John Barnes

Ada Rationale 2012

Predefined Library

Courtesy of

AdaCore

Rationale for Ada 2012: 6 Predefined library

John Barnes
John Barnes Informatics, 11 Albert Road, Caversham, Reading RG4 7AN, UK; Tel: +44 118 947 4125; email: jgpb@jbinfo.demon.co.uk

Abstract

This paper describes various relatively minor improvements to the predefined library in Ada 2012. The major changes concerning the container library will be described in a later paper.

Keywords: rationale, Ada 2012

1 Overview of changes

The WG9 guidance document [1] does not specifically identify problems in this area other than through a general exhortation to remedy shortcomings.

We have already discussed the additional library packages in the area of tasking and real-time in a previous paper. There are also many additional library packages concerning containers and these will be discussed in a later paper. The following Ada issues cover the relevant changes in other areas and are described in detail in this paper:

1 Bounded containers and other container issues

31 Add a From parameter to Find_Token

49 Extend file name processing in Ada.Directories

127 Adding locale capabilities

137 String encoding package

185 Wide_Character and Wide_Wide_Character classification and folding

233 Questions on locales

266 Use latest version of ISO/IEC 10646

283 Stream_IO should be preelaborated

285 Defaulted environment variable queries

286 Internationalization of Ada

These changes can be grouped as follows.

A number of enhancements concern strings and characters. These include comprehensive new packages to support conversions between strings (and wide strings and wide-wide strings) and the UTF-8 and UTF-16 encodings (137). It is important to note that Ada 2012 directly supports source code in UTF-8 (286). Additional facilities are also provided for the classification of characters and new packages added for similar operations on wide characters and wide wide characters (185, 266). A minor change is the provision of a further procedure Find_Token with an additional parameter giving the start of the search (31).

The file name processing in Ada.Directories is enhanced to overcome some shortcomings (49).

A new package is added to enable a program to identify the locale in which it is being used (127, 233).

There are a number of additional facilities regarding hashing and case insensitive comparisons. The hashing issues really relate to containers but are briefly mentioned here for completeness (1, 286).

Finally, other improvements are that the package Ada.Streams.Stream_IO is now preelaborated (283) and that an additional function Value is added to the package Ada.Environment_Variables (285).

2 Strings and characters

Ada 95 added a number of packages for manipulating strings and characters. Three child packages of Ada.Strings enable the manipulation of fixed length, bounded and unbounded strings. They are Ada.Strings.Fixed, Ada.Strings.Bounded and Ada.Strings.Unbounded. The packages have many subprograms with similar facilities.

In particular there are functions Index and Index_Non_Blank which search through a string and return the index of the first character satisfying some criteria and procedures Find_Token which search through a string and find the first instance of a slice satisfying some other criteria.

As originally defined in Ada 95 these subprograms all started the search at the beginning of the string. This proved to be somewhat inconvenient and so in Ada 2005, versions of the functions Index and Index_Non_Blank with an extra parameter From were added to enable the search to be started at any position. However, the fact that versions of the procedures Find_Token with an extra parameter From should also have been added was overlooked. This is remedied in Ada 2012.

So in Ada 2012 corresponding additional subprograms Find_Token are added to the appropriate packages. They are

```
procedure Find_Token(Source: in String;
                     Set: in Maps.Character_Set;
                     From: in Positive;
                     Test: in Membership;
                     First: out Positive;
                     Last: out Natural);

procedure Find_Token(Source: in Bounded_String;
                     Set: in Maps.Character_Set;
                     From: in Positive;
                     Test: in Membership;
                     First: out Positive;
                     Last: out Natural);

procedure Find_Token(Source: in Unbounded_String;
                     Set: in Maps.Character_Set;
                     From: in Positive;
                     Test: in Membership;
                     First: out Positive;
                     Last: out Natural);
```

Note also that the wording for Find_Token is modified to make it clear that the values of First and Last identify the longest possible slice starting at From. If no characters satisfy the criteria then First is set to From and Last is set to zero.

The existing procedures Find_Token are now defined as calls of the new ones with From set to Source'First.

The encodings UTF-8 and UTF-16 are now widely used but Ada 2005 provides no mechanisms to convert between these encodings and the types String, Wide_String, and Wide_Wide_String.

