Comparing and duplicating objects

21.1 OVERVIEW

The just studied Creation instruction is the basic language mechanism for obtaining new objects at run time; it produces fresh direct instances of a given class, initialized from scratch.

Sometimes you will need instead to copy the contents of an existing object onto those of another. This is the **copying** operation.

A variant of copying is **cloning**, which produces a fresh object by duplicating an existing one.

For both copying and cloning, the default variants are "shallow", affecting only one object, but **deep** versions are available to duplicate an object structure recursively.

A closely related problem is that of *comparing* two objects for shallow or deep equality.

The copying, cloning and comparison operations rely on only one language construct (the object equality operator ~) and are entirely defined through language constructs but through routines that developer-defined classes inherit from the universal class *ANY*. This makes it possible, through feature redefinitions, to adapt the semantics of copying, cloning and comparing objects to the specific properties of any class.

← *"ANY", 6.6, page 172; see also chapter 35 for more details.*

21.2 COPYING AN OBJECT

---- MOVE AND REWRITE The first operation copies the fields of an object onto those of another. It is provided by the procedure *copy* from class *ANY*. Descendant classes may redefine *copy* to provide a form of copy specific to object of the corresponding types, but the original version is always available through the frozen variant *identical_copy*.

21.3 EQUALITY EXPRESSIONS

--- MOVED FROM EXPRESSION CHAPTER, NOT UPDATED --

Object comparison features from *ANY*

The features whose contract views appear below are provided by class *ANY*.

```
default_is_equal (other: ? like Current)
        -- Is other attached to object field-by-field equal
        -- to current object?
    ensure
        same_type: Result implies same_type (other)
        symmetric: Result = ((other /= Void) and then
                    other.default_is_equal (Current))
        consistent: Result implies is_equal (other)

is_equal (other: ? like Current)
        -- Is other attached to object considered equal
        -- to current object?
    ensure
        same_type: Result implies same_type (other)
        symmetric: Result = ((other /= Void) and then
                    other.is_equal (Current))
        consistent: default_is_equal (other) implies Result
```

The original version of *is_equal* in *ANY* has the same effect as *default_is_equal*.

These are the two basic object comparison operations. The difference is that *default_is_equal* is frozen, always returning the value of field-by-field identity comparison (for non-void *other*); any class may, on the other hand, redefine *is_equal*, in accordance with the pre- and postcondition, to reflect a more specific notion of equality.

Both functions accept a void argument and will in that case, as the header comment implies, return **False**.

An Equality expression serves to test equality of values with the symbol =, or their inequality with the symbol /=. Typical examples are

```
border_color = Black_color
window.height /= 0
```

The syntax is straightforward:

Equality expressions

Equality ≜ Expression Comparison Expression

Comparison ≜ "=" | "/=" | "~" | "/~"

The operators =, /= and ~ have the same precedence as relational operators such as < and >=, higher than the boolean operators such as **and** and **or**, and lower than arithmetic operators such as + and *.

→ *"Precedence and Parenthesized Form", page 758*

There is no constraint on equality expressions. In particular it is not necessary that either of the operands conform to the other. If they don't (or if one is void and the other attached to an object) the result will be false, but that doesn't make the expression illegal: whatever the answer, it's permitted to ask the question.

Equality Expression Semantics

The Boolean_expression *e* ~ *f* has value true if and only if the values of *e* and *f* are both attached and such that *e*.*is_equal* (*f*) holds.

The Boolean_expression *e* = *f* has value true if and only if the values of *e* and *f* are one of:

1 • Both void.

2 • Both attached to the same object with reference semantics.

3 • Both attached to objects with copy semantics, and such that *e* ~ *f* holds.

The form with ~ always denotes object equality. The form with = denotes reference equality if applicable, otherwise object equality. Both rely, for object equality, on function *is_equal* — the version that can be redefined locally in any class to account for a programmer-defined notion of object equality adapted to the specific semantics of the class.

The semantics of the equality operators and ~ was explored in detail as part of the discussion on reattachment. As a reminder, *e* ~ *f* is true if and only if *e* and *f* are attached to equal objects, according to the *is_equal* function from class *ANY*; as to *e* = *f* :

See 22.16, page 610.

1 • If both *e* and *f* are of reference types, the expression denotes reference equality, true if and only if *e* and *f* are either both void or attached to the same object.

2 • If either *e* or *f* is of an expanded type, the expression denotes object equality; it returns the same result as .

If you need a different notion of equality you will, instead of *e* = *f*, use *equal* (*e*, *f*) which takes into account possible redefinitions of *equal*.

Inequality is defined in terms of equality:

Inequality Expression Semantics

The expression *e* /= *f* has value true if and only if *e* = *f* has value false.

The expression *e* /~ *f* has value true if and only if *e* ~ *f* has value false.

