The file test.txt is empty
3 [2018-02-07 20:39:51] INFO : X.Y - Opening file test.txt
4 [2018-02-07 20:39:51] DEBUG: X.Y - Reading line ......
```

When the `layout` configuration is set to `level-message`, only the message and its level are reported.

```
1 ERROR: Cannot open file test.txt: File does not exist
2 WARN : The file test.txt is empty
3 INFO : X.Y - Opening file test.txt
4 DEBUG: X.Y - Reading line ......
```

The last possible configuration for `layout` is `message` which only prints the message.

```
Cannot open file test.txt: File does not exist
The file test.txt is empty
Opening file test.txt
Reading line ......
```

4.4.1 Console appender

The `Console` appender uses either `Ada.Text_IO` or a direct write on console to write messages. The default is to use `Ada.Text_IO` and the appender expects standard Ada strings encoded in Latin-1 in the configuration. When the appender gets UTF-8 strings, it should be configured for a direct write on the console. The console appender recognises the following configurations:

Name	Description
layout	Defines the format of the message printed by the appender.
level	Defines the minimum level above which messages are printed.
utc	When 'true' or '1', print the date in UTC instead of local time
stderr	When 'true' or '1', use the console standard error, by default the appender uses the standard output.
utf8	When 'true', use a direct write on the console and avoid using `Ada.Text_IO`.

Example of configuration:

```
log4j.appender.console=Console
log4j.appender.console.level=WARN
log4j.appender.console.layout=level-message
log4j.appender.console.utc=false
```

4.4.2 File appender

The `File` appender recognises the following configurations:

Name	Description
layout	Defines the format of the message printed by the appender.

Name	Description
level	Defines the minimum level above which messages are printed.
utc	When 'true' or '1', print the date in UTC instead of local time
File	The path used by the appender to create the output file.
append	When 'true' or '1', the file is opened in append mode otherwise it is truncated (the default is to truncate).
immediateFlush	When 'true' or '1', the file is flushed after each message log. Immediate flush is useful in some situations to have the log file updated immediately at the expense of slowing down the processing of logs.

4.4.3 Rolling file appender

The `RollingFile` appender recognises the following configurations:

Name	Description
layout	Defines the format of the message printed by the appender.
level	Defines the minimum level above which messages are printed.
utc	When 'true' or '1', print the date in UTC instead of local time
fileName	The name of the file to write to. If the file, or any of its parent directories, do not exist, they will be created.
filePattern	The pattern of the file name of the archived log file. The pattern can contain '%i' which are replaced by a counter incremented at each rollover, a '%d' is replaced by a date pattern.
append	When 'true' or '1', the file is opened in append mode otherwise it is truncated (the default is to truncate).
immediateFlush	When 'true' or '1', the file is flushed after each message log. Immediate flush is useful in some situations to have the log file updated immediately at the expense of slowing down the processing

Name	Description
	of logs.
policy	The triggering policy which drives when a rolling is performed. Possible values are: `none`, `size`, `time`, `size-time`
strategy	The strategy to use to determine the name and location of the archive file. Possible values are: `ascending`, `descending`, and `direct`. Default is `ascending`.
policyInterval	How often a rollover should occur based on the most specific time unit in the date pattern. This indicates the period in seconds to check for pattern change in the `time` or `size-time` policy.
policyMin	The minimum value of the counter. The default value is 1.
policyMax	The maximum value of the counter. Once this values is reached older archives will be deleted on subsequent rollovers. The default value is 7.
minSize	The minimum size the file must have to roll over.

A typical rolling file configuration would look like:

```
log4j.rootCategory=DEBUG,applogger,apperror
log4j.appender.applogger=RollingFile
log4j.appender.applogger.layout=level-message
log4j.appender.applogger.level=DEBUG
log4j.appender.applogger.fileName=logs/debug.log
log4j.appender.applogger.filePattern=logs/debug-%d{YYYY-MM}/debug-%{dd}-%i.log
log4j.appender.applogger.strategy=descending
log4j.appender.applogger.policy=time
log4j.appender.applogger.policyMax=10
log4j.appender.apperror=RollingFile
log4j.appender.apperror.layout=level-message
log4j.appender.apperror.level=ERROR
log4j.appender.apperror.fileName=logs/error.log
log4j.appender.apperror.filePattern=logs/error-%d{YYYY-MM}/error-%{dd}.log
log4j.appender.apperror.strategy=descending
log4j.appender.apperror.policy=time
```

With this configuration, the error messages are written in the `error.log` file and they are rotated on

a day basis and moved in a directory whose name contains the year and month number. At the same time, debug messages are written in the `debug.log` file.

4.5 Custom appender

It is possible to write a customer log appender and use it in the generation of logs. This is done in two steps:

- first by extending the `Util.Log.Appenders.Appender` tagged type and overriding some of the methods to implement the custom log appender and by writing a `Create` function whose role is to create instances of the appender and configure them according to the user configuration.
- second by instantiating the `Util.Log.Appenders.Factories` package. The package is instantiated with the appender's name and the `Create` function. It contains an elaboration body that registers automatically the factory.

For example, the first step could be implemented as follows (methods are not shown):

```
1  type Syslog_Appender (Length : Positive) is
2     new Util.Log.Appenders.Appender (Length) with null record;
3  function Create (Name       : in String;
4                   Properties : in Util.Properties.Manager;
5                   Default    : in Util.Log.Level_Type)
6                return Util.Log.Appenders.Appender_Access;
```

Then, the package is instantiated as follows:

```
1  package Syslog_Factory is
2    new Util.Log.Appenders.Factories (Name   => "syslog",
3                                      Create => Create'Access)
4    with Unreferenced;
```

The appender for `syslog` can be configured as follows to report all errors to `syslog`:

```
1  log4j.logger.X.Y=INFO,console,syslog
2  log4j.appender.test=syslog
3  log4j.appender.test.level=ERROR
```

See the log.adb and syslog_appender.adb example.

4.6 Custom formatter

The formatter is responsible for preparing the message to be displayed by log appenders. It takes the message string and its arguments and builds the message. The same formatted message is given to each log appender. This step is handled by `Format_Message`. Then, each log appender can use the

log appender to format the log event which is composed of the log level, the date of the event, the logger name. This step is handled by `Format_Event`.

Using a custom formatter can be useful to change the message before it is formatted, translate messages, filter messages to hide sensitive information and so on. Implementing a custom formatter is made in three steps:

- first by extending the `Util.Log.Formatters.Formatter` tagged type and overriding one of the `Format_XXX` procedures. The procedure gets the log message passed to the `Debug`, `Info`, `Warn` or `Error` procedure as well as every parameter passed to customize the final message. It must populate a `Builder` object with the formatted message.
- second by writing a `Create` function that allocates an instance of the formatter and customizes it with some configuration properties.
- third by instantiating the `Util.Log.Formatters.Factories` generic package. It contains an elaboration body that registers automatically the factory.

For example, a formatter that translates the message when it is printed can be created. The two first steps could be implemented as follows (method bodies are not shown):

```
1  type NLS_Formatter (Length : Positive) is
2     new Util.Log.Formatters.Formatter (Length) with null record;
3  function Create (Name       : in String;
4                   Properties : in Util.Properties.Manager;
5                   Default    : in Util.Log.Level_Type)
6                return Util.Log.Formatters.Formatter_Access;
```

Then, the package is instantiated as follows:

```
1  package NLS_Factory is
2    new Util.Log.Appenders.Factories (Name   => "NLS",
3                                      Create => Create'Access)
4    with Unreferenced;
```

To use the new registered formatter, it is necessary to declare some minimal configuration. A `log4j.formatter.<name>` definition must be declared for each named formatter where `<name>` is the logical name of the formatter. The property must indicate the factory name that must be used (example: `NLS`). The named formatter can have custom properties and they are passed to the `Create` procedure when it is created. Such properties can be used to customize the behavior of the formatter.