The encoding UTF-8 works in terms of raw bytes and is straightforward; it is defined in Annex D of ISO/IEC 10646. However, UTF-16 comes in two forms according to whether the arrangement of two bytes into a 16-bit word uses big-endian or little-endian packing. So there are two forms UTF-16BE and UTF-16LE; they are defined in Annex C of ISO/IEC 10646.

The different encodings can be distinguished by a special value known as a BOM (Byte Order Mark) at the start of the string. So we have BOM_8, BOM_16BE, BOM_16LE, and just BOM_16 (for wide strings).

To support these encodings, Ada 2012 includes the following five new packages

Ada.Strings.UTF_Encoding
Ada.Strings.UTF_Encoding.Conversions
Ada.Strings.UTF_Encoding.Strings
Ada.Strings.UTF_Encoding.Wide_Strings
Ada.Strings.UTF_Encoding.Wide_Wide_Strings

The first package declares items that are used by the other packages. It is

```
package Ada.Strings.UTF_Encoding is
  pragma Pure(UTF_Encoding);

  type Encoding_Scheme is (UTF_8, UTF_16BE, UTF_16LE);

  subtype UTF_String is String;
  subtype UTF_8_String is String;
  subtype UTF_16_Wide_String is Wide_String;

  Encoding_Error: exception;

  BOM_8: constant UTF_8_String :=
                                  Character'Val(16#EF#) &
                                  Character'Val(16#BB#) &
                                  Character'Val(16#BF#);

  BOM_16BE: constant UTF_String :=
                                  Character'Val(16#FE#) &
                                  Character'Val(16#FF#);

  BOM_16LE: constant UTF_String :=
                                  Character'Val(16#FF#) &
                                  Character'Val(16#FE#);

  BOM_16: constant UTF_16_Wide_String := (1 => Wide_Character'Val(16#FEFF#);

  function Encoding(Item: UTF_String; Default: Encoding_Scheme := UTF_8)
                                                                  return
Encoding_Scheme;

end Ada.Strings.UTF_Encoding;
```

Note that the encoded forms are actually still held in objects of type String or Wide_String. However, in order to aid understanding, the subtypes UTF_String, UTF_8_String and UTF_16_Wide_String are introduced and these should be used when referring to objects holding the encoded forms.

The type Encoding_Scheme defines the various schemes. Note that an encoded string might or might not start with the identifying BOM; it is optional. The function Encoding takes a UTF_String (that is a plain old string), checks the BOM if present and returns the value of Encoding_Scheme identifying the scheme. If there is no BOM then it returns the value of the parameter Default which itself by default is UTF_8.

Note carefully that the function Encoding does not do any encoding – that is done be functions Encode in the other packages which will be described in a moment. Note also that there is no

corresponding function Encoding for wide strings; that is because there is only one relevant scheme corresponding to UTF_16_Wide_String, namely that with BOM_16.

We will now look at the other packages. The package UTF_Encoding.Strings contains functions Encode and Decode which convert between the raw type String and the UTF forms. Similar packages apply to wide and wide wide strings. The package UTF_Encoding.Conversions contains functions Convert which convert between the various UTF forms.

The package for the type String is

```
package Ada.Strings.UTF_Encoding.Strings is
  pragma Pure(Strings);

  function Encode(Item: String; Output_Scheme: Encoding_Scheme;
                                        Output_BOM: Boolean := False) return
UTF_String;

  function Encode(Item: String; Output_BOM: Boolean := False) return UTF_8_String;

  function Encode(Item: String; Output_BOM: Boolean := False)
                                                                  return
UTF_16_Wide_String;

  function Decode(Item: UTF_String; Input_Scheme: Encoding_Scheme) return String;

  function Decode(Item: UTF_8_String;) return String;

  function Decode(Item: UTF_16_Wide_String;) return String;

end Ada.Strings.UTF_Encoding.Strings;
```

The functions Encode take a string and return it encoded. The first function has a parameter Output_Scheme which determines whether the encoding is to be to UTF_8, UTF_16BE or UTF_16LE. The second function is provided as a convenience for the common case of encoding to UTF_8 and the third function is necessary for encoding to UTF_16_Wide_String. In all cases there is a final optional parameter indicating whether or not an appropriate BOM is to be placed at the start of the encoded string.

The functions Decode do the reverse. Thus the first function takes a value of subtype UTF_String and a parameter Input_Scheme giving the scheme to be used and returns the decoded string. If a BOM is present which does not match the Input_Scheme, then the exception Encoding_Error is raised. The second function is a convenience for the common case of decoding from UTF_8 and the third function is necessary for decoding from UTF_16_Wide_String; again, if a BOM is present that does not match the expected scheme then Encoding_Error is raised.

