8

The run-time structure: objects

In the previous chapter we saw that classes may have instances, called objects. We must now turn our attention to these objects and, more generally, to the run-time model of object-oriented computation.

Where the previous chapters were mostly concerned with conceptual and structural issues, the present one will, for the first time in this book, include implementation aspects. In particular it will describe how the execution of object-oriented software uses memory — a discussion continued by the study of garbage collection in the next chapter. As already noted, one of the benefits of object technology is to restore implementation issues to their full status; so even if your interest is mostly in analysis and design topics you should not be afraid of this excursion into implementation territory. It is impossible to understand the method unless you have some idea of its influence on run-time structures.

The study of object structures in this chapter indeed provides a particularly good example of how wrong it is to separate implementation aspects from supposedly higher-level issues. Throughout the discussion, whenever we realize the need for a new O-O technique or mechanism, initially introduced for some implementation-related purpose, the real reason will almost always turn out to be deeper: we need the facility just as much for purely descriptive, abstract purposes. A typical example will be the distinction between references and expanded values, which might initially appear to be an obscure programming technique, but in reality provides a general answer to the question of sharing in whole-to-parts relations, an issue that figures prominently in many discussions of object-oriented analysis.

This contribution of implementation is sometimes hard to accept for people who have been influenced by the view, still prevalent in the software literature, that all that counts is analysis. But it should not be so surprising. To develop software is to develop models. A good implementation technique is often a good modeling technique as well; it may be applicable, beyond software systems, to systems from various fields, natural and artificial.

More than implementation in the strict sense of the term, then, the theme of this chapter is modeling: how to use object structures to construct realistic and useful operational descriptions of systems of many kinds.

8.1 OBJECTS

At any time during its execution, an O-O system will have created a certain number of objects. The run-time structure is the organization of these objects and of their relations. Let us explore its properties.

What is an object?

First we should recall what the word "object" means for this discussion. There is nothing vague in this notion; a precise technical definition was given in the previous chapter:

The definition appeared on page 166. See also the Object rule, page 171.

Definition: object

An object is a run-time instance of some class.

A software system that includes a class *C* may at various points of its execution create (through creation and cloning operations, whose details appear later in this chapter) instances of *C*; such an instance is a data structure built according to the pattern defined by *C*; for example an instance of the class *POINT* introduced in the previous chapter is a data structure consisting of two fields, associated with the two attributes *x* and *y* declared in the class. The instances of all possible classes constitute the set of objects.

The above definition is the official one for object-oriented software. But "object" also has a more general meaning, coming from everyday language. Any software system is related to some external system, which may contain "objects": points, lines, angles, surfaces and solids in a graphics system: employees, pay checks and salary scales in a payroll system; and so on. Some of the objects created by the software will be in direct correspondence with such external objects, as in a payroll system that includes a class *EMPLOYEE*, whose run-time instances are computer models of employees.

This dual use of the word "object" has some good consequences, which follow from the power of the object-oriented method as a modeling tool. Better than any other method, object technology highlights and supports the modeling component of software development. This explains in part the impression of naturalness which it exudes, the attraction it exerts on so many people, and its early successes — still among the most visible — in such areas as simulation and user interfaces. The method here enjoys the *direct mapping* property which an earlier chapter described as a principal requirement of good modular design. With software systems considered to be direct or indirect models of real systems, it is not surprising that some classes will be models of external object types from the problem domain, so that the software objects (the instances of these classes) are themselves models of the corresponding external objects.

"Direct Mapping", page 47.

But we should not let ourselves get too carried away by the word "object". As always in science and technology, it is a bit risky to borrow words from everyday language and give them technical meanings. (The only discipline which seems to succeed in this delicate art is mathematics, which routinely hijacks such innocent words as "neighborhood", "variety" or "barrel" and uses them with completely unexpected meanings — perhaps the reason why no one seems to have any trouble.) The term "object" is so overloaded with everyday meanings that in spite of the benefits just mentioned its use in a technical software sense has caused its share of confusion. In particular:

- As pointed out in the discussion of direct mapping, not all classes correspond to object types of the problem domain. The classes introduced for design and implementation have no immediate counterparts in the modeled system. They are often among the most important in practice, and the most difficult to find.

See chapter 20 about the form-based system. About the notion of command, see chapter 21.

- Some concepts from the problem domain may yield classes in the software (and objects in the software's execution) even though they would not necessarily be classified as objects in the usual sense of the term if we insist on a concrete view of objects. A class such as *STATE* in the discussion of the form-based interactive system, or *COMMAND* (to be studied in a later chapter in connection with undo-redo mechanisms) fall in this category.

When the word "object" is used in this book, the context will clearly indicate whether the usual meaning or (more commonly) the technical software meaning is intended. When there is a need to distinguish, one may talk about *external objects* and *software objects*.

Basic form

A software object is a rather simple animal once you know what class it comes from.

Let O be an object. The definition on the previous page indicates that it is an instance of some class. More precisely, it is a **direct instance** of just one class, say *C*.

Because of inheritance, O will then be an instance, direct or not, of other classes, the ancestors of *C*; but that is a matter for a future chapter, and for the present discussion we only need the notion of direct instance. The word "direct" will be dropped when there is no possible confusion.

C is called the generating class, or just **generator**, of O. *C* is a software text; O is a run-time data structure, produced by one of the object creation mechanisms studied below.

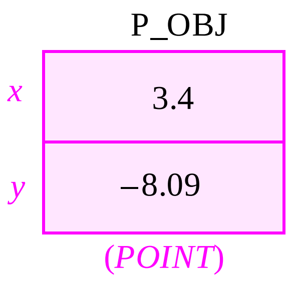
Among its features, *C* has a certain number of attributes. These attributes entirely determine the form of the object: O is simply a collection of components, or **fields**, one for each attribute.

Consider class *POINT* from the previous chapter. The class text was of the form:

For the text of class POINT see page 176.

```
class POINT feature
        x, y: REAL
        … Routine declarations …
end
```

The routines have been omitted, and for good reason: the form of the corresponding objects (the direct instances of the class) is solely determined by the attributes, although the *operations* applicable to the objects depend on the routines. Here the class has two attributes, *x* and *y*, both of type *REAL*, so a direct instance of *POINT* is an object with two fields containing values of that type, for example:



Notice the conventions used here and in the rest of this book for representing an object as a set of fields, shown as adjacent rectangles containing the associated values. Below the object the name of the generating class, here *POINT*, appears in parentheses and in italics; next to each field, also in italics, there appears the name of the corresponding attribute, here *x* and *y*. Sometimes a name in roman (here P_OBJ) will appear above the object; it has no counterpart in the software but identifies the object in the discussion.

See "Graphical conventions", page 271.

In diagrams used to show the structure of an object-oriented system, or more commonly of some part of such a system, classes appear as ellipses. This convention, already used in the figures of the previous chapter, avoids any confusion between classes and objects.

Simple fields

Both attributes of class *POINT* are of type *REAL*. As a consequence, each of the corresponding fields of a direct instance of *POINT* contains a real value.

This is an example of a field corresponding to an attribute of one of the "basic types". Although these types are formally defined as classes, their instances take their values from predefined sets implemented efficiently on computers. They include:

- *BOOLEAN*, which has exactly two instances, representing the boolean values true and false.
- *CHARACTER*, whose instances represent characters.
- *INTEGER*, whose instances represent integers.
- *REAL* and *DOUBLE*, whose instances represent single-precision and double-precision floating-point numbers.

Another type which for the time being will be treated as a basic type, although we will later see that it is actually in a different category, is *STRING*, whose instances represent finite sequences of characters.

"STRINGS", 13.5, page 456.

For each of the basic types we will need the ability to denote the corresponding values in software texts and on figures. The conventions are straightforward:

- For *BOOLEAN*, the two instances are written *True* and *False*.
- To denote an instance of *CHARACTER* you will write a character enclosed in single quotes, such as '*A*'.

- To denote an instance of *STRING*, write a sequence of characters in double quotes, as in "*A STRING*".
- To denote an instance of *INTEGER*, write a number in an ordinary decimal notation with an optional sign, as in *34*, *–675* and *+4*.
- You can also write an instance of *REAL* or *DOUBLE* in ordinary notation, as in *3.5* or *–0.05*. Use the letter *e* to introduce a decimal exponent, as in *–5.e–2* which denotes the same value as the preceding example.

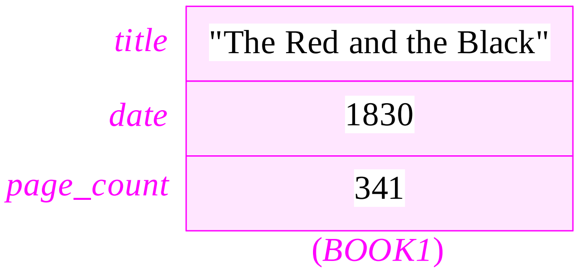
A simple notion of book

Here is a class with attribute types taken from the preceding set:

```
class BOOK1 feature
    title: STRING
    date, page_count: INTEGER
end
```

A typical instance of class *BOOK1* may appear as follows:

An object representing a book



Since for the moment we are only interested in the structure of objects, all the features in this class and the next few examples are attributes — none are routines.

This means that our objects are similar at this stage to the records or structure types of non-object-oriented languages such as Pascal and C. But unlike the situation in these languages there is little we can do with such a class in a good O-O language: because of the information hiding mechanisms, a client class has no way of assigning values to the fields of such objects. In Pascal, or in C with a slightly different syntax, a record type with a similar structure would allow a client to include the declaration and instruction

Warning: not permitted in the O-O notation! For discussion only.

```
b1: BOOK1
…
b1.page_count := 355
```

which at run time will assign value 355 to the *page_count* field of the object attached to *b1*. With classes, however, we should not provide any such facility: letting clients change object fields as they please would make a mockery of the rule of information hiding, which implies that the author of each class controls the precise set of operations that clients may execute on its instances. No such direct field assignment is possible in an O-O context; clients will perform field modifications through procedures of the class. Later in this chapter we will add to *BOOK1* a procedure that gives clients the effect of the above assignment, if the author of the class indeed wishes to grant them such privileges.

We have already seen that C++ and Java actually permit assignments of the form *b1*.*page_count* := *355*. But this simply reflects the inherent limits of attempts to integrate object technology in a C context.

As the designers of Java themselves write in their book about the language: "*A programmer could still mess up the object by setting* [*a public*] *field, because the field* [*is*] *subject to change*" through direct assignment instructions. Too many languages require such "don't do this" warnings. Rather than propose a language and then explain at length how not to use it, it is desirable to define hand in hand the method and a notation that will support it.

[Arnold 1996], *page 40.*

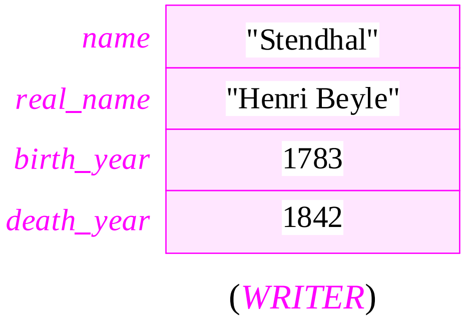
See also "If it is baroque, fix it", page 670.

In proper O-O development, classes without routines, such as *BOOK1*, have little practical use (except as ancestors in an inheritance hierarchy, where descendants will inherit the attributes and provide their own routines; or to represent external objects which the O-O part can access but not modify, for example sensor data in a real-time system). But they will help us go through the basic concepts; then we will add routines.

Writers

Using the types mentioned above, we can also define a class *WRITER* describing a simple notion of book author:

```
class WRITER feature
    name, real_name: STRING
    birth_year, death_year: INTEGER
end
```



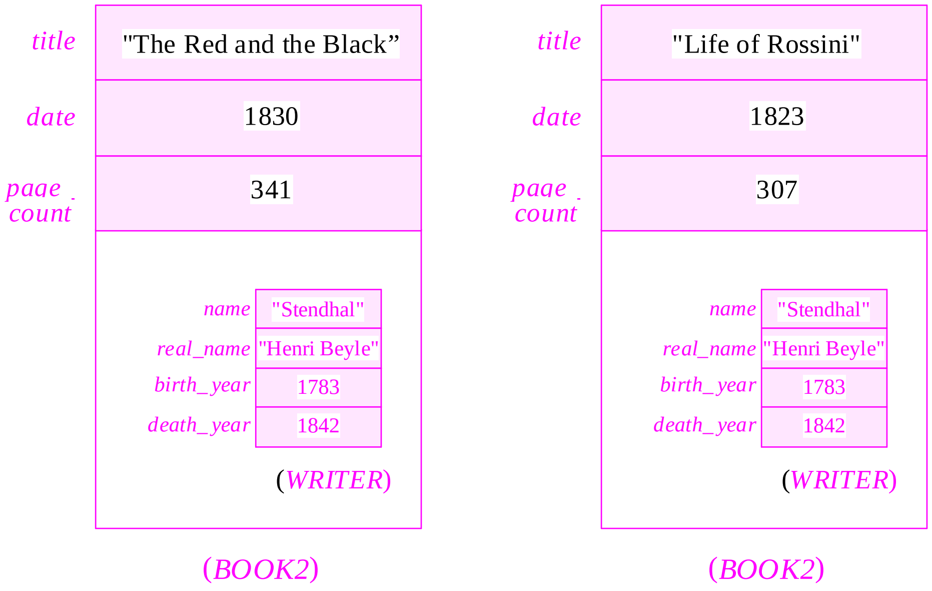
A "writer" object

References

Objects whose fields are all of basic types will not take us very far. We need objects with fields that represent other objects. For example we will want to represent the property that a book has an author — denoted by an instance of class *WRITER*.

A possibility is to introduce a notion of subobject. For example we might think of a book object, in a new version *BOOK2* of the book class, as having a field *author* which is itself an object, as informally suggested by the following picture:

Two "book" objects with "writer" subobjects



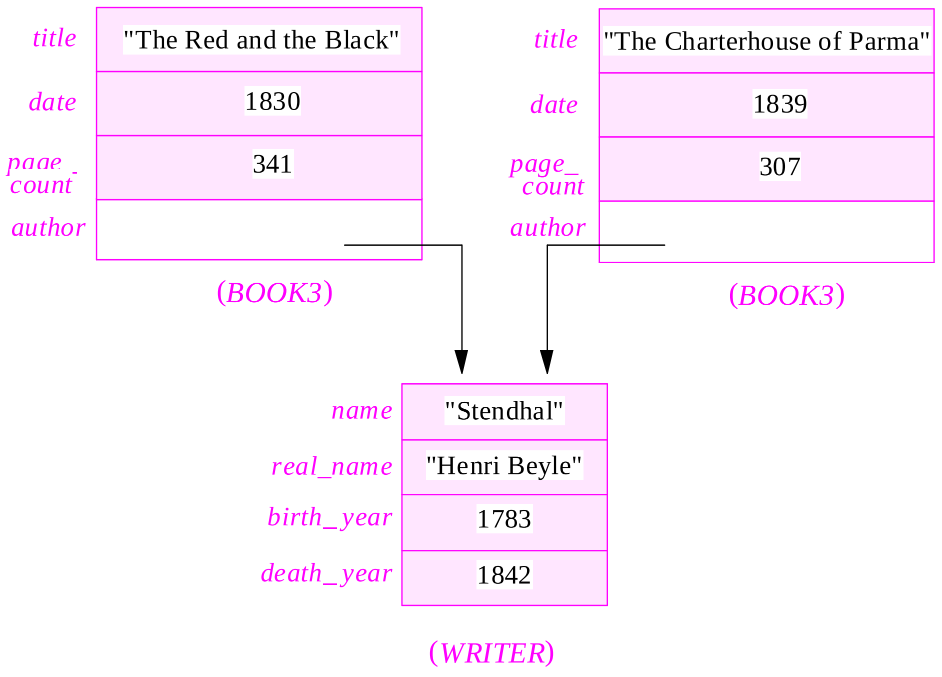
Such a notion of subobject is indeed useful and we will see, later in this chapter, how to write the corresponding classes.

But here it is not exactly what we need. The example represents two books with the same author; we ended up duplicating the author information, which now appears as two subobjects, one in each instance of *BOOK2*. This duplication is probably not acceptable:

- It wastes memory space. Other examples would make this waste even more unacceptable: imagine for example a set of objects representing people, each one with a subobject representing the country of citizenship, where the number of people represented is large but the number of countries is small.

- Even more importantly, this technique fails to account for the need to express **sharing**. Regardless of representation choices, the *author* fields of the two objects refer to the same instance of *WRITER*; if you update the *WRITER* object (for example to record an author's death), you will want the change to affect all book objects associated with the given author.

Here then is a better picture of the desired situation, assuming yet another version of the book class, *BOOK3*:



Two "book" objects with references to the same "writer" object

The *author* field of each instance of *BOOK3* contains what is known as a **reference** to a possible object of type *WRITER*. It is not difficult to define this notion precisely:

Definition: reference

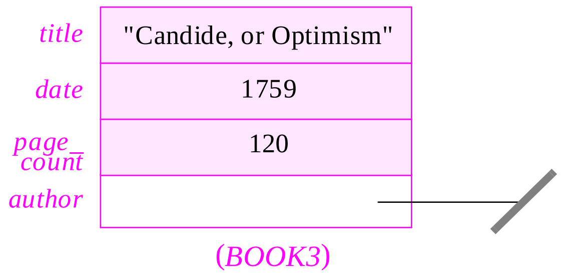
A reference is a run-time value which is either **void** or **attached**.

