Constants

29.1 OVERVIEW

Expressions, just studied, include the special case of constants, whose values cannot directly be changed by execution-time actions. This discussion goes through the various kinds. Particular attention will be devoted to the various forms, single- and multi-line, of *string* constant.

Along with constants proper, we will study two notations for "manifest" objects given by the list of their items: manifest tuples and manifest arrays, both using the syntax [*item*$_1$, … *item*$_n$].

29.2 GENERAL FORM OF CONSTANTS

A Constant expression has a value that does not change at run time, and is the same for all instances of a class.

A Constant is required as "inspect constant" in Multi_branch instructions (chapter 17).

The form is:

Constants

Constant ≜ Manifest_constant |
Constant_attribute

Constant_attribute ≜ Feature_name

A Constant_attribute denotes a constant value, specified in the attribute's declaration as a Manifest_constant. The use of an identifier as Constant_attribute is subject to an obvious constraint:

Constant attributes were discussed in chapter 18

Constant Attribute rule *VWCA*

A Constant_attribute appearing in a class *C* is valid if and only if its Feature_name is the final name of a constant attribute of *C*.

← The rules for recognizing constant attributes and other feature categories were given in 5.12, page 145.

To apply this rule, you must look at the declaration of the attribute and check that, according to the rules for distinguishing between various kinds of feature, it indeed defines a constant attribute.

If not a Constant_attribute, a Constant will be a Manifest_constant, whose form directly determine both a type and a value:

Manifest constants

Manifest_constant	≜	[Manifest_type] Manifest_value
Manifest_type	≜	"**{**" Type "**}**"
Manifest_value	≜	Boolean_constant | Character_constant | Integer_constant | Real_constant | Manifest_string | Manifest_type
Sign	≜	"+" | "–"
Integer_constant	≜	[Sign] Integer
Character_constant	≜	"'" Character "'"
Boolean_constant	≜	*True* | *False*
Real_constant	≜	[Sign] Real

→ *The syntax for* Manifest_string *appears later in this chapter, page 785.*

For clarity and consistency, and to avoid mistakes, we put a restriction — not expressible in BNF-E — on the use of signs:

← *A similar rule applied to operators: "Syntax (non-production): Alias Syntax rule", page 151. See also rules on characters*

Syntax (non-production): Sign Syntax rule

If present, the Sign of an Integer_constant or Real_constant must immediately precede the associated Integer or Real, with no intervening tokens or components (such as breaks or comments).

Similarly, for characters:

Syntax (non-production): Character Syntax rule

The quotes of a Character_constant must immediately precede and follow the Character, with no intervening tokens or components (such as breaks or comments).

In general, breaks or comment lines may appear between components prescribed by a BNF-E production, making the last two rules necessary to complement the grammar: for signed constants, you must write –5, not – 5 etc. This helps avoid confusion with operators in arithmetic expressions, which may of course be followed by spaces, as in *a* – *b*. Similarly, you must write a character constant as '*A*', not ' *A* '.

To avoid any confusion about the syntax of Character_constant, it is important to note that a character code such as %N (New Line) constitutes a single Character token.

→ *"Syntax (non-production): Manifest character", page 885.*

The following sections study the role of the optional Type in braces, then the various cases, except for Boolean_constant, about which it suffices to note that this construct only has two specimens, *True* and *False*, whose values are different (when compared for equality).

Beyond basic types, there is also a need for specifying constant values of complex types. This is addressed through **once functions**. The body of a once function is executed at most once, to compute the result of the first call (if any). All subsequent calls return the same result as the first, without further computation.

On once functions (and once routines in general) see 8.5, page 218, and the semantics of calls in chapter 23.

For functions of reference types, this yields constant *references*. The scheme is particularly useful for objects containing shared information and may be illustrated as follows:

```
shared: SOME_REFERENCE_TYPE
            -- A reference to an object shared by
            -- all instances of the enclosing class
      once
            create Result … (…);
            … Further operations on Result, if needed, to update
                the attached object …
      end
```

Calls to *shared* always return a reference to the object created and initialized by the first call. Only the reference is constant here, not the object itself since clients can change its fields through procedure calls

```
shared.some_procedure (…)
```

29.3 FORCING A TYPE ON A CONSTANT

The syntax for Manifest_constant on the preceding page specifies

Manifest types

Manifest_constant ≜ [Manifest_type] Manifest_value

A Manifest_value directly specifies a value, for example *3.141592* or "*ABC*". That value always determines a type: *REAL* and *STRING* in these two examples. Usually this is also the type that you want for the Manifest_constant as a whole and, if so, you don't need to qualify the Manifest_value further. For example if you use 1, a specimen of the lexical construct Integer, as a constant, the type rules imply that it will be understood as a Manifest_constant of type *INTEGER*. This is usually the desired result; but if you want the expression to be of a different type — a sized integer variant — you may specify a Manifest_type, as in

```
{INTEGER_8} 1
```

This syntax does not imply a conversion, but simply forces an explicit type. You will need it in some cases, for example, to specify large integer values. Depending on a global setting that you can override through a compilation option, *INTEGER* is a synonym for either *INTEGER_32* or *INTEGER_64*. Assume it is means *INTEGER_32*. The maximum value of that type is approximately 2^{31}. If you want to express the value 2^{32}, representable as an *INTEGER_64* but not as an *INTEGER_32*, you may not write 4_294_967_296 (even though that's the correct mathematical value) since it's invalid as an *INTEGER_32*, being beyond the bounds. You may, however, use

← See also "EXPRESSION CONVERTIBILITY: THE ROLE OF PRECONDITIONS", 15.10, page 412.