```
1  log4j.formatter.nlsFormatter=NLS
2  log4j.formatter.nlsFormatter.prop1=value1
3  log4j.formatter.nlsFormatter.prop2=value2
```

Once the named formatter is declared, it can be selected for one or several logger by appending the string `:<name>` after the log level. For example:

```
log4j.logger.X.Y=WARN:nlsFormatter
```

With the above configuration, the `X.Y` and its descendant loggers will use the formatter identified by `nlsFormatter`. Note that it is also possible to specify an appender for the configuration:

```
log4j.logger.X.Y=INFO:nlsFormatter,console
```

The above configuration will use the `nlsFormatter` formatter and the `console` appender to write on the console.

A formatter can also be configured specifically for an appender by using the following configuration:

```
log4j.appender.console.formatter=nlsFormatter
```

In that configuration, the `Format_Event` function of the configured formatter will be called to format the log level, date and logger's name.

5 Property Files

The `Util.Properties` package and children implements support to read, write and use property files either in the Java property file format or the Windows INI configuration file. Each property is assigned a key and a value. The list of properties are stored in the `Util.Properties.Manager` tagged record and they are indexed by the key name. A property is therefore unique in the list. Properties can be grouped together in sub-properties so that a key can represent another list of properties. To use the packages described here, use the following GNAT project:

```
1 with "utilada_base";
```

5.1 File formats

The property file consists of a simple name and value pair separated by the = sign. Thanks to the Windows INI file format, list of properties can be grouped together in sections by using the `[section-name]` notation.

```
1 test.count=20
2 test.repeat=5
3 [FileTest]
4 test.count=5
5 test.repeat=2
```

5.2 Using property files

An instance of the `Util.Properties.Manager` tagged record must be declared and it provides various operations that can be used. When created, the property manager is empty. One way to fill it is by using the `Load_Properties` procedure to read the property file. Another way is by using the `Set` procedure to insert or change a property by giving its name and its value.

In this example, the property file `test.properties` is loaded and assuming that it contains the above configuration example, the `Get ("test.count")` will return the string `"20"`. The property `test.repeat` is then modified to have the value `"23"` and the properties are then saved in the file.

```
1 with Util.Properties;
2 ...
3    Props : Util.Properties.Manager;
4    ...
5       Props.Load_Properties (Path => "test.properties");
6       Ada.Text_IO.Put_Line ("Count: " & Props.Get ("test.count");
7       Props.Set ("test.repeat", "23");
```

```
8        Props.Save_Properties (Path => "test.properties");
```

To be able to access a section from the property manager, it is necessary to retrieve it by using the `Get` function and giving the section name. For example, to retrieve the `test.count` property of the `FileTest` section, the following code is used:

```
1     FileTest : Util.Properties.Manager := Props.Get ("FileTest");
2     ...
3        Ada.Text_IO.Put_Line ("[FileTest] Count: "
4                              & FileTest.Get ("test.count");
```

When getting or removing a property, the `NO_PROPERTY` exception is raised if the property name was not found in the map. To avoid that exception, it is possible to check whether the name is known by using the `Exists` function.

```
1     if Props.Exists ("test.old_count") then
2        ... --  Property exist
3     end if;
```

5.3 Reading JSON property files

The `Util.Properties.JSON` package provides operations to read a JSON content and put the result in a property manager. The JSON content is flattened into a set of name/value pairs. The JSON structure is reflected in the name. Example:

```
1 { "id": "1",                                    id         -> 1
2   "info": { "name": "search",                   info.name  -> search
3             "count": "12",                      info.count -> 12
4             "data": { "value": "empty" }},      info.data.value  -> empty
5   "count": 1                                    count      -> 1
6 }
```

To get the value of a JSON property, the user can use the flatten name. For example:

```
1   Value : constant String := Props.Get ("info.data.value");
```

The default separator to construct a flatten name is the dot (`.`) but this can be changed easily when loading the JSON file by specifying the desired separator:

```
1   Util.Properties.JSON.Read_JSON (Props, "config.json", "|");
```

Then, the property will be fetch by using:

```
1   Value : constant String := Props.Get ("info|data|value");
```

5.4 Property bundles

Property bundles represent several property files that share some overriding rules and capabilities. Their introduction comes from Java resource bundles which allow to localize easily some configuration files or some message. When loading a property bundle a locale is defined to specify the target language and locale. If a specific property file for that locale exists, it is used first. Otherwise, the property bundle will use the default property file.

A rule exists on the name of the specific property locale file: it must start with the bundle name followed by _ and the name of the locale. The default property file must be the bundle name. For example, the bundle `dates` is associated with the following property files:

```
dates.properties           Default values (English locale)
dates_fr.properties        French locale
dates_de.properties        German locale
dates_es.properties        Spain locale
```

Because a bundle can be associated with one or several property files, a specific loader is used. The loader instance must be declared and configured to indicate one or several search directories that contain property files.

```
with Util.Properties.Bundles;
...
   Loader : Util.Properties.Bundles.Loader;
   Bundle : Util.Properties.Bundles.Manager;
   ...
   Util.Properties.Bundles.Initialize (Loader,
                                       "bundles;/usr/share/bundles");
   Util.Properties.Bundles.Load_Bundle (Loader, "dates", "fr", Bundle);
   Ada.Text_IO.Put_Line (Bundle.Get ("util.month1.long");
```

In this example, the `util.month1.long` key is first searched in the `dates_fr` French locale and if it is not found it is searched in the default locale.

The restriction when using bundles is that they don't allow changing any value and the `NOT_WRITEABLE` exception is raised when one of the `Set` operation is used.

When a bundle cannot be loaded, the `NO_BUNDLE` exception is raised by the `Load_Bundle` operation.

5.5 Advance usage of properties

The property manager holds the name and value pairs by using an Ada Bean object.

It is possible to iterate over the properties by using the `Iterate` procedure that accepts as parameter

a `Process` procedure that gets the property name as well as the property value. The value itself is passed as an `Util.Beans.Objects.Object` type.

6 Date Utilities

The `Util.Dates` package provides various date utilities to help in formatting and parsing dates in various standard formats. It completes the standard `Ada.Calendar.Formatting` and other packages by implementing specific formatting and parsing. To use the packages described here, use the following GNAT project:

```
with "utilada_base";
```

6.1 Date Operations

Several operations allow to compute from a given date:

- `Get_Day_Start`: The start of the day (0:00),
- `Get_Day_End`: The end of the day (23:59:59),
- `Get_Week_Start`: The start of the week,
- `Get_Week_End`: The end of the week,
- `Get_Month_Start`: The start of the month,
- `Get_Month_End`: The end of the month

The `Date_Record` type represents a date in a split format allowing to access easily the day, month, hour and other information.

```
Now        : Ada.Calendar.Time := Ada.Calendar.Clock;
Week_Start : Ada.Calendar.Time := Get_Week_Start (Now);
Week_End   : Ada.Calendar.Time := Get_Week_End (Now);
```

6.2 RFC7231 Dates

The RFC 7231 defines a standard date format that is used by HTTP headers. The `Util.Dates.RFC7231` package provides an `Image` function to convert a date into that target format and a `Value` function to parse such format string and return the date.