If attached, a reference identifies a single object. (It is then said to be attached to that particular object.)

In the last figure, the *author* reference fields of the *BOOK3* instances are both attached to the *WRITER* instance, as shown by the arrows, which are conventionally used on such diagrams to represent a reference attached to an object. The following figure has a void reference (perhaps to indicate an unknown author), showing the graphical representation of void references:

An object with a void reference field

(*"Candide" was published anonymously.*)



The definition of references makes no mention of implementation properties. A reference, if not void, is a way to identify an object; an abstract *name* for the object. This is similar to a social security number that uniquely identifies a person, or an area code that identifies a phone area. Nothing implementation-specific or computer-specific here.

The reference concept of course has a counterpart in computer implementations. In machine-level programming it is possible to manipulate addresses; many programming languages offer a notion of pointer. The notion of reference is more abstract. Although a reference may end up being represented as an address, it does not have to; and even when the representation of a reference includes an address, it may include other information.

Another property sets references apart from addresses, although pointers in typed languages such as Pascal and Ada (not C) also enjoy it: as will be explained below, a reference in the approach described here is typed. This means that a given reference may only become attached to objects of a specific set of types, determined by a declaration in the software text. This idea again has counterparts in the non-computer world: a social security number is only meant for persons, and area codes are only meant for phone areas. (They may look like normal integers, but you would not *add* two area codes.)

Object identity

The notion of reference brings about the concept of object identity. Every object created during the execution of an object-oriented system has a unique identity, independent of the object's value as defined by its fields. In particular:

I1 • Two objects with different identities may have identical fields.

I2 • Conversely, the fields of a certain object may change during the execution of a system; but this does not affect the object's identity.

These observations indicate that a phrase such as "*a* denotes the same object as *b*" may be ambiguous: are we talking about objects with different identities but the same contents (I1)? Or about the states of an object before and after some change is applied to its fields (I2)? We will use the second interpretation: a given object may take on new values for its constituent fields during an execution, while remaining "the same object". Whenever confusion is possible the discussion will be more explicit. For case I1 we may talk of equal (but distinct) objects; equality will be defined more precisely below.

A point of terminology may have caught your attention. It is not a mistake to say (as in the definition of I2) that the fields of an object may change. The term "field" as defined above denotes one of the values that make up an object, not the corresponding field identifier, which is the name of one of the attributes of the object's generating class.

For each attribute of the class, for example *date* in class *BOOK3*, the object has a field, for example *1832* in the object of the last figure. During execution the attributes will never change, so each object's division into fields will remain the same; but the fields themselves may change. For example an instance of *BOOK3* will always have four fields, corresponding to attributes *title*, *date*, *page_count*, *author*; these fields — the four values that make up a given object of type *BOOK3* — may change.

"Object identity", page 1052.

The study of how to make objects *persistent* will lead us to explore further properties of object identity.

Declaring references

Let us see how to extend the initial book class, *BOOK1*, which only had attributes of basic types, to the new variant *BOOK3* which has an attribute representing references to potential authors. Here is the class text, again just showing the attributes; the only difference is an extra attribute declaration at the end:

```
class BOOK3 feature
    title: STRING
    date, page_count: INTEGER
    author: WRITER            -- This is the new attribute.
end
```

The type used to declare *author* is simply the name of the corresponding class: *WRITER*. This will be a general rule: whenever a class is declared in the standard form

```
class C feature ... end
```

then any entity declared of type *C* through a declaration of the form

```
x: C
```

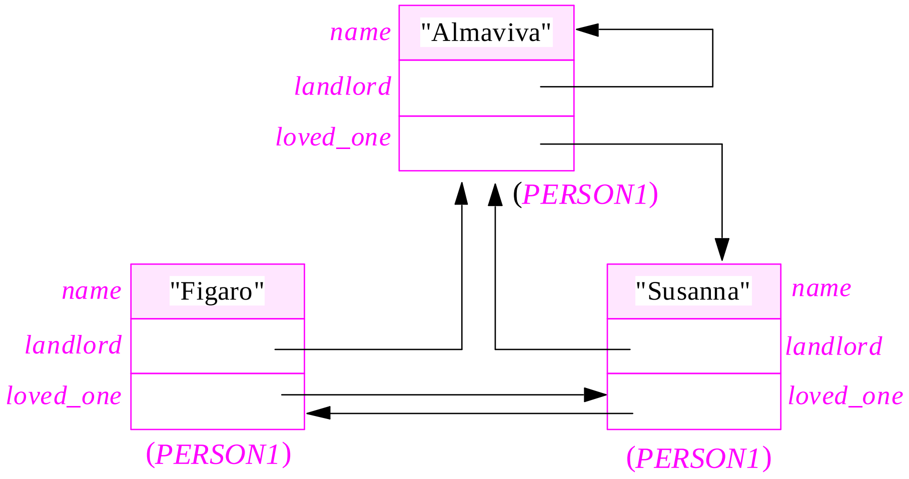
denotes values that are **references** to potential objects of type *C*. The reason for this convention is that using references provides more flexibility, and so are appropriate in the vast majority of cases. You will find further examination of this rule (and of the other possible conventions) in the discussion section of this chapter.

See page 272.

Self-reference

Nothing in the preceding discussion precludes an object O1 from containing a reference field which (at some point of a system's execution) is attached to O1 itself. This kind of self-reference can also be indirect. In the situation pictured below, the object with "Almaviva" in its *name* field is its own landlord (direct reference cycle); the object "Figaro" loves "Susanna" which loves "Figaro" (indirect reference cycle).

Direct and indirect self-reference



Such cycles in the dynamic structure can only exist if the client relation among the corresponding classes also has direct or indirect cycles. In the above example, the class declaration is of the form

```
class PERSON1 feature
    name: STRING
    loved_one, landlord: PERSON1
end
```

showing a direct cycle (*PERSON1* is a client of *PERSON1*).

The reverse property is not true: the presence of a cycle in the client relation does not imply that the run-time structure will have cycles. For example you may declare a class

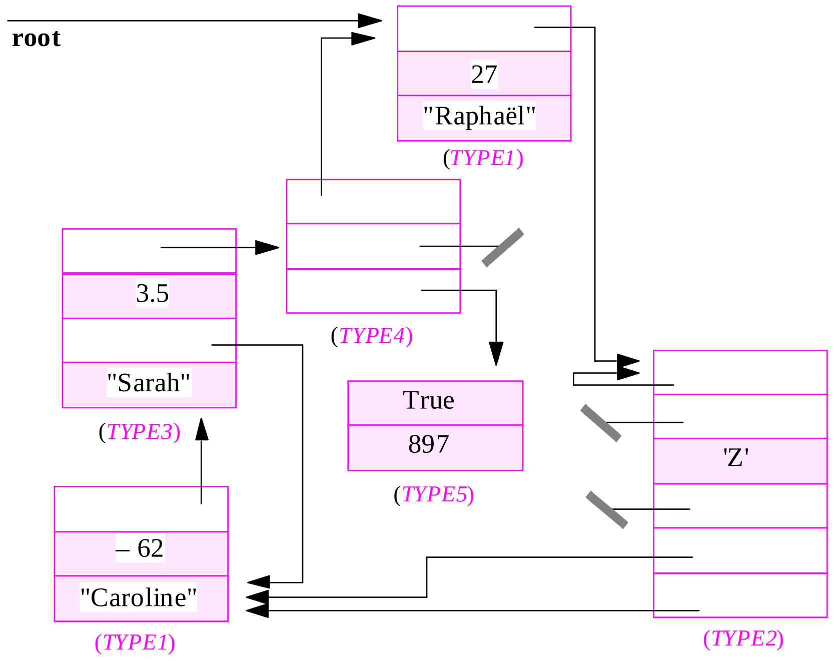
```
class PERSON2 feature
    mother, father: PERSON2
end
```

which is a client of itself; but if this models the relations between people suggested by the attributes' names, there can be no reference cycle in the run-time structure, as it would imply that a certain person is his own parent or indirect ancestor.

A look at the run-time object structure

From what we have seen so far emerges a first picture of the structure of an object-oriented system during its execution.

A possible run-time object structure



The system is made of a certain number of objects, with various fields. Some of these fields are values of basic types (integer fields such as *27*, character fields such as 'Z' and so on); others are references, some void, others attached to objects. Each object is an instance of some type, always based on a class and indicated below the object in the figure. Some types may be represented by just one instance, but more commonly there will be many instances of a given type; here *TYPE1* has two instances, the others only one. An object may have reference fields only; this is the case here with the *TYPE4* instance, or basic fields only, as with the *TYPE5* instance. There may be self-references: direct, as with the top field of the *TYPE2* instance, or indirect, as with the clock-wise reference cycle starting from and coming back to the *TYPE1* instance at the top.

This kind of structure may look unduly complicated at first — an impression reinforced by the last figure, which is meant to show many of the available facilities and does not purport to model any real system. The expression "spaghetti bowl" comes to mind.

But this impression is not justified. The concern for simplicity applies to the software text and not necessarily to the run-time object structure. The text of a software system embodies certain relations (such as "is child of", "loves", "has as landlord"); a particular run-time object structure embodies what we may call an instance of these relations — how the relations hold between members of a certain set of objects. The relations modeled by the software may be simple even if their instances for a particular set of objects are complex. Someone who considers the basic idea behind the relation "loves" fairly simple might find the instance of the relation for a particular group of people — the record of who loves whom — hopelessly entangled.

So it is often impossible to prevent the run-time object structures of our O-O systems from becoming big (involving large numbers of objects) and complex (involving many references with a convoluted structure). A good software development environment will provide tools that help explore object structures for testing and debugging.

Such run-time complexity does not have to affect the static picture. We should try to keep the software itself — the set of classes and their relations — as simple as possible.

The observation that simple models can have complex instances is in part a reflection on the power of computers. A small software text can describe huge computations; a simple O-O system can at execution time yield millions of objects connected by many references. A cardinal goal of software engineering is to keep the software simple even when its instances are not.

8.2 OBJECTS AS A MODELING TOOL

We can use the techniques introduced so far to improve our understanding of the method's modeling power. It is important in particular two clarify two aspects: the various worlds touched by software development; and the relationship of our software to external reality.

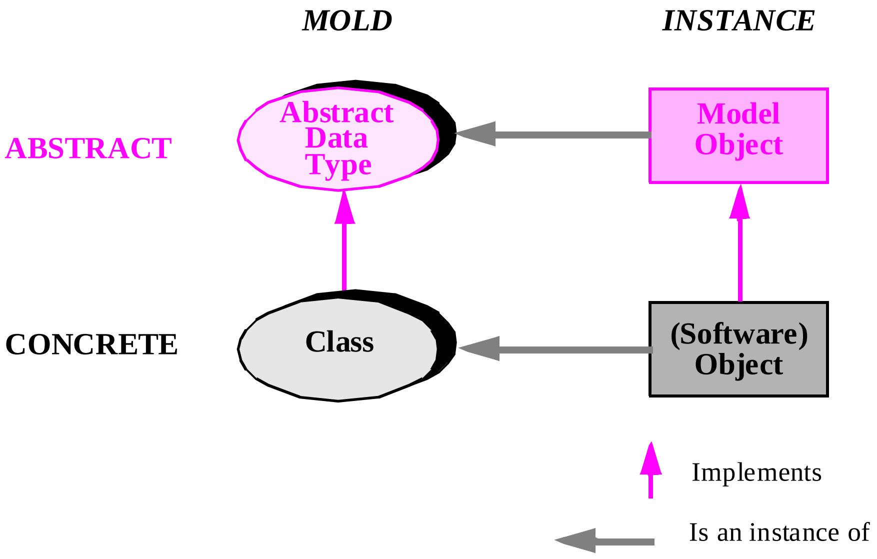
The four worlds of software development

From the preceding discussions it appears that when talking about object-oriented software development we should distinguish between four separate worlds:

- The modeled system, also known as the external system (as opposed to the software system) and described through object types and their abstract relations.
- A particular instantiation of the external system, made of objects between which relations may hold.
- The software system, made of classes connected by the relations of the object-oriented method (client and inheritance).
- An object structure, as may exist during the execution of the software system, made of software objects connected through references.

The following picture suggests the mappings that exist between these worlds.

Molds and their instances



On both the software level (lower part of the picture) and the external level (higher part) it is important to distinguish between the general notions (classes and abstract relations, appearing on the left) and their specific instances (objects and relation instances, appearing on the right). This point has already been emphasized in the previous chapter's discussion of the comparative role of classes and objects. It also applies to relations: we must distinguish between the abstract relation *loved_one* and the set of *loved_one* links that exist between the elements of a certain set of objects.

This distinction is emphasized neither by the standard mathematical definitions of relations nor, in the software field, by the theory of relational databases. Limiting ourselves to binary relations, a relation is defined in both mathematics and relational databases as a set of pairs, all of the form $<x, y>$ where every x is a member a given set TX

and every *y* is a member of a given set *TY*. (In software terminology: all *x* are of type *TX* and all *y* are of type *TY*.) Appropriate as such definitions may be mathematically, they are not satisfactory for system modeling, as they fail to make the distinction between an abstract relation and one of its particular instances. For system modeling, if not for mathematics and relational databases, the *loves* relation has its own general and abstract properties, quite independent of the record of who loves whom in a particular group of people at a particular time.

This discussion will be extended in a later chapter when we look at *transformations* on both abstract and concrete objects and give a name to the vertical arrows of the preceding figure: the *abstraction function*.

"The abstraction function", page 376.

Reality: a cousin twice removed

You may have noted how the above discussion (and previous ones on neighboring topics) stayed clear of any reference to the "real world". Instead, the expression used above in reference to what the software represents is simply "the modeled system".

This distinction is not commonly made. Many of the discussions in information modeling talk about "modeling the real world", and similar expressions abound in books about O-O analysis. So we should take a moment to reflect on this notion. Talking about the "reality" behind a software system is deceptive for at least four reasons.

First, reality is in the eyes of the beholder. Without being accused of undue chauvinism for his profession, a software engineer may with some justification ask his customers why *their* systems are more real than his. Take a program that performs mathematical computations — proving the four-color conjecture in graph theory, integrating some differential equations, or solving geometrical problems in a four-dimensional Riemann surface. Are we, the software developers, to quarrel with our mathematician friends (and customers) as to whose artefacts are more real: a piece of software written in some programming language, or a complete subspace with negative curvature?

Second, the notion of real world collapses in the not infrequent case of software that solves software problems — reflexive applications, as they are sometimes called. Take a C compiler written in Pascal. The "real" objects that it processes are C programs. Why should we consider these programs more real than the compiler itself? The same observation applies to other systems handling objects that only exist in a computer: an editor, a CASE tool, even a document processing system (since the documents it manipulates are computer objects, the printed version being only their final form).

The third reason is a generalization of the second. In the early days of computers, it may have been legitimate to think of software systems as being superimposed on a pre-existing, independent reality. But today the computers and their software are more and more a part of that reality. Like a quantum physicist finding himself unable to separate the measure from the measurement, we can seldom treat "the real world" and "the software" as independent entities. The MIS field (Management Information Systems, that is to say, business data processing) provides some of the most vivid evidence: although it may have

See also "DISCUSSION", 20.6, page 693 on the dangers of staying too *close to reality.*

been the case with the first MIS applications, a few decades ago, that companies introduced computers and the associated software simply with the aim of automating existing procedures, the situation today is radically different, as many existing procedures already involve computers and their software. To describe the operations of a modern bank is to describe mechanisms of which software is a fundamental component. The same is true of most other application areas; many of the activities of physicists and other natural scientists, for example, rely on computers and software not as auxiliary tools but as a fundamental part of the operational process. One may reflect here about the expression "virtual reality", and its implication that what software produces is no less real than what comes from the outside world. In all such cases the software is not disjoint from the reality, as if we had a feedback loop in which operating the software injects some new and important inputs into the model.

The last reason is even more fundamental. A software system is not a model of reality; it is at best a model of a model of some part of some reality. A hospital's patient monitoring system is not a model of the hospital, but the implementation of someone's view of how certain aspects of the hospital management should be handled — a *model* of a *model* of a *subset* of the hospital's reality. An astronomy program is not a model of the universe; it is a software model of someone's model of some properties of some part of the universe. A financial information system is not a model of the stock exchange; it is a software transposition of a model devised by a certain company to describe those aspects of the stock exchange which are relevant to the company's goals.

See "BEYOND SOFTWARE", 6.6, page 147.

The general theme of the object-oriented method, abstract data types, helps understand why we do not need to delude ourselves with the flattering but illusory notion that we deal with the real world. The first step to object orientation, as expressed by the ADT theory, is to toss out reality in favor of something less grandiose but more palatable: a set of abstractions characterized by the operations available to clients, and their formal properties. (This gave the ADT modeler's motto — tell me not what you are but what you have.) Never do we make any pretense that these are the only possible operations and properties: we choose the ones that serve our purposes of the moment, and reject the others. *To model is to discard*.

To a software system, the reality that it addresses is, at best, a cousin twice removed.