→ The optional underscores let you group digits for readability. See "INTEGERS", 32.16, page 889.

```
{INTEGER_64} 4_294_967_296
```

29.4 THE TYPE OF A CONSTANT

---- EXPLAIN

Type of a manifest constant

The type of a Manifest_constant of Manifest_value *mv* is:

1 • For {*T*} *mv*, with the optional Manifest_type present: *T*. The remaining cases assume this optional component is absent, and only involve *mv*.

2 • If *mv* is a Boolean_constant: *BOOLEAN*.

3 • If *mv* is a Character_constant: *CHARACTER*.

4 • If *mv* is an Integer_constant: *INTEGER*.

5 • If *mv* is a Real_constant: *REAL*.

6 • If *mv* is a Manifest_string: *STRING*.

7 • If *mv* is a Manifest_type {*T*}: *TYPE* [*T*].

As a consequence of cases 3 to 6, the type of a character, string or numeric constant is never one of the sized variants but always the fundamental underlying type (*CHARACTER*, *INTEGER*, *REAL*, *STRING*). Language mechanisms are designed so that you can use such constants without hassle — for example, without explicit conversions — even in connection with specific variants. For example:

- You can assign an integer constant such as 10 to a target of a type such as *INTEGER_8* as long as it fits (as enforced by validity rules).
- You can use such a constant for discrimination in a Multi_branch even if the expression being discriminated is of a specific sized variant; here too the compatibility is enforced statically by the validity rules.

Case 7 involves the Kernel Library class *TYPE*.

The Manifest_type notation is only applicable to manifest constants of types with sized variants:

Manifest-Type Qualifier rule *VWMQ*

It is valid for a Manifest_constant to be of the form {*T*} *v* (with the optional Manifest_type qualifier present) if and only if the type *U* of *v* (as determined by cases 2 to 7 of the definition of the type of a manifest constant) is one of *CHARACTER*, *STRING*, *INTEGER* and *REAL*, and *T* is one of the sized variants of *U*.

→ *“Basic types and their sized variants”, page 807*.

The rule states no restriction on the value, even though an example such as {*INTEGER_8*} 256 is clearly invalid, since 256 is not representable as an *INTEGER_8*. The Manifest Constant rule addresses this.

Do not confuse this Manifest_type notation {*T*} *const*, for a constant *const*, with the mechanism for expressing conversions explicitly: ----- REWRITE THIS FOR NEW NOTATION ----{*T*} [*exp*] which simply applies a function *adapted* from class *TYPE* to the target *exp*, an arbitrary expression, triggering any necessary conversion in the process. The {*T*} *const* notation only applies to constants and **will not cause a conversion**, as noted above in the example of {*INTEGER_8*} 1.

← *“CONVERTING AN EXPRESSION EXPLICITLY”, 15.9, page 408*..

The effect of adding a Manifest_type follows from the informal description:

Manifest Constant Semantics

The **value** of a Manifest_constant *c* listing a Manifest_value *v* is:

1 • If *c* is of the form {*T*} *v* (with the optional Manifest_type qualifier present): the value of type *T* denoted by *v*.

2 • Otherwise (*c* is just *v*): the value denoted by *v*.

--- In case 2, the lexical rules of the language ensure that the form of the constants uniquely determines one of the types listed. For example, *123* is an integer, but *12.3* is a real; *{T}* is a *TYPE*; and so on.

Sometimes we just need to refer to the value explicitly listed for a constant, ignoring any Manifest_type. The following definition captures this notion:

Manifest value of a constant

The **manifest value** of a constant is:

1 • If it is a Manifest_constant: its value.

2 • If it is a constant attribute: (recursively) the manifest value of the Manifest_constant listed in its declaration.

29.5 INTEGER CONSTANTS

An integer constant – a specimen of Integer_constant – consists of an Integer, possibly preceded by a sign. (Integer, a lexical construct, describes unsigned integers.) Example specimens of Integer_constant are:

Construct Integer is described as part of the lexical specification in 32.15, page 888.

```
0
253
-57
+253
```

29.6 REAL CONSTANTS

A real constant – a specimen of Real_constant – consists of a Real, possibly preceded by a sign.

Real, a lexical construct, describes floating-point numbers. Without a scaling factor, the possible forms of Real are

Real *is described in 32.16, page 889.*

```
a.
.b
a.b
```

where *a* and *b* are specimens of Integer. Any of these may be followed by the letter *E*, an optional sign and an Integer to indicate scaling by a power of ten.

Here are some example specimens of Real_constant:

```
46.
54.
24.36
-34.65
-34.65E-12
45.21E2
+45.21E2
```

29.7 CHARACTER CONSTANTS

A Character_constant is a character enclosed in single quotes, as in

```
'c'
```

The following constant attribute declarations define symbolic names for some specimens of Character_constant:

```
Upper_z: CHARACTER is 'Z';
Dollar_sign: CHARACTER is '$';
blank: CHARACTER is ' ';
```

A Character_constant consists of exactly three characters: the first and the third are single quotes '; the middle one is a Character other than a single quote.

Allowing a quote would not cause any ambiguity (since there are always exactly three characters altogether), but the rule is consistent with the convention for double quotes in a Manifest_string, as studied in the next section.

The value of a Character_constant is its middle Character.

To understand the above syntactic definition, you must realize that a Character is either a key corresponding directly to a printable character (such as *A* or *$*) or one of a set of multiple-key special character codes beginning with the percent sign *percent*. Examples of such codes are:

→ *"Special characters and their codes", page 887*.

- %*N* for a new-line.
- %*'* for a single quote.
- %*B* for a backspace.
- %/ "*91*" /% for the character of ASCII code 91.