Copying and cloning features from *ANY*

The features whose contract views appear below are provided by class *ANY* as secret features.

```
copy (other: ? like Current)
        -- Update current object using fields of object
        -- attached to other, to yield equal objects.
    require
        exists: other /= Void
        same_type: other.same_type (Current)
    ensure
        equal: is_equal (other)

frozen default_copy (other: ? like Current)
        -- Update current object using fields of object
        -- attached to other, to yield identical objects.
    require
        exists: other /= Void
        same_type: other.same_type (Current)
    ensure
        equal: default_is_equal (other)

frozen cloned: like Current
        -- New object equal to current object
        -- (relies on copy)
    ensure
        equal: is_equal (Result)

frozen default_cloned: like Current
        -- New object equal to current object
        -- (relies on default_copy)
    ensure
        equal: default_is_equal (Result)
```

The original versions of *copy* and *cloned* in *ANY* have the same effect as *default_copy* and *default_cloned* respectively.

Procedure *copy* is called in the form *x*.*copy* (*y*) and overrides the fields of the object attached to *x*. Function *cloned* is called as *x*.*cloned* and returns a new object, a "clone" of the object attached to *x*. These features can be adapted to a specific notion of copying adapted to any class, as long as they produce a result equal to the source, in the sense of the — also redefinable — function *is_equal*. You only have to redefine *copy*, since *cloned* itself is frozen, with the guarantee that it will follow any redefined version of *copy*; the semantics of *cloned* is to create a new object and apply *copy* to it.

In contrast, *default_copy* and *default_cloned*, which produce field-by-field identical copies of an object, are frozen and hence always yield the original semantics as defined in *ANY*.

All these features are **secret in their original class** *ANY*. The reason is that exporting copying and cloning may violate the intended semantics of a class, and concretely its invariant. For example the correctness of a class may rely on an invariant property such as

some_circumstance **implies** (*some_attribute* = *Current*)

stating that under *some_circumstance* (a boolean property) the field corresponding to *some_attribute* is cyclic (refers to the current object itself). Copying or cloning an object will usually not preserve such a property. The class should then definitely not export *default_copy* and *default_cloned*, and should not export *copy* and *cloned* unless it redefines *copy* in accordance with this invariant; such redefinition may not be possible or desirable. Because these features are secret by default, software authors must decide, class by class, whether to re-export them.

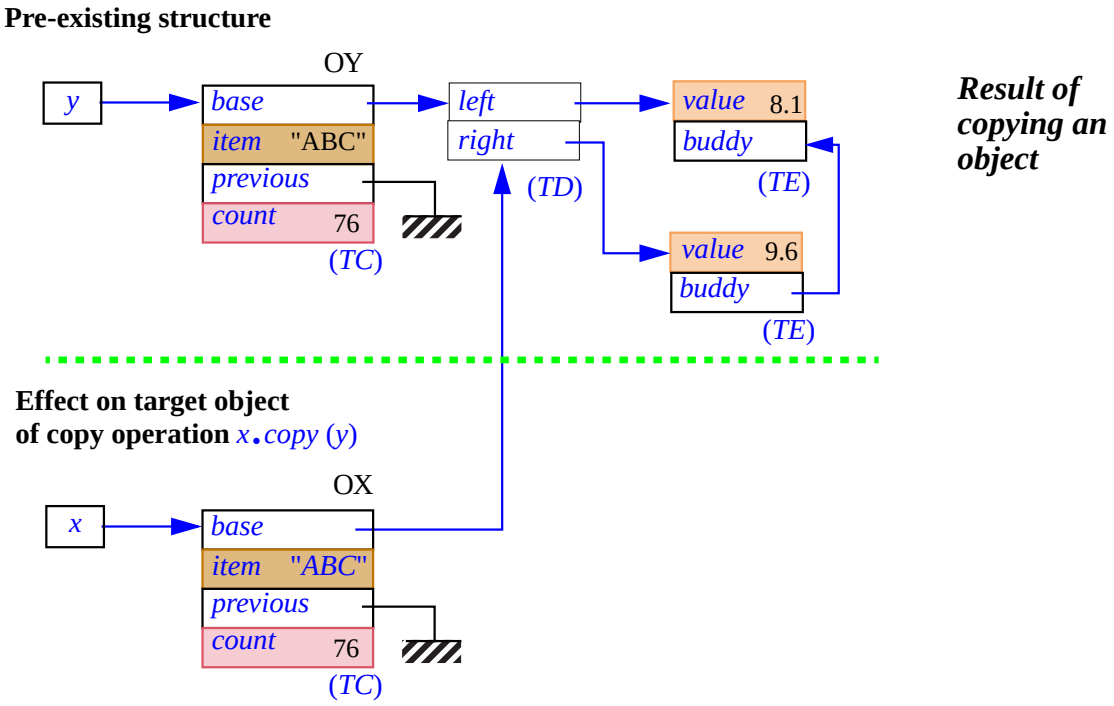
Deep equality, copying and cloning

The feature *is_deep_equal* of class *ANY* makes it possible to compare object structures recursively; the features , *deep_copy* and *deep_cloned* duplicate an object structure recursively. Detailed descriptions are part of the ELKS specification.