8.3 MANIPULATING OBJECTS AND REFERENCES

Let us come back to more mundane matters and see how our software systems are going to deal with objects so as to create and use flexible data structures.

Dynamic creation and reattachment

What the description of the run-time object structure has not yet shown is the highly dynamic nature of a true object-oriented model. As opposed to static and stack-oriented policies of object management, illustrated at the programming language level by Fortran and Pascal respectively, the policy in a proper O-O environment is to let systems create objects as needed at run time, according to a pattern which is usually impossible to predict by a mere static examination of the software text.

From an initial state in which (as described in the previous chapter) only one object has been created — the root object — a system will repetitively perform such operations on the object structure as creating a new object, attach a previously void reference to an object, make a reference void, or reattach a previously attached reference to a different object. The dynamic and unpredictable nature of these operations is part of the reason for the flexibility of the approach, and its ability to support the dynamic data structures that are necessary if we are to use advanced algorithms and model the fast-changing properties of many external systems.

The next sections explore the mechanisms needed to create objects and manipulate their fields, in particular references.

The creation instruction

Let us see how to create an instance of a class such as *BOOK3*. This can only be done by a routine of a class which is a client of *BOOK3*, such as

```
class QUOTATION feature
    source: BOOK3
    page: INTEGER
    make_book is
            -- Create a BOOK3 object and attach source to it.
        do
            … See below …
        end
end
```

which might serve to describe a quotation of a book, appearing in another publication and identified by two fields: a reference to the quoted book and the number of the page which quotes the book.

The (soon to be explained) mechanism that creates an instance of type *QUOTATION* will also by default initialize all its fields. An important part of the default initialization rule is that any reference field, such as the one associated with attribute *source*, will be initialized to a void reference. In other words, creating an object of type *QUOTATION* does not by itself create an object of type *BOOK3*.

The general rule is indeed that, unless you do something to it, a reference remains void. To change this, you may create a new object through a creation instruction. This can be done by procedure *make_book*, which should then read as follows:

```
make_book is
        -- Create a BOOK3 object and attach source to it.
    do
        !! source
    end
```

This illustrates the simplest form of the creation instruction: *!! x*, where *x* is an attribute of the enclosing class or (as will be seen later) a local entity of the enclosing routine. We will see a few extensions to this basic notation later.

The symbol ! is usually read aloud as "bang", so that !! is "bang bang". The entity *x* named in the instruction (*source* in the above example) is called the **target** of the creation instruction.

This form of the creation instruction is known as a "basic creation instruction". (Another form, involving a call to a procedure of the class, will appear shortly.) Here is the precise effect of a basic creation instruction:

Effect of a basic creation instruction

The effect of a creation instruction of the form *!! x*, where the type of the target *x* is a reference type based on a class *C*, is to execute the following three steps:

C1 • Create a new instance of *C* (made of a collection of fields, one for each attribute of *C*). Let OC be the new instance.

C2 • Initialize each field of OC according to the standard default values.

C3 • Attach the value of *x* (a reference) to OC.

The "standard default values" mentioned in step C2 appear in the next box.

Step C1 will create an instance of *C*. Step C2 will set the values of each field to a predetermined value, which depends on the type of the corresponding attribute. Here are these values:

Default initialization values

For a reference, the default value is a void reference.

For a *BOOLEAN*, the default value is *False*.

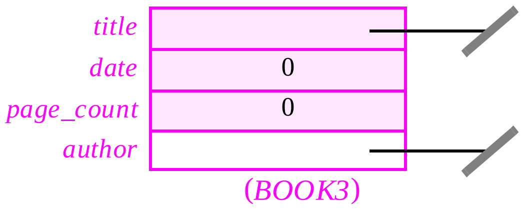
For a *CHARACTER*, the default value is the null character.

For a number (of type *INTEGER*, *REAL* or *DOUBLE*), the default value is zero (that is to say, the zero value of the appropriate type).

So for a target *source* of type *BOOK3*, where the above class declaration read

```
class BOOK3 feature
    title: STRING
    date, page_count: INTEGER
    author: WRITER
end
```

the creation instruction !! *source*, executed as part of a call to procedure *make_book* of class *QUOTATION*, will yield an object of the following form:



A newly created and initialized object

The integer fields have been initialized to zero. The reference field for *author* has been initialized to a void reference. The field for *title*, a *STRING*, also shows a void reference. This is because type *STRING* (of which the above initialization rules said nothing) is in fact a reference type too, although as noted we may for most practical purposes treat it as a basic type.

"STRINGS", 13.5, page 456.

The global picture

It is important not to lose track of the order in which things happen. For the above instance of *BOOK3* to be created, the following two events must occur:

B1 • An instance of *QUOTATION* gets created. Let Q_OBJ be that instance and let *a* be an entity whose value is a reference attached to Q_OBJ.

B2 • Some time after step B1, a call of the form *a*.*make_book* executes procedure *make_book* with Q_OBJ as its target.

It is legitimate of course to ask how we ever get to step B1 — how Q_OBJ itself will be created. This only pushes the problem further. But by now you know the answer to this question: it all comes back to the Big Bang. To execute a system, you must provide a root class and the name of a procedure of that class, the creation procedure. At the start of the execution, you are automatically provided with one object, the root object — an instance of the root class. The root object is the only one that does not need to be created by the software text itself; it comes from the outside, as an *objectus ex machina*. Starting with that one providential object, the software can now create other objects in the normal way, through routines that execute creation instructions. The first routine to be executed is the creation procedure, automatically applied to the root object; in all but the most trivial cases it will include at least one creation instruction so as to start what the previous chapter compared to a giant firework: the process of producing as many new objects as a particular execution will need.

See "PUTTING EVERYTHING TOGETHER", 7.9, page 194.

Why explicit creation?

Object creation is explicit. Declaring an entity such as

```
b: BOOK3
```

does not cause an object to be created at run time: creation will only occur when some element of the system executes an operation

```
!! b
```

You may have wondered why this was so. Should the declaration of *b* not be sufficient if we need an object at run time? What good is it to declare an entity if we do not create an object?

A moment's reflection, however, shows that the distinction between declaration and creation is actually the only reasonable solution.

The first argument is by *reductio ad absurdum*. Assume that somehow we start processing the declaration of *b* and immediately create the corresponding book object. But this object is an instance of class *BOOK3*, which has an attribute *author*, itself of a reference type *WRITER*, so that the *author* field is a reference, for which we must create an object right away. Now this object has reference fields (remember that *STRING* is in fact a reference type) and they will require the same treatment: we are starting on a long path of recursive object creation before we have even begun any useful processing!

This argument would be even more obvious with a self-referential class, such as *PERSON1* seen above:

```
class PERSON1 feature
    name: STRING
    loved_one, landlord: PERSON1
end
```

Treating every declaration as yielding an object would mean that every creation of an instance of *PERSON1* would cause creation of two more such objects (corresponding to *loved_one* and *landlord*), entering into an infinite loop. Yet we have seen that such self-referential definitions, either direct as here or indirect, are common and necessary.

Another argument simply follows from a theme that runs through this chapter: the use of object technology as a powerful modeling technique. If every reference field were initialized to a newly created object, we would have room neither for void references nor for multiple references attached to a single object. Both are needed for realistic modeling of practical systems:

- In some cases the model may require that a certain reference be left not attached to any object. We used this technique when leaving the *author* field void to indicate that a book is by an unknown author.

See the figure on page 226.

- In other cases two references should be attached, again for conceptual reasons coming from the model, to the same object. In the self-reference example we saw the *loved_one* fields of two *PERSON1* instances attached to the same object. It would not make sense in that case to create an object for each field on creation; what you need is, rather than a creation instruction, an assignment operation (studied later in this chapter) that attaches a reference to an already existing object. This observation applies even more clearly to the self-referential field from the same example (field *landlord* for the top object).

The object management mechanism never attaches a reference implicitly. It creates objects through creation instructions (or *clone* operations, seen below and explicit too), initializing their reference fields to void references; only through explicit instructions will these fields, in turn, become attached to objects.

In the discussion of inheritance we will see that a creation instruction may use the syntax ! *T* ! *x* to create an object whose type *T* is a descendant of the type declared for *x*.

"Polymorphic creation", page 479.

8.4 CREATION PROCEDURES

All the creation instructions seen so far relied on default initializations. In some cases, you may be unhappy with the language-defined initializations, wanting instead to provide specific information to initialize the created object. Creation procedures address this need.

Overriding the default initializations

To use an initialization other than the default, give the class one or more creation procedures. A creation procedure is a procedure of the class, which is listed in a clause starting with the keyword **creation** at the beginning of the class, before the first feature clause. The scheme is this:

```
indexing
        …
class C creation
        p1, p2, …
feature
        … Feature declarations, including declarations for procedures p1, p2, …
end
```

A style suggestion: the recommended name for creation procedures in simple cases is *make*, for a class that has only one creation procedure; for a class that has two or more creation procedures it is generally desirable to give them a name starting with *make_* and continuing with some qualifying word, as in the *POINT* example that follows.

"CHOOSING THE RIGHT NAMES", 26.2, page 879.

The corresponding creation instruction is not just !! *x* any more, but of the form

!! *x*.*p* (…)

where *p* is one of the creation procedures listed in the **creation** clause, and (…) is a valid actual argument list for *p*. The effect of such an instruction is to create the object using the default values as in the earlier form, and to apply *p*, with the given arguments, to the result. The instruction is called a **creation call**; it is a combination of creation instruction and procedure call.

We can for example add creation procedures to the class *POINT* to enable clients to specify initial coordinates, either cartesian or polar, when they create a point object. We will have two creation procedures, *make_cartesian* and *make_polar*. Here is the scheme:

Original version of POINT in "The class", page 176.

```
class POINT1 creation
    make_cartesian, make_polar
feature
    … The features studied in the preceding version of the class:
      x, y, ro, theta, translate, scale, …
feature {NONE} -- See explanations below about this export status.
    make_cartesian (a, b: REAL) is
                -- Initialize point with cartesian coordinates a and b.
            do
                x := a; y := b
            end
    make_polar (r, t: REAL) is
                -- Initialize point with polar coordinates r and t.
            do
                x := r * cos (t); y := r * sin (t)
            end
end -- class POINT1
```

With this class text, a client will create a point through such instructions as

```
!! my_point.make_cartesian (0, 1)
!! my_point.make_polar (1, Pi/2)
```

both having the same effect if *Pi* has the value suggested by its name.

Here is the rule defining the effect of such creation calls. The first three steps are the same as for the basic form seen earlier:

Effect of a creation call

The effect of a creation call of the form !! *x*.*p* (…), where the type of the target *x* is a reference type based on a class *C*, *p* is a creation procedure of class *C*, and (…) represents a valid list of actual arguments for this procedure if necessary, is to execute the following four steps:

C1 • Create a new instance of *C* (made of a collection of fields, one for each attribute of *C*). Let OC be the new instance.

C2 • Initialize each field of OC according to standard default values.

C3 • Attach the value of *x* (a reference) to OC.

The new step → C4 • Call procedure *p*, with the arguments given, on OC.

The export status of creation procedures

In *POINT1* the two creation procedures have been declared in a feature clause starting with **feature** {*NONE*}. This means they are secret, but only for normal calls, not for creation calls. So the two example creation calls just seen are valid; normal calls of the form *my_point*.*make_cartesian* (*0*, *1*) or *my_point*.*make_polar* (*1*, *Pi*/*2*) are invalid since the features have not been made available for calling by any client.

On the {NONE} construct see "Style for declaring secret features", page 192.

The decision to make the two procedures secret means we do not want clients, once a point object exists, to set their coordinates directly, although they may set them indirectly through the other procedures of the class such as *translate* and *scale*. Of course this is only one possible policy; you may very well decide to export *make_cartesian* and *make_polar* in addition to making them creation procedures.

It is possible to give a procedure a selective creation status as well by including a set of classes in braces in its **creation** clause, as in

```
class C creation {A, B, …}
        p1, p2,
…
```

although this is less frequent than limiting the export status of a feature through the similar syntax **feature** {*A*, *B*, …} or **feature** {*NONE*}. Remember in any case that the creation status of a procedure is independent of its call export status.

Rules on creation procedures

The two forms of creation instructions, the basic form !! *x* and the creation call !! *x*.*p* (…), are mutually exclusive. As soon as a class has a **creation** clause, then only the creation call is permitted; the basic form will be considered invalid and rejected by the compiler.

This convention may seem strange at first, but is justified by considerations of object consistency. An object is not just a collection of fields; it is the implementation of an abstract data type, which may impose consistency constraints on the fields. Here is a typical example. Assume an object representing a person, with a field for the birth year and another for the age. Then you cannot set these two fields independently to arbitrary values, but must ensure a consistency constraint: the sum of the age field and the birth year field must equal either the current year or the one before. (In a later chapter we will learn how to express such constraints, often reflecting axioms from the underlying ADT, as **class invariants**.) A creation instruction must *always* yield a consistent object. The basic form of the creation instruction — !! *x* with no call — is only acceptable if setting all the fields to the default values yields a consistent object. If this is not the case, you will need creation procedures, and should disallow the basic form of the creation instruction.

See "CLASS INVARIANTS", 11.8, page 364, in particular "The role of creation procedures", page 372.

In some infrequent cases you may want to accept the default initializations (as they satisfy the class invariant) while also defining one or more creation procedures. The technique to apply in this case is to list *nothing* among the creation procedures. Feature *nothing* is a procedure without arguments, inherited from the universal class *ANY*, which has an empty body (the feature declaration is simply: *nothing* **is do end**) so that it does exactly what the name indicates. Then you can write:

```
class C creation
        nothing, some_creation_procedure, some_other_creation_procedure…
feature
        …
```

Although the form !! *x* is still invalid in this case, clients can achieve the intended effect by writing the instruction as !! *x*.*nothing*

Finally, note that as a special case the rule on creation instructions gives a way to define a class that *no client* will be permitted to instantiate. A class declaration of the form

```
class C creation
        -- There is nothing here!
feature
        … Rest of class text …
end
```

has a creation clause — an empty one. The above rule states that if there is a **creation** clause the only permitted creation instructions are creation calls using a creation procedure; here, since there are no creation procedures, no creation call is permitted.

Being able to disallow class instantiation is of little interest if we limit ourselves to the object-oriented mechanisms seen so far. But when we move on to inheritance this little facility may prove handy if we want to specify that a certain class should only be used as ancestor to other classes, never directly to create objects.

See "What to do with deferred classes", page 487 and exercise E14.5, page 519.

Another way to achieve this is to make the class *deferred*, but a deferred class must have at least one deferred feature, and we will not always have a role for such a feature.

Multiple creation and overloading

In advance of the discussion section, it is illuminating to compare the mechanism of multiple creation procedures with the C++/Java approach. The need is universal: providing several ways to initialize an object on creation. C++ and Java, however, rely on a different technique, name overloading.

In these languages all the creation procedures of a class (its "constructors") have the same name, which is in fact the class name; if a class *POINT* contains a constructor with two real arguments corresponding to *make_cartesian*, the expression **new** *POINT* (*0*, *1*) will create a new instance. To differentiate between two constructors, the languages rely on the signatures (the types of the arguments).

See "Syntactic overloading", page 93.

The problem is of course, as we saw in the discussion of overloading, that the argument signature is not the appropriate criterion: if we also want a constructor providing the equivalent of *make_polar* we are stuck, since the arguments would be the same, two real numbers. This is the general problem of overloading: using the same name for different operations, thereby causing potential ambiguity — compounded here by the use of that name as a class name as well as a procedure name.

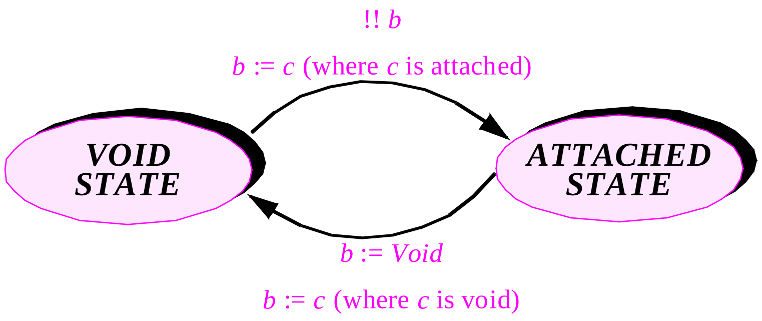
The technique developed earlier seems preferable in all respects: minimum hassle (no creation procedure) if default initializations suffice; prevent creation, if desired, through an empty **creation** clause; to provide several forms of creation, define as many creation procedures as needed; do not introduce any confusion between class names and feature names; let the effect of every operation stand out clearly from its names, as with *make_polar*.

8.5 MORE ON REFERENCES

The run-time model gives an important role to references. Let us examine some of their properties, in particular the notion of void reference, and some of the issues they raise.

States of a reference

A reference may be in either of two states: void and attached. We have seen that a reference is always void initially and can be come attached through creation. Here is a more complete picture.


