This draft: Version 4.82-00-00, 21 September 02 12:15 (Zürich time)

Attributes

18.1 OVERVIEW

Attributes are one of the two kinds of feature.

← The other is routines, studied in chapter 8.

When, in the declaration of a class, you introduce an attribute of a certain type, you specify that, for every instance of the class that may exist at execution time, there will be an associated value of that type.

Attributes are of two kinds: **variable** and **constant**. The difference affects what may happen at run time to the attribute's values in instances of the class: for a variable attribute, the class may include routines that, applied to a particular instance, will change the value; for a constant attribute, the value is the same for every instance, and cannot be changed at run time.

This chapter discusses the properties of both two kinds of attribute.

18.2 GRAPHICAL REPRESENTATION

In graphical system representations, you may mark a feature that you know is a variable attribute by putting its name in a box.

Representing attributes

The figure illustrates this convention for attributes *first* and *first_element* in a class *LINKED_LIST* similar to the one from EiffelBase. (This is a partial representation of the class.)

As illustrated by this example, putting the attributes of a class next to each other, each boxed in a rectangle, yields a bigger rectangle that suggests the form of an **instance** of the class with all its fields. So we get a picture of both the class (elliptic) and the corresponding *objects* (rectangular).

Not boxing a feature does not mean that it is not a attribute. In some cases, you may choose to leave unspecified whether a particular feature is an attribute or a routine. Then the standard representation for features, unboxed, is appropriate. In the example illustrated above, *previous* and *next* may be attributes just as well as functions without arguments.

→ *Principle of uniform access*: 23.5, page 502.

18.3 VARIABLE ATTRIBUTES

Declaring a variable attribute in a class prescribes that every instance of the class should contain a field of the corresponding type. Routines of the class can then execute assignment instructions to set this field to specific values.

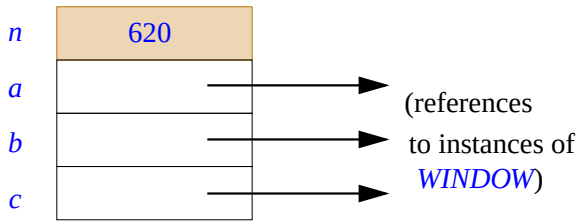
Here are some variable attribute declarations:

```
n: INTEGER
a, b, c: WINDOW
```

The first introduces a single attribute *n* of type *INTEGER*. The second (equivalent, because of the Multiple Declaration semantics rule, to three separate declarations) introduces three attributes, all of type *WINDOW*.

← "Multiple Declaration semantics", page 93.

If these declarations appear in the Features clause of a class *C*, all instances of *C* will have associated values of the corresponding types; an instance will look like this:



An instance with its fields

More generally, as you may remember, a feature declaration is a variable attribute declaration if it satisfies the following conditions:

← "HOW TO RECOGNIZE FEATURES", 5.11, page 82.

- There is no Formal_arguments part.
- There is a Type_mark part.
- There is no Constant_or_routine part.

18.4 CONSTANT ATTRIBUTES

Declaring a constant attribute in a class associates a certain value with every instance of the class. Because the value is the same for all instances, it does not need to be actually stored in each instance.

The construct Constant_attribute is introduced in 27.2, page 613, as part of the discussion of expressions.

Since you must specify the value in the attribute's declaration, the type of a constant attribute must be one for which the language offers a lexical mechanism to denote values explicitly. This means one of the following:

- *BOOLEAN*, with values written *True* and *False*.
- *CHARACTER*, with values written as characters in single quotes, such as *'A'*.
- *INTEGER*, with values written using decimal digits possibly preceded by a sign, such as *–889*.
- *REAL*, with values such as *–889.72*.
- *DOUBLE*, with values written in the same way as for *REAL*.
- *STRING*, with values made of character strings in double quotes such as *"A SEQUENCE OF $CHARACTERS#"*.

All these types except STRING are called basic types. See 11.11, page 244

All these examples use "manifest" constants; see below.

For types other than these, you may obtain an effect similar to that of constants by using a once function. For example, assuming a class

→ See 23.12, page 512, about the effect of calling a once function.

```
class COMPLEX creation
        make_cartesian, …
feature -- Initialization
        make_cartesian (a, b: REAL) is
                        -- Initialize to real part a, imaginary part b.
                do
                        x := a; y := b
                end
feature -- Access
        x, y: REAL
        … Other features and invariant …
end -- class COMPLEX
```

you may, in another class, define the once function

```
i: COMPLEX is
                -- Complex number of real part 0, imaginary part 1
        once
                create Result.makecartesian (0, 1)
        end
```

which creates a *COMPLEX* object on its first call; this call and any subsequent one return a reference to that object.

Returning to true constant attributes: the declaration of a constant attribute must determine the attribute's value. You may specify such a constant value in either of two ways:

- If you want to choose the value yourself, use a **manifest** constant.
- You may also let language processing tools select an appropriate value. This is the policy for **unique** values.

The next sections examine these two cases.

18.5 CONSTANT ATTRIBUTES WITH MANIFEST VALUES

A Manifest_constant is a constant given by its explicit value. It may be a Boolean_constant, Character_constant, Integer_constant, Real_constant or Manifest_string.

Chapter 29 describes the precise form of manifest constants.

There is no lexical difference between a real constant and a double constant.

Here are some constant attribute declarations using Manifest_constant values:

```
Terminal_count: INTEGER is 247
Cross: CHARACTER is 'X'
No: BOOLEAN is False
Height: REAL is 1.78
Message: STRING is "No such file"
```

More generally, a feature declaration is a constant attribute declaration if it satisfies the following conditions:

← *"HOW TO RECOGNIZE FEATURES", 5.11, page 82.*

- There is no Formal_arguments part.
- There is a Type_mark part.
- There is a Constant_or_routine part, which contains either a Manifest_constant or a Unique.