The default versions of the earlier features — *default_is_equal*, *default_copy*, *default_cloned* and the original versions of their non-*default* variants — are "shallow": they compare or copy only one source object. The *deep* version recursively compare or copy entire object structures.

Effect of a copy operation

For the copy operation to succeed, both the source and the target must be attached to objects. (Cloning, however, will work for void sources or targets.)



The figure illustrates the effect of a copy operation with *x* as target and *y* as source. If *copy* has not been redefined for the generating class of the object OX attached to *x*, you may obtain this effect through the call

```
x.copy (y)
```

Before the call, *y* was attached to the object labeled OY; *x* was attached to the object labeled OX. What the fields of OX contained then does not matter (since the call overwrites them), but this object must exist. The call copies every field of OY onto the corresponding field of OX.

Since the argument of *copy* is declared of type **like** *Current*, the type of OY conforms to the type of OX, but actually a precondition of *copy* requires more: the types of the two objects must be identical, so that their fields will be in one-to-one correspondence. On the figure, the type of OX and OY is called *TC*.

The fields of OY include expanded values, such as the integer *count*, of value *76*, and references such as *base* and *previous*. In both cases, the copy operation will simply copy the field. For reference fields, no attempt is made to duplicate the data structure recursively: as a result, the *base* fields of both OX and OY will, after the call, be attached to the same object of type *TD*. Applying *copy* to any object containing reference fields will, indeed, always cause sharing of references; later in this chapter we will encounter recursive copy routines, *deep_copy* and *deep_clone*, which duplicate an entire object structure, following references recursively.

→ *See 21.5, page 571 below, about deep copy and clone.*

--- FIX --- As noted, *copy* requires a non-void source and target. For the target, this is simply part of the general requirement on Call instructions: in the above example, *x*, like the target of any other call, must be non-void under penalty of raising an exception. For the source, the requirement is expressed by the precondition of *copy*. A void source will trigger an exception if the execution monitors preconditions.

Specification of default copy

We can now examine the exact specification of *copy*. First, the interface of the procedure's version in class *ANY*:

```
copy, frozen identical_copy (other: like Current)
            -- Copy fields of other onto corresponding fields
            -- of current object.
        require
            other_not_void: other /= Void
            type identity: same_type (other)
        ensure
            equal: is_equal (other)
```

→ *See next about the function same_type used in the precondition and 21.6, page 572 about the function is_equal used in the postcondition.*

Setting the style for other duplication and comparison routines, *copy* has two versions: one redefinable, the other (whose name begins with *identical_*) frozen.

← *Chapter 5 discussed frozen features.*

The second precondition clause uses function *same_type* of *ANY*. For *x* and *y* attached to objects OX and OY, *x*.*same_type* (*y*) has value true if and only if the type of OX has exactly the same type as OY.

→ *same_type is discussed in "OBJECT PROPERTIES", 35.4, page 919*

Here now is the precise effect of the standard version. Assume *copy* has not been redefined and consider a call *x*.*copy* (*y*).

1 • As with any call, the target *x* must be non-void (if it were void the call would cause an exception); the first precondition clause of *copy* states that *y* must also be non-void. Let OX and OY be the attached objects at the time of the call.

2 • The precondition *same_type* requires that OX and OY have the same type; let *T* be that type.

3 • If *T* is a basic type (*BOOLEAN*, *CHARACTER*, *INTEGER*, *REAL* or *POINTER*), the effect of the call is to copy the value of OY onto OX.

4 • If OX and OY are special objects (sequences of values used to represent strings or arrays), it is the implementation's responsibility to ensure that whenever such a situation arises — as a result of copying other objects — the size of OX is at least as large as the size of OY. Then the call copies the value of OY onto OX.

Special objects are not directly accessible to software texts. See 19.2, page 498.

5 • In the remaining cases OX and OY are objects made of zero or more fields, and the second precondition clause, *other_same_type*, implies that the types of OX and OY are identical, so that for every field of OX there is a field of the same type in OY. Then the call copies onto every field of OX the corresponding field of OY.

With repeated inheritance, an attribute of TX may yield two fields in OY. The Select *subclause, 16.5, page 434, determines which one is the field "corresponding" to the relevant OX field.*

As a consequence of the precondition *other_same_type*, you cannot use a copy operation to perform a conversion; a call *your_real*.*copy* (*your_integer*) is incorrect.

→*"CONVERSIONS", 22.6, page 583*.

Tuning copy semantics

Any class may redefine *copy* to provide a copying operation consistent with the notion of object equality that has been deemed appropriate for the class.