!! *b*
b := *c* (where *c* is attached)
VOID STATE
ATTACHED STATE
b := *Void*
b := *c* (where *c* is void)

The possible states of a reference

Other than creation, a reference may change state through assignment, as will be studied shortly. For the moment, please make sure you understand the difference between the three notions — object, reference and entity — which recur through this chapter:

- "Object" is a run-time notion; any object is an instance of a certain class, created at execution time and made of a number of fields.
- "Reference" is also a run-time notion: a reference is a value that is either void or attached to an object. We have seen a precise definition of "attached": a reference is attached to an object if it identifies that object unambiguously.
- In contrast, "entity" is a static notion — that is to say, applying to the software text. An entity is an identifier appearing in the text of a class, and representing a run-time value or a set of successive run-time values. (Readers used to traditional forms of software development may think of the notion of entity as covering variables, symbolic constants, routine arguments and function results.)

Full definition of "entity": page 213.

If *b* is an entity of reference type, its run-time value is a reference, which may be attached to an object O. By an abuse of language we can say that *b* itself is attached to O.

Void references and calls

In most situations we expect a reference to be attached to an object, but the rules also permit a reference to be void. Void references play an important role — if only by making a nuisance of themselves — in the object-oriented model of computation. As discussed extensively in the previous chapter, the fundamental operation in that model is feature call: apply to an instance of a class a feature of that class. This is written

```
some_entity.some_feature (arg1, …)
```

where *some_entity* is attached to the desired target object. For the call to work, *some_entity* must indeed be attached to an object. If *some_entity* is of a reference type and happens to have a void value at the time of the call, the call cannot proceed, as *some_feature* needs a target object.

See chapter 12, in particular "Sources of exceptions", page 412.

To be correct, an object-oriented system must never attempt at run time to execute a feature call whose target is void. The effect will be an **exception**; the notion of exception, and the description of how it is possible to recover from an exception, will be discussed in a later chapter.

It would be desirable to let compilers check the text of a system to guarantee that no such event will occur at run time, in the same way that they can check the absence of type incompatibilities by enforcing type rules. Unfortunately such a general goal is currently beyond the reach of compilers (unless we place unacceptable restrictions on the language). So it remains the software developer's responsibility to ensure that the execution will never attempt a feature call on a void target. There is of course an easy way to do so: always write *x.f* (…) as

The test "x is not void" may be written simply as x /= Void. See

```
if "x is not void" then
    x.f (…)
else
    …
end
```

but this is too unwieldy to be acceptable as a universal requirement. Sometimes (as when a call *x.f* immediately follows a creation !! *x*) it is clear from the context that *x* is not void, and you do not want to test.

The question of non-vacuity of references is part of the larger question of software correctness. To prove a system correct, it is necessary to prove that no call is ever applied to a void reference, and that all the software's assertions (as studied in a later chapter) are satisfied at the appropriate run-time instants. For non-vacuity as well as for assertion correctness, it would be desirable to have an automatic mechanism (a program prover, either integrated with the compiler or designed as a separate software tool) to ascertain that a software system is correct. In the absence of such tools, the result of a violation is a run-time error — an exception. Developers may protect their software against such situations in two ways:

- When writing the software, trying to prevent the erroneous situations from arising at run time, using all means possible: systematic and careful development, class inspections, use of tools that perform at least partial checks.
- If any doubt remains and run-time failures are unacceptable, equipping the software with provisions for handling exceptions.

8.6 OPERATIONS ON REFERENCES

We have seen one way of changing the value of a reference *x*: using a creation instruction of the form !! *x*, which creates a new object and attaches *x* to it. A number of other interesting operations are available on references.

Attaching a reference to an object

So far the classes of this chapter have had attributes but no routines. As noted, this makes them essentially useless: it is not possible to change any field in an existing object. We need ways to modify the value of references, without resorting to instructions of the Pascal-C-Java-C++ form *my_beloved*.*loved_one* := *me* (to set the *loved_one* field of an object directly), which violates information hiding and is syntactically illegal in our notation.

To modify fields of foreign objects, a routine will need to call other routines that the authors of the corresponding classes have specifically designed for that purpose. Let us adapt class *PERSON1* to include such a procedure, which will change the *loved_one* field to attach it to a new object. Here is the result:

```
class PERSON2 feature
    name: STRING

    loved_one, landlord: PERSON2

    set_loved (l: PERSON2) is
            -- Attach the loved_one field of current object to l.
        do
            loved_one := l
        end
end
```

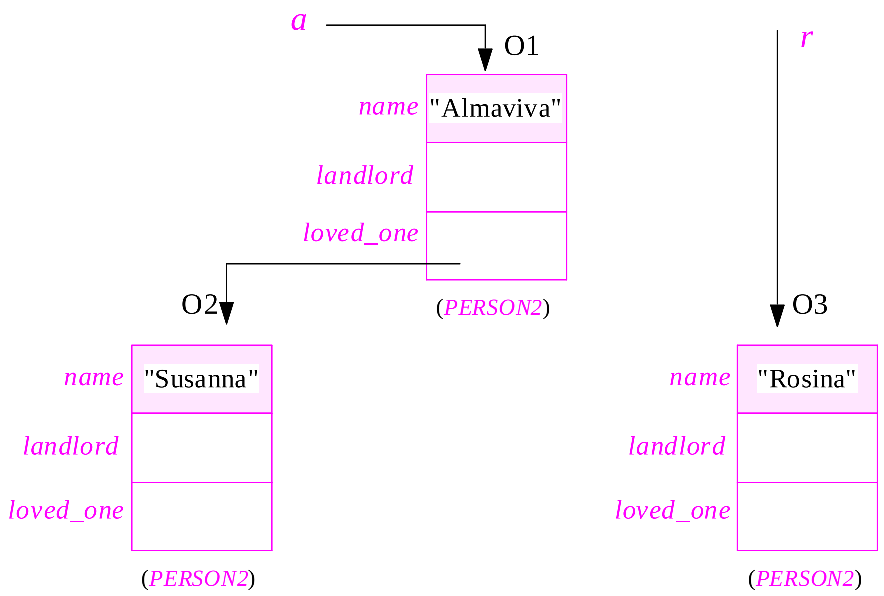
Procedure *set_loved* assigns to the *loved_one* field of the current instance of *PERSON2*, a reference field, the value of another reference, *l*. Reference assignments (like assignments of simple values such as integers) rely on the := symbol, with the assignment's source on the right and the target on the left. In this case, since both source and target are of reference types, the assignment is said to be a reference assignment.

The effect of a reference assignment is exactly what the name suggests: the target reference gets reattached to the object to which the source reference is attached — or becomes void if the source was void. Assume for example that we start with the situation shown at the top of the facing page; to avoid cluttering the picture, the *landlord* fields and the irrelevant *loved_one* fields have been left blank.

Assume that we execute the procedure call

```
a.set_loved (r)
```

Before reference assignment

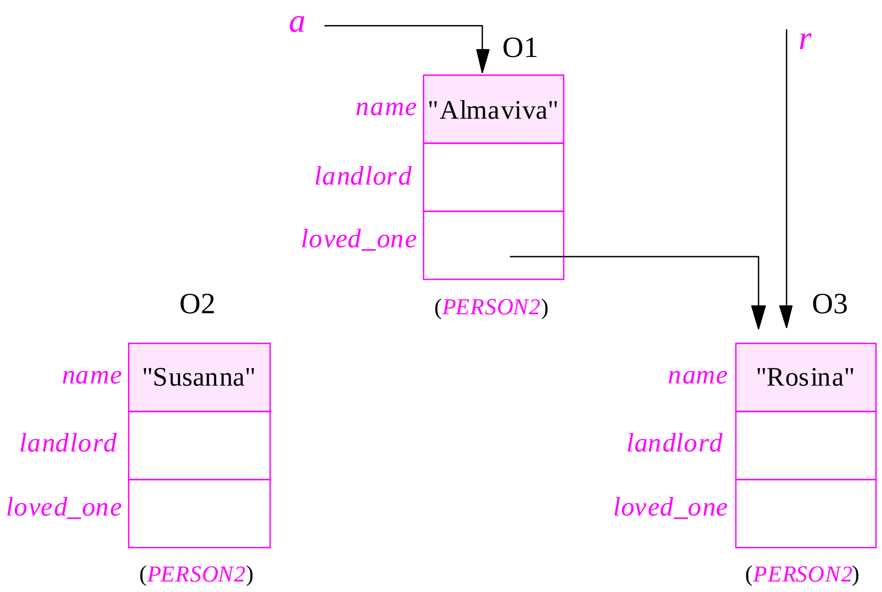


where *a* is attached to the top object (O1) and *r* to the bottom-right object (O3). From the way *set_loved* has been written, this will execute the assignment

loved_one := *l*

with O1 as the current object and *l* having the same value as *r*, a reference to O3. The result is to reattach the *loved_one* field of O1:

After reference assignment



If *r* had been a void reference, the assignment would have made the *loved_one* field of O1 void too.

A natural question at this stage is: what happens to the object to which the modified field was initially attached — O2 in the figure? Will the space it occupies be automatically recycled for use by future creation instructions?

This question turns out to be so important as to deserve a chapter of its own — the next chapter, on memory management and garbage collection. So please hold your breath until then. But it is not too early for a basic observation: regardless of the final answer, a policy that would always recycle the object's space would be incorrect. In the absence of further information about the system from which the above run-time structure is extracted, we do not know whether some other reference is still attached to O2. So a reference assignment by itself does not tell us what to do with the previously attached object; any mechanism for recycling objects will need more context.

Reference comparison

In the same way that we have an operation (the := assignment) to attach a reference to an object, we need a way to test whether two references are attached to the same object. This is simply provided by the usual equality operator =.

If *x* and *y* are entities of reference types, the expression

x = *y*

is true if and only if the corresponding references are either both void or both attached to the same objects. The opposite operator, "not equal", is written /= (a notation borrowed from Ada).

For example, the expression

r = *a*•*loved_one*

has value true on the last figure, where both sides of the = sign denote references attached to the object O3, but not on the next-to-last figure, where *a*•*loved_one* is attached to O2 and *r* is attached to O3.

In the same way that an assignment to a reference is a reference operation, not an operation on objects, the expressions *x* = *y* and *x* /= *y* compare references, not objects. So if *x* and *y* are attached to two distinct objects, *x* = *y* has value false even if these objects are field-by-field identical. Operations which compare objects rather than reference will be introduced later.

The void value

Although it is easy to get a void reference — since all reference fields are by default initialized to *Void* –, we will find it convenient to have a name for a reference value accessible in all contexts and known always to be void. The predefined feature

Void

will play that role.

Two common uses of *Void* are to test whether a certain reference is void, as in

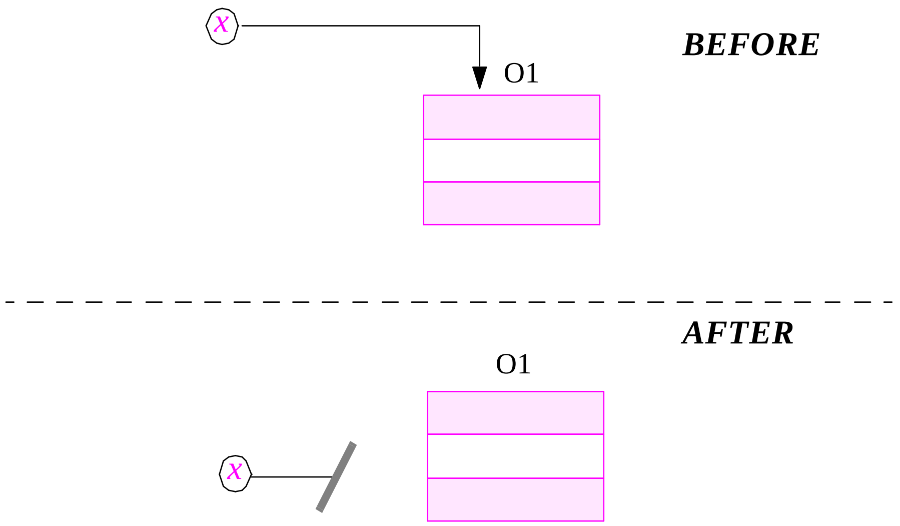
if *x* = *Void* **then** …

and to make a reference void, using the assignment

x := *Void*

This last assignment has the effect of putting the reference back to the void state, and so of de-attaching it from the attached object, if any:

De-attaching a reference from an object



The comment made in the general discussion of reference assignment is worth repeating here: the assignment of *Void* to *x* has no immediate effect on the attached object (O1 in the figure); it simply cuts the link between the reference and the object. It would be incorrect to understand it as freeing the memory associated with O1, since some other reference may still be attached to O1 even after *x* has been de-attached from it. See the discussion of memory management in the next chapter.

Object cloning and equality

Reference assignments may cause two or more references to become attached to a single object. Sometimes you will need a different form of assignment, which works on the object itself: rather than attaching a reference to an existing object, you will want to create a new copy of an existing object.

This goal is achieved through a call to a function called *clone*. If *y* is attached to an object OY, the expression

clone (*y*)

denotes a new object OX, such that OX has the same number of fields as OY, each field of OX being identical to the corresponding field of OY. If *y* is void, the value of *clone* (*y*) is also void.

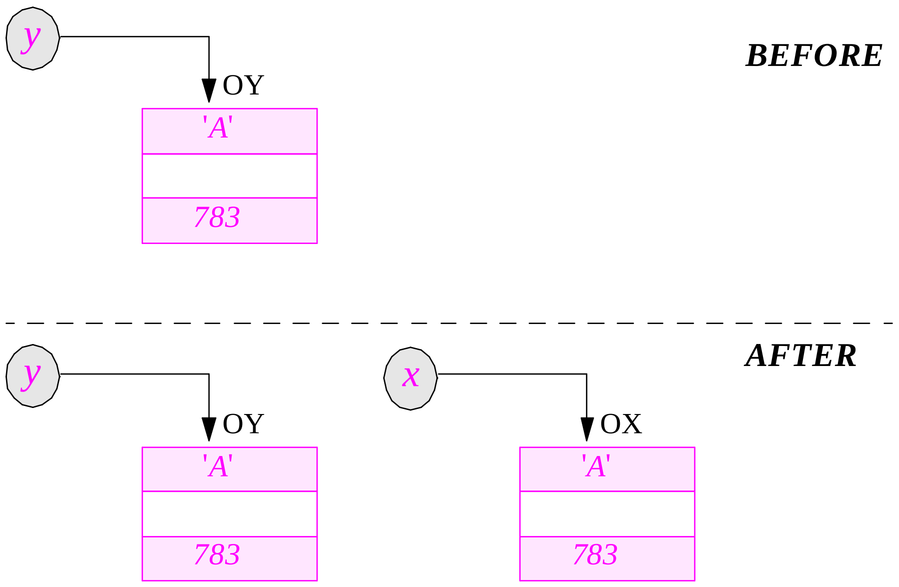
To duplicate the object attached to *y* and attach the resulting object to *x* (or make *x* void if *y* is void), you may use a call to *clone* in an assignment:

[1]

x := *clone* (*y*)

Here is an illustration of this mechanism.

Cloning an object



We similarly need a mechanism to compare two objects. The expression *x* = *y*, as noted, fulfills another purpose: comparing references. For objects, we will use function *equal*. The call

equal (*x*, *y*)

returns a boolean value, true if and only if *x* and *y* are either both void, or attached to two objects whose corresponding fields have the same values. If a system executes the clone assignment [1], the state immediately following that assignment will satisfy *equal* (*x*, *y*).

You may wonder why function *clone* has an argument, and *equal* two arguments treated symmetrically, rather than being called under forms closer to the usual object-oriented style, for example *y*.*twin* and *x*.*is_equal* (*y*). The answer appears in the discussion section, but it is not too early to try to guess it.

"The form of clone and equality operations", page 274.

Object copying

Function *clone* creates a new object as a carbon copy of an existing one. Sometimes the target object already exists; all we want to do is to overwrite its fields. Procedure *copy* achieves this. It is called through the instruction

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

for *x* and *y* of the same type; its effect is to copy the fields of the object attached to *y* onto the corresponding ones of the object attached to *x*.

As with all feature calls, any call to *copy* requires the target *x* to be non-void. In addition, *y* must also be non-void. This inability to deal with void values distinguishes *copy* from *clone*.

See chapter 11 about assertions.

The requirement that *y* must be non-void is so important that we should have a way to express it formally. The problem is in fact more general: how a routine can state the **preconditions** on the arguments passed by its callers. Such preconditions, a case of the more general notion of assertion, will be discussed in detail in a later chapter. Similarly, we will learn to express as **postconditions** such fundamental semantic properties as the observation made above that the result of a *clone* will satisfy *equal*.

Procedure *copy* may be considered more fundamental than function *clone* in the sense that we can, at least for a class with no creation procedure, express *clone* in terms of *copy* through the following equivalent function:

```
clone (y: SOME_TYPE) is
        -- Void if y is void; otherwise duplicate of object attached to y
    do
        if y /= Void then
            !! Result           -- Valid only in the absence of creation procedures
            Result.copy (y)
        end
    end
```

On execution of a function call, *Result* is automatically initialized using the same rules defined above for attributes. This is the reason why the **if** needs no **else**: since *Result* is initialized to *Void*, the result of the above function is a void value if *y* is void.

Deep clone and comparison

The form of copy and comparison achieved by routines *clone*, *equal* and *copy* may be called **shallow** since these operations work on an object at the first level only, never trying to follow references. There will also be a need for **deep** variants which recursively duplicate an entire structure.