91 is the (decimal) code of the opening bracket [, which you may also write as %(.

For example, a class text may include constant attribute declarations such as

```
New_line: CHARACTER is ’%N’;
Single_quote: CHARACTER is ’%’’
```

Because a new-line is not a Character, the three characters of a Character_constant must appear on the same line. Of course, you may define a constant whose value is a new-line character by using %*N* as middle Character.

In spite of appearances, the presence of %*’* in a Character_constant, as in the declaration of *Single_quote* above, does not violate the prohibition of the quote character as a constant’s Character: %*’*, as all the special character codes, is considered to be a single character, although it consists of two signs (percent and single quote). This is explained in detail in the specification of characters.

29.8 MANIFEST STRINGS

A Manifest_string denotes an instance of the Kernel Library class *STRING*, studied in a later chapter.

→ See 36.7, page 927, about class STRING.

As the following syntax indicates, there are two ways to write a manifest string:

- A Basic_manifest_string, the most common case, is a sequence of characters in double quotes, as in "*This text*". Some of the characters may be special character codes, such as %N representing a new line. This variant is useful for such frequent applications as object names, texts of simple messages to be displayed, labels of buttons and other user interface elements, generally using fairly short and simple sequences of characters. You may write the string over several lines by ending an interrupted line with a percent character % and starting the next one, after possible blanks and tabs, by the same character.
- A Verbatim_string is a sequence of lines to be taken exactly as they are (hence the name), bracketed by "{ at the end of the line that precedes the sequence and }" at the beginning of the line (or "[and "] to left-align the lines). No special character codes apply. This is useful for embedding multi-line texts; applications include *description* entries of Notes clauses, inline C code, SQL or XML queries to be passed to some external program.

An example of the first of these uses is:

```
note
    description: "[
            Constants covering kinds of user interface events.
                This class is meant to be used as ancestor
                by classes needing its facilities.
            ]"
```

Here are the syntax rules for these two variants:

Manifest strings

```
       Manifest_string ≜ Basic_manifest_string |
                         Verbatim_string

 Basic_manifest_string ≜ ' " ' String_content ' " '

        String_content ≜ {Simple_string Line_wrapping_part ...}+

       Verbatim_string ≜ Verbatim_string_opener
                         Line_sequence
                         Verbatim_string_closer

Verbatim_string_opener ≜ ' " ' [Simple_string] Open_bracket

Verbatim_string_closer ≜ Close_bracket [Simple_string] ' " '

          Open_bracket ≜ "[" | "{"

         Close_bracket ≜ "]" | "}"
```

In the “basic” case, most examples of String_content involve just one Simple_string (a sequence of printable characters, with no new lines). For generality, however, String_content is defined as a repetition, with successive Simple_string components separated by Line_wrapping_part to allow writing a string on several lines. Details below.

→ *The “extreme case” arises when one of the lines in the* Line_sequence *begins with* **]"**. *See “Verbatim strings”, page 788.*

In the “verbatim” case, Line_sequence is a lexical construct denoting a sequence of lines with arbitrary text. The reason for the Verbatim_string_opener and the Verbatim_string_closer is to provide an escape sequence for an extreme case (a Line_sequence that begins with **]"**), but most of the time the opener is just **"[** or **"{** and the closer **]"** or **"}**. The difference between brackets and braces is that with **"{ ... }"** the Line_sequence is kept exactly as is, whereas with **"[...]"** the lines are left-aligned (stripped of any common initial blanks and tabs). Details below.

← *Similar rules applied to operators, in “Syntax (non-production): Alias Syntax rule”, page 151, and to signed constants, in “Syntax (non-production): Sign Syntax rule”, page 778.*

As with some other constructs, we need to clarify the use of breaks through the definition of Line_sequence:

Syntax (non-production): Line sequence

A specimen of Line_sequence is a sequence of one or more Simple_string components, each separated from the next by a single New_line.

and a consistency constraint on manifest strings:

Syntax (non-production): Manifest String rule

In addition to the properties specified by the grammar, every Manifest_string must satisfy the following properties:

1 • The Simple_string components of its Line_sequence may not include a double quote character except as part of the character code **%"** (denoting a double quote).

2 • A Verbatim_string_opener or Verbatim_string_closer may not contain any break character.

Like other "non-production" syntax rules, the last two rules capture simple syntax requirements not expressible through BNF-E productions.

Because a Line_sequence is made of simple strings separated by a single New_line in each case, a line in a Verbatim_string that looks like a comment is not a comment but a substring of the Verbatim_string.

The details of both variants now follow, after an explanation of the **once** keyword.

Basic manifest strings

To denote the actual string content, you have the choice between Basic_manifest_string and Verbatim_string.

An example Basic_manifest_string is:

```
"This Manifest_string contains 43 characters"
```

The value is a *STRING* object, which represents a sequence of characters. In this example the sequence contains all the characters given except for the two enclosing double quotes, which play a purely syntactical role.

← *"CHARACTER CONSTANTS", 29.7, page 783.*

Any of the characters may be a character code as discussed earlier for Character_constant. This enables you to include a double quote character into the string: use its code, %". Similarly

```
"First line%NNew line"
```

describes a string with two lines, separated by a newline character represented as %N.

Whether the **value** of a Basic_manifest_string is text extending over just one line or several, you have the freedom to write the **specification** of the string in the Eiffel text over one or more lines. In particular you may, for readability, write a long Basic_manifest_string over several lines in the source text. For this you will use one or more **line wrapping parts**, interrupting the string with a percent sign % at the end of a line, and resuming it with a percent sign on the next line.

