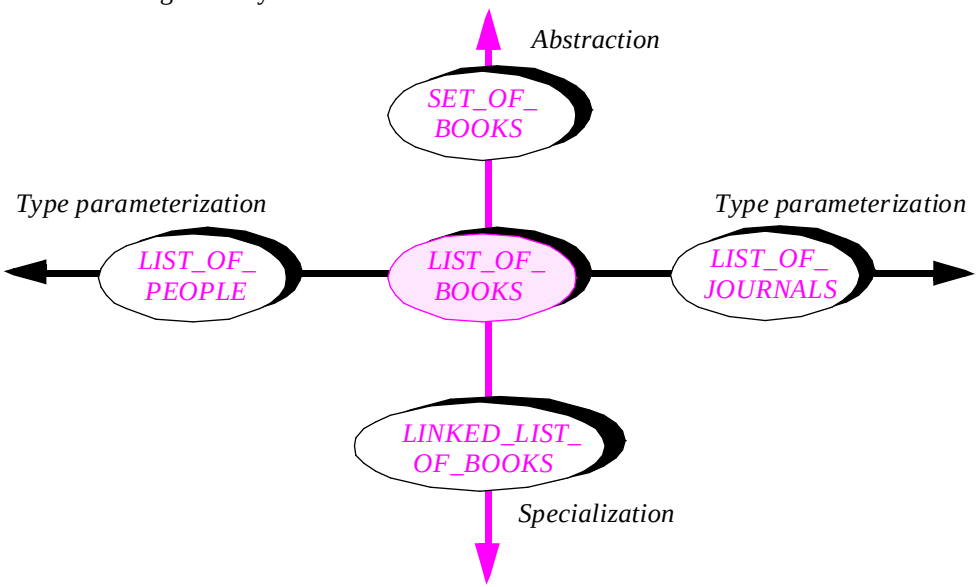
10

Genericity

From the merging of module and types concepts, we have been able to develop a powerful notion of class, which serves as the basis of the object-oriented method and can already, as it stands, enable us to do much. But to achieve our goals of extendibility, reusability and reliability we must make the class construct more flexible, an effort that will proceed in two directions. One, vertical in the figure below, represents abstraction and specialization; it will give rise to the study of inheritance in subsequent chapters. The present chapter studies the other dimension, horizontal in the figure: type parameterization, also known as genericity.

Dimensions of generalization



10.1 HORIZONTAL AND VERTICAL TYPE GENERALIZATION

With the mechanisms studied so far we have all that we need to write the class at the center of the figure, *LIST_OF_BOOKS*, of which an instance represents a list of book objects. We know what kinds of feature it would have: *put* to add an element, *remove* to delete an

element, *count* to find out how many elements are present and so on. But it is easy to see two ways of generalizing the notion of *LIST_OF_BOOKS*:

- Lists are a special case of "container" structure, of which other examples (among many) include trees, stacks and arrays. A more abstract variant might be described by a class *SET_OF_BOOKS*. A more specialized variant, covering a particular choice of list representation, might be described by a class *LINKED_LIST_OF_BOOKS*. This is the vertical dimension of our figure — the dimension of inheritance.

- Lists of books are a special case of lists of objects of any particular kind, of which other examples (among many) include lists of journals, lists of people, lists of integers. This is the horizontal dimension of our figure — the dimension of genericity, our topic for the rest of this chapter. By giving classes parameters representing arbitrary types, we will avoid the need to write many quasi-identical classes — such as *LIST_OF_BOOKS* and *LIST_OF_PEOPLE* — without sacrificing the safety afforded by static typing.

The relation between these two mechanisms is an elusive question for students of object-oriented concepts. Should inheritance and genericity be viewed as comrades or competitors in the rush towards more flexible software? That question is the subject of an appendix. In the present chapter we concentrate on genericity; this will also enable us to take a closer look at one of the most common examples of generic structure: arrays.

Appendix B.

10.2 THE NEED FOR TYPE PARAMETERIZATION

Genericity is not really a new concept in this discussion, although we have not yet seen it applied to classes. We encountered the idea a first time when reviewing traditional approaches to reusability; and when we studied the mathematical model — abstract data types — we saw the need to define an ADT as parameterized by types.

See "Genericity", page 96, and again "Genericity", page 131.