To understand the differences assume for example that we start with the object structure appearing in black (except for the attribute and class names) under **A** in the figure on the facing page, where the entity *a* is attached to the object labeled O1.

For purposes of comparison, consider first the simple reference assignment

b := *a*

As pictured under **B**, this simply attaches the assignment's target *b* to the same object O1 to which the source *a* was attached. No new object is created.

Next consider the cloning operation

c := *clone* (*a*)

This instruction will, as shown under **C**, create a single new object O4, field-by-field identical to O1. It copies the two reference fields onto the corresponding fields of O4, yielding references that are attached to the same objects O1 and O3 as the originals. But it does not duplicate O3 itself, or any other object other than O1. This is why the basic *clone* operation is known as shallow: it stops at the first level of the object structure.

Note that a self-reference has disappeared: the *landlord* field of O1 was attached to O1 itself. In O4 this field becomes a reference to the original O1.

In other cases, you may want to go further and duplicate a structure recursively, without introducing any sharing of references such as occurred in the creation of O4. The function *deep_clone* achieves this. Instead of stopping at the object attached to *y*, the process of creating *deep_clone* (*y*) recursively follows any reference fields contained in that object and duplicates the entire structure. (If *y* is void the result is void too.) The function will of course process cyclic reference structures properly.

The bottom part of the figure, labeled **D**, illustrates the result of executing

d := *deep_clone* (*a*)

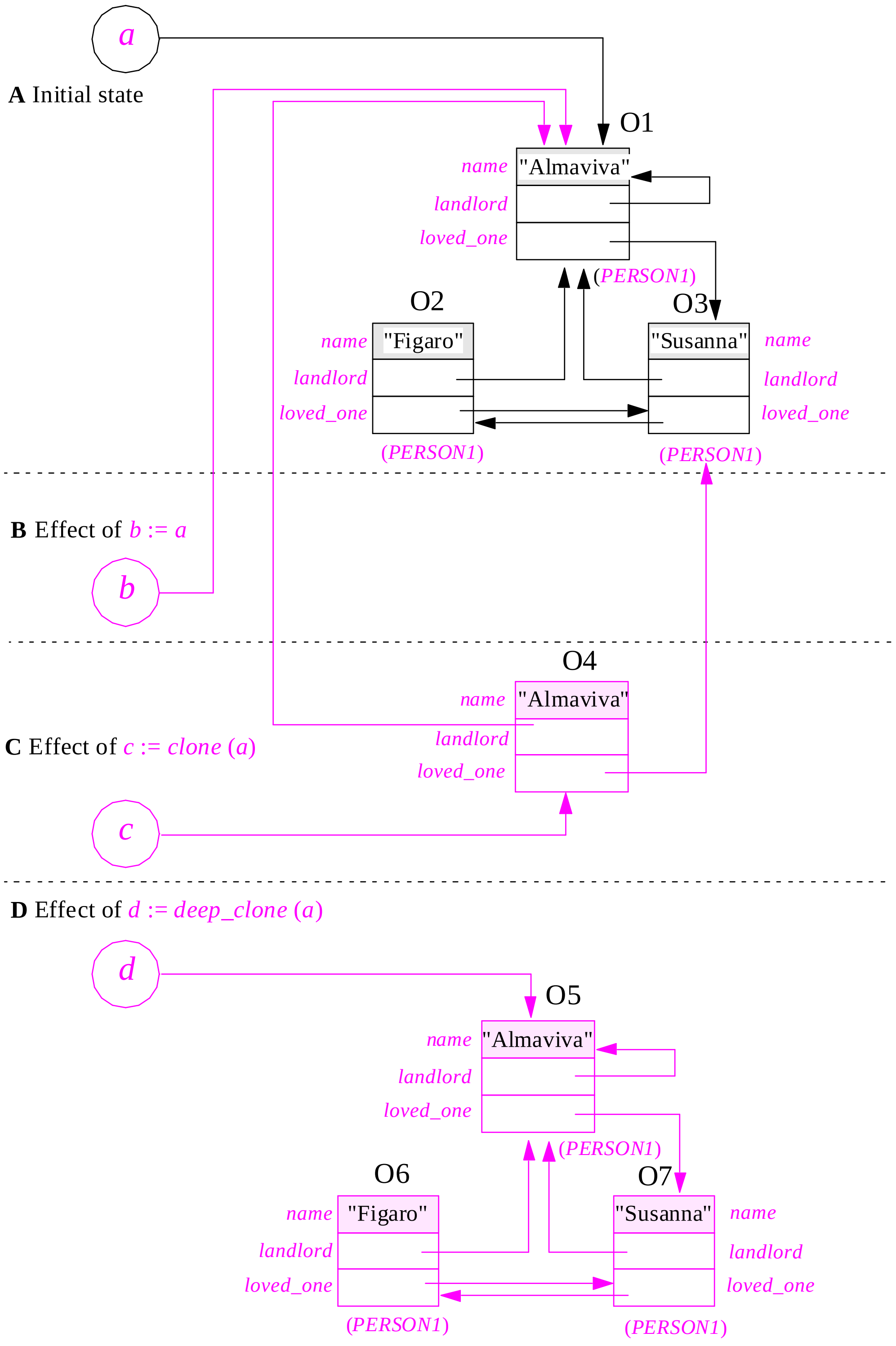
This case introduces no new sharing; all the objects accessible directly or indirectly from O1 (the object attached to *a*) will be duplicated, yielding new objects O5, O6 and O7. There is no connection between the old objects (O1, O2 and O3) and the new. Object O5, mimicking O1, has a self-reference.

In the same way that we need both deep and shallow clone operations, equality must have a deep variant. The *deep_equal* function compares two object structures to determine whether they are structurally identical. In the figure's example, *deep_equal* holds between any two of *a*, *b* and *d*; but whereas *equal* (*a*, *c*) is true, since the corresponding objects O1 and O4 are field-by-field identical, *equal* (*a*, *d*) is false. In fact *equal* does not hold between *d* and any of the other three. (Both *equal* (*a*, *b*) and *equal* (*b*, *c*) hold.) In the general case we may note the following properties:

- After an assignment *x* := *clone* (*y*) or a call *x*.*copy* (*y*), the expression *equal* (*x*, *y*) has value true. (For the first assignment this property holds whether or not *y* is void.)
- After *x* := *deep_clone* (*y*), the expression *deep_equal* (*x*, *y*) has value true.

These properties will be expressed as postconditions of the corresponding routines.

Various forms of assignment and cloning



Deep storage: a first view of persistence

The study of deep copy and equality leads to another mechanism which, in environments where it is available, provides one of the great practical advantages of the O-O method.

So far, the discussion has not examined the question of input and output. But of course an object-oriented system will need to communicate with other systems and with the rest of the world. Since the information it manipulates is in the form of objects, this means it must be able to write and read objects to and from files, databases, communication lines and various devices.

For simplicity this section will assume that the problem is to write to and write from files, and will use the terms "storage" and "retrieval" for these operations ("input" and "output" would also be adequate.) But the mechanisms studied must also be applicable for exchanging objects with the outside world through other means of communication, for example by sending and receiving objects through a network.

For instances of such classes as *POINT* or *BOOK1*, storage and retrieval of objects raise no particular novelty. These classes, used as the first examples at the beginning of this chapter, have attributes of types such as *INTEGER*, *REAL* and *STRING*, for which well-understood external representations are available. Storing an instance of such a class into a file, or retrieving it from that file, is similar to performing an output or input operation on a Pascal record or a C structure. Account must be taken, of course, of the peculiarities of data representations on different machines and in different languages (C, for example, has a special convention for strings, which the language expects to be terminated by a null character); but these are well-known technical problems for which standard solutions exist. So it is reasonable to expect that for such objects a good O-O environment could provide general-purpose procedures, say *read* and *write*, which, in the manner of *clone*, *copy* and consorts, would be available to all classes.

But such mechanisms will not take us very far because they do not handle a major component of the object structure: references. Since references can be represented in memory (as addresses or otherwise) it is possible to find an external representation as well. That is not the difficult part of the problem. What matters is the meaning of these references. A reference attached to an object is worthless without that object.

So as soon as we start dealing with non-trivial objects — objects that contain references — we cannot satisfy ourselves any more with a storage and retrieval mechanism that would just work on individual objects; the mechanism must process, together with an object, all its dependents according to the following definition:

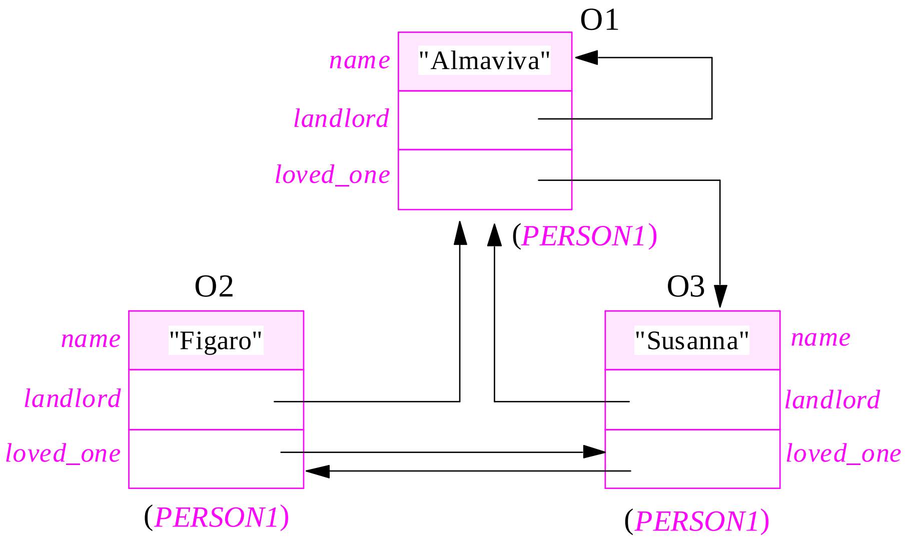
Definition: direct dependents, dependents

The direct dependents of an object are the objects attached to its reference fields, if any.

The dependents of an object are the object itself and (recursively) the dependents of its direct dependents

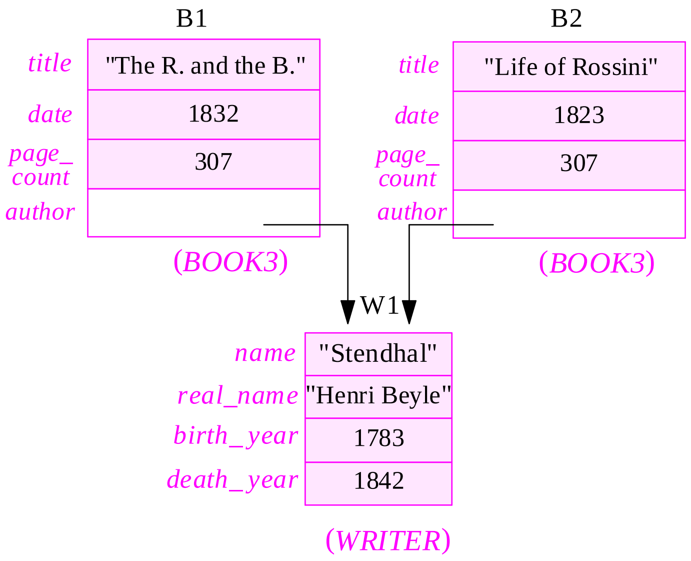
With the object structure shown below (identical to earlier examples), it would be meaningless to store into a file, or transmit over a network, just the object O1. The operation must also include the dependents of O1: O2 and O3.

Three mutually dependent objects



In this example any one of the three objects has the other two as dependents. In the *BOOK3* example reproduced below, we may store W1 by itself, and whenever we store B1 or B2 we must store W1 as well.

"Book" and "Writer" objects



The notion of dependent was implicitly present in the presentation of *deep_equal*. Here is the general rule:

Persistence Closure principle

Whenever a storage mechanism stores an object, it must store with it the dependents of that object. Whenever a retrieval mechanism retrieves a previously stored object, it must also retrieve any dependent of that object that has not yet been retrieved.

The basic mechanism which will achieve this for our purposes is known as the *STORABLE* facility from the name of the Base library class which includes the corresponding features. The basic features of *STORABLE* are of the form:

store (*f*: *IO_MEDIUM*)

retrieved (*f*: *IO_MEDIUM*): *STORABLE*

The effect of a call of the form *x*.*store* (*f*) is to store the object attached to *x*, together with all its dependents, in the file associated with *f*. The object attached to *x* is said to be the **head object** of the stored structure. The generating class of *x* must be a descendant of *STORABLE* (that is to say, it must inherit directly or indirectly from *STORABLE*); so you will have to add *STORABLE* to the list of its parents if it is not already there. This applies only to the generating class of the head object; there is no particular requirement on the generating classes of the dependent objects — fortunately, since a head object can have an arbitrary number of direct and indirect dependents, instances of arbitrary classes.

Class *IO_MEDIUM* is another Base library class, covering not only files but also structures for network transmission. Clearly *f* must be non-void and the attached file or transmission medium must be writable.

The result of a call *retrieved* (*f*) is an object structure recursively identical, in the sense of *deep_clone*, to the complete object structure stored in *f* by an earlier call to *store*. Feature *retrieved* is a function; its result is a reference to the head object of the retrieved structure.

If you have already acquired a basic understanding of inheritance and of the associated type rules, you may have noted that *retrieved* raises a typing problem. The result of this function is of type *STORABLE*; but it seems that its normal use will be in assignments of the form *x* := *retrieved* (*f*) where the type of *x* is a proper descendant of *STORABLE*, not *STORABLE* itself, even though the type rules will permit *x* := *y* only if the type of *y* is a descendant of the type of *x* — not the other way around. The key to this problem will be an important construct, the **assignment attempt**. All this will be examined in detail when we study inheritance and the associated type rules.

See "ASSIGNMENT ATTEMPT", 16.5, page 591.

The *STORABLE* mechanism is our first example of what is known as a **persistence** facility. An object is persistent if it survives individual sessions of the systems that manipulate it. *STORABLE* only provides a partial solution to the persistence problem, suffering from several limitations:

- In the structure stored and retrieved, only one object is known individually: the head object. It may be desirable to retain the identity of other objects too.
- As a consequence, the mechanism is not directly usable to retrieve objects selectively through contents-based or keyword-based queries as in database management systems.
- A call to *retrieved* recreates the entire object structure. This means that you cannot use two or more such calls to retrieve various parts of a structure, unless they are disjoint.

Chapter 31.

To address this problem is to move from a mere persistence mechanism to the notion of object-oriented database, presented in a later chapter, which also discusses a number of issues associated with *STORABLE* and other persistence mechanisms, such as schema evolution (what happens when you retrieve an object and its class has changed?) and persistent object identity.

But the above limitations should not obscure the considerable practical benefits of the *STORABLE* mechanism as described above. In fact one may conjecture that the absence of such a mechanism has been one of the major obstacles to the use of sophisticated data structures in traditional development environments. Without *STORABLE* or its equivalent, storing a data structure becomes a major programming effort: for every kind of structure that you want to endow with persistence properties you must write a special input and output mechanism, including a set of mutually recursive procedures (one for each type) and special-purpose traversal mechanisms (which are particularly tricky to write in the case of possibly cyclic structures). But the worst part is not even the work that you have to do initially: as usual, the real trouble comes when the structure changes and you have to update the procedures.

With *STORABLE* a predefined mechanism is available regardless of your object structure, its complexity, and the software's evolution.

A typical application of the *STORABLE* mechanism is a SAVE facility. Consider an interactive system, for example a text editor, a graphical editor, a drafting program or a computer-aided design system; it needs to provide its users with a SAVE command to store the state of the current session into a file. The information stored should be sufficient to restart the session at any later time, so it must include all the important data structures of the system. Writing such a procedure in an ad hoc fashion suffers from the difficulties mentioned; in particular, you will have to update it whenever you change a class during development. But with the *STORABLE* mechanism and a good choice of head object, you can implement the SAVE facility using a single instruction:

```
head.store (save_file)
```

Just by itself, this mechanism would suffice to recommend an object-oriented environment over its more traditional counterparts.

8.7 COMPOSITE OBJECTS AND EXPANDED TYPES

The preceding discussion described the essentials of the run-time structure. It gives an important role to references. To complete the picture, we must see how to handle values which are *not* references to objects, but the objects themselves.

References are not sufficient

The values considered so far, save for integers, booleans and the like, were references to objects. Two reasons suggest that we may also need entities whose values are objects:

- An important goal announced in the last chapter is to have a completely uniform type system, in which basic types (such as *BOOLEAN* and *INTEGER*) are handled in the same way as developer-defined types (such as *POINT* or *BOOK*). But if you use an entity *n* to manipulate an integer, you will almost always want the value of *n* to be an integer, for example 3, not a reference to an object containing the value 3. The reason is partly efficiency — think of the penalty in both time and space that we would have to incur if every integer access were indirect; just as important in this case is the goal of faithful modeling. An integer is conceptually not the same thing as a reference to an integer.
- Even with complex, developer-defined objects, we may prefer in some cases to consider that object O1 contains a subobject O2, rather than a reference to another object O2. The reason again may be efficiency, faithful modeling or both.