An example using two line wrapping parts (shown by the shaded zones) is the Manifest_string

First wrapping part: shaded on first two lines. Second part: shaded on last two lines.

```
"This Manifest_string con%
                       %tains 43 %
                       %characters"
```

There is a space at the end of the second line, after '43'.

The sole purpose of the line wrapping form is to enable you to write a Manifest_string on several lines, but without retaining the line separations in the string that it denotes. So the contents of the last example do not tell a lie: the resulting string is indeed exactly the same as in with the first example of this section; it is simply written on three lines rather than one. Note that the initial blanks or tabs on the second and third lines, before the percent sign, are there only for layout and do not contribute to the string.

More generally, the division of the string into two or more lines in the source text has no effect on the value of the string. In the above example, the value is a one-line string (which we may also write as just "*This Manifest_string contains 43 characters*"). By adding %N characters we would turn this into a multi-line string values. In contrast, the Verbatim_strings to be seen next will retain the line formating of the string as it appears in the Eiffel text.

Here are the definition and rules formalizing the preceding properties. The syntax of Basic_Manifest_string involves one or more Line_wrapping_part, with the following definition:

Line_wrapping_part

A Line_wrapping_part is a sequence of characters consisting of the following, in order: **%** (percent character); zero or more blanks or tabs; New_line; zero or more blanks or tabs; **%** again.

This construct requires such a definition since it can't be specified through a context-free syntax formalism such as BNF-E.

The use of Line_wrapping_part as separator between a Simple_string and the next in a Basic_manifest_string allows you to split a string across lines, with a % at the end of an interrupting line and another one at the beginning of the resuming line. The definition allows blanks and tabs before the final **%** of a Line_wrapping_part although they will not contribute to the contents of the string. This makes it possible to apply to the Basic_manifest_string the same indentation as to the neighboring elements. The definition also permits blanks and tabs after the initial % of a Line_wrapping_part, partly for symmetry and partly because it doesn't seem justified to raise an error just because the compiler has detected such invisible but probably harmless characters.

As to the semantics:

Manifest string semantics

The value of a Basic_manifest_string is the sequence of characters that it includes, in the order given, excluding any line wrapping parts, and with any character code replaced by the corresponding character.

→ *"Special characters and their codes", page 887*.

Verbatim strings

A Verbatim_string is a sequence of lines meant to be retained exactly as they are (except for possible left-alignment).

→ *A break character is a space or a tab. See "BREAKS", 32.5, page 871*.

You may bracket a Verbatim_string between a line ending with **"[**, possibly followed by break characters, and a line beginning with **]"**, possibly preceded by break characters.

The earlier example showed a *description* entry in a Notes clause:

```
note
    description: "[
                Constants covering kinds of user interface events.
                    This class is meant to be used as ancestor
                    by classes needing its facilities.
                ]"
```

The reason for allowing break characters after the initial **"[** and before the final **]"** is, as above, to avoid raising an error in harmless cases.

As the examples below illustrate, there may be other Eiffel elements preceding the initial **"[** on the same line, and following the final **]"** on the same line.

The beginning and ending delimiters of a Verbatim_string will usually be **"[** and **]"** as above, but the syntax gave a more general convention:

← *This syntax first appeared on page 785*.

Verbatim strings

```
      Verbatim_string ≜ Verbatim_string_opener
                        Line_sequence
                        Verbatim_string_closer

Verbatim_string_opener≜ '"' [Simple_string] Open_bracket
Verbatim_string_closer≜ Close_bracket [Simple_string] ' " '
         Open_bracket ≜ "[" | "{"
        Close_bracket ≜ "]" | "}"
```

In this general form the string is bracketed by "α{ and }α", or "α[and]α", where α — the Simple_string of the Verbatim_string_opener and Verbatim_string_closer — is any sequence of characters not including a double quote ". In most cases, including all the examples in this section, α is an empty string, but a non-empty α is useful in the case — unavoidable if we want to have a completely general mechanism — in which one of the lines of the string begins with the closing delimiter]" (or }" if the opening delimiter was "{).To handle such a string as a Verbatim_string, choose a string α such that no line of the text begins with α]" (or α}"), possibly preceded by breaks.

This convention is an application of the general language design rule that any convention for quoting text, using specific characters or character sequences as delimiters, must have an **escape** convention making it possible to quote a text that includes the delimiters themselves.

Here is an example of this convention:

```
note
    description: "++[
            This class, from a hypothetical Eiffel parser, is in
            charge of parsing Verbatim strings that end with
            ]" or some variant thereof.
            ]++"
```

Because of the highlighted line beginning, we can't use **"[** and **]"** as delimiters; instead we choose **"++[** and **++]"**, making sure that no line in the string begins with **]++"**. This is of course an extreme case, and most uses of Verbatim_string will rely on the default delimiters.

This observation leads to the constraint on verbatim strings::

Verbatim String rule *VWVS*

A Verbatim_string is valid if and only if it satisfies the following conditions, where α is the (possibly empty) Simple_string appearing in its Verbatim_string_opener:

1 • The Close_bracket is **]** if the Open_bracket is **[**, and **}** if the Open_bracket is **{**.

2 • Every character in α is printable, and not a double quote **"**.

3 • If α is not empty, the string's Verbatim_string_closer includes a Simple_string identical to α.

Truly identical: in strings, letter case is significant.

Next, the semantics. Whatever the delimiters, the value of a Verbatim_string is given by its Line_sequence (the sequence of lines that make it up, delimiters excluded), taken exactly as it is, except for left-alignment if the delimiters used brackets **[ ]** rather than braces. The characters in the Line_sequence are retained exactly as they are. No special character codes apply: if %N (for example) appears in the Line_sequence, it will be understood as two characters, a percent and an *N*.