In all cases all the strings returned have a lower bound of 1.

The packages UTF_Encoding.Wide_Strings and UTF_Encoding.Wide_Wide_Strings are identical except that the type String is replaced by Wide_String or Wide_Wide_String throughout.

Finally, the package for converting between the various UTF forms is as follows

```
package Ada.Strings.UTF_Encoding.Conversions is
  pragma Pure(Conversions);

  function Convert(Item: UTF_String;
                   Input_Scheme: Encoding_Scheme
                   Output_Scheme: Encoding_Scheme;
                   Output_BOM: Boolean := False)
                                          return UTF_String;
```

```
      function Convert(Item: UTF_String;
                       Input_Scheme: Encoding_Scheme
                       Output_BOM: Boolean := False) return UTF_16_Wide_String;

      function Convert(Item: UTF_8_String;
                       Output_BOM: Boolean := False) return UTF_16_Wide_String;

      function Convert(Item: UTF_16_Wide_String;
                       Output_Scheme: Encoding_Scheme;
                       Output_BOM: Boolean := False) return UTF_String;

      function Convert(Item: UTF_16_Wide_String;
                       Output_BOM: Boolean := False) return UTF_8_String;

   end Ada.Strings.UTF_Encoding.Conversions;
```

The purpose of these should be obvious. The first converts between encodings held as strings with parameters indicating both the Input_Scheme and the Output_Scheme. If the input string has a BOM that does not match the Input_Scheme then the exception Encoding_Error is raised. The final optional parameter indicates whether or not an appropriate BOM is to be placed at the start of the converted string.

The other functions convert between UTF encodings held as strings and wide strings. Two give the explicit Input_Scheme or Output_Scheme and two are provided for convenience for the common case of UTF_8.

The final topic in this section concerns the classification and folding of characters and strings. The package Ada.Characters.Handling was introduced in Ada 95; this contains various classification functions such as Is_Lower, Is_Digit and so on. This package also contains functions such as To_Upper and To_Lower which convert characters to upper case or lower case; such conversions are often referred to as case folding operations.

These facilities are extended in Ada 2012 by the addition of a few more classification functions in the package Ada.Characters.Handling plus similar packages named Ada.Wide_Characters.Handling for dealing with wide characters and Ada.Wide_Wide_Characters.Handling for dealing with wide wide characters.

It should be noticed that these new packages are children of Ada.Wide_Characters and Ada.Wide_Wide_Characters respectively. These packages were introduced in Ada 2005 but are empty other than for pragmas Pure.

The additional functions in Ada.Characters.Handling are

```
   function Is_Line_Terminator ...
   function Is_Mark(Item: Character) return Boolean;
   function Is_Other ...
   function Is_Punctuation_Connector ...
   function Is_Space ...
```

In each case they have a single parameter Item of type Character and return a result of type Boolean.

The meanings are as follows

Is_Line_Terminator – returns True if Item is one of Line_Feed (10), Line_Tabulation (11), Form_Feed (12), Carriage_Return (13), or Next_Line (133).

Is_Mark – always returns False.

Is_Other_Format – returns True if Item is Soft_Hyphen (171).

Is_Punctuation_Connector – returns True if Item is Low_Line (95); this is often known as Underscore.

Is_Space – returns True if Item is Space (32) or No_Break_Space (160).

Readers might feel that Is_Mark is a foolish waste of time. However, it is introduced because the corresponding functions in the new packages for wide and wide wide characters can return True.

An important point is that these classifications enable a compiler to analyze Ada source code without direct reference to the definition of ISO/IEC 10646. Note further that case insensitive text comparison which is useful for the analysis of identifiers is now provided by new functions described in Section 5 below.

The new package Wide_Characters.Handling is very similar to the package Characters.Handling (as modified by the additional functions just described) with Character and String everywhere replaced by Wide_Character and Wide_String. However, there are no functions corresponding to Is_Basic, Is_ISO_646, To_Basic and To_ISO_646. In the case of Is_Basic this is because there is no categorization of Basic in 10646. In the case of ISO-646 it is not really necessary because it would seem rather unlikely that one would want to check a wide character WC to see if it was one of the 7-bit ISO-646 set. In any event, one could always write

```
WC in Wide_Characters'POS(0) .. Wide_Characters'POS(127)
```

The package Wide_Characters.Handling also has the new function Character_Set_Version thus

```
function Character_Set_Version return String;
```

The string returned identifies the version of the character set standard being used. Typically it will include either "10646:" or "Unicode". The reason for introducing this function is because the categorization of some wide characters depends upon the version of 10646 or Unicode being used. So rather than specifying that the package uses a particular set (which might be a nuisance in the future if the character set standard changes), it seemed more appropriate to enable the program to find out exactly which version is being used. For most programs, it won't matter at all of course.