```
   Now  : constant Ada.Calendar.Time := Ada.Calendar.Clock;
   S    : constant String := Util.Dates.RFC7231.Image (Now);
   Date : Ada.Calendar.time := Util.Dates.RFC7231.Value (S);
```

A `Constraint_Error` exception is raised when the date string is not in the correct format.

6.3 ISO8601 Dates

The ISO8601 defines a standard date format that is commonly used and easily parsed by programs. The `Util.Dates.ISO8601` package provides an `Image` function to convert a date into that target format and a `Value` function to parse such format string and return the date.

```
1   Now  : constant Ada.Calendar.Time := Ada.Calendar.Clock;
2   S    : constant String := Util.Dates.ISO8601.Image (Now);
3   Date : Ada.Calendar.time := Util.Dates.ISO8601.Value (S);
```

A `Constraint_Error` exception is raised when the date string is not in the correct format.

6.4 Localized date formatting

The `Util.Dates.Formats` provides a date formatting and parsing operation similar to the Unix `strftime`, `strptime` or the `GNAT.Calendar.Time_IO`. The localization of month and day labels is however handled through `Util.Properties.Bundle` (similar to the Java world). Unlike `strftime` and `strptime`, this allows to have a multi-threaded application that reports dates in several languages. The `GNAT.Calendar.Time_IO` only supports English and this is the reason why it is not used here.

The date pattern recognizes the following formats:

Format	Description
%a	The abbreviated weekday name according to the current locale.
%A	The full weekday name according to the current locale.
%b	The abbreviated month name according to the current locale.
%h	Equivalent to %b. (SU)
%B	The full month name according to the current locale.
%c	The preferred date and time representation for the current locale.
%C	The century number (year/100) as a 2-digit integer. (SU)
%d	The day of the month as a decimal number (range 01 to 31).
%D	Equivalent to %m/%d/%y
%e	Like %d, the day of the month as a decimal number, but a leading zero is replaced by a space. (SU)
%F	Equivalent to %Y-%m-%d (the ISO 8601 date format). (C99)

Format	Description
%G	The ISO 8601 week-based year
%H	The hour as a decimal number using a 24-hour clock (range 00 to 23).
%I	The hour as a decimal number using a 12-hour clock (range 01 to 12).
%j	The day of the year as a decimal number (range 001 to 366).
%k	The hour (24 hour clock) as a decimal number (range 0 to 23);
%l	The hour (12 hour clock) as a decimal number (range 1 to 12);
%m	The month as a decimal number (range 01 to 12).
%M	The minute as a decimal number (range 00 to 59).
%n	A newline character. (SU)
%p	Either "AM" or "PM"
%P	Like %p but in lowercase: "am" or "pm"
%r	The time in a.m. or p.m. notation.
	In the POSIX locale this is equivalent to %I:%M:%S %p. (SU)
%R	The time in 24 hour notation (%H:%M).
%s	The number of seconds since the Epoch, that is,
	since 1970-01-01 00:00:00 UTC. (TZ)
%S	The second as a decimal number (range 00 to 60).
%t	A tab character. (SU)
%T	The time in 24 hour notation (%H:%M:%S). (SU)
%u	The day of the week as a decimal, range 1 to 7,
	Monday being 1. See also %w. (SU)
%U	The week number of the current year as a decimal
	number, range 00 to 53
%V	The ISO 8601 week number
%w	The day of the week as a decimal, range 0 to 6,
	Sunday being 0. See also %u.
%W	The week number of the current year as a decimal number,

Format	Description
	range 00 to 53
%x	The preferred date representation for the current locale without the time.
%X	The preferred time representation for the current locale without the date.
%y	The year as a decimal number without a century (range 00 to 99).
%Y	The year as a decimal number including the century.
%z	The timezone as hour offset from GMT.
%Z	The timezone or name or abbreviation.

The following strftime flags are ignored:

Format	Description
%E	Modifier: use alternative format, see below. (SU)
%O	Modifier: use alternative format, see below. (SU)

SU: Single Unix Specification C99: C99 standard, POSIX.1-2001

See strftime (3) and strptime (3) manual page

To format and use the localize date, it is first necessary to get a bundle for the `dates` so that date elements are translated into the given locale.

```
1  Factory     : Util.Properties.Bundles.Loader;
2  Bundle      : Util.Properties.Bundles.Manager;
3  ...
4     Load_Bundle (Factory, "dates", "fr", Bundle);
```

The date is formatted according to the pattern string described above. The bundle is used by the formatter to use the day and month names in the expected locale.

```
1  Date : String := Util.Dates.Formats.Format (Pattern => Pattern,
2                                              Date    => Ada.Calendar.
                                                  Clock,
3                                              Bundle  => Bundle);
```

To parse a date according to a pattern and a localization, the same pattern string and bundle can be used and the `Parse` function will return the date in split format.

```
1  Result : Date_Record := Util.Dates.Formats.Parse (Date    => Date,
2                                                    Pattern => Pattern,
3                                                    Bundle  => Bundle);
```

7 Ada Beans

A Java Bean(http://en.wikipedia.org/wiki/JavaBean) is an object that allows to access its properties through getters and setters. Java Beans rely on the use of Java introspection to discover the Java Bean object properties.

An Ada Bean has some similarities with the Java Bean as it tries to expose an object through a set of common interfaces. Since Ada does not have introspection, some developer work is necessary. The Ada Bean framework consists of:

- An `Object` concrete type that allows to hold any data type such as boolean, integer, floats, strings, dates and Ada bean objects.
- A `Bean` interface that exposes a `Get_Value` and `Set_Value` operation through which the object properties can be obtained and modified.
- A `Method_Bean` interface that exposes a set of method bindings that gives access to the methods provided by the Ada Bean object.

The benefit of Ada beans comes when you need to get a value or invoke a method on an object but you don't know at compile time the object or method. That step being done later through some external configuration or presentation file.

The Ada Bean framework is the basis for the implementation of Ada Server Faces and Ada EL. It allows the presentation layer to access information provided by Ada beans.

To use the packages described here, use the following GNAT project:

```
1 with "utilada_base";
```

7.1 Objects

The `Util.Beans.Objects` package provides a data type to manage entities of different types by using the same abstraction. The `Object` type allows to hold various values of different types.

An `Object` can hold one of the following values:

- a boolean,
- a long long integer,
- a date,
- a string,
- a wide wide string,
- an array of objects,
- a generic data bean,

- a map of objects,
- a vector of object

Several operations are provided to convert a value into an `Object`.