This rule assumes a view of strings as plain sequences of characters, line separations being marked by a special "new line" character.

Here is the semantic rule that states these properties:

Verbatim string semantics

The value of a Line_sequence is the string obtained by concatenating the characters of its successive lines, with a "new line" character inserted between any adjacent ones.

The value of a Verbatim_string using braces **{ }** as Open_bracket and Close_bracket is the value of its Line_sequence.

The value of a Verbatim_string using braces **[ ]** as Open_bracket and Close_bracket is the value of the left-aligned form of its Line_sequence.

This semantic definition is **platform-independent**: even if an environment has its own way of separating lines (such as two characters, carriage return %R and new line %N, on Windows) or represents each line as a separate element in a sequence (as in older operating systems still used on mainframes), the semantics yields a single string — a single character sequence — where each successive group of characters, each representing a line of the original, is separated from the next one by a single %N.

(“**Left-aligned form**” should be intuitively clear, but is defined rigorously below.) The difference between the two kinds of brackets follows from whether you want to left-align the string. If you don’t, use braces, as in

“Prefix, longest break prefix, left-aligned form”, page 794.

```
note
	description: "{
		Constants covering kinds of user interface events.
			This class is meant to be used as ancestor
			by classes needing its facilities.
			}"
```

The shaded rectangles show the positions of tab characters.

where the various tab positions have been highlighted; the value of the string retains all these tabs:

```
		Constants covering kinds of user interface events.
			This class is meant to be used as ancestor
			by classes needing its facilities.
```

This is the value of the Verbatim_string *of the above example.*

Originally, however, we had written this example using square brackets **"[** … **]"** to request left alignment:

```
note
	description: "[
		Constants covering kinds of user interface events.
			This class is meant to be used as ancestor
			by classes needing its facilities.
			]"
```

This means that the string’s value doesn’t include the two initial tabs (highlighted above as lightly shaded rectangles) common to all three lines. It does, however, include the extra tab (dark rectangle) that appears only on the last two lines. So that value is

```
Constants covering kinds of user interface events.
	This class is meant to be used as ancestor
	by classes needing its facilities.
```

Why go into all this trouble by having two kinds of delimiters, one implying left alignment and the other not? The reason is cosmetic. In many case, you want the left-aligned version of a verbatim string. This example, involving *description* entries of Notes clauses, is typical. Language processing tools may store the values into a reuse repository to facilitate content-based retrieval of classes; in ISE Eiffel, for example, all such entries are turned into **META** tags of the generated HTML documentation, so that you can retrieve them through Web search engines. But without automatic left-alignment you would have to start your class text as

WARNING: *ugly layout, not recommended. See text.*

```
note
        description: "{
Constants covering kinds of user interface events.
        This class is meant to be used as ancestor
        by classes needing its facilities.
                }"
        author: "Jane Programmer"
class YOUR_CLASS …
```

which messes up the indentation and layout. Using the brackets **"[** … **]"** instead of the braces **"{** … **}"** solves the problem: you indent the text in a way that looks nice in your software text

WARNING: *ugly layout, not recommended. See text.*

```
note
        description: "[
                Constants covering kinds of user interface events.
                        This class is meant to be used as ancestor
                        by classes needing its facilities.
                        ]"
        author: "Jane Programmer"
class YOUR_CLASS …
```

The same observation applies if you use a Verbatim_string for inline C code in an external function, as in this extract from an example in the discussion of interfacing Eiffel with C, involving two verbatim strings:

→ *"Specifying C code inline", page 834.*

```
        an_inline_function (x,y: INTEGER): INTEGER
                external "[
                        C
                                inline
                                use <stdio.h>, /path/user/her_include.h
                        ]"
                alias "[
                                if ($x > cvar) {
                                        some_c_function ($y, cvar++);
                                }
                        ]"
                end
```

Warning: the content of the **alias** *clause represents C, not Eiffel. The indentation is shown as it will appear in the class text (where a feature declaration is already indented one step, as part of a* Feature_clause.

or in a hypothetical example using an embedded SQL string:

```
        example
                do
                        sql_statement := database.prepare ("[
                                SELECT emp
                                FROM EMPLOYEE
                                WHERE salary >= 50000
                                        ]")
        end
```

The indentation in these last two examples should not be retained in the strings actually passed to the appropriate tools — a C compiler, an SQL query processor for a Database Management System. (The C compiler probably don't care, but other tools might!), The indentation is only beneficial to the reader Eiffel text, who might cringe when seeing a version left-aligned for the sake of the external tool, such as

```
example
        do
                sql_statement := database.prepare ("{
SELECT emp
FROM EMPLOYEE
WHERE salary >= 50000
                }")
        end
```

WARNING: ugly layout, not recommended. See text.

which breaks the layout of Eiffel texts. Using the bracket form **"[…]"** solves this problem by allowing you to keep a pleasant layout without retaining the extra initial tabs or spaces in the string's semantics.

→ *"LAYOUT", 34.9, page 905.*

About "tabs or spaces": the recommended practice is always to use tabs for indentation. Some older text editors, however, prefer spaces.

The following auxiliary definitions give a precise meaning to the notion of "left-aligned form" used above to specify the semantics of a verbatim strings using brackets as delimiters:

← "Verbatim String rule", page 790.

Prefix, longest break prefix, left-aligned form

A **prefix** of a string *s* is a string *p* of some length *n* ($n \geq 0$) such that the first *n* characters of *s* are the corresponding characters of *p*.

The **longest break prefix** of a sequence of strings *ls* is the longest string *bp* containing no characters other than spaces and tabs, such that *bp* is a prefix of every string in *ls*. (The longest break prefix is always defined, although it may be an empty string.)