A straightforward validity constraint governs such declarations:

Manifest Constant rule *CQMC*

A declaration of a feature *f* introducing a manifest constant is valid if and only if the Manifest_constant *m* used in the declaration matches the type *T* declared for *f* in one of the following ways:

1 • *m* is a Boolean_constant and *T* is *BOOLEAN*.

2 • *m* is a Character_constant and *T* is *CHARACTER*.

3 • *m* is an Integer_constant and *T* is *INTEGER*.

4 • *m* is a Real_constant and *T* is *REAL* or *DOUBLE*.

5 • *m* is a Manifest_string and *T* is *STRING*.

18.6 UNIQUE ATTRIBUTES

Unique constants describe positive integer values chosen by the language processing tool (for example a compiler) rather than by the programmer.

Unique attributes are useful for sets of codes describing different variants of a certain phenomenon. You are guaranteed to get different values for all unique attributes declared in the same class, without having to choose — or know — these values.

You may for example use declarations of the following form:

```
Full_time, Part_time: INTEGER is unique
Blue, Red, Green, Yellow: INTEGER is unique
```

Such features are constant attributes of type *INTEGER*. The difference with a Manifest_constant declaration is that here the constant values are chosen by the language processing tool; so you avoid the need to invent integer codes, as in

Warning: Not necessary; use the above Unique *declaration instead.*

```
Blue_code: INTEGER is 1
Red_code: INTEGER is 2
Green_code: INTEGER is 3
Yellow_code: INTEGER is 4
```

To discriminate between unique values, you may use a Multi_branch instruction, such as this one assuming an expression *n* of type *INTEGER*:

← *"A NOTE ON SELECTION INSTRUCTIONS", 16.6, page 366*.

```
inspect
        n
when Blue then
        some_action
when Red, Green, Yellow then
        other_action
else
        default_action
end
```

This will execute *some_action* if *n* has the value of *Blue*, *other_action* if it has any of the values of *Red, Green, Yellow*, and *default_action* otherwise.

Unique values are guaranteed to be positive, and different for all unique attributes introduced in a given class. Furthermore, for unique attributes declared as part of the same Feature_declaration, such as *Blue*, *Red*, *Green*, *Yellow* above, the values are guaranteed to be consecutive; this means that you can safely use intervals in a Multi_branch instruction, and rewrite the preceding example as

```
inspect
        n
when Blue then
        some_action
when Red .. Yellow then
        other_action
else
        default_action
end
```

Here now are the precise rules. A unique attribute is an integer constant attribute whose value is simply expressed by the keyword **unique**:

Unique ≜ **unique**

The validity constraint limits this mechanism to integers:

Unique Declaration rule *CQUD*

A declaration of a feature *f* introducing a Unique constant is valid if and only if the type declared for *f* is *INTEGER*.

The semantics is defined by a rule which provides the missing case of the Multiple Declaration semantics rule:

← *The "Multiple Declaration semantics", page 93 mentioned this as the only case of a multiple declaration having more properties than the sequence of its individual declarations.*

Unique Declaration semantics

The value of an attribute declared as unique is a positive integer. Two unique attributes belonging to the same system have different values. Unique attributes declared as part of the same Feature_declaration are guaranteed to have consecutive values, in the order given.

Because of this guarantee of consecutiveness, you may for example use the first and last elements of a list of unique attributes declared together to set the dimensions of an array, as in

A Unique declaration does not introduce an "enumerated type". Nothing binds the different attributes together; they are simply integer constants whose values are not chosen by the author of the class.

```
a: ARRAY [INTENSITY]
…
create a.make (Blue, Yellow)
```

which declares and creates an array *a*, using the creation procedure *make* to set the bounds to *Blue* and *Yellow*. Then you know that the legal indices for *a* are *Blue*, *Red*, *Green* and *Yellow* as declared above.

Another example that relies on the consecutiveness of values is their use in an interval for a Multi_branch instruction, as in the branch **when** *Red* **..** *Yellow* above.

Because the values of unique attributes are guaranteed to be positive, you can safely use a set of unique attributes to handle a number of specific cases, and zero or negative codes to handle other cases. For example, if *n* is an integer entity, a conditional instruction of the form

```
if n <= 0 then
        action
end
```

will not execute *action* if *n* has been assigned any one among the values of *Blue*, *Red*, *Green* and *Yellow*.

The only properties that you may expect of the values of unique attributes are those specified above: uniqueness of the values of all unique attributes in a given system; all values positive; consecutive values for unique attributes declared as part of the same Feature_declaration. Any further properties are dependent on the implementation of unique values in a particular language processing tool. In particular:

- The implementation may choose any positive values for unique attributes as long as they satisfy the Unique Declaration semantics. In particular, values do not have to start at 1.
- You may not make any assumption on the values of unique attributes declared in separate Feature_declaration clauses, even if the declarations are in the same class and appear consecutively. For example, the above declaration of *Full_time* and *Part_time* was separate from that of *Blue*, *Red*, *Green*, *Yellow*; this means that, although the values of *Full_time* and *Part_time* will be consecutive, and so will be those of *Blue*, *Red*, *Green*, *Yellow*, you may not assume any connection between those two sets of values.

Unique attributes and the Multi_branch instruction should only be used in cases of simple multi-value discriminations where the set of choices is frozen. In more complex cases, using this mechanism could defeat the goals of extendibility and reusability, which are at the heart of the Eiffel method. For such situations you should rely on the dynamic binding of redefined routines, as explained in the discussion of conditional and multi-branch instructions.

← *"A NOTE ON SELECTION INSTRUCTIONS", 16.6, page 366*.