Note that there is no corresponding function in Ada.Characters.Handling. This is because the set used for the type Character is frozen as at 1995 and the classification functions defined for the type Character are frozen as well. It might be that classifications for wide and ever wider characters might change in the future for some obscure characters but the programmer can rest assured that Character is for ever reliable.

So Wide_Characters.Handling in essence is

```
package Ada.Wide_Characters.Handling is
  pragma Pure(Handling);

  function Character_Set_Version return String;

  function Is_Control(Item: Wide_Character) return Boolean;

  ...          -- and so on

  function To_Upper(Item: Wide_String) return Wide_String);

end Ada.Wide_Characters.Handling.
```

The new package Wide_Wide_Characters.Handling is the same as Wide_Characters.Handling with Wide_Character and Wide_String replaced by Wide_Wide_Character and Wide_Wide_String throughout.

3 Directories

The package Ada.Directories was introduced in Ada 2005. However, experience with its use has revealed a number of shortcomings which are rectified in Ada 2012.

Three specific problems are mentioned in AI-49.

First, it is not possible to concatenate a root directory such as "/tmp" with a relative pathname such as "public/file.txt" using the procedure Compose thus

```
The_Path: String := Compose("/tmp", "public/file.txt");
```

This is because the second parameter of Compose has to be a simple name such as just "file" if there is no extension parameter. If we supply the extension parameter thus

```
The_Path: String := Compose("/tmp", "public/file", "txt");
```

then the second parameter has to be just a base name such as "public".

Another problem is that there is no sensible way to check for a root directory. Thus suppose the string S is a directory name and we want to see whether it is just a root such as "/" in Unix then the only thing that we can do is write

```
Containing_Directory(S)
```

which will raise Use_Error which is somewhat ugly.

We could write **if** S ="/" **then** but this would not be portable from Unix to other systems. Indeed, the whole purpose of providing file name operations in Ada.Directories is so that file names can be manipulated in an abstract manner without fiddling with text strings.

The third problem concerns case sensitivity. At the moment it is not possible to write portable programs because operating systems differ in their approach to this issue.

This last problem is solved by adding an enumeration type Name_Case_Kind and a function Name_Case_Equivalence to the file and directory name operations of the package Ada.Directories. So in outline we now have

```
with Ada.IO_Exceptions;  with Ada.Calendar;
package Ada.Directories is
   ...
-- File and directory name operations:
   function Full_Name(Name: String) return String;
   function Simple_Name(Name: String) return String;
   function Containing_Directory(Name: String) return String;
   function Extension(Name: String) return String;
   function Base_Name(Name: String) return String;
   function Compose(Containing_Directory: String := "";
                                    Name: String;
                                    Extension: String := "") return String;
   type Name_Case_Kind := (Unknown, Case_Sensitive, Case_Insensitive, Case_Preserving);
   function Name_Case_Equivalence(Name: String) return Name_Case_Kind;
   -- File and directory queries:
   -- and so on
end Ada.Directories;
```

The function Name_Case_Equivalence returns the file name equivalence rule for the directory containing Name. It raises Name_Error if Name is not a Full_Name.

It returns Case_Sensitive if file names that differ only in the case of letters are considered to be different. If file names that differ only in the case of letters are considered to be the same, then it returns Case_Preserving if the name has the case of the file name used when a file is created and Case_Insensitive otherwise. It returns Unknown if the name equivalence rule is not known.

We thus see that Unix and Linux are Case_Sensitive, Windows is Case_Preserving, and historic systems such as CP/M and early MS/DOS were Case_Insensitive.