```
1  with Util.Beans.Objects;
2    Value : Util.Beans.Objects.Object
3       := Util.Beans.Objects.To_Object (String '("something"));
4    Value := Value + To_Object (String '("12"));
5    Value := Value - To_Object (Integer (3));
```

The package provides various operations to check, convert and use the `Object` type.

Name	Description
Is_Empty	Returns true if the object is the empty string or empty list
Is_Null	Returns true if the object does not contain any value
Is_Array	Returns true if the object is an array
Get_Type	Get the type of the object
To_String	Converts the object to a string
To_Wide_Wide_String	Convert to a wide wide string
To_Unbounded_String	Convert to an unbounded string
To_Boolean	Convert to a boolean
To_Integer	Convert to an integer
To_Long_Integer	Convert to a long integer
To_Long_Long_Integer	Convert to a long long integer
To_Float	Convert to a float
To_Long_Float	Convert to a long float
To_Long_Long_Float	Convert to a long long float
To_Duration	Convert to a duration
To_Bean	Convert to an access to the Read_Only_Bean'Class

Conversion to a time or enumeration is provided by specific packages.

The support for enumeration is made by the generic package `Util.Beans.Objects.Enums` which must be instantiated with the enumeration type. Example of instantiation:

```
with Util.Beans.Objects.Enum;
...
   type Color_Type is (GREEN, BLUE, RED, BROWN);
   package Color_Enum is
      new Util.Beans.Objects.Enum (Color_Type);
```

Then, two functions are available to convert the enum value into an `Object` or convert back the `Object` in the enum value:

```
Color : Object := Color_Enum.To_Object (BLUE);
C : Color_Type := Color_Enum.To_Value (Color);
```

7.2 Object maps

The `Util.Beans.Objects.Maps` package provides a map of objects with a `String` as key. This allows to associated names to objects. To create an instance of the map, it is possible to use the `Create` function as follows:

```
with Util.Beans.Objects.Maps;
...
   Person : Util.Beans.Objects.Object := Util.Beans.Objects.Maps.Create
      ;
```

Then, it becomes possible to populate the map with objects by using the `Set_Value` procedure as follows:

```
Util.Beans.Objects.Set_Value (Person, "name",
                              To_Object (Name));
Util.Beans.Objects.Set_Value (Person, "last_name",
                              To_Object (Last_Name));
Util.Beans.Objects.Set_Value (Person, "age",
                              To_Object (Age));
```

Getting a value from the map is done by using the `Get_Value` function:

```
Name : Util.Beans.Objects.Object := Get_Value (Person, "name");
```

It is also possible to iterate over the values of the map by using the `Iterate` procedure or by using the iterator support provided by the `Util.Beans.Objects.Iterators` package.

7.3 Object vectors

The `Util.Beans.Objects.Vectors` package provides a vector of objects. To create an instance of the vector, it is possible to use the `Create` function as follows:

```
with Util.Beans.Objects.Vectors;
...
   List : Util.Beans.Objects.Object := Util.Beans.Objects.Vectors.
      Create;
```

7.4 Datasets

The `Datasets` package implements the `Dataset` list bean which defines a set of objects organized in rows and columns. The `Dataset` implements the `List_Bean` interface and allows to iterate over its rows. Each row defines a `Bean` instance and allows to access each column value. Each column is associated with a unique name. The row `Bean` allows to get or set the column by using the column name.

```
with Util.Beans.Objects.Datasets;
...
   Set : Util.Beans.Objects.Datasets.Dataset_Access
       := new Util.Beans.Objects.Datasets.Dataset;
```

After creation of the dataset instance, the first step is to define the columns that composed the list. This is done by using the `Add_Column` procedure:

```
Set.Add_Column ("name");
Set.Add_Column ("email");
Set.Add_Column ("age");
```

To populate the dataset, the package only provide the `Append` procedure which adds a new row and calls a procedure whose job is to fill the columns of the new row. The procedure gets the row as an array of `Object`:

```
procedure Fill (Row : in out Util.Beans.Objects.Object_Array) is
begin
   Row (Row'First) := To_Object (String '("Yoda"));
   Row (Row'First + 1) := To_Object (String '("Yoda@Dagobah"));
   Row (Row'First + 2) := To_Object (Integer (900));
end Fill;
Set.Append (Fill'Access);
```

The dataset instance is converted to an `Object` by using the `To_Object` function. Note that the default behavior of `To_Object` is to take the ownership of the object and hence it will be released automatically.

```
List : Util.Beans.Objects.Object
   := Util.Beans.Objects.To_Object (Set);
```

7.5 Object iterator

Iterators are provided by the `Util.Beans.Objects.Iterators` package. The iterator instance is created by using either the `First` or `Last` function on the object to iterate.

```
with Util.Beans.Objects.Iterators;
...
   Iter : Util.Beans.Objects.Iterators.Iterator
      := Util.Beans.Objects.Iterators.First (Object);
```

The iterator is used in conjunction with its `Has_Element` function and either its `Next` or `Previous` procedure. The current element is obtained by using the `Element` function. When the object being iterated is a map, a key can be associated with the element and is obtained by the `Key` function.

```
while Util.Beans.Objects.Iterators.Has_Element (Iter) loop
   declare
      Item : Object := Util.Beans.Objects.Iterators.Element (Iter);
      Key  : String := Util.Beans.Objects.Iterators.Key (Iter);
   begin
      ...
      Util.Beans.Objects.Iterators.Next (Iter);
   end;
end loop;
```

7.6 Bean Interface

An Ada Bean is an object which implements the `Util.Beans.Basic.Readonly_Bean` or the `Util.Beans.Basic.Bean` interface. By implementing these interface, the object provides a behavior that is close to the Java Beans: a getter and a setter operation are available.

8 Command Line Utilities

The `Util.Commands` package provides a support to help in writing command line applications. It allows to have several commands in the application, each of them being identified by a unique name. Each command has its own options and arguments. The command line support is built around several children packages.

The `Util.Commands.Drivers` package is a generic package that must be instantiated to define the list of commands that the application supports. It provides operations to register commands and then to execute them with a list of arguments. When a command is executed, it gets its name, the command arguments and an application context. The application context can be used to provide arbitrary information that is needed by the application.

The `Util.Commands.Parsers` package provides the support to parse the command line arguments.

The `Util.Commands.Consoles` package is a generic package that can help for the implementation of a command to display its results. Its use is optional.

8.1 Command arguments

The `Argument_List` interface defines a common interface to get access to the command line arguments. It has several concrete implementations. This is the interface type that is used by commands registered and executed in the driver.

The `Default_Argument_List` gives access to the program command line arguments through the `Ada.Command_Line` package.

The `String_Argument_List` allows to split a string into a list of arguments. It can be used to build new command line arguments.

8.2 Command line driver

The `Util.Commands.Drivers` generic package provides a support to build command line tools that have different commands identified by a name. It defines the `Driver_Type` tagged record that provides a registry of application commands. It gives entry points to register commands and execute them.

The `Context_Type` package parameter defines the type for the `Context` parameter that is passed to the command when it is executed. It can be used to provide application specific context to the command.

The `Config_Parser` describes the parser package that will handle the analysis of command line options. To use the GNAT options parser, it is possible to use the `Util.Commands.Parsers.GNAT_Parser` package.

The `IO` package parameter allows to control the operations that the command line support will use to print some message on the console. When strings are encoded in Latin-1, it is possible to give the `Util.Commands.Text_IO` package that will use the standard `Ada.Text_IO` package to write on the console. When strings are encoded in UTF-8, it is best to use the `Util.Commands.Raw_IO` package that will write the strings unmodified on the console.

8.3 Command line parsers

Parsing command line arguments before their execution is handled by the `Config_Parser` generic package. This allows to customize how the arguments are parsed.

The `Util.Commands.Parsers.No_Parser` package can be used to execute the command without parsing its arguments.

The `Util.Commands.Parsers.GNAT_Parser.Config_Parser` package provides support to parse command line arguments by using the `GNAT Getopt` support.

8.4 Example

First, an application context type is defined to allow a command to get some application specific information. The context type is passed during the instantiation of the `Util.Commands.Drivers` package and will be passed to commands through the `Execute` procedure.