Copy and equality are indeed intricately connected: the postcondition of *copy*, given on the previous page, states that the copy must make the target object equal to the source in the sense of function *is_equal*, another feature of *ANY* covered in detail later in this chapter. Clearly, if you redefine either one of *copy* and *is_equal*, you must redefine the other as well, to maintain consistent semantics for copying and equality according to the postcondition redefinition rules.

←*"REDECLARATION AND ASSERTIONS", 10.17, page 277*.

Redefinitions of *copy* and *is_equal* are of two kinds, going in reverse directions: one makes the semantics more "shallow" and the other makes it more "deep":

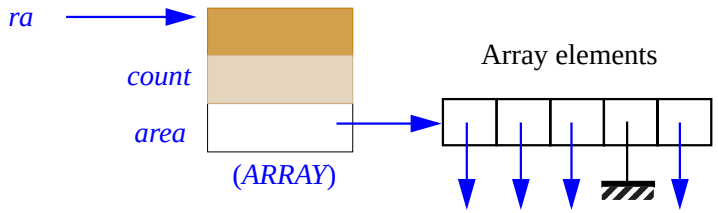
- Sometimes you want to loosen the condition under which two instances of a class are considered equal, by ignoring some fields. Then copy can be redefined to copy only the relevant fields.
- You may instead want copy and equality to involve not only the original objects but also others to which they contain references.

  In some cases you might want both: ignore some fields of the original objects, but involve some other objects as well.

Many classes of the Kernel Library and EiffelBase provide examples of the second kind, as they describe objects which are just headers for complex structures; copy and equality will then involve complete structures, not just the headers. For example the semantics of class *ARRAY* suggests an implementation as illustrated:

Array descriptor and array elements

(*See chapter 36.*)



The array object on the left is a header containing some general information such as the number of elements, *count*, and a reference *area* to the special object containing the array elements (which are references on the figure, but could be expanded values, for example of basic types). The default *copy* would only copy the *ARRAY* object; the procedure *copy* as redefined in class *ARRAY* also duplicates the special object containing the array elements. The same scheme applies to class *STRING*; the version of *copy* in list classes such as *LINKED_LIST* and *ARRAYED_LIST* copies not only the list headers but the list cells themselves.

Arrays, strings and the supporting Kernel Library classes are covered in chapter 36 and and10.

The copy algorithm stops there, however: it doesn't recursively duplicate the actual contents of the list. It's the same for arrays: in the above figure, *rb*.*copy* (*ra*) will copy the special object shown under "Array elements", but not the objects to which its references are attached. For fully recursive duplication, you can use *deep_copy*, presented later in this chapter.

→ *"DEEP COPYING AND CLONING", 21.5, page 571.*

Part of the reason for redefinining *copy* is indeed that sometimes the default version — available as *identical_copy* — doesn't duplicate enough, while *deep_copy* duplicates too much. By redefining *copy* you can prescribe the exact depth you want — in accordance with your desired notion of equality, expressed by a redefinition of *is_equal* — for habitual copy operations.

21.4 CLONING AN OBJECT

Instead of copying an object, you can clone it; this creates a new object rather than updating the fields of an existing object. In class *ANY*, feature *clone* is a function, so a call of the form

```
clone (y)
```

is syntactically an expression; evaluating it will return a new object, which is a copy of the object attached to *y* if any. If *y* is void, the result is void.

Using cloning

The most obvious use of *clone* is in an assignment:

```
x := clone (y)
```

where the type of *y* must be a descendant of the type of *x*. The figure used to illustrate *x.copy* (*y*) also describes the effect of this assignment; only now the object OX represents a new object created by the assignment.

← *Page 563.*

Another use of *clone* is to pass a fresh copy of an existing object as argument to a call, as in

```
some_routine (…, clone (y), …)
```

Although closely related, copy and clone differ in three respects:

C1 •Copy modifies an existing object, whereas clone creates a new object. In the above assignment, any earlier attachment between *x* and some object is lost.

C2 •For copy to work, the target must be non-void; this is expressed syntactically by the nature of *copy*, a procedure in *ANY*. In contrast, *clone* is a function and does not by itself have a target; it simply produces a result. When used as part of an assignment of target *x* as above, it does not care whether *x* is void or attached.

C3 •Finally, because *clone* does not presuppose an existing target object, it can handle a void source. The result in this case is simply a void reference.

Like *copy*, *clone* does not attempt to follow references for fields of reference types, but simply copies the fields; a "deep" version is available.

→ *"DEEP COPYING AND CLONING", 21.5, page 571.*

As with a Creation instruction, a call to *clone* will fail, triggering an exception (the same one, of type *NO_MORE_MEMORY*) if it attempts to create a new object and no memory is available for it.