The **left-aligned form** of a sequence of strings *ls* is the sequence of strings obtained from the corresponding strings in *ls* by removing the first *n* characters, where *n* is the length of the longest break prefix of *ls* ($n \geq 0$).

Obviously, if you are passing a verbatim string to an external tool that attaches a semantic value to initial tabs and spaces on a line, and want to control the string character-by-character without any intervention from the left-aligning process implied by these definitions, you should use the brace form **"{ … }"** even if this means uglifying the Eiffel text layout. (Yes, semantic correctness should in the end win over esthetic appeal.) But in that case you are probably better off anyway using the non-verbatim form of manifest strings, control characters and all.

Choosing between basic and verbatim manifest strings

You can indeed write any Verbatim_string as a Basic_manifest_string; for example to get the same result as our earlier Verbatim_string, you may use:

← This refers to the non-left-aligned version, using braces, on page 791.

```
note
    description: "%TConstants covering kinds of user %
        %interface events.%N%
            %%T%TThis class is meant to be used%
            % as ancestor%N%
            %%T%Tby classes needing its facilities"
```

Note the need to include explicit codes *%N* (new line) and *%T* (tab) and, if the text is too long for pleasant formating, break it into several lines with "line wrapping parts". The two divisions do not necessarily coincide: in this example the string has been broken after *of user* and *used*, but there is no new line at those positions in the string's value.

Since the words being sent to separate lines (*user* and *interface*, *used* and *as*) must be separated by a space, you mustn't forget to include that space — highlighted above — at the end of the interrupted line or the beginning of the continuation line. It's this need to code everything explicitly that makes Basic_manifest_string tedious to use for describing multi-line strings; Verbatim_string avoids the problem.

"Once" string expressions

The syntax for expressions in the preceding chapter included the Once_string expression, of the form **once** *some_manifest_string*, for example **once** "*This text*". This is actually an expression, not a constant; you can use it in contexts that expect an expression, for example an assignment or argument passing

```
your_string_entity := once "This text"
your_procedure (once "This text")
```

but not where only a Manifest_string will do, for example a manifest declaration *your_constant* **is** "*This text*", a Debug key, an Obsolete message. But since this notion is closely connected to the semantics of strings it is appropriate to study it here.

The possibility of qualifying a manifest string constant by **once** when you use in an expressions corresponds to different semantics for the manifest string:

- Without **once**, every evaluation of the string creates a new object.
- With **once**, only the first evaluation creates an object; every subsequent one yields a reference to that initial object.

This is of course the same difference as between a function declared with **do** and one declared with **once**. In fact you may understand a manifest string "*This text*" as a call to a function of the form\

← See "ROUTINE BODY", 8.5, page 218, about once routines.

```
new_string: STRING
      do create Result.make_from_string ("This text") end
```

whereas the manifest string **once** "*This text*" is equivalent to a call to

```
once_string: STRING
      once create Result.make_from_string ("This text") end
```

Which one of the two forms should you use? Each has advantages and drawbacks depending on the circumstances. If you find yourself using a text message in a loop, as in

```
from… loop … until
        …
        print (once "Message text")
end
```

you should probably use the **once** form, to avoid creating lots of identical objects.

It is almost always better in this case to declare a constant attribute:

```
Your_message: STRING is "Message text"
                -- Note that once "Message text" is valid here,
                -- but would make no difference.
…
from… loop … until
        …
        print (Your_message)
end
```

which avoids the problem altogether and is in line with general methodology principles (don't use literal constants in algorithms!). The **once** form is available, however, for designers who feel they truly need manifest strings with no symbolic names.

Because they are shared, however, once manifest strings can cause some surprises. Consider the following extract, with *text* of type *STRING*:

```
text := ""
text.append (i.out)
                        -- i.out is the string representation of the integer i
```

Assume that this appears in the body of a loop, and that each iteration of the loop increments *i* by one. You probably expect that each loop iteration will reinitialize *text* to an empty string, then extend it with the string value of *i*, setting it to "*1*"the first time, "*2*" the second time and so on. This is indeed what happens with the extract as given since the empty string is recreated each time. But if for the first instruction you use

```
text := once ""
```

then only one string is ever created, so that in loop iterations the string will take successive values "*1*", "*12*", "*123*"and so on. This is seldom the desired result and explains why the **once** behavior is not the default. (If novices are going to have bad surprises, better be it because of bad initial performance — which can be fixed through a **once** declaration — than because of wild and seemingly buggy behavior.)

The general rule, then, is:

- Use **once** for manifest strings that will not change at all — although it is even better in most cases to give them symbolic names such as *Your_message* above, and stop worrying.
- Use non-once strings as soon as anything can change, keeping in mind then that each use of the constant will create a new object.

Run-time model for manifest strings

As explained above, a Manifest_string *m* defines an associated Simple_string *s*. For example, in the Manifest_string appearing in the declaration

```
Message: STRING is "Example 1"
```

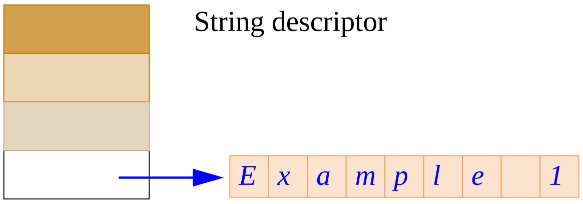
m is "*Example 1*" and *s* is the String made of the nine characters *Example 1*.

For most practical purposes you may view *s* as the value of *m*. In a more precise specification of the semantics, however, this is not quite correct. Although the nuance is somewhat fine, you should understand it even if this is your first reading, because of a potential confusion that has been known to surprise newcomers.