```
1  type Context_Type is limited record
2     ... --  Some application specific data
3  end record;
4  package Drivers is
5    new Util.Commands.Drivers
6      (Context_Type  => Context_Type,
7       Config_Parser => Util.Commands.Parsers.GNAT_Parser.Config_Parser,
8       IO            => Util.Commands.Text_IO,
9       Driver_Name   => "tool");
```

Then an instance of the command driver must be declared. Commands are then registered to the command driver so that it is able to find them and execute them.

```
1  Driver : Drivers.Driver_Type;
```

A command can be implemented by a simple procedure or by using the `Command_Type` abstract tagged record and implementing the `Execute` procedure:

```
procedure Command_1 (Name    : in String;
                     Args    : in Argument_List'Class;
                     Context : in out Context_Type);
type My_Command is new Drivers.Command_Type with null record;
procedure Execute (Command : in out My_Command;
                   Name    : in String;
                   Args    : in Argument_List'Class;
                   Context : in out Context_Type);
```

Commands are registered during the application initialization. And registered in the driver by using the `Add_Command` procedure:

```
Driver.Add_Command (Name => "cmd1",
                    Description => "",
                    Handler => Command_1'Access);
```

A command is executed by giving its name and a list of arguments. By using the `Default_Argument_List` type, it is possible to give to the command the application command line arguments.

```
Ctx   : Context_Type;
Args  : Util.Commands.Default_Argument_List (0);
...
Driver.Execute ("cmd1", Args, Ctx);
```

9 Serialization of data structures in CSV/JSON/XML

9.1 Introduction

The `Util.Serialize` package provides a customizable framework to serialize and de-serialize data structures in CSV, JSON and XML. It is inspired from the Java XStream library.

9.2 Record Mapping

The serialization relies on a mapping that must be provided for each data structure that must be read. Basically, it consists in writing an enum type, a procedure and instantiating a mapping package. Let's assume we have a record declared as follows:

```
type Address is record
  City      : Unbounded_String;
  Street    : Unbounded_String;
  Country   : Unbounded_String;
  Zip       : Natural;
end record;
```

The enum type shall define one value for each record member that has to be serialized/deserialized.

```
 type Address_Fields is (FIELD_CITY, FIELD_STREET, FIELD_COUNTRY,
    FIELD_ZIP);
```

The de-serialization uses a specific procedure to fill the record member. The procedure that must be written is in charge of writing one field in the record. For that it gets the record as an in out parameter, the field identification and the value.

```
procedure Set_Member (Addr  : in out Address;
                      Field : in Address_Fields;
                      Value : in Util.Beans.Objects.Object) is
begin
   case Field is
     when FIELD_CITY =>
       Addr.City := To_Unbounded_String (Value);

     when FIELD_STREET =>
       Addr.Street := To_Unbounded_String (Value);

     when FIELD_COUNTRY =>
       Addr.Country := To_Unbounded_String (Value);

     when FIELD_ZIP =>
        Addr.Zip := To_Integer (Value);
   end case;
```

```
18 end Set_Member;
```

The procedure will be called by the CSV, JSON or XML reader when a field is recognized.

The serialization to JSON or XML needs a function that returns the field value from the record value and the field identification. The value is returned as a **Util.Beans.Objects.Object** type which can hold a string, a wide wide string, a boolean, a date, an integer or a float.

```
1  function Get_Member (Addr  : in Address;
2                       Field : in Address_Fields) return Util.Beans.
                           Objects.Object is
3  begin
4     case Field is
5        when FIELD_CITY =>
6           return Util.Beans.Objects.To_Object (Addr.City);
7
8        when FIELD_STREET =>
9           return Util.Beans.Objects.To_Object (Addr.Street);
10
11       when FIELD_COUNTRY =>
12          return Util.Beans.Objects.To_Object (Addr.Country);
13
14       when FIELD_ZIP =>
15          return Util.Beans.Objects.To_Object (Addr.Zip);
16
17    end case;
18 end Get_Member;
```

A mapping package has to be instantiated to provide the necessary glue to tie the set procedure to the framework.

```
1  package Address_Mapper is
2    new Util.Serialize.Mappers.Record_Mapper
3       (Element_Type        => Address,
4        Element_Type_Access => Address_Access,
5        Fields              => Address_Fields,
6        Set_Member          => Set_Member);
```

Note: a bug in the gcc compiler does not allow to specify the **!Get_Member** function in the generic package. As a work-arround, the function must be associated with the mapping using the **Bind** procedure.

9.3 Mapping Definition

The mapping package defines a `Mapper` type which holds the mapping definition. The mapping definition tells a mapper what name correspond to the different fields. It is possible to define several mappings for the same record type. The mapper object is declared as follows:

```
Address_Mapping : Address_Mapper.Mapper;
```

Then, each field is bound to a name as follows:

```
Address_Mapping.Add_Mapping ("city", FIELD_CITY);
Address_Mapping.Add_Mapping ("street", FIELD_STREET);
Address_Mapping.Add_Mapping ("country", FIELD_COUNTRY);
Address_Mapping.Add_Mapping ("zip", FIELD_ZIP);
```

Once initialized, the same mapper can be used read several files in several threads at the same time (the mapper is only read by the JSON/XML parsers).

9.4 De-serialization

To de-serialize a JSON object, a parser object is created and one or several mappings are defined:

```
Reader : Util.Serialize.IO.JSON.Parser;
...
   Reader.Add_Mapping ("address", Address_Mapping'Access);
```

For an XML de-serialize, we just have to use another parser:

```
Reader : Util.Serialize.IO.XML.Parser;
...
   Reader.Add_Mapping ("address", Address_Mapping'Access);
```

For a CSV de-serialize, we just have to use another parser:

```
Reader : Util.Serialize.IO.CSV.Parser;
...
   Reader.Add_Mapping ("", Address_Mapping'Access);
```

The next step is to indicate the object that the de-serialization will write into. For this, the generic package provided the `Set_Context` procedure to register the root object that will be filled according to the mapping.

```
Addr : aliased Address;
...
  Address_Mapper.Set_Context (Reader, Addr'Access);
```

The `Parse` procedure parses a file using a CSV, JSON or XML parser. It uses the mappings registered by `Add_Mapping` and fills the objects registered by `Set_Context`. When the parsing is successful, the `Addr` object will hold the values.

```
   Reader.Parse (File);
```

9.5 Parser Specificities

9.5.1 XML

XML has attributes and entities both of them being associated with a name. For the mapping, to specify that a value is stored in an XML attribute, the name must be prefixed by the **@** sign (this is very close to an XPath expression). For example if the `city` XML entity has an `id` attribute, we could map it to a field `FIELD_CITY_ID` as follows:

```
1 Address_Mapping.Add_Mapping ("city/@id", FIELD_CITY_ID);
```

9.5.2 CSV

A CSV file is flat and each row is assumed to contain the same kind of entities. By default the CSV file contains as first row a column header which is used by the de-serialization to make the column field association. The mapping defined through `Add_Mapping` uses the column header name to indicate which column correspond to which field.

If a CSV file does not contain a column header, the mapping must be created by using the default column header names (Ex: A, B, C, ..., AA, AB, ...). The parser must be told about this lack of column header:

```
1    Parser.Set_Default_Headers;
```

10 HTTP

The `Util.Http` package provides a set of APIs that allows applications to use the HTTP protocol. It defines a common interface on top of CURL and AWS so that it is possible to use one of these two libraries in a transparent manner.

10.1 Client

The Util.Http.Clients package defines a set of API for an HTTP client to send requests to an HTTP server.

10.1.1 GET request

To retrieve a content using the HTTP GET operation, a client instance must be created. The response is returned in a specific object that must therefore be declared:

```
1 Http     : Util.Http.Clients.Client;
2 Response : Util.Http.Clients.Response;
```

Before invoking the GET operation, the client can setup a number of HTTP headers.