Expanded types

The answer to the need for modeling composite objects is simple. Let *C* be a class declared, as all classes so far, under the form

```
class C feature
        …
end
```

C may be used as a type. Any entity declared of type *C* represents a reference; for that reason *C* is called a **reference type**.

Now assume that we need an entity *x* whose value at run time will be an instance of *C* — not a reference to such an instance. We may obtain this effect by declaring *x* as

```
x : expanded C
```

This notation uses a new keyword, **expanded**. The notation **expanded** *C* denotes a type. The instances of this type are exactly the same as the instances of *C*. The only difference affects declarations using these types: an entity of type *C* denotes a reference which may become attached to an instance of *C*; an entity of type **expanded** *C*, such as *x* above, directly denotes an instance of *C*.

This mechanism adds the notion of composite object to the structure defined in the preceding sections. An object O is said to be composite if one or more of its fields are themselves objects — called **subobjects** of O. The following example class (routines again omitted) shows how to describe composite objects:

class *COMPOSITE* **feature**

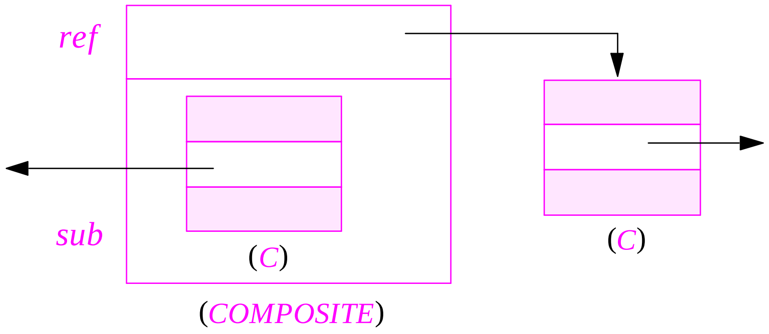
    ref: *C*

    sub: **expanded** *C*

end

This class relies on *C* declared as above. *COMPOSITE* has two attributes: *ref*, denoting a reference, and *sub*, denoting a subobject; *sub* is what makes the class composite. Any direct instance of *COMPOSITE* may look like this:

A composite object with one subobject



The *ref* field is a reference attached to an instance of *C* (or void). The *sub* field (which cannot be void) contains an instance of *C*.

A notational extension is convenient here. You may sometimes write a class *E* with the intention that all entities declared of type *E* should be expanded. To make this intention explicit, declare the class as

expanded class *E* **feature**

    … The rest as for any other class …

end

A class defined in this manner is said to be an expanded class. Here too the new declaration changes nothing for instances of *E*: they are the same as if the class had been declared as just **class** *E* … But an entity declared of type *E* will now denote an object, not a reference. As a consequence of this new possibility, the notion of "expanded type" includes two cases:

Definition: expanded type

A type is said to be expanded in the following two cases:

- It is of the form **expanded** *C*.
- It is of the form *E*, where *E* is an expanded class.

It is not a mistake to declare an entity *x* as being of type **expanded** *E* if *E* is an expanded class, just useless, since the result in this case is the same as if you declare *x* to be just of type *E*.

We now have two kinds of type; a type which is not expanded is a **reference type** (a term already used in this chapter). We may apply the same terminology to the entities correspondingly declared: reference entities and expanded entities. Similarly, a class is an expanded class if it has been declared as **expanded class**…, a reference class otherwise.

The role of expanded types

Why do we need expanded types? They play three major roles:

- Improving efficiency.
- Providing better modeling.
- Supporting basic types in a uniform object-oriented type system.

The first application may be the most obvious at first: without expanded types, you would have to use references every time you need to describe composite objects. This means that accessing their subobjects would require an operation to follow a reference — “dereferencing”, as it is sometimes called – which implies a time penalty. There is also a space penalty, as the run-time structure must devote space to the references themselves.

This performance argument is not, however, the prime justification. The key argument, in line with this chapter’s general emphasis on object-oriented software construction as a modeling activity, is the need to model composite objects separately from objects that contain references to other objects. This is not an implementation issue but a conceptual one.

Consider the two attribute declarations

D1 • *ref*: *S*

D2 • *exp*: **expanded** *S*

appearing in a class *C* (and assuming that *S* is a reference class). Declaration D1 simply expresses that every instance of *C* “knows about” a certain instance of *S* (unless *ref* is void). Declaration D2 is more committing: it states that every instance of C **contains** an instance of *S*. Aside from any implementation issue, this is a quite different relation.

In particular, the “contains” relation as provided by expanded types does not allow any **sharing** of the contained elements, whereas the “knows about” relation allows two or more references to be attached to the same object.

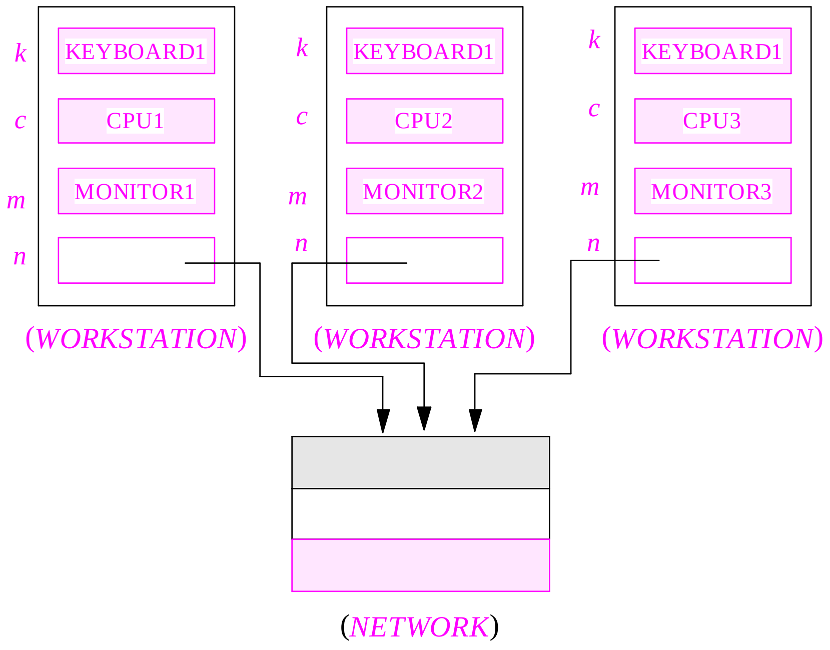
You may apply this property to ensure proper modeling of relations between objects. Consider for example this class declaration:

All classes shown are assumed to be reference (non-expanded) classes.

```
class WORKSTATION feature
    k: expanded KEYBOARD
    c: expanded CPU
    m: expanded MONITOR
    n: NETWORK
    …
end
```

Under this model a computer workstation has a keyboard, a CPU (central processing unit) and a monitor, and is attached to a network. The keyboard, CPU and monitor are part of a single workstation, and cannot be shared between two or more workstations. The network component, however, is shared: many workstations can be hooked up to the same network. The class definition reflects these properties by using expanded types for the first three attributes, and a reference type for the network attribute.

"Knows about" and "contains" relations between objects



So the concept of expanded type, which at first sight appears to be an implementation-level technique, actually helps describe some of the relations used in information modeling. The "contains" relation, and its inverse often known as "is-part-of", are central to any effort at building models of external systems; they appear in analysis methods and in database modeling.

See "A UNIFORM TYPE SYSTEM", 7.4, page 171. The outline of class REAL was on page 189.

The third major application of expanded types is in fact a special case of the second. The previous chapter emphasized the desirability of a uniform type system, based on the notion of class, which must encompass both developer-defined types and basic types. The example of *REAL* was used to show how, with the help of infix and prefix features, we can indeed model the notion of real number as a class; we can do the same for the other basic types *BOOLEAN*, *CHARACTER*, *INTEGER*, *DOUBLE*. But a problem remains. If these classes were treated as reference classes, an entity declared of a basic type, such as

r: *REAL*

would at run time denote a reference to a possible object containing a value (here of type *REAL*). This is unacceptable: to conform to common practice, the value of *r* should be the real value itself. The solution follows from the earlier discussion: define class *REAL* as expanded. Its declaration will be

```
expanded class REAL feature
        … Feature declarations exactly as given earlier (see page 189) …
end
```

All the other basic types are similarly defined by expanded classes.

Aggregation

In some areas of computing science — databases, information modeling, requirements analysis — authors have developed a classification of the relations that may hold between elements of a modeled system. Often mentioned in this context is the "aggregation" relation, which serves to express that every object of a certain type is a combination (an aggregate) of zero or more objects, each of a specified type. For example we might define "car" as an aggregation of "engine", "body" etc.

Expanded types provide the equivalent mechanism. We may for example declare class *CAR* with features of types **expanded** *ENGINE* and **expanded** *BODY*. Another way to express this observation is to note that aggregation is covered by the "expanded client" relation, where a class *C* is said to be an expanded client of a class *S* if it contains a declaration of a feature of type **expanded** *S* (or just *S* if *S* is expanded). One advantage of this modeling approach is that "expanded client" is just a special case of the general client relation, so that we can use a single framework and notation to combine aggregation-like dependencies (that is to say, dependencies on subobjects, such as the relation between *WORKSTATION* and *KEYBOARD* in the earlier example) with dependencies that permit sharing (such as the relation between *WORKSTATION* and *NETWORK*).

With the object-oriented approach, one can avoid the multiplicity of relations found in the information modeling literature, and cover all possible cases with just two relations: client (expanded or not) and inheritance.

Properties of expanded types

Consider an expanded type *E* (of either form) and an expanded entity *x* of type *E*.

Since the value of *x* is always an object, it can never be void. So the expression

x = *Void*

will always yield the value false, and a call of the form *x*.*some_feature* (*arg1*, …) will never raise the exception "call on void target" that was possible in the case of references.

"Void references and calls", page 240.

Let object O be the value of *x*. As with the case of a non-void reference, *x* is said to be attached to O. So for any non-void entity we may talk of the attached object, whether the entity is of reference or expanded type.

What about creation? The instruction

```
!! x
```

See "Effect of a basic creation instruction", page 233.

may be applied to an expanded *x*. For reference *x*, its effect was to perform three steps: (C1) create a new object; (C2) initialize its fields to the default values; (C3) attach it to *x*. For expanded *x*, step C1 is inappropriate, and step C3 is unneeded; so the only effect is to set all fields to their default values.

More generally, the presence of expanded types affects the default initialization performed as part of C2. Assume a class, expanded or not, having one or more expanded attributes:

```
class F feature
        u: BOOLEAN
        v: INTEGER
        w: REAL
        x: C
        y: expanded C
        z: E
        …
end
```

where *E* is expanded but *C* is not. The initialization of a direct instance of *F* involves setting the *u* field to false, the *v* field to 0, the *w* field to 0.0, the *x* field to a void reference, and the *y* and *z* to instances of *C* and *E* respectively, whose fields are themselves initialized according to the standard rules. This initialization process is to be applied recursively, since *C* and *E* may themselves include expanded fields.

Cycles in the client relation were studied in "Self-reference", page 226.

As you may have realized, a restriction is necessary for expanded types to be usable (to ensure that the recursive process just defined always remains finite): although, as discussed earlier, the client relation may in general include cycles, such cycles must make no use of expanded attributes. For example it is not permitted for class *C* to have an attribute of type **expanded** *D* if class *D* has an attribute of type **expanded** *C*; this would mean that every object of type *C* includes a subobject of type *D* and conversely — a clear impossibility. Hence the following rule, based on the notion of "expanded client" already introduced informally above:

Expanded Client rule

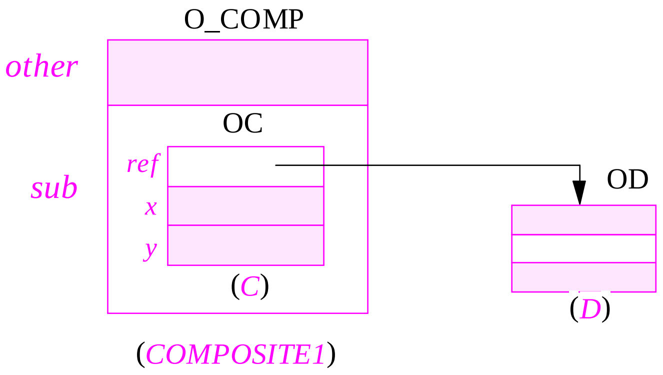
Let "expanded client" the relation between classes be defined as follows: *C* is an expanded client of *S* if some attribute of *C* is of an expanded type based on *S* (that is to say **expanded** *S*, or just *S* if *S* is an expanded class).

Then the expanded client relation may not include any cycles.

In other words there may not be a set of classes *A*, *B*, *C*, … *N* such that *A* is an expanded client of *B*, *B* an expanded client of *C* etc., with *N* being an expanded client of *A*. In particular, *A* may not have an attribute of type **expanded** *A*, as this would make *A* an expanded client of itself.

No references to subobjects

A final comment about expanded types will answer the question of how to mix references and subobjects. An expanded class, or an expanded type based on a reference class, may have reference attributes. So a subobject may contain references attached to objects:



A subobject with a reference to another object

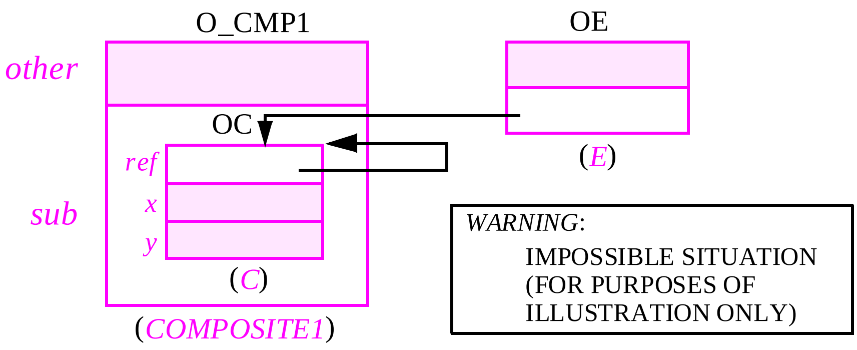
The situation pictured assumes the following declarations:

```
class COMPOSITE1 feature
    other: SOME_TYPE
    sub: expanded C
end

class C feature
    ref: D
    x: OTHER_TYPE; y: YET_ANOTHER_TYPE
end

class D feature
    …
end
```

Each *COMPOSITE* instance, such as O_COMP in the figure, has a subobject (OC in the figure) containing a reference *ref* which may be attached to an object (OD in the figure).

But the reverse situation, where a reference would become attached to a subobject, is impossible. (This will follow from the rules on assignment and argument passing, studied in the next section.) So the run-time structure can never come to the state described by the picture on the facing page, where OE contains a reference to OC, a subobject of O_CMP1, and OC similarly contains a reference to itself.

A reference to a subobject



This rule is open to criticism since it limits the modeling power of the approach. Earlier versions of this book's notation did in fact permit references to subobjects. But this possibility was found to cause more problems than it was worth:

Garbage collection is studied in the next chapter.

- From the implementation's perspective, the garbage collection mechanism must be prepared to deal with subobject references even if in a given execution there are few such references, or none at all. This caused a significant performance degradation.
- From the viewpoint of modeling, excluding subobject references actually turned out to simplify system descriptions by defining a single unit of referencing, the object.

The discussion will point out what precise attachment rule would have to be modified to revert to the scheme in which references may be attached to subobjects.

8.8 ATTACHMENT: REFERENCE AND VALUE SEMANTICS

If skipping, go to "DEALING WITH REFERENCES: BENEFITS AND DANGERS", 8.9, page 265.

(This section covers more specialized information and you may skip it on first reading.)

The introduction of expanded types means that we must take a second look at two fundamental operations studied earlier in this chapter: assignment, written :=, which attaches a reference to an object, and the associated comparison operation, written =. Since entities may now denote objects as well as references to objects, we must decide what assignment and equality will mean in the first of these cases.

Attachment

The semantics of assignment will actually cover more than this operation. Another case in which the value of an entity may change is argument passing in routine calls. Assume a routine (procedure or function) of the form

r (…, *x*: *SOME_TYPE*, …)

Here entity *x* is one of the **formal arguments** of *r*. Now consider a particular call to *r*, of one of the possible two forms (unqualified and qualified):

r (…, *y*, …)
t•*r* (…, *y*, …)

where expression *y* is the **actual argument** having the same position in the list of actual arguments as *x* has in the list of formal arguments.

Whenever *r* gets started as a result of one of these calls, it initializes each of its formal arguments with the value of the corresponding actual argument, such as *y* for *x*.

For simplicity and consistency, the rules governing such actual-formal argument associations are the same as the rules governing assignment. In other words, the initial effect on *x* of such a call is exactly as if *x* were the target of assignment of the form

x := *y*

This rule yields a definition:

Definition: attachment

An attachment of *y* to *x* is either of the following two operations:

- An assignment of the form *x* := *y*.
- The initialization of *x* at the time of a routine call, where *x* is a formal argument of a routine and *y* is the corresponding actual argument in the call.

In both cases, *x* is the **target** of the attachment and *y* its **source**.

Exactly the same rules will be applicable in both cases to determine whether an attachment is valid (depending on the types of its target and source) and, if it is, what effect it will have at execution time.