← *"CREATION SEMANTICS", 20.12, page 548.*

Twin

The description of *clone* indicates (property C3 above) that *Void* is a valid argument, for which the function will return *Void* as its result. This is convenient in the vast majority of cases. If you do know that the source of the clone operation is not void, you may, instead of *clone* (*y*), use

```
y.twin
        -- Defined only if y /= Void; then has same value as clone (y)
```

The only advantage of *twin* over *clone* — apart from being a little more concise — is that its implementation doesn't need to test for *Void*, so it will normally be slightly faster. But you should make sure to reserve *twin* for cases in which the target is known for sure to be non-void, since a void target would cause a run-time exception. If the case may arise, use *clone*, which handles void references gracefully.

Whenever one of the routines of this chapter handles a certain type of value and it is possible to define a reasonable default response for cases in which that value is void, the routine follows the example of *clone* and treats that value as an argument, not as the target of calls.

For a non-void *y*, *clone* (*y*) and *y.twin*, both applicable, are guaranteed always to yield the same value, thanks to the rules seen next.

Specification of default cloning

Here are the interfaces of function *clone* and its *twin* variant:

```
frozen clone (other: ANY): like other
            --Void if other void; otherwise, new object equal to
            -- object attached to other.
        ensure
            equal: equal (Result, other)
            preserves_void: (other = Void) implies (Result =Void)
            same_as_twin: (other /= Void) implies
                                    equal (Result, other.twin)
```

The function 'equal' used in the postcondition is derived from 'is_equal'. See below.

```
frozen twin: like Current
            -- New object equal to current object
        ensure
            not_void: Result /= Void
            equal: Result.is_equal (Current)
            same_as_clone: identical (Result, clone (Current))
```

Why are *clone* and *twin* frozen? The reason is not that their effect is immutable, but that you can change that effect without redefining the functions. To guarantee compatible semantics for cloning and copying, *clone* and *twin* are defined in terms of *copy*, and so will follow any redefinition of *copy*.

A frozen routine may, of course, call routines which are not frozen; it will then be affected by their redefinitions.

More precisely, here is the definition of the semantics of a call *clone* (*y*):

1 • If the value of *y* is void, the call returns a void value.

2 • If the value of *y* is attached to an object OY, the call returns a newly created object of the same type as OY, initialized by applying *copy* to that object with OY as source.

The second case also defines the semantics of *y.twin*. (For void *y* the general rules on routine call imply that the call will trigger an exception.)

In exactly the same way, function function *equal*, used in the postcondition of *clone*, will automatically follow any redefinition of *is_equal*, used in the postcondition of *copy*. As we'll see in the discussion of equality, *equal* is to *is_equal* like *clone* to *twin*: it accepts a void target, but for non-void target returns the same result.

→ *"OBJECT EQUALITY", 21.6, page 572*.

To guarantee the original semantics of field-by-field duplication and ignore any redefinition of *copy*, you may use function *identical_clone*, which has the same signature as *clone*, and is defined in terms of *identical_copy* exactly as *clone* is defined from *copy*.

In principle, *clone* is superfluous: you could in most cases use a Creation and copy instead, replacing

```
create y …
y.copy (x)
```

In practice, however, several reasons justify a separate *clone* facility:

- It's more concise to use *clone* than a creation followed by a copy, particularly in an expression, or in an argument to a routine call *r* (…, *clone* (*x*), …) where the other form would be much more verbose, requiring the declaration of a local variable *y* and two extra instructions.
- If the associated class has two or more creation procedures, a Creation instruction forces you to choose one, although the choice is irrelevant.
- The creation procedure may do some extra work, justified when you create an object from scratch, but unneeded or harmful when all you need is a duplicate of an existing object.
- A Creation forces the client to specify the exact type of the new object, whereas a call to *clone*, as emphasized next, may dynamically produce an object of one among several possible types, depending on the type of the source, selected at run time. This is especially interesting for Formal_generic_name types, since *clone* may be applicable even when plain creation isn't.

← *"CREATING INSTANCES OF FORMAL GENERICS", 20.9, page 535*.

Cloning, types and factories

If *x* is an expression of type *T*, and its value is not void, the generating type of the object created by a call to *clone* (*x*) is not necessarily *T*: it is the type *U* of the object to which *x* is attached. *U* will always (ignoring the conversion case) conform to *T*, but may be based on a proper descendant. In fact *T* might be deferred, in which case there are no objects of generating type *T*.

The generating type of an object is the type of which it is a direct instance. See 19.2, page 498.

Assume *fig1* and *fig2* declared of the deferred type *FIGURE*, with *fig1* attached, at some point during execution, to an instance of an effective descendant of *FIGURE*, such as *CIRCLE*. Executing

```
fig2 := clone (fig1)
```

will attach *fig2* to another *CIRCLE*.

In such cases you don't need to know the exact dynamic type of the source (here *fig1*) when writing the instruction; because of polymorphism, that type may be different for successive executions of the same instruction.