```
1 Http.Add_Header ("X-Requested-By", "wget");
```

The GET operation is performed when the Get procedure is called:

```
1 Http.Get ("http://www.google.com", Response);
```

Once the response is received, the Response object contains the status of the HTTP response, the HTTP reply headers and the body. A response header can be obtained by using the Get_Header function and the body using Get_Body:

```
1 Body : constant String := Response.Get_Body;
```

11 Streams

The `Util.Streams` package provides several types and operations to allow the composition of input and output streams. Input streams can be chained together so that they traverse the different stream objects when the data is read from them. Similarly, output streams can be chained and the data that is written will traverse the different streams from the first one up to the last one in the chain. During such traversal, the stream object is able to bufferize the data or make transformations on the data.

The `Input_Stream` interface represents the stream to read data. It only provides a `Read` procedure. The `Output_Stream` interface represents the stream to write data. It provides a `Write`, `Flush` and `Close` operation.

To use the packages described here, use the following GNAT project:

```
1 with "utilada_sys";
```

11.1 Buffered Streams

The `Output_Buffer_Stream` and `Input_Buffer_Stream` implement an output and input stream respectively which manages an output or input buffer. The data is first written to the buffer and when the buffer is full or flushed, it gets written to the target output stream.

The `Output_Buffer_Stream` must be initialized to indicate the buffer size as well as the target output stream onto which the data will be flushed. For example, a pipe stream could be created and configured to use the buffer as follows:

```
1 with Util.Streams.Buffered;
2 with Util.Streams.Pipes;
3 ...
4    Pipe   : aliased Util.Streams.Pipes.Pipe_Stream;
5    Buffer : Util.Streams.Buffered.Output_Buffer_Stream;
6    ...
7       Buffer.Initialize (Output => Pipe'Unchecked_Access,
8                          Size => 1024);
```

In this example, the buffer of 1024 bytes is configured to flush its content to the pipe input stream so that what is written to the buffer will be received as input by the program. The `Output_Buffer_Stream` provides write operation that deal only with binary data (`Stream_Element`). To write text, it is best to use the `Print_Stream` type from the `Util.Streams.Texts` package as it extends the `Output_Buffer_Stream` and provides several operations to write character and strings.

The `Input_Buffer_Stream` must also be initialized to also indicate the buffer size and either an input stream or an input content. When configured, the input stream is used to fill the input stream

buffer. The buffer configuration is very similar as the output stream:

```
with Util.Streams.Buffered;
with Util.Streams.Pipes;
...
   Pipe   : aliased Util.Streams.Pipes.Pipe_Stream;
   Buffer : Util.Streams.Buffered.Input_Buffer_Stream;
   ...
      Buffer.Initialize (Input => Pipe'Unchecked_Access, Size => 1024);
```

In this case, the buffer of 1024 bytes is filled by reading the pipe stream, and thus getting the program's output.

11.2 Texts

The `Util.Streams.Texts` package implements text oriented input and output streams. The `Print_Stream` type extends the `Output_Buffer_Stream` to allow writing text content.

The `Reader_Stream` type extends the `Input_Buffer_Stream` and allows to read text content.

11.3 File streams

The `Util.Streams.Files` package provides input and output streams that access files on top of the Ada `Stream_IO` standard package. The `File_Stream` implements both the `Input_Stream` and `Output_Stream` interfaces. The stream is opened by using the `Open` or `Create` procedures.

```
with Util.Streams.Files;
...
  In_Stream : Util.Streams.Files.File_Stream;
  In_Stream.Open (Mode => Ada.Streams.Stream_IO.In_File, "cert.pem");
```

11.4 Pipes

The `Util.Streams.Pipes` package defines a pipe stream to or from a process. It allows to launch an external program while getting the program standard output or providing the program standard input. The `Pipe_Stream` type represents the input or output stream for the external program. This is a portable interface that works on Unix and Windows.

The process is created and launched by the `Open` operation. The pipe allows to read or write to the process through the `Read` and `Write` operation. It is very close to the *popen* operation provided by the C stdio library. First, create the pipe instance:

```
with Util.Streams.Pipes;
...
   Pipe : aliased Util.Streams.Pipes.Pipe_Stream;
```

The pipe instance can be associated with only one process at a time. The process is launched by using the `Open` command and by specifying the command to execute as well as the pipe redirection mode:

- `READ` to read the process standard output,
- `WRITE` to write the process standard input.

For example to run the `ls -l` command and read its output, we could run it by using:

```
Pipe.Open (Command => "ls -l", Mode => Util.Processes.READ);
```

The `Pipe_Stream` is not buffered and a buffer can be configured easily by using the `Input_Buffer_Stream` type and connecting the buffer to the pipe so that it reads the pipe to fill the buffer. The initialization of the buffer is the following:

```
with Util.Streams.Buffered;
...
   Buffer : Util.Streams.Buffered.Input_Buffer_Stream;
   ...
   Buffer.Initialize (Input => Pipe'Unchecked_Access, Size => 1024);
```

And to read the process output, one can use the following:

```
Content : Ada.Strings.Unbounded.Unbounded_String;
...
Buffer.Read (Into => Content);
```

The pipe object should be closed when reading or writing to it is finished. By closing the pipe, the caller will wait for the termination of the process. The process exit status can be obtained by using the `Get_Exit_Status` function.

```
Pipe.Close;
if Pipe.Get_Exit_Status /= 0 then
   Ada.Text_IO.Put_Line ("Command exited with status "
                         & Integer'Image (Pipe.Get_Exit_Status));
end if;
```

You will note that the `Pipe_Stream` is a limited type and thus cannot be copied. When leaving the scope of the `Pipe_Stream` instance, the application will wait for the process to terminate.

Before opening the pipe, it is possible to have some control on the process that will be created to configure:

- The shell that will be used to launch the process,

- The process working directory,
- Redirect the process output to a file,
- Redirect the process error to a file,
- Redirect the process input from a file.

All these operations must be made before calling the `Open` procedure.

11.5 Sockets

The Util.Streams.Sockets package defines a socket stream.

11.6 Raw files

The Util.Streams.Raw package provides a stream directly on top of file system operations read and write.

11.7 Part streams

The `Input_Part_Stream` is an input stream which decomposes an input stream in several parts separated by well known and fixed boundaries. It can be used to read multipart streams, certificate files, private and public keys and others. The example below shows how to read a file composed of several parts separated by well defined text boundaries.

```
1  with Util.Streams.Files;
2  with Util.Streams.Buffered.Parts;
3  ...
4    In_Stream   : aliased Util.Streams.Files.File_Stream;
5    Part_Stream : Util.Streams.Buffered.Parts.Input_Part_Stream;
```

With the above declarations, the `Input_Part_Stream` is configured to read from the `File_Stream` by using the `Initialize` procedure and giving a buffer size. The buffer size must be large enough to hold the largest fixed boundary plus some extra.