Generic abstract data types

Our working ADT example, *STACK*, was declared as *STACK* [*G*], meaning that any actual use requires you to specify an "actual generic parameter" representing the type of the objects stored in a particular stack. The name *G* as used in the ADT's specification stands for any possible type that these stack elements may have; it is called the **formal generic parameter** of the class. With this approach you can use a single specification for all possible stacks; the alternative, hard to accept, would be to have a class *INTEGER_STACK*, a class *REAL_STACK* and so on.

Any ADT describing "container" structures — data structures such as sets, lists, trees, matrices, arrays and many others that serve to keep objects of various possible types — will be similarly generic.

The same concerns, applied to the container classes of our software systems rather than to the container ADTs of our mathematical models, will yield a similar solution.

The issue

Let us keep the stack example, no longer as a mathematical ADT but as a software class. We know how to write a class *INTEGER_STACK* describing the notion of stack of integers. Features will include *count* (number of elements), *put* (push a new element), *item* (top element), *remove* (pop the top element), *empty* (is this stack empty?).

Type *INTEGER* will be used frequently in this class. For example it is the type of the argument of *put* and of the result of *item*:

```
put (element: INTEGER) is
            -- Push element on top.
        do … end

item: INTEGER is
            -- Item at top
        do … end
```

These appearances of type *INTEGER* follow from the rule of explicit declaration that we have used in developing the notation: any time you introduce an entity, denoting possible run-time objects, you must write an explicit type declaration for it, such as *element*: *INTEGER*. Here this means that you must specify a type for the query *item*, for the argument *element* of procedure *put*, and for other entities denoting possible stack elements.

But as a consequence you must write a different class for every sort of stack: *INTEGER_STACK*, *REAL_STACK*, *POINT_STACK*, *BOOK_STACK*… All such stack classes will be identical except for the type declarations of *item*, *element* and a few other entities: since the basic operations on a stack are the same regardless of the type of stack elements, nothing in the bodies of the various routines depends on the choice of *INTEGER*, *REAL*, *POINT* or *BOOK* as the type of stack element. For anyone concerned with reusability, this is not attractive.

The issue, then, is the contradiction that container classes seem to cause between two of the fundamental quality goals introduced at the beginning of this book:

- Reliability: retaining the benefits of type safety through explicit type declarations.
- Reusability: being able to write a single software element covering variants of a given notion.

The role of typing

Chapter 17.

Why insist on explicit type declarations (the first of the two requirements)? This is part of the general question of typing, to which an entire chapter is devoted later in this book. It is not too early to note the two basic reasons why an O-O notation should be statically typed:

- The *readability* reason: explicit declarations tell the reader, loud and clear, about the intended use of every element. This is precious to whoever — the original author, or someone else — needs to understand the element, for example to debug or extend it.

- The *reliability* reason: thanks to explicit type declarations, a compiler will be able to detect erroneous operations before they have had a chance to strike. In the fundamental operations of object-oriented computation, feature calls of the general form *x*.*f* (*a*, …), where *x* is of some type *TX*, the potential for mischief is manyfold: the class corresponding to *TX* might not have a feature called *f*; the feature might exist but be secret; the number of arguments might not coincide with what has been declared for *f* in the class; the type for *a* or another argument might not be compatible with what *f* expects. In all such cases, letting the software text go through unopposed — as in a language without static typechecking — would usually mean nasty consequences at run time, such as the program crashing with a diagnostic of the form "*Message not understood*" (the typical outcome in Smalltalk, a non-statically-typed O-O language). With explicit typing, the compiler will not let the erroneous construct through.

The key to software reliability, as was pointed out in the discussion of that notion, is prevention more than cure. Many studies have found that the cost of correcting an error grows astronomically when the time of detection is delayed. Static typing, which enables the early detection of type errors, is a fundamental tool in the quest for reliability.

Without these considerations we would not need explicit declarations, and so we would not need genericity. As a consequence the rest of this chapter only applies to *statically typed* languages, that is to say languages which require all entities to be declared and enforce rules enabling compilers to detect type inconsistencies prior to execution. In a non-statically-typed language such as Smalltalk, there is no role for genericity; this removes a language construct, but also removes any protection against schemes such as

```
my_stack.put (my_circle)
my_account := my_stack.item
my_account.withdraw (5000)
```

where an element is retrieved from the top of the stack and treated as if it were a bank account even though it is in reality (because of the first instruction) a circle, so that the software ends up trying to withdraw five thousand dollars from a circle on the screen.