Reference and copy attachment

We have seen a first rule for the effect of attachment when studying reference assignment. If both source and target are references, then the effect of an assignment

x := *y*

and of the corresponding argument passing is to make *x* denote the same reference as *y*. This was illustrated through several examples. If *y* is void prior to the attachment, the operation will make *x* void too; if *y* is attached to an object, *x* will end up attached to the same object.

What now if the types of *x* and *y* are expanded? Reference assignment would not make sense, but a copy (the shallow form) is possible. The meaning of an attachment of an expanded source to an expanded target will indeed be a copy. With the declarations

x, *y*: **expanded** *SOME_CLASS*

the assignment *x* := *y* will copy every field of the object attached to *y* onto the corresponding field of the object attached to *x*, producing the same effect as

x.*copy* (*y*)

which of course is still legal in this case. (In the case of reference types, *x* := *y* and *x*.*copy* (*y*) are both legal but have different effects.)

This copy semantics for expanded types yields the expected effect in the case of the basic types which, as noted above, are all expanded. For example if *m* and *n* have been declared of type *INTEGER*, you will expect the assignment *m* := *n*, or a corresponding argument passing, to copy the value of *n* onto that of *m*.

The analysis just applied to attachment transposes immediately to a related operation: comparison. Consider the boolean expressions *x* = *y* and *x* /= *y*, which will have opposite values. For *x* and *y* of reference types, as already noted, the tests compare references: *x* = *y* yields true if and only if *x* and *y* are either both void or both attached to the same object. For expanded *x* and *y*, this would not make sense; the only acceptable semantics is to use field-by-field comparison, so that in this case *x* = *y* will have the same value as *equal* (*x*, *y*).

"Fixed semantics for copy, clone and equality features", page 583.

It is possible, as we will see in the discussion of inheritance, to adapt the semantics of *equal* to support a specific notion of equality for the instances of some class. This has no effect on the semantics of =, which, for safety and simplicity, is always that of the original function *standard_equal*.

The basic rule for attachment and comparison, then, is summarized by the following observation:

> An attachment of *y* to *x* is a copy of objects *x* if *x* and *y* are of expanded types (including any of the basic types). It is a reference attachment if *x* and *y* are of reference types.
>
> Similarly, an equality or inequality test *x* = *y* or *x* /= *y* is a comparison of objects for *x* and *y* of expanded types; it is a comparison of references if *x* and *y* are of reference types.

Hybrid attachments

In the cases seen so far, the source and target types of an attachment are of the same category — both expanded or both reference. What if they are of different categories?

See chapter 12, in particular "Sources of exceptions", page 412.

First consider *x* := *y* where the target *x* is of an expanded type and the source *y* is of a reference type. Because reference assignment does not make sense for *x*, the only acceptable semantics for this attachment is copy semantics: copy the fields of the object attached to *y* onto the corresponding fields of the object attached to *x*. This is indeed the effect of the assignment in this case; but it only makes sense if *y* is non-void at the time of execution (otherwise there is no attached object). If *y* is void, the result will be to trigger an exception. The effect of exceptions, and the specification of how to recover from an exception, are discussed in a later chapter.

For expanded *x*, the test *x* = *Void* does not cause any abnormal event; it simply yields the result false. But there is no way we can find an acceptable semantics for the assignment *x* := *Void*, so any attempt at executing it causes an exception.

Now consider the other case: *x* := *y* where *x* is of a reference type and *y* is of an expanded type. Then at run time *y* is always attached to an object, which we may call OY, and the attachment should also attach *x* to an object. One possibility would be to attach *x* to OY. This convention, however, would introduce the possibility of references to subobjects, as in routine *reattach* below:

```
class C feature
          …
end

class COMPOSITE2 feature
          x: C
          y: expanded C

          reattach is
                    do x := y end
end
```

If, as suggested earlier, we prohibit references to subobjects, we may in such a case prescribe that the attachment perform a **clone** of OY. This will indeed be the effect of the attachment for expanded source and reference target: attach the target to a clone of the source object.

The following table summarizes the semantics of attachment in the cases studied:

Effect of attachment *x := y*

Type of source *y* → / ↓ ***Type of target*** *x*	**Reference**	**Expanded**
Reference	Reference attachment	Clone; effect of *x := clone (y)*
Expanded	Copy; effect of *x.copy (y)* (will fail if *y* is void)	Copy; effect of *x.copy (y)*

To allow references to subobjects, it would suffice to replace the clone semantics defined in the top-right entry by the semantics of reference attachment.

Equality comparison

The semantics of equality comparison (the = and /= signs) should be compatible with the semantics of attachment: if *y* /= *z* is true and you execute *x* := *y*, then both *x* = *y* and *x* /= *z* should be true immediately after the assignment.

Besides =, we have seen that there is an operation *equal* applicable to objects. Which of these operations is available depends on the circumstances:

E1 • If *x* and *y* are references, you can test both for reference equality and, if the references are not void, for object equality. We have defined the operation *x* = *y* as denoting reference equality in this case. The *equal* function was introduced to cover object equality; for completeness it also applies when *x* or *y* is void (returning true in this case only if both are).

E2 • If *x* and *y* are expanded, the only operation that makes sense is object comparison.

E3 • If *x* is a reference and *y* is expanded, object equality is also the only meaningful operation — again extended to accept void *x*, in which case it will return false since *y* cannot be void.

This analysis yields the desirable interpretation for = in all cases. For object comparison, *equal* is always available, conveniently extended to deal with cases in which one or both operands are void. = serves to apply reference comparison when it makes sense, defaulting to *equal* in other cases:

Meaning of comparison
x = *y*

***Type of** y* → / ↓ ***Type of** x*	**Reference**	**Expanded**
Reference	Reference comparison	*equal* (*x*, *y*) i.e. object comparison if *x* non-void, false if *x* void.
Expanded	*equal* (*x*, *y*) i.e. object comparison if *y* non-void, false if *y* void.	*equal* (*x*, *y*) i.e. object comparison.

By comparing with the preceding table, you may check that = and /= are indeed compatible with := in the sense defined above. Recall in particular that *equal* (*x*, *y*) will be true as a result of *x* := *clone* (*y*) or *x*•*copy* (*y*).

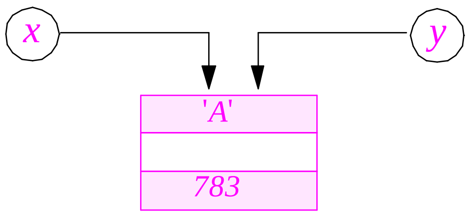
This issue that we have just settled arises in any language which includes pointer or references (such as Pascal, Ada, Modula-2, C, Lisp etc.), but is particularly acute in an object-oriented language in which all non-basic types are reference types; in addition, for reasons explained in the discussion section, the syntax does not explicitly show them to be references, so we need to be particularly careful.

8.9 DEALING WITH REFERENCES: BENEFITS AND DANGERS

Two properties of the run-time model, as introduced in the preceding sections, deserve further examination. One is the important role of references; the other is the dual semantics of basic operations such as assignment, argument passing and equality tests which, as we have seen, produce different effects for reference and expanded operands.

Dynamic aliasing

If *x* and *y* are of reference types and *y* is not void, the assignment *x* := *y*, or the corresponding attachment in a call, causes *x* and *y* to be attached to the same object.



Sharing as a result of an attachment

The result is to bind *x* and *y* in a durable way (until any further assignment to any of them). In particular, an operation of the form *x*.*f*, where *f* is some feature of the corresponding class, will have the same effect as *y*.*f* since they affect the same object.

The attachment of *x* to the same object as *y* is known as dynamic aliasing: aliasing because the assignment makes an object accessible through two references, like a person known under two names; dynamic because the aliasing occurs at run time.

Static aliasing, where a software text specifies that two names will always denote the same value regardless of what happens at execution time, is also possible in some programming languages: the Fortran *EQUIVALENCE* directive states that two variables will always denote the contents of the same memory location; and the C preprocessor directive *#define x y* specifies that any further occurrence of *x* in the program text means exactly the same thing as *y*.

Because of dynamic aliasing, attachment operations have a more far-reaching effect on entities of reference types than on those of expanded types. If *x* and *y* are of type *INTEGER*, an example of expanded type, the assignment *x* := *y* only resets the value of *x* using that of *y*; but it does not durably bind *x* and *y*. For reference types, the assignment causes *x* and *y* to become aliases for the same object.

The semantics of aliasing

A somewhat shocking consequence of aliasing (static or dynamic) is that an operation may affect an entity that it does not even cite.

Models of computation that do not involve aliasing enjoy a pleasant property: the correctness of such extracts as

[NO SURPRISE]

```
        -- Assume that here P (y) holds
    x := y
    C (x)
        -- Then here P (y) still holds.
```

This example assumes that *P* (*y*) is an arbitrary property of *y*, and *C* (*x*) some operation whose textual description in the software may involve *x* but does not involve *y*. Correctness here means that the property of “NO SURPRISE” expressed by the comments is indeed satisfied: if *P* (*y*) is true initially, then no action on *x* can invalidate this property. An operation on *x* does not affect a property of *y*.

With entities of expanded types, property NO SURPRISE indeed holds. Here is s typical example, assuming *x* and *y* of type *INTEGER*:

```
-- Assume that here y >= 0
x := y
x := –1
-- Then here y >= 0 still holds.
```

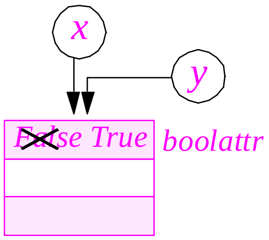
In no way can the assignment to *x* have any effect on *y* in this case. But now consider a similar one involving dynamic aliasing. Let *x* and *y* be of type *C*, where class *C* is of the form

```
class C feature
    boolattr: BOOLEAN
            -- Boolean attribute, modeling some object property.
    set_true is
            -- Make boolattr true.
        do
            boolattr := True
        end
    … Other features …
end
```

Assume that *y* is of type *C* and that its value at some run-time instant is not void. Then the following instance of the above scheme violates property NO SURPRISE:

[SURPRISE, SURPRISE!]

```
        -- Assume that y.boolattr is false.
    x := y
        -- Here it is still true that y.boolattr is false.
    x.set_true
        -- But then here y.boolattr is true!
```



The last instruction of this extract does not involve *y* in any way; yet one of its effects is to change the properties of *y*, as indicated by the final comment.

Coming to terms with dynamic aliasing

Having seen the disturbing consequences of reference assignments and dynamic aliasing, one may legitimately ask why we should keep such a facility in our model of computation.

The answer is twofold — partly theoretical and partly practical:

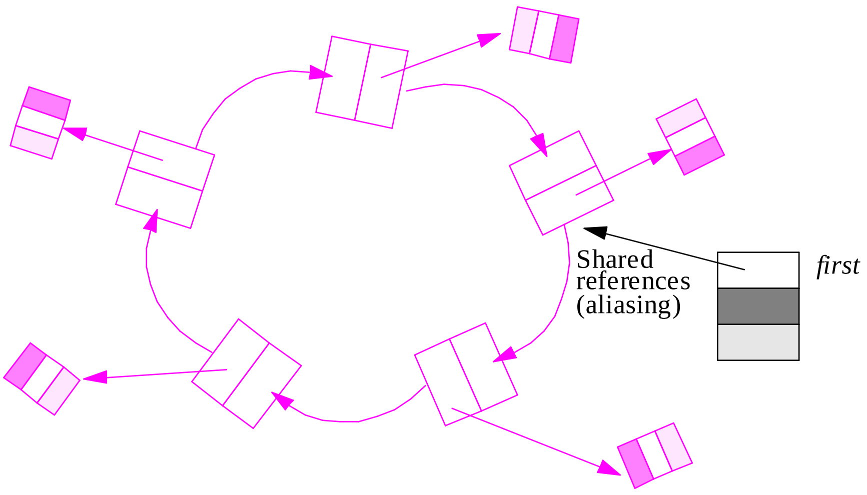
- We need reference assignments if we are to benefit from the full power of the object-oriented method, in particular to describe complex data structures. The issue here is again to make sure that our tools are versatile enough for our modeling needs.
- In the practice of object-oriented software construction, encapsulation makes it possible to avoid the dangers of reference manipulations.

Let us examine these two important aspects in turn.

Aliasing in software and elsewhere

The first observation is simply that many of the data structures we will need require references and reference sharing. Some standard data structures, for example, include cyclically chained elements, which you cannot implement without references. In representing list and tree structures, it is often convenient to let every node contain a reference to its neighbor or parent. The figure below shows a circular list representation, combining both of these ideas. Open any textbook on fundamental data structures and algorithms, as used in introductory computing science courses, and you will find many such examples. With object technology we will want, if anything, to use even more sophisticated structures.

A linked circular list



In fact the need for references, reference attachment and reference sharing already arises with quite unsophisticated data structures. Recall the classes used above to describe books; one of the variants was

Page 226.

```
class BOOK3 feature
        … Other features; …
        author: WRITER
end
```

Here the need for reference sharing is simply a consequence of the property that two or more books may have the same author. Many of the examples of this chapter also cause sharing; in the *PERSON* case, several people may have the same landlord. The question, as already noted, is modeling power, not just the requirements of implementation.

But then if *b1* and *b2* are two instances of *BOOK3* with the same author, we have a case of aliasing: *b1*.*author* and *b2*.*author* are two references attached to the same object, and using any of them as target of a feature call will have exactly the same effect as using the other. Seen in this light, dynamic aliasing appears less as a potentially dangerous software facility than as a fact of life, the price to pay for the convenience of being able to refer to things under more than one name.

It is indeed easy to find violations of the above NO SURPRISE property without ever entering the software field. Consider the following property and operation, defined for any book *b*:

- *NOT_NOBEL* (*b*) stands for: "the author of *b* has never received the Nobel prize".
- *NOBELIZE* (*b*) stands for: "Give the Nobel prize to the author of *b*".

Now assume *rb* denotes the book *The Red and the Black* and *cp* denotes *The Charterhouse of Parma*. Then the following is a correct development:

Stendhal lived prior to the establishment of the prize, of course — and would probably not have got it anyway; he did not even make it to the Académie.

```
[SURPRISE IN OSLO]
            -- Assume that here NOT_NOBEL (rb) holds
      NOBELIZE (cp)
            -- Then here NOT_NOBEL (rb) does not hold any more!
```

An operation on *cp* has changed a property of a different entity, *rb*, not even named in the instruction! The consequences on *rb* may actually be quite significant (with a Nobel author an out-of-print book will be reprinted, its price may rise etc.). In this non-software case exactly the same thing happens as when the operation *x*.*set_true*, in the earlier software example, produced an important effect on *y* even though it did not refer to *y*.

So dynamic aliasing is not just a consequence of programmers' dirty tricks with references or pointers. It is a consequence of the human ability to *name* things ("objects" in the most general sense of the word), and to give many names to one thing. In classical rhetoric, this was known as a *polyonymy*, as with the use of "Cybele", "Demeter" and "Ceres" for the same goddess, and *antonomasia*, the ability to refer to an object through indirect phrases, as with "The beautiful daughter of Agammemnon" for Helena of Troy. Polyonymy, antonomasia and the resulting dynamic aliasing are not restricted to gods and heroes; if in the cafeteria you overhear two conjectures from separate conversations, one stating that the spouse of the engineering Vice President just got a big promotion and the other that the company has fired its accountant, you will not realize the contradiction — unless you know that the accountant is the VP's husband.

Encapsulating reference manipulations

By now we have accumulated enough evidence that any realistic framework for modeling and software development must support the notion of reference, and consequently dynamic aliasing. How then do we cope with the unpleasant consequences of these mechanisms? The inability to ensure the NO SURPRISE property illustrates how references and aliasing endanger our ability to reason systematically about our software, that is to say, to infer run-time properties of the software's execution, in a safe and simple way, by examining the software text.

To find an answer it helps to understand first how much of this issue is specific to the object-oriented method. If you are familiar with such programming languages as Pascal, C, PL/I, Ada and Lisp you will probably have noted that much of the above discussion applies to them as well. They all have a way of allocating objects dynamically (although in C the corresponding function, *malloc*, is in the library rather than the language proper) and of letting objects contain references to other objects. The level of abstraction of the language mechanisms varies significantly: C and PL/I pointers are scantily dressed machine addresses; Pascal and Ada use typing rules to wrap pointers in more respectable attire, although they do not need much prompting to return to their original state.

What then is new with object-oriented development? The answer lies not in the theoretical power of the method (whose run-time structures are similar to those of Pascal or Ada, with the important difference of garbage collection, studied in the next chapter) but in the practice of software construction. O-O development implies reuse. In particular, any project in which many application classes perform tricky manipulations (such as reference manipulation) is a flawed use of the object-oriented approach. Such operations should be encapsulated once and for all in library classes.

Regardless of the application domain, if a system includes object structures requiring non-trivial reference operations, the vast majority of these structures are not application-specific but merely instances of such frequently needed and well-known structures as lists of various kinds, trees under various representations, graphs, hash tables and a few others. In a good O-O environment a library will be readily available, offering many implementations of these structures; appendix A will sketch an example, the Base library. The classes of such a library may contain many operations on references (think for example of the reference manipulations needed to insert or delete an element in a linked list, or a node in a tree using linked representation). The library should have been patiently crafted and validated, so as to take care of the tricky problems once and for all.