```
1  Part_Stream.Initialize (Input => In_Stream'Unchecked_Access, Size =>
       4096);
```

Once it is configured, the first boundary to stop at is configured by using the `Set_Boundary` procedure. The example below is intended to extract the certificate from a PEM file. The certificate (encoded in Base64) is enclosed in two different markers. The first boundary is first defined as follows:

```
1  Part_Stream.Set_Boundary ("-----BEGIN CERTIFICATE-----" & ASCII.LF);
```

After calling `Set_Boundary`, we can start reading the `Part_Stream` and it will stop when the boundary string is found. If we want to drop content until the first boundary is found, we can loop until the boundary is found. To extract the certificate content, we want to skip everything until the first boundary is found in the stream:

```
while not Part_Stream.Is_Eob loop
  Part_Stream.Read (Item);
end loop;
```

Once a boundary is reached, trying to read from the stream will raise the standard `Data_Error` exception. We can either use `Next_Part` to prepare and read for a next part with the same boundary or call `Set_Boundary` with another boundary. To extract the certificate content, we can do:

```
Part_Stream.Set_Boundary ("-----END CERTIFICATE-----" & ASCII.LF);
Part_Stream.Read (Content);
```

11.8 Encoder Streams

The `Util.Streams.Buffered.Encoders` is a generic package which implements an encoding or decoding stream through the `Transformer` interface. The generic package must be instantiated with a transformer type. The stream passes the data to be written to the `Transform` method of that interface and it makes transformations on the data before being written.

The AES encoding stream is created as follows:

```
package Encoding is
  new Util.Streams.Buffered.Encoders (Encoder => Util.Encoders.AES.
      Encoder);
```

and the AES decoding stream is created with:

```
package Decoding is
  new Util.Streams.Buffered.Encoders (Encoder => Util.Encoders.AES.
      Decoder);
```

The encoding stream instance is declared:

```
   Encode : Util.Streams.Buffered.Encoders.Encoder_Stream;
```

The encoding stream manages a buffer that is used to hold the encoded data before it is written to the target stream. The `Initialize` procedure must be called to indicate the target stream, the size of the buffer and the encoding format to be used.

```
 Encode.Initialize (Output => File'Access, Size => 4096, Format => "
     base64");
```

The encoding stream provides a `Produces` procedure that reads the encoded stream and write the result in another stream. It also provides a `Consumes` procedure that encodes a stream by reading its content and write the encoded result to another stream.

11.9 Base16 Encoding Streams

The `Util.Streams.Base16` package provides streams to encode and decode the stream using Base16.

11.10 Base64 Encoding Streams

The `Util.Streams.Base64` package provides streams to encode and decode the stream using Base64.

11.11 AES Encoding Streams

The `Util.Streams.AES` package define the `Encoding_Stream` and `Decoding_Stream` types to encrypt and decrypt using the AES cipher. Before using these streams, you must use the `Set_Key` procedure to setup the encryption or decryption key and define the AES encryption mode to be used. The following encryption modes are supported:

- AES-ECB
- AES-CBC
- AES-PCBC
- AES-CFB
- AES-OFB
- AES-CTR

The encryption and decryption keys are represented by the `Util.Encoders.Secret_Key` limited type. The key cannot be copied, has its content protected and will erase the memory once the instance is deleted. The size of the encryption key defines the AES encryption level to be used:

- Use 16 bytes, or `Util.Encoders.AES.AES_128_Length` for AES-128,
- Use 24 bytes, or `Util.Encoders.AES.AES_192_Length` for AES-192,
- Use 32 bytes, or `Util.Encoders.AES.AES_256_Length` for AES-256.

Other key sizes will raise a pre-condition or constraint error exception. The recommended key size is 32 bytes to use AES-256. The key could be declared as follows:

```
Key : Util.Encoders.Secret_Key
          (Length => Util.Encoders.AES.AES_256_Length);
```

The encryption and decryption key are initialized by using the `Util.Encoders.Create` operations or by using one of the key derivative functions provided by the `Util.Encoders.KDF` package. A simple string password is created by using:

```
Password_Key : constant Util.Encoders.Secret_Key
          := Util.Encoders.Create ("mysecret");
```

Using a password key like this is not the good practice and it may be useful to generate a stronger key by using one of the key derivative function. We will use the PBKDF2 HMAC-SHA256 with 20000 loops (see RFC 8018):

```
Util.Encoders.KDF.PBKDF2_HMAC_SHA256 (Password => Password_Key,
                                      Salt     => Password_Key,
                                      Counter  => 20000,
                                      Result   => Key);
```

To write a text, encrypt the content and save the file, we can chain several stream objects together. Because they are chained, the last stream object in the chain must be declared first and the first element of the chain will be declared last. The following declaration is used:

```
   Out_Stream   : aliased Util.Streams.Files.File_Stream;
   Cipher       : aliased Util.Streams.AES.Encoding_Stream;
   Printer      : Util.Streams.Texts.Print_Stream;
```

The stream objects are chained together by using their `Initialize` procedure. The `Out_Stream` is configured to write on the `encrypted.aes` file. The `Cipher` is configured to write in the `Out_Stream` with a 32Kb buffer. The `Printer` is configured to write in the `Cipher` with a 4Kb buffer.

```
   Out_Stream.Initialize (Mode => Ada.Streams.Stream_IO.In_File,
                          Name => "encrypted.aes");
   Cipher.Initialize (Output => Out_Stream'Unchecked_Access,
                      Size   => 32768);
   Printer.Initialize (Output => Cipher'Unchecked_Access,
                       Size   => 4096);
```

The last step before using the cipher is to configure the encryption key and modes:

```
   Cipher.Set_Key (Secret => Key, Mode => Util.Encoders.AES.ECB);
```

It is now possible to write the text by using the `Printer` object:

```
Printer.Write ("Hello world!");
```

12 Encoders

The `Util.Encoders` package defines the `Encoder` and `Decoder` types which provide a mechanism to transform a stream from one format into another format. The basic encoder and decoder support `base16`, `base32`, `base64`, `base64url` and `sha1`. The following code extract will encode in `base64`:

```
C : constant Encoder := Util.Encoders.Create ("base64");
S : constant String := C.Encode ("Ada is great!");
```

and the next code extract will decode the `base64`:

```
D : constant Decoder := Util.Encoders.Create ("base64");
S : constant String := D.Decode ("QWRhIGlzIGdyZWF0IQ==");
```

To use the packages described here, use the following GNAT project:

```
with "utilada_sys";
```

12.1 URI Encoder and Decoder

The `Util.Encoders.URI` package provides operations to encode and decode using the URI percent encoding and decoding scheme. A string encoded using percent encoding as described in RFC 3986 is simply decoded as follows:

```
Decoded : constant String := Util.Encoders.URI.Decode ("%20%2F%3A");
```

To encode a string, one must choose the character set that must be encoded and then call the `Encode` function. The character set indicates those characters that must be percent encoded. Two character sets are provided,

- `HREF_STRICT` defines a strict character set to encode all reserved characters as defined by RFC 3986. This is the default.
- `HREF_LOOSE` defines a character set that does not encode the reserved characters such as `-_.+!*'(),%#@?=;:/&$`.

```
Encoded : constant String := Util.Encoders.URI.Encode (" /:");
```

12.2 Error Correction Code

The `Util.Encoders.ECC` package provides operations to support error correction codes. The error correction works on blocks of 256 or 512 bytes and can detect 2-bit errors and correct 1-bit error. The

ECC uses only three additional bytes. The ECC algorithm implemented by this package is implemented by several NAND Flash memory. It can be used to increase the robustness of data to bit-tempering when the data is restored from an external storage (note that if the external storage has its own ECC correction, adding another software ECC correction will probably not help).

The ECC code is generated by using the `Make` procedure that gets a block of 256 or 512 bytes and produces the 3 bytes ECC code. The ECC code must be saved together with the data block.

