Draft 5.02.00-0, 15 August 2005 (Santa Barbara). Extracted from ongoing work on future third edition of "Eiffel: The Language". Copyright Bertrand Meyer 1986-2005. Access restricted to purchasers of the first or second printing (Prentice Hall, 1991). Do not reproduce or distribute.

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).

→ *Principle of uniform access*: 23.4, page 616.

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.

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.

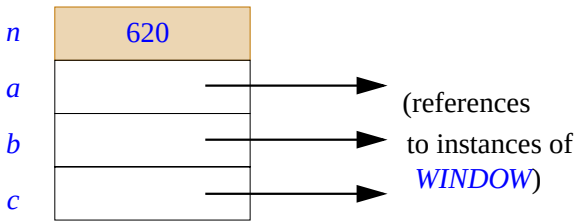
Here are some variable attribute declarations:

```
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*.

← *"Unfolded form of a possibly multiple declaration", page 158.*

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", 5.12, page 145.*

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

18.4 ATTRIBUTES IN FULL FORM

Attribute bodies

Attribute ≜ **attribute** Compound

The Compound is empty in most usual cases, but it is required for an attribute of an attached type (including the case of an expanded type) that does not provide *default_create* as a creation procedure; it will then serve to initialize the corresponding field, on first use for any particular object, if that use occurs prior to an explicit initialization. To set that first value, assign to **Result** in the Compound.

Such a Compound is executed at most once on any particular object during a system execution.

18.5 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 29.2, page 777, 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*.
- *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 , page 330

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 23.15, page 638, about the effect of calling a once function.*

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

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, using a **manifest** constant.

The next section details this case.

18.6 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 32 describes the precise form of manifest constants.

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", 5.12, page 145*.

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

A straightforward validity constraint governs such declarations:

Manifest Constant rule *VQMC*

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 one of the sized variants of *CHARACTER* for which *m* is a valid value.

3 • *m* is an Integer_constant and *T* is one of the sized variants of *INTEGER* for which *m* is a valid value.

4 • *m* is a Real_constant and *T* is one of the sized variants of *REAL* for which *m* is a valid value.

5 • *m* is a Manifest_string and *T* is one of the sized variants of *STRING* for which *m* is a valid value.

6 • *m* is a Manifest_type, of the form {*Y*} for some type *Y*, and *T* is *TYPE* [*X*] for some constant type *X* to which *Y* conforms.

The “valid values” are determined by each basic type’s semantics; for example 1000 is a valid value for *INTEGER_16* but not for *INTEGER_8*.

In case 6, we require the type listed in a Manifest_type {*Y*} to be *constant*, meaning that it does not involve any formal generic parameter or anchored type, as these may represent different types in different generic derivations or different descendants of the original class. This would not be suitable for a constant attribute, which must have a single, well-defined value.