An earlier discussion introduced an important application of these properties: how to implement a **factory of objects** through the **clonable array technique**. The idea was simply to obtain a fresh instance of a type, selected from a set of variants by a certain *code*, by writing

← *"Single choice and factory objects", page 529 ; final, simplified form on page 530.*

```
x := clone (factory @ code)
```

where the *factory* is an array automatically created on first use — thanks to the beauties of once functions and creation expressions — through a simple function, worth showing again:

```
factory: ARRAY [FIGURE]
        once
                Result.make (Low_id, High_id)

                        -- Create and enter an instance of each desired kind:
                Result.put (create {SEGMENT}.make (…), Segment_id)
                Result.put (create {TRIANGLE}.make (…), Triangle_id)
                … Similarly for each variant …
        end
```

← *First shown on page 530. The example involves a set of figure types.*

This provide a simple and easily extendible scheme, compatible with the Single Choice principle and much preferable to the first form shown, which used explicit discrimination through a Multi_branch.

← *"Single choice and factory objects", page 529 ; original form [1], page 527.*

21.5 DEEP COPYING AND CLONING

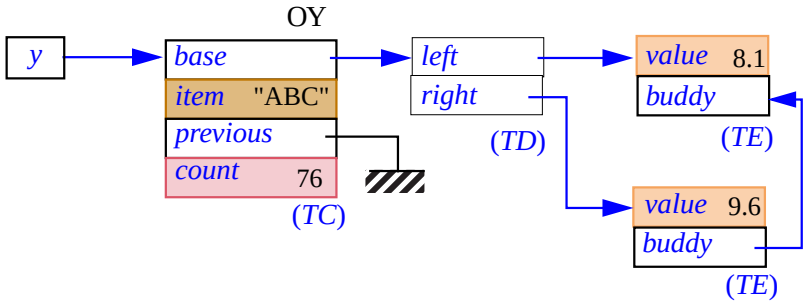
The default *clone* and *copy* are, as noted, *shallow*: they do not follow references, just copy fields of the source object as they appear.

You may in some cases need deep versions of these operations, which will recursively duplicate an entire structure. The routines *deep_clone* and *deep_copy* of class *ANY*, with the same signatures as *clone* and *copy* respectively, fulfill this need. They will replicate an entire data structure, creating as many new objects as needed.

If we take as an illustration the original example used to present shallow copying, here is the result of a deep clone on the same structure:

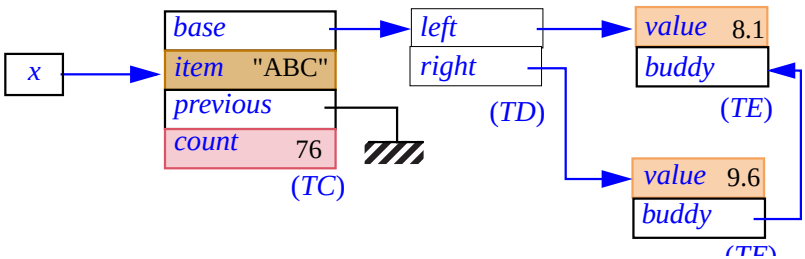
← *Page 563.*

Pre-existing structure



Result of deep cloning

Structure created by a deep-clone operation *x* := *deep_clone* (*y*)



Your usual supplier of memory upgrades and discount disks will be happy to provide your staff, at no charge, with a full training session on the use of deep_copy and deep_clone.

Unlike their shallow counterparts *clone* and *copy*, *deep_clone* and *deep_copy* cannot cause sharing of references between source and target.

The deep versions are frozen. Their postconditions involve *deep_equal*, studied below.

→ *"DEEP EQUALITY", 21.7, page 574.*

21.6 OBJECT EQUALITY

---- OBSOLETE SECTION, REWRITE WITH ~ -----The discussion of cloning and copying transposes readily to the problem of comparing objects for equality. To determine if the objects attached to *x* and *y* are equal, you may use the expression

```
equal (x, y)
```

SEMANTICS

Here is the result of applying the default *equal* to two values *x* and *y*.

For convenience, the short form of the function equal appears after this semantic specification.

1 • If any one of *x* and *y* is void, the result is true if the other is also void, and false otherwise. Cases 2 to 5 assume that both arguments are attached to respective objects OX and OY.

2 • .If the types of OX and OY are not identical, the result is false. For cases 3 to 5 let *T* be their common type.

3 • If *T* is a basic type, the result is true if and only if OX and OY are the same value

← *"Basic types", , page 330*

4 • If OX and OY are special objects (sequences of values used to represent strings or arrays), the result is true if and only if the sequences have the same length, and every field in one is identical to the field at the same position in the other.

Special objects are not directly accessible to software texts. See 19.2, page 498.

5 • Otherwise OX and OY are standard complex objects, and conformance of *TY* to *TX* implies that for every field of OX there is a corresponding field in OY. Then the result is true if and only if every reference field of OX is attached to the same object as the corresponding field in OY, and every subobject field of OX is (recursively) equal to the corresponding field in OY.