The problem is that the value of *m*, like any other value, must be an object or a reference to an object, and that a String (a sequence of characters) is not appropriate for this purpose. The desired object should be an instance of some class, so that you can apply features to it: for example a routine to which you pass *Message* as actual argument may need to access properties of *Message* such as its length, through the features of some appropriate class. But there is no class whose instances are just arbitrary sequences of characters.

There does exist a class meant for representing character strings: the Kernel Library class *STRING*, which indeed served as type for *Message* in the above example declaration. But an instance of *STRING* is not a sequence of characters: it is a string descriptor, which must of course provide access to the characters but may also include other information such as the string length. Most importantly, because a string descriptor is an instance of a class, all the features of that class are applicable to it. Class *STRING* offers many routines for operations on strings such as accessing the character at a given position, extracting a substring and appending another string.

The details of class STRING, its representation and its features are given in 36.7, page 927.



The details of string representation do not matter for this discussion, although the above figure shows a possible implementation, where the string descriptor includes, among various possible fields, a reference to the actual character sequence.

The character sequence is a "special object", as introduced in 19.2, page 498. See the discussion of strings in 36.7.

What does matter is that the value of an entity declared of *STRING* type, such as *Message* above, is a string descriptor, not a character sequence.

What difference does it make? Only one consequence is of practical concern: a Manifest_string is not a "constant" in the sense that most people would expect. It will always refer to the same string **descriptor**, but not necessarily to the same *character sequence*; this is because the contents of the descriptor may be changed through procedure calls.

Here is an example showing how this can happen:

```
Message: STRING is "Example 1"
…
Message.put ('2', 9)
```

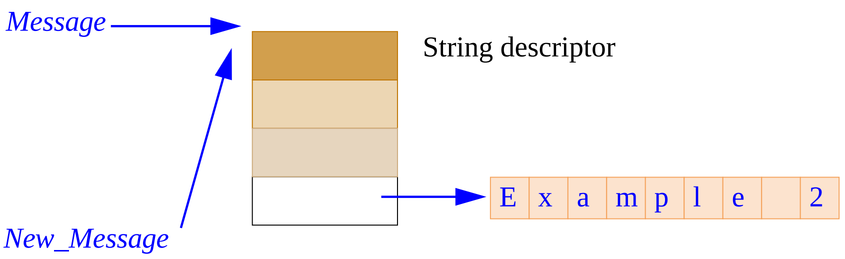
where the call to *put*, a procedure of class *STRING*, replaces the character at position 9 (originally *1*) by the character *2*. As a result, if a later instruction prints *Message*, the output will be

```
Example 2
```

The same potential problem may arise through a feature call applying to another entity than the original "constant" if there has been an assignment; for example, the assignment

```
New_message := Message;
```

produces a situation where both entities refer to the same character sequence:



The characters are assumed to be Example 2 *as a result of the call to* put *in the previous assignment..*

Then any operation on *Message* will have the same effect as the corresponding operation on *New_message*. For example, after

```
Message.put ('3' , 9)
```

an instruction to print *New_message* will produce *Example 3*.

29.9 MANIFEST TUPLES

The next form of Expression is the Manifest_tuple, defining a tuple through a list of expressions representing the tuple's successive items.

An example of Manifest_tuple, of type *TUPLE* [*INTEGER*, *REAL*, *INTEGER*], is

```
[27, 3.5, m+ n]
```

for *m* and *n* of type *INTEGER*. The value of this expression is a tuple of three elements, having the values given.

If the list of expressions in a Manifest_tuple is empty, it describes an array with no elements:

```
[ ]
```

Among other applications, you may use a Manifest_tuple to obtain the effect of a variable number of arguments for a routine: if one of the formal arguments of a routine is declared with the type *ARRAY* [*T*] for some *T*, then an actual argument may be an expression list

See 8.4, page 217.

```
[e1, e2, … en]
```

such that every *ei* is of a type conforming to *T*. The number *n* of elements in the list is arbitrary, so that you indeed obtain the same effect as if routines were permitted to have a variable number of arguments. Examples of this technique appeared in the discussion of routines.

The *ei* do not need, of course, to be all of the same type, as long as their types all conform to *T*. By choosing a more specific or more general *T* (based on a class lower or higher in the inheritance graph), you restrict or extend the set of acceptable types for the *ei*.

Here again is the syntax of manifest tuples:

← From the tuple chapter, page 365.

Manifest tuples

Manifest_tuple ≜ "**[**" Expression_list "**]**"

Expression_list ≜ {Expression "," …}

There is no constraint on manifest tuples.[TO BE COMPLETED -- FOLLOWING RULE REMOVED]: This is the rule that may be used in practice to ascertain whether a Manifest_tuple is appropriate as actual argument to a routine, or whether an assignment of the form

```
a := [e1, e2, .. en]
```

is valid. For example, with the routine specification

```
average_age (group: ARRAY [PERSON]): INTEGER
```

then the call in

```
group_average := average_age
          ([Fiordiligi, Dorabella, Guglielmo, Ferrando, Alfonso])
```

is valid if *Fiordiligi* and *Dorabella* are of type *LADY*, *Guglielmo* and *Ferrando* of type *GENTLEMAN*, Alfonso of type *PHILOSOPHER*, and all these classes are descendants of *PERSON*. If you add to the Manifest_tuple an expression whose type does not conform to *PERSON*, the Manifest_tuple ceases to be a valid expression of type *ARRAY* .[*PERSON*].