```
1 Code : Util.Encoders.ECC.ECC_Code;
2 ...
3 Util.Encoders.ECC.Make (Data, Code);
```

When reading the data block, you can verify and correct it by running again the `Make` procedure on the data block and then compare the current ECC code with the expected ECC code produced by the first call. The `Correct` function is then called with the data block, the expected ECC code that was saved with the data block and the computed ECC code.

```
1 New_Code : Util.Encoders.ECC.ECC_Code;
2 ...
3 Util.Encoders.ECC.Make (Data, New_Code);
4 case Util.Encoders.ECC.Correct (Data, Expect_Code, New_Code) is
5    when NO_ERROR | CORRECTABLE_ERROR => ...
6    when others => ...
7 end case;
```

13 Other utilities

13.0.1 Nullable types

Sometimes it is necessary to represent a simple data type with an optional boolean information that indicates whether the value is valid or just null. The concept of nullable type is often used in databases but also in JSON data representation. The Util.Nullables package provides several standard type to express the null capability of a value.

By default a nullable instance is created with the null flag set.

13.1 Text Builders

The `Util.Texts.Builders` generic package was designed to provide string builders. The interface was designed to reduce memory copies as much as possible.

- The `Builder` type holds a list of chunks into which texts are appended.
- The builder type holds an initial chunk whose capacity is defined when the builder instance is declared.
- There is only an `Append` procedure which allows to append text to the builder. This is the only time when a copy is made.
- The package defines the `Iterate` operation that allows to get the content collected by the builder. When using the `Iterate` operation, no copy is performed since chunks data are passed passed by reference.
- The type is limited to forbid copies of the builder instance.

First, instantiate the package for the element type (eg, String):

```
1  package String_Builder is new Util.Texts.Builders (Character, String);
```

Declare the string builder instance with its initial capacity:

```
1  Builder : String_Builder.Builder (256);
```

And append to it:

```
1  String_Builder.Append (Builder, "Hello");
```

To get the content collected in the builder instance, write a procedure that receives the chunk data as parameter:

```
1  procedure Collect (Item : in String) is ...
```

And use the `Iterate` operation:

```
String_Builder.Iterate (Builder, Collect'Access);
```

13.2 Listeners

The `Listeners` package implements a simple observer/listener design pattern. A subscriber registers to a list. When a change is made on an object, the application can notify the subscribers which are then called with the object.

13.2.1 Creating the listener list

The listeners list contains a list of listener interfaces.

```
L : Util.Listeners.List;
```

The list is heterogeneous meaning that several kinds of listeners could be registered.

13.2.2 Creating the observers

First the `Observers` package must be instantiated with the type being observed. In the example below, we will observe a string:

```
package String_Observers is new Util.Listeners.Observers (String);
```

13.2.3 Implementing the observer

Now we must implement the string observer:

```
type String_Observer is new String_Observer.Observer with null record;
procedure Update (List : in String_Observer; Item : in String);
```

13.2.4 Registering the observer

An instance of the string observer must now be registered in the list.

```
O : aliased String_Observer;
L.Append (O'Access);
```

13.2.5 Publishing

Notifying the listeners is done by invoking the `Notify` operation provided by the `String_Observers` package:

```
1  String_Observer.Notify (L, "Hello");
```

13.3 Timer Management

The Util.Events.Timers package provides a timer list that allows to have operations called on regular basis when a deadline has expired. It is very close to the Ada.Real_Time.Timing_Events package but it provides more flexibility by allowing to have several timer lists that run independently. Unlike the GNAT implementation, this timer list management does not use tasks at all. The timer list can therefore be used in a mono-task environment by the main process task. Furthermore you can control your own task priority by having your own task that uses the timer list.

The timer list is created by an instance of Timer_List:

```
1  Manager : Util.Events.Timers.Timer_List;
```

The timer list is protected against concurrent accesses so that timing events can be setup by a task but the timer handler is executed by another task.

13.3.1 Timer Creation

A timer handler is defined by implementing the Timer interface with the Time_Handler procedure. A typical timer handler could be declared as follows:

```
1  type Timeout is new Timer with null record;
2  overriding procedure Time_Handler (T : in out Timeout);
3  My_Timeout : aliased Timeout;
```

The timer instance is represented by the Timer_Ref type that describes the handler to be called as well as the deadline time. The timer instance is initialized as follows:

```
1  T : Util.Events.Timers.Timer_Ref;
2  Manager.Set_Timer (T, My_Timeout'Access, Ada.Real_Time.Seconds (1));
```

13.3.2 Timer Main Loop

Because the implementation does not impose any execution model, the timer management must be called regularly by some application's main loop. The Process procedure executes the timer handler

that have elapsed and it returns the deadline to wait for the next timer to execute.

```
Deadline : Ada.Real_Time.Time;
loop
   ...
   Manager.Process (Deadline);
   delay until Deadline;
end loop;
```

13.4 Executors

The `Util.Executors` generic package defines a queue of work that will be executed by one or several tasks. The `Work_Type` describes the type of the work and the `Execute` procedure will be called by the task to execute the work. After instantiation of the package, an instance of the `Executor_Manager` is created with a number of desired tasks. The tasks are then started by calling the `Start` procedure.

A work object is added to the executor's queue by using the `Execute` procedure. The work object is added in a concurrent fifo queue. One of the task managed by the executor manager will pick the work object and run it.

14 Performance Measurements

Performance measurements is often made using profiling tools such as GNU gprof or others. This profiling is however not always appropriate for production or release delivery. The mechanism presented here is a lightweight performance measurement that can be used in production systems.

The Ada package `Util.Measures` defines the types and operations to make performance measurements. It is designed to be used for production and multi-threaded environments.

14.1 Create the measure set

Measures are collected in a `Measure_Set`. Each measure has a name, a counter and a sum of time spent for all the measure. The measure set should be declared as some global variable. The implementation is thread safe meaning that a measure set can be used by several threads at the same time. It can also be associated with a per-thread data (or task attribute).

To declare the measure set, use:

```
1  with Util.Measures;
2     ...
3     Perf : Util.Measures.Measure_Set;
```

14.2 Measure the implementation

A measure is made by creating a variable of type `Stamp`. The declaration of this variable marks the beginning of the measure. The measure ends at the next call to the `Report` procedure.

```
1  with Util.Measures;
2  ...
3    declare
4       Start : Util.Measures.Stamp;
5    begin
6       ...
7       Util.Measures.Report (Perf, Start, "Measure for a block");
8    end;
```

When the `Report` procedure is called, the time that elapsed between the creation of the `Start` variable and the procedure call is computed. This time is then associated with the measure title and the associated counter is incremented. The precision of the measured time depends on the system. On GNU/Linux, it uses `gettimeofday`.

If the block code is executed several times, the measure set will report the number of times it was executed.

14.3 Reporting results

After measures are collected, the results can be saved in a file or in an output stream. When saving the measures, the measure set is cleared.

```
1  Util.Measures.Write (Perf, "Title of measures",
2                       Ada.Text_IO.Standard_Output);
```

14.4 Measure Overhead

The overhead introduced by the measurement is quite small as it does not exceeds 1.5 us on a 2.6 Ghz Core Quad.

14.5 What must be measured

Defining a lot of measurements for a production system is in general not very useful. Measurements should be relatively high level measurements. For example:

- Loading or saving a file
- Rendering a page in a web application
- Executing a database query