The other problems are solved by the introduction of an optional child package for dealing with systems with hierarchical file names. Its specification is

```
package Ada.Directories.Hierarchical_File_Names is

   function Is_Simple_Name(Name: String) return Boolean;
   function Is_Root_Directory_Name(Name: String) return Boolean;
   function Is_Parent_Directory_Name(Name: String) return Boolean;
   function Is_Current_Directory_Name(Name: String) return Boolean;
   function Is_Full_Name(Name: String) return Boolean;
   function Is_Relative_Name(Name: String) return Boolean;

   function Simple_Name(Name: String) renames Ada.Directories.Simple_Name;
   function Containing_Directory(Name: String)
                                                  renames
Ada.Directories.Containing_Directory;

   function Initial_Directory(Name: String) return String;
   function Relative_Name(Name: String) return String;

   function Compose(Directory: String := "";
                                  Relative_Name: String;
                                  Extension: String := "") return String;

end Ada.Directories.Hierarchical_File_Names;
```

Note that the six functions, Full_Name, Simple_Name, Containing_Directory, Extension, Base_Name and Compose in the existing package Ada.Directories just manipulate strings representing file names and do not in any way interact with the actual external file system. The same applies to many of the new functions such as Is_Simple_Name.

In particular, Is_Root_Directory_Name returns true if the string is syntactically a root and so cannot be decomposed further. It therefore solves the second problem mentioned earlier. Thus

```
Is_Root_Directory_Name("/")
```

returns true for Unix. In the case of Windows "C:\" and "\\Computer\Share" are roots.

The function Is_Parent_Directory_Name returns true if and only if the Name is ".." for both Unix and Windows.

The function Is_Current_Directory_Name returns true if and only if Name is "." for both Unix and Windows.

The function Is_Full_Name returns true if the leftmost part of Name is a root whereas Is_Relative_Name returns true if Name allows identification of an external file but is not a full name. Note that relative names include simple names as a special case.

The functions Simple_Name and Containing_Directory are just renamings of those in the parent package and are provided for convenience.

Finally, the functions Initial_Directory, Relative_Name and Compose provide the ability to manipulate relative file names and so solve the problem with Compose mentioned at the beginning of this section.

Thus Initial_Directory returns the leftmost directory part of Name and Relative_Name returns the entire full name apart from the initial directory portion.

If we apply Relative_Name to a string that is just a single part of a name then Name_Error is raised. In particular this happens if Relative_Name is applied to a name which is a Simple Name, a Root Directory Name, a Parent Directory Name or a Current Directory Name.

The function Compose is much like Compose in the parent package except that it takes a relative name rather than a simple name. It therefore allows us to write

```
The_Path: String := Compose("/tmp", "public/file.txt");
```

as required.

The result of calling Compose is a full name if Is_Full_Name(Directory) is true and otherwise is a relative name.

4 Locale

When writing portable software it is often necessary to know the locality in which the software is to be run. Two key items are the country and the language (human language that is, not programming language).

To enable this to be done, Ada 2012 includes the following package

```
package Ada.Locales is
   pragma Preelaborate(Locales);
   pragma Remote_Types(Locales);

   type Language_Code is array (1 .. 3) of Character range 'a' .. 'z';

   type Country_Code is array (1 .. 2) of Character range 'A' .. 'Z';

   Language_Unknown: constant Language_Code := "und";
   Country_Unknown: constant Country_Code := "ZZ";

   function Language return Language_Code;
   function Country return Country-Code;

end Ada.Locales;
```

The various country codes and language codes are defined in ISO/IEC 3166-1:2006 and ISO/IEC 639-3:2007 respectively.

Knowledge of the locale is important for writing programs where the convention for certain information varies. Thus in giving a date we might want to add the name of the day of the week and clearly in order to do this we need to know what language to use. An earlier (really grotesque) attempt at providing this information introduced a host of packages addressing many issues. However, it was decided that for simplicity and indeed reliability all that is really needed is to know the language to use and the country.

Canada is interesting in that it has just one country code ("CA") but two language codes ("eng" and "fra"). In Quebec, a decimal value for a million dollars and one cent is written as $1.000.000,01 whereas in English language parts it is written as $1,000,000.01 with the comma and stop interchanged.

Sometimes, several locales might be available on a target. Some environments define a system locale and a locale for the current user. In the case of an Ada program the active locale is the one associated with the partition of the current task.

5 Hashing and comparison

New library functions are added for case insensitive comparisons and hashing. Thus we have

```
function Ada.Strings.Equal_Case_Insensitive(Left, Right: String) return Boolean;
pragma Pure(Ada.Strings.Equal_Case_Insensitive);
```

This simply compares the strings Left and Right for equality but ignoring case. Thus

```
Equal_Case_Insensitive("Pig", "PIG")
```

is true.