This definition of *equal*'s semantics closely parallels the semantic definition of *copy*; the five cases in both specifications match each other. The two are indeed designed to be compatible since, as noted, a call of the form *x*.*copy* (*y*) must ensure the postcondition *equal* (*x*, *y*).

Like copying, equality does not take conversions into account. The expression *equal* (*0.0*, *0*) — with a real argument and an an integer argument — will return false. To get different behavior you must take care of the conversion yourselves.

The rejection of any conversions is part of a more general decision reflected in clause 2 above: equality may only hold for objects of the exact same type. You may be interested to know that the policy was more lax in early versions of Eiffel (as reflected in the first edition of this book): it specified the value true for *equal* (*x*, *y*) if the type of *y* conforms to the type of *x* and two objects have equal fields for the attributes of *x*'s type, even though OY may have more fields. This policy was more flexible, and did not cause any major problems; it went well, in particular, with the use of *is_equal* as the basic equality operation, explained next. It was abandoned, however, when critics pointed out that it made *equal* a non-symmetric property — it could result in *equal* (*x*, *y*) being true while *equal* (*x*, *y*) is not — whereas equality, in mathematics, is always symmetric. Hence the change to a more restrictive view of equality.

The short form of *equal* has not yet been given because in its postcondition it mentions the next function of interest, *is_equal*. Here it is:

```
frozen equal (some: ANY; other: like some): BOOLEAN
            -- Are some and other either both void or attached
            -- to equal objects?
        ensure
            definition: Result = (some = Void and other =Void) or
                (some /= Void and other /= Void and then
                some.is_equal (other))
            symmetric: Result = equal (some, other)
```

Function *is_equal*, for its part, has a frozen synonym *is_identical*, and the interface form

```
is_equal ( other: like Current): BOOLEAN
            -- Is other attached to an object equal to current object?
        ensure
            only_if_not_void: Result implies other /= Void
            same_type: Result implies same_type (other)
            symmetric: Result = other.is_equal (Current)
            consistent: is_identical (other) implies Result
```

To change the semantics of equality in a particular class, just redefine *is_equal*; you cannot directly redefine *equal* — as you can see above, it's frozen — but its postcondition guarantees that *equal* will follow automatically. An obvious way to implement *equal* is indeed to rely on *is_equal*:

```
if some = Void then
        Result := (other = Void)
else
        Result := some.is_equal (other)
end
```

Function *is_equal* has the same relationship to *equal* as *twin* to *clone*: it works on a target and an argument as in *x*.*is_equal* (*y*), where *equal* uses two arguments as in *equal* (*x*, *y*). For non-void *x*, the two will always yield the same result, as defined above, but only *equal* accepts a void *x*; *is_equal* requires a non-void target. So it is the more basic of the two, but *equal* is more general.

Use *equal* (*x*, *y*) when there is any chance that *x* could be void. Otherwise you can still use *equal* except if you are concerned about the small overhead of testing for *Void*. Function *is_equal* is the one to redefine to introduce a specific semantics of equality for instances of a certain class. As noted, this almost always implies an associated redefinition of *copy*.

Earlier on, we encountered library classes — *ARRAY*, *STRING*, list implementations — that redefine *copy* to duplicate not just the header of an object structure but some of its contents too. These same classes redefine *is_equal* to compare the contents and not just the header.

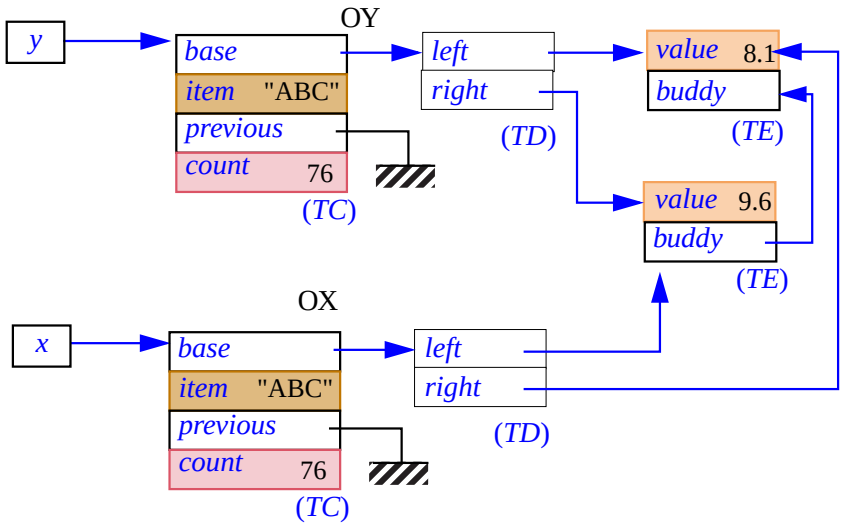
Like *copy* and *clone*, functions *equal* and *is_equal* have frozen synonyms: *identical* and *is_identical*, both guaranteeing the original semantics of exact field-by-field comparison.