"Reuse consumers, reuse producers", page 69.

If, as you are building the application, you recognize the need for complex object structures which are not adequately covered by the available libraries, you should look at them as requiring new general-purpose classes. You should design and check them carefully, under the expectation that in due time they will become part of some library. Using the terminology introduced in an earlier chapter, such a case is an example of moving from a consumer's to a producer's view of reuse.

The remaining reference manipulations in application-dependent classes should be restricted to simple and safe operations. (The bibliographical notes cite an article by Suzuki which explores this idea further.)

8.10 DISCUSSION

This chapter has introduced a number of rules and notations for manipulating objects and the corresponding entities. Some of these conventions may have surprised you. So it is useful to conclude our exploration of objects and their properties by examining the issues involved and the reasons behind the choices made. Although I hope you will in the end agree with these choices, the more important goal of this discussion is to make sure that you fully understand the underlying problems, so that even if you prefer a different solution you choose it with your eyes open.

Graphical conventions

To warm up let us begin with a little notational issue — a detail, really, but software is sometimes in the details. This particular detail is the set of conventions used to illustrate classes and objects in graphical representations.

"The mold and the instance", page 167.

The previous chapter emphasized the importance of not confusing the notions of class and object. Accordingly, the graphical representations are different. Objects are represented as rectangles. Classes, as they appear in system architecture diagrams, are represented by ellipses (connected by arrows representing the relations between classes: single arrow for the inheritance relation, double arrow for the client relation).

Class and object representations appear in different contexts: a class ellipse will be part of a diagram representing the structure of a software system; an object rectangle will be part of a diagram representing a snapshot of the state of a system during its execution. Because these two kinds of diagram address completely different purposes, there is usually no opportunity in paper presentations such as the present book for having both class and object representations appear in the same context. But the situation is different with interactive CASE tools: during the execution of a software system, you may want (for example for debugging purposes) to look at an object, and then display its generating class to examine the features, parents or other properties of that class.

On BON see the bibliographical notes and chapter 27.

The graphical conventions used for classes and objects are compatible with the standard established by Nerson and Waldén's BON method. In BON (Business Object Notation), which is meant for use in interactive CASE tools as well as for paper documentation, class bubbles can be stretched vertically so as to reveal a class's features, invariant, indexing words, and other properties.

As with any choice of graphical representation, there is no absolute justification for the conventions used in BON and in this book. But if the graphical symbols at our disposal are ellipses and rectangles, and the elements to be represented are classes and objects, then it does appear preferable to assign rectangles to objects: an object is a set of fields, so we can represent each field by a small rectangle and glue together a set of fields to make up a bigger rectangle which represents an object.

A further convention, illustrated by the figures of this chapter, is to make expanded fields appear shaded, whereas references fields are blank; subobjects appear as smaller embedded rectangles, containing their own fields. All these conventions follow from the decision to use rectangles for objects.

From a review of Martin and Odell's "Object-Oriented Analysis and Design", in OOPS (British Computer Society O-O interest group newsletter), 16, Winter 1992, pages 35-37.

On the lighter side, it is hard to resist quoting the following non-scientific argument, from Ian Graham's critique of an O-O analysis book that uses a different convention:

> *Nor do I like showing classes as sharp cornered triangles. I like to think that instances have sharp corners because if you drop them on your foot they hurt, whereas classes can't hurt anyone and therefore have rounded corners.*

References and simple values

An important syntactical question is whether we should deal differently with references and simple values. As noted, assignment and equality test have different meanings for references and for values of expanded types — the latter including values of basic types: integers and the like. Yet the same symbols are used in both cases: *:=*, *=*, */=*. Is this not dangerous? Would it not be preferable to use different sets of symbols to remind the reader that the meanings are different?

Using two sets of symbols was indeed the solution of Simula 67. Transposing the notation slightly so as to make it compatible with those of the present book, the Simula solution is to declare an entity of a reference type *C* as

Simula, covered in chapter 35, abbreviates **reference** *to* **ref**.

x: **reference** *C*

where the keyword **reference** reminds the reader that instances of *x* will be references. Assuming the declarations

m, *n*: *INTEGER*
x, *y*: **reference** *C*

then different notations are used for operations on simple and reference types, as follows:

Simula-style notations for operations on reference and expanded values

OPERATION	EXPANDED OPERANDS	REFERENCE OPERANDS
Assignment	*m* := *n*	*x* :– *y*
Equality test	*m* = *n*	*x* == *y*
Inequality test	*m* /= *n*	*x* =/= *y*

The Simula conventions remove any ambiguity. Why not keep them then? The reason is that in practice they turn out in spite of the best intentions to cause more harm than help. The problems begin with a mundane matter: typing errors. The two sets of symbols are so close that one tends to make syntactical oversights, such as using := instead of :–. Such errors will be caught by the compiler. But although compiler-checkable restrictions in programming languages are meant to help programmers, the checks are of no use here: either you know the difference between reference and value semantics, in which case the obligation to prove again, each time you write an assignment or equality, that you did understand this difference, is rather annoying; or you do not understand the difference, but then the compiler message will not help you much!

The remarkable aspect of the Simula convention is that you do not in fact have a choice: for references, no predefined construct is available that would give value semantics. It might have seemed reasonable to allow two sets of operations on entities *a* and *b* of reference types:

- *a* :– *b* for reference assignment, and *a* == *b* for reference comparison.
- *a* := *b* for copy assignment (the equivalent, in our notation, of either *a* := *clone* (*b*) or *a*.*copy* (*b*)), and *a* = *b* for object comparison (the equivalent of our *equal* (*a*, *b*)).

But this is not the case; for operands of reference types, with one exception, Simula only provides the first set of operations, and any attempt to use := or = will produce a syntactical error. If you need operations of the second set (copy or clone, object comparison), you must write specific routines corresponding to our *clone*, *copy* and *equal* for each target class. (The exception is the *TEXT* type, representing character strings, for which Simula does offer both sets of operations.)

On further examination, by the way, the idea of allowing both sets of operations for all reference types does not appear so clever. It would mean that a trivial oversight such as typing := for :– would now go undetected by the compiler but produce an effect quite different from the programmer's intent, for example a *clone* where a reference assignment was intended.

As a result of this analysis, the notation of this book uses a different convention from Simula's: the same symbols apply for expanded and reference types, with different semantics (value in one case, reference in the other). You can achieve the effect of value semantics for objects of reference types by using predefined routines, available on all types:

- *a* := *clone* (*b*) or *a*•*copy* (*b*) for object assignment.

- *equal* (*a*, *b*) for object (field-by-field) comparison.

These notations are sufficiently different from their reference counterparts (:= and =, respectively) to avert any risk of confusion.

Beyond the purely syntactical aspects, this issue is interesting because it typifies some of the tradeoffs that arise in language design when a balance must be found between conflicting criteria. One criterion, which won in the Simula case, may be stated as:

- "Make sure different concepts are expressed by different symbols".

But the opposing forces, which dominated in the design of our notation, say:

- "Avoid bothering the software developer."

- "Weigh carefully any new restriction against the actual benefits that it will bring in terms of security and other quality factors." Here the restriction is the prohibition of := and similar operators for references.

- "Make sure that the most common operations can be expressed by short and simple notations." The application of this principle requires some care, as the language designer may be wrong in his guesses of what cases will be the most common. But in the present example it seems clear that on entities of expanded types (such as *INTEGER*) value assignment and comparison are the most frequent operations, whereas on references entities reference assignment and comparison are more frequent than clone, copy and object comparison. So it is appropriate to use := and = for the fundamental operations in both cases.

• "To keep the language small and simple, do not introduce new notations unless they are absolutely necessary". This applies in particular if, as in this example, existing notations will do the job and there is no danger of confusion.

• "If you know there is a serious risk of confusion between two facilities, make the associated notations as different as possible." This leads us to avoid making both :– and := available for the same operands with different semantics.

One more reason plays a role in the present case, although it involves mechanisms that we have not yet studied. In later chapters we will learn to write generic classes, such as *LIST* [*G*], where *G*, known as a formal generic parameter, stands for an arbitrary type. Such a class may manipulate entities of type *G* and use them in assignments and equality tests. Clients that need to use the class will do so by providing a type to serve as actual generic parameter; for example they may use *LIST* [*INTEGER*] or *LIST* [*POINT*]. As these examples indicate, the actual generic parameter may be an expanded type (as in the first case) as well as a reference type (as in the second case). In the routines of such a generic class, if *a* and *b* are of type *G*, it is often useful to use assignments of the form *a* := *b* or tests of the form *a* = *b* with the intent of obtaining value semantics if the actual generic parameter is expanded (as with *INTEGER*) and reference semantics if it is a reference type (as with *POINT*).

An example of a routine which needs such dual behavior is a procedure for inserting an element *x* into a list. The procedure creates a new list cell; if *x* is an integer, the cell must contain a copy of that integer, but if *x* is a reference to an object the cell will contain a reference to the same object.

In such a case the rules defined above ensure the desired dual behavior, which would have been impossible to achieve if a different syntax had been required for the two kinds of semantics. If, on the other hand, you want a single identical behavior in all cases, you can specify it too: that behavior can only be value semantics (since reference semantics does not make sense for expanded types); so in the appropriate routines you should use not := and = but *clone* (or *copy*) and *equal*.

The form of clone and equality operations

A small point of style which may have surprised you is the form under which routines *clone* and *equal* are called. The notations

clone (*x*)

equal (*x*, *y*)

do not look very O-O at first; a dogmatic reading of the previous chapter would suggest conventions that seem more in line with what was there called "the object-oriented style of computation"; for example:

"THE OBJECT-ORIENTED STYLE OF COMPUTATION", 7.7, page 181.

x•*twin*

x•*is_equal* (*y*)

In a very early version of the notation, these were indeed the conventions. But they raise the problem of void references. A feature call of the form *x*.*f* (…) cannot be executed correctly if, at run time, the value of *x* is void. (In that case the call will trigger an exception which, unless the class contains specific provisions to recover from the exception, will cause the execution of the entire system to terminate abnormally.) So the second set of conventions would only work for non-void *x*. Because in many cases *x* may indeed be void, this would mean that most uses of *twin* would in practice be of the form

```
if x = Void then
        z := Void
else
        z := x.twin
end
```

and most uses of *is_equal* of the form

and then is a variant of and. See "Non-strict boolean operators", page 454.

```
if
        ((x = Void) and (y = Void)) or
        ((x /= Void) and then x.is_equal (y))
then
        …
```

Needless to say, these conventions were not kept for long. We quickly became tired of having to write such convoluted expressions — and even more of having to face the consequences (run-time errors) when we forgot. The conventions finally retained, described earlier in this chapter, have the pleasant property of giving the expected results for void *x*: in that case *clone* (*x*) is a void value, and *equal* (*x*, *y*) is true if and only if *y* is also void.

Procedure *copy*, called under the form *x*.*copy* (*y*), raises no particular problem: it requires *x* (and also *y*) to be non-void, but this requirement is acceptable because it is a consequence of the semantics of *copy*, which copies an object onto another and so does not makes sense unless both objects exist. The condition on *y*, as explained in a later chapter, is captured by an official precondition on *copy* and so is present in a clear form in the documentation for this procedure.

It should be noted that a function *is_equal* as introduced above exists. The reason is that it is often convenient to define specific variants of equality, adapted to a class and overriding the default semantics of field-by-field comparison. To obtain this effect it suffices to redefine function *is_equal* in the desired classes. Function *equal* is defined in terms of *is_equal* (through the expression shown above to illustrate the use of *is_equal*), and so will follow its redefinitions.

See also "Fixed semantics for copy, clone and equality features", page 583.

In the case of *clone*, there is no need for *twin*. This is because *clone* is simply defined as a creation plus a call to *copy*. So to adapt the meaning of *clone* to the specific needs of a class it suffices to redefine procedure *copy* for that class; *clone* will automatically follow.

The status of universal operations

The last comments have partly lifted the veil on a question that have may caught your attention: what is the status of the universal operations *clone*, *copy*, *equal*, *is_equal*, *deep_clone*, *deep_equal*?

Although fundamental in practice, these operations are not language constructs. They come from a Kernel library class, *ANY*, which has the special property that every class written by a software developer automatically inherits (directly or indirectly) from *ANY*. This is why it is possible to redefine the features mentioned to support a particular view of equality or copying.

"THE GLOBAL INHERITANCE STRUCTURE", 16.2, page 580.

We need not concern ourselves with the details here, as they will be studied together with inheritance. But it is useful to know that, thanks to the inheritance mechanism, we can rely on library classes to provide facilities that are then made available to any class — and can be adapted by any class to suit its own specific purposes.

8.11 KEY CONCEPTS INTRODUCED IN THIS CHAPTER

- Object-oriented computation is characterized by a highly dynamic run-time structure, where objects are created on demand rather than pre-allocated.
- Some of the objects manipulated by the software are (usually quite indirect) models of outside objects. Others serve design and implementation purposes only.
- An object is made of a number of values called fields. Each field corresponds to an attribute of the object's generator (the class of which the object is a direct instance).
- A value, in particular a field of an object, is either an object or a reference.
- A reference is either void or attached to an object. The test *x* = *Void* tells which of the two cases holds. A call with target *x*, such as *x*•*f* (…), can only be executed correctly if *x* is non-void.
- If the declaration of a class begins with **class** *C* …, an entity declared of type *C* will denote a reference, which may become attached to instances of *C*. If the declaration begins with **expanded class** *D* …, an entity declared of type *D* will denote an object (an instance of *D*), and will never be void.
- The basic types (*BOOLEAN*, *CHARACTER*, *INTEGER*, *REAL*, *DOUBLE*) are defined by expanded classes.
- Expanded declarations also make it possible to define composite objects: objects with subobjects.
- Object structures may contain cyclic chains of references.
- The creation instruction !! *x* creates an object, initializes its field to default values (such as void for references and zero for numbers), and attaches *x* to it. If the class has defined creation procedures, The instruction will also perform, in the form !! *x*•*creatproc* (…), any desired specific initializations.

- On entities of reference types, assignment (:=) and equality test (=) are reference operations. On entities of expanded types, they represent copy and field-by-field comparison. They also have the appropriate semantics for mixed operands.
- Reference operations cause dynamic aliasing, which makes it more difficult to reason formally about software. In practice, most non-trivial reference manipulations should be encapsulated in library classes.

8.12 BIBLIOGRAPHICAL NOTES

The notion of object identity plays an important role in databases, especially object-oriented databases. See chapter 31 and its bibliographical notes.

The graphical conventions of the BON method (Business Object Notation), designed by Jean-Marc Nerson and Kim Waldén, appear in [Waldén 1995]. James McKim and Richard Bielak expound the merits of multiple creation procedures in [Bielak 1994].

The risks caused by unfettered pointer or reference operations have worried software methodologists for a long time, prompting the inevitable suggestion that they are the data equivalent of what abhorred **goto** instructions represent on the control side. A surprisingly little-known article by Nori Suzuki [Suzuki 1982] explores whether a disciplined approach, using higher-level operations (in the same way that one avoids **goto** by sticking to the "structured programming" constructs of sequence, conditional and loop), could avoid the troubles of dynamic aliasing. Although the results are somewhat disappointing — by the author's own admission — the article is useful reading.

I am indebted to Ross Scaife from the University of Kentucky for help with rhetorical terms. See his page at *http*://*www*.*uky*.*edu*/*ArtsSciences*/*Classics*/*rhetoric*.*html*.

EXERCISES

E8.1 Books and authors

Starting from the various sketches given in this chapter, write classes *BOOK* and *WRITER* covering a useful view of books and their authors. Be sure to include the relevant routines (not just the attributes as in most of this chapter).

E8.2 Persons

Write a class *PERSON* covering a simple notion of person, with attributes *name* (a *STRING*), *mother*, *father* and *sibling* (describing the next older sibling if any). Include routines which will find (respectively) the list of names of ancestors, direct cousins, cousins direct or indirect, uncles or aunts, siblings-in-laws, parents-in-laws etc. of a given person. **Hint**: write recursive procedures (but make sure to avoid infinite recursion where the relations, for example direct or indirect cousin, are cyclic.).

E8.3 Notation design

Assume you are frequently using comparisons of the form *x*.*is_equal* (*y*) and want to simplify the notation to take advantage of infix features (applicable here since *is_equal* is a function with one argument). With an infix feature using some operator §, the call will be written *x* § *y*. This little exercise asks you to invent a symbol for §, compatible with the rules on infix operators. There are of course many possible answers, and deciding between them is partly (but only partly) a matter of taste.

See "Operator features", page 187 about infix features and permissible operators.

Hint: The symbol should be easy to remember and somehow suggest equality; but perhaps even more importantly it should be different enough from = to avoid mistakes. Here you can benefit from the study of C and C++ which, departing from mathematical tradition, use = for assignment rather than equality comparison, but for the latter operation introduce a similar-looking symbol, ==. The matter is made even more delicate by the rule that permits treating an assignment as an expression, whose value is the value being assigned to the target, and by the rule accepting values such as integers as boolean expressions, meaning true if non-zero, so that compilers will accept a text of the form

```
if (x = y) then …
```

although in most practical cases it is in error (mistakenly using = for ==), and will have the probably incorrect effect of assigning the value of *y* to *x*, returning true if and only if that value is non-zero.