As another example of applying the Manifest Array rule, this time recursively, here is a valid expression of type *ARRAY* [*ARRAY* [*INTEGER*]]:

```
<< <<-3, 41>>, <<0>>, <<45, 31, -27>>, << >> >>
```

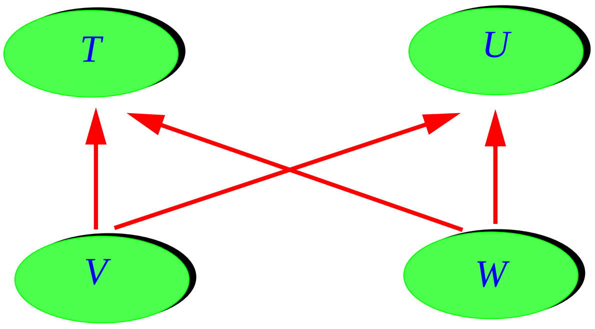
describing an array whose elements are integer arrays of two, one, three and zero elements successively.

As you may have noted, the Manifest Array rule departs slightly from the style using elsewhere in this book to talk about types, since it does not define "the type of" a Manifest_tuple, but instead tells us how to ascertain whether a Manifest_tuple is "a valid expression of type" *ARRAY* [*T*] for given *T*. This is because, in contrast with the other expressions studied in this chapter (and any other Eiffel component that has a value), a Manifest_tuple does not have a single type of the form *ARRAY* [*T*] for a single *T*. You may see this by considering the Manifest_tuple in

```
v1: V; w1: W
…
some_routine (<<v1, w1>>)
```

where *V* and *W* are non-generic classes with the inheritance structure illustrated on the adjacent page.

Here the given Manifest_tuple is a valid argument for *some_routine* if the formal argument is declared either as *ARRAY* [*T*] or as *ARRAY* [*U*] (as well as *ARRAY* [*X*] for any *X* to which *T*, *U* or both conform). If we tried to define "the type of the Manifest_tuple, however, *ARRAY* [*T*] and *ARRAY* [*U*] would be equally good candidates.



Fortunately, this inability to settle for a single type will not cause any difficulty in the two situations which require obtaining type information about a Manifest_tuple *ma*:

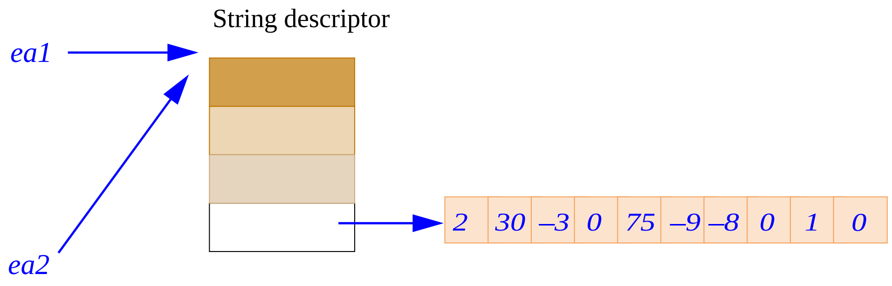
The Manifest Array rule enables us to ascertain conformance of *ma* to a certain type (the type of the target in an assignment, or of a routine's formal argument) without any ambiguity, as illustrated by the above examples.

The dynamic type set of *ma*, needed to ascertain system-level call validity, is the set of all types of the form *ARRAY* .[T] for every type *T* in the dynamic type set of any of the elements of *ma*. This is in line with the handling of genericity in the definition of the dynamic type set.

Arrays and class ARRAY are discussed in 36.4, page 924.

The value of a Manifest_tuple made of *N* expressions is an array of bounds 1 and *N*, whose elements are the values of the successive expressions in the Manifest_tuple. In this definition, an "array" is an instance of the Kernel Library class *ARRAY*.

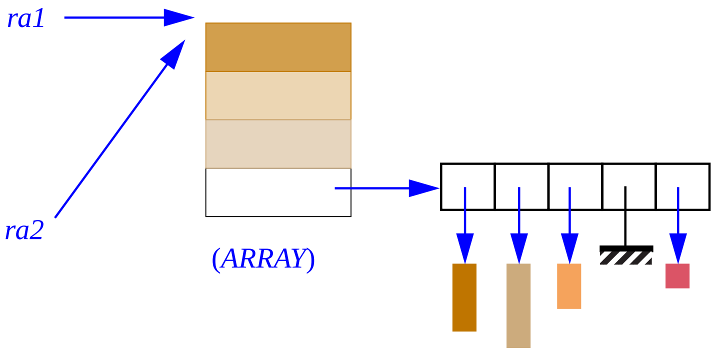
The situation here is similar to what we encountered above for strings: an instance of *ARRAY* is in fact an array descriptor, which must of course provide access to the actual elements, but may also include other information such as the number of elements and the bounds. This also means that an array descriptor is a normal object – an instance of class *ARRAY*, with all the features of this class applicable to it.



The last figure illustrates the notion of array descriptor; it is a conceptual representation of the situation resulting from

```
ea1, ea2: ARRAY {INTEGER];
…
ea1 := <<27, 54, -3, 7, 1, 0, 10, 546, -40>>
ea2 := ea1
```

In this case the array's elements are objects (instances of the expanded type *INTEGER*). Below is an illustration of the effect of similar operations on arrays *ra1* and *ra2* of type *ARRAY* .[*T*] for some reference type *T*. =



This could be the result of:

```
ra1, ra2: ARRAY [T];
a, b, c, d, T;
…
create {T} a …
create {U} b …
create {V} c …
create {W} d …
ra1 := <<a, b, c, d>>;
ra2 := ra1
```

U, V, W are types conforming to T.

29.10 SEMANTICS OF CONSTANT ATTRIBUTES

--- To be completed ---