21.7 DEEP EQUALITY

Like the shallow forms of copy and clone, the just explored shallow form of equality testing has a deep counterpart in *ANY*:

```
frozen deep_equal (some: ANY; other: like some): BOOLEAN
        -- Are some and other either both void
        -- or attached to isomorphic object structures?
```

What exactly are " isomorphic structures"? Clearly, *deep_equal* should yield true if one of the arguments results from a *deep_clone* or *deep_copy* applied to the other, as *x* and *y* on the figure that illustrated deep_clone. But we shouldn't limit ourselves to this case, because it excludes any sharing between the two object structures, as in the following figure below, where we are entitled to expect that *deep_equal* (*x*, *y*) will yield true.



A case of deep equality

Here is the definition of deep equality (yielding true for such cases). It is convenient to define the notion separately for references and for objects.

Two references *x* and *y* are deep-equal if and only if they are either both void or attached to deep-equal objects.

Two objects OX and OY are deep-equal and only if they satisfy the following four conditions:

1 • OX and OY have the same exact type.

2 • The objects obtained by setting all the reference fields of OX and OY (if any) to void references are equal.

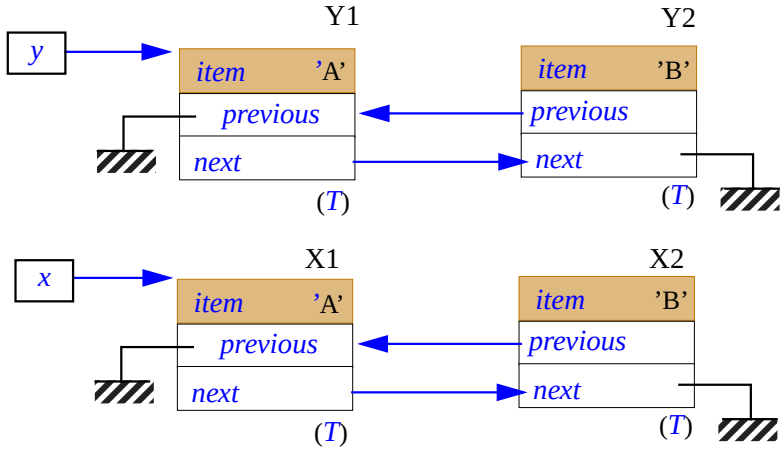
3 • For every void reference field of OX, the corresponding field of OY is void.

4 • For every non-void reference field of OX, attached to an object PX, the corresponding field of OY is attached to an object PY, and it is possible (recursively) to show, under the assumption that OX is deep-equal to OY, that PX is deep-equal to PY.

Condition **1** is the same as for *equal*: we want the types to be identical.

Conditions **2** and **3** express that every expanded or void field must be equal to the field in the other object.

Condition **4** handles the non-void reference fields. It is a bit subtle, as often when recursion is involved. The phrasing seems strange: why not just state that in this case PX must recursively be deep-equal to *PY*?



Another case of deep equality

The problem is that such a condition, although not wrong, would be impossible to prove, or disprove, for any cyclic data structures. Consider the situation picture above, which might be the result of a *deep_clone* operation. How can we check that the objects labeled X1 and Y1 are deep-equal — which they clearly should be?

Condition **1** will raise no problem since all objects are of the same type *T*. Condition **2** is readily satisfied since the only non-reference fields in X1 and Y1, the *item* fields, are equal. Condition 3 is also immediate since both *previous* fields are void. For condition 4, we must check recursively that the objects X2 and Y2 are deep-equal.

Conditions **2** and **3** again hold trivially, covering fields *item* and *next*. There remains to check condition **4**, in other words, that the *previous* fields of X2 and Y2 are attached to deep-equal objects. But now you see the problem: those attached objects are none other than X1 and Y1, and we are back to square one.

The phrasing of condition **4** gets us out of this potentially endless reasoning loop: when checking condition **4** on the original objects X1 and Y1, we only have to check that X2 and Y2 are deep-equal **under the assumption** that X1 and Y1 are themselves deep-equal. So here the equality of the *item* and *next* fields suffices to terminate the proof.

If you are looking at this with a programmer's rather than a mathematician's eyes, you will have understood this clause as meaning that in an abstract traversal algorithm designed to check deep-equality of objects, you may *mark* every previously encountered object so as not to explore it again, avoiding infinite looping.

If, on the other hand, you also master the theoretical background, you will have recognized the idea of self-conditional recursive proof: a technique whereby, to prove a property *R*, you must first prove a property of the form "if *R* holds, then *P* holds" for some other property *P*. This is exactly the scheme used, in *axiomatic* specifications of programming language semantics, to prove the correctness of a recursive routine.

On this theoretical perspective, see the book "Introduction to the Theory of Programming Language"", eparticularly its section : 9.10.6 and the example in 9.10.9.