The function Ada.Strings.Fixed.Equal_Case_Insensitive is a renaming of the above. There are also similar functions Ada.Strings.Bounded.Equal_Case_Insensitive for bounded strings and Ada.Strings.Unbounded.Equal_Case_Insensitive for unbounded strings. And, as expected, there are similar functions for wide and wide wide versions.

Note that the comparison for strings can be phrased as convert to lower case and then compare. But this does not always work for wide and wide wide strings. The proper terminology is "locale-independent case folding and then compare".

Although it comes to the same thing for Latin-1 characters there are problems with some character sets where there is not a one-one correspondence between lower case and upper case. This used to apply to English with the two forms of lower case S and still applies to the corresponding letters in Greek where the upper case character is Σ and there are two lower case versions namely σ and ς. So

```
Ada.Wide_Strings.Equal_Case_Insensitive("ΣΟΣ", "σος")
```

returns true. Note that if we convert to lower case first then it would not be true.

Furthermore there is also

```
function Ada.Strings.Less_Case_Insensitive(Left, Right: String) return Boolean;
pragma Pure(Ada.Strings.Less_Case_Insensitive);
```

which does a lexicographic comparison.

As expected there are similar functions for fixed, bounded and unbounded strings and, naturally, for wide and wide wide versions.

Ada 2005 has functions for hashing such as

```
with Ada.Containers;
function Ada.Strings.Hash(Key: String) return Containers.Hash_Type;
```

Ada 2012 adds case insensitive versions as well such as

```
with Ada.Containers;
function Ada.Strings.Hash_Case_Insensitive(Key: String) return Containers.Hash_Type;
```

There are also fixed, bounded and unbounded versions and the inevitable wide and wide wide ones as well.

6 Miscellanea

The first item is that the package Stream_IO should be marked as preelaborated. So in Ada 2012 it now begins

```
with Ada.IO_Exceptions;
package Ada.Streams.Stream_IO is
   pragma Preelaborate(Stream_IO);
   ...
```

The reason for making this change concerns the use of input–output in preelaborated packages. The normal input–output packages such as Text_IO are not preelaborated and so cannot be used in packages that are themselves preelaborated. This makes preelaborated packages awkward to debug since they cannot do straightforward output for monitoring purposes. To make packages such as Text_IO preelaborated is essentially impossible because they involve local state. However, no such problem exists with Stream_IO, and so making it preelaborated means that it can be used to implement simple logging facilities in other preelaborated packages.

In principle, there is a similar problem with pure units. But they cannot change state anyway and so cannot do output since that changes the state of the environment. They just have to be written correctly in the first place.

(I have been told that there are naughty ways around this with pure packages but I will not contaminate innocent minds with the details.)

The package Ada.Environment_Variables was introduced in Ada 2005 as follows

```
package Ada.Environment_Variables is
   pragma Preelaborate(Environment_Variables);

   function Value(Name: String) return String;
   function Exists(Name: String) return Boolean;
   procedure Set(Name: in String; Value: in String);
   procedure Clear(Name: in String);
   procedure Clear;

   procedure Iterate(Process: not null access procedure (Name, Value: in String));

end Ada.Environment_Variables;
```

If we do not know whether an environment variable exists then we can check by calling Exists prior to accessing the current value. Thus a program might be running in an environment where we might expect an environment variable "Ada" whose value indicates the version of Ada currently supported.

So as in [2] we might write

```
if not Exists("Ada") then
   raise Horror;
end if;
Put("Current Ada is ");
Put_Line(Value("Ada"));
```

But this raises a possible race condition. After determining that Ada does exist some malevolent process (such as another Ada task or an external human agent) might execute Clear("Ada"); and then the call of Value("Ada") will raise Constraint_Error.

The other race condition might arise as well. Having decided that Ada does not exist and so taking remedial action some kindly process might have created Ada.

These problems are overcome in Ada 2012 by the introduction of an additional function Value with a default parameter

```
   function Value(Name: String; Default: String);
```

Calling this version of Value returns the value of the variable if it exists and otherwise returns the value of Default.

References

[1] ISO/IEC JTC1/SC22/WG9 N498 (2009) *Instructions to the Ada Rapporteur Group from SC22/WG9 for Preparation of Amendment 2 to ISO/IEC 8652.*

[2] John Barnes (2006) *Programming in Ada 2005*, Addison-Wesley.