Static typing protects us against such mishaps; combining it with the reusability requirement implies that we develop a mechanism for genericity.

10.3 GENERIC CLASSES

Reconciling static typing with the requirement of reusability for classes describing container structures means, as illustrated by the stack example, that we want both to:

- Declare a type for every entity appearing in the text of a stack class, including entities representing stack elements.
- Write the class so that it does not give out any clue about the elements' type, and hence that it can be used to build stacks of arbitrary elements.

At first sight these requirements seem irreconcilable but they are not. The first one commands us to declare a type; it does not assume that the declaration is exact! As soon as we have provided a type name, we will have pacified the type checking mechanism. ("Name your fear, and it will go away".) Hence the idea of genericity: to obtain a type-parameterized class, equip it with the name of a fictitious type, called the formal generic parameter.

Declaring a generic class

By convention the generic parameter will use the name *G* for Generic; this is a style recommendation, not a formal rule. If we need more generic parameters they will be called *H*, *I* and so on.

The syntax will include the formal generic parameters in square brackets after the class name, as with generic ADTs in a previous chapter. Here is an example:

```
indexing
        description: "Stacks of elements of an arbitrary type G"
class STACK [G] feature
        count: INTEGER
                -- Number of elements in stack
        empty: BOOLEAN is
                --Are there no items?
            do … end
        full: BOOLEAN is
                -- Is representation full?
            do … end
        item: G is
                -- Top element
            do … end
        put (x: G) is
                -- Add x on top.
            do … end
        remove is
                -- Remove top element.
            do … end
end -- class STACK
```

In the class, you may use a formal generic parameter such as *G* in declarations: not only for function results (as in *item*) and formal arguments of routines (as in *put*), but also for attributes and local entities.

Using a generic class

A client may use a generic class to declare entities of its own, such as an entity representing a stack. In such a case, the declaration must provide types, called **actual generic parameters** — as many as the class has formal generic parameters, here just one:

sp: *STACK* [*POINT*]

Providing an actual generic parameter to a generic class so as to produce a type, as here, is called a **generic derivation**, and the resulting type, such as *STACK* [*POINT*], is said to be generically derived.

A generic derivation both produces and requires a type:

- The result of the derivation, *STACK* [*POINT*] in this example, is a type.
- To produce this result, you need an existing type to serve as actual generic parameter, *POINT* in the example.

The actual generic parameter is an arbitrary type. Nothing prevents us, in particular, from choosing a type that is itself generically derived; assuming another generic class *LIST* [*G*], we can define a stack of lists of points:

slp: *STACK* [*LIST* [*POINT*]]

or even, using *STACK* [*POINT*] itself as the actual generic parameter, a stack of stacks of points:

ssp: *STACK* [*STACK* [*POINT*]]

There is no limit — other than suggested by the usual guideline that software texts should remain simple — to the depth of such nesting.

Terminology

To discuss genericity, we need to be precise about the terms that we use:

- To produce a type such as *STACK* [*POINT*] by providing a type, here *POINT*, as actual generic parameter for a generic class, here *STACK*, is to perform a generic derivation. You may encounter the term “generic instantiation” for that process, but it is confusing because “instantiation” normally denotes a run-time event, the production of an object — an instance — from its mold (a class). Generic derivation is a static mechanism, affecting the text of the software, not its execution. So it is better to use completely different terms.
- This book uses the term “parameter” exclusively to denote the types that parameterize generic classes, never to denote the values that a routine call may pass to that routine, called *arguments*. In traditional software parlance “parameter” and “argument” are synonymous. Although the decision of which term to use for routines and which for generic classes is a matter of convention, it is desirable to stick to a consistent rule to avoid any confusion.

Type checking

Using genericity, you can guarantee that a data structure will only contain elements of a single type. Assuming a class contains the declarations

sc: *STACK* [*CIRCLE*]; *sa*: *STACK* [*ACCOUNT*]; *c*: *CIRCLE*; *a*: *ACCOUNT*

then the following are valid instructions in routines of that class:

```
sc.put (c)          -- Push a circle onto a stack of circles
sa.put (a)          -- Push an account onto a stack of accounts
c := sc.item        -- Assign to a circle entity the top of a stack of circles
```

but each of the following is invalid and will be rejected:

```
sc.put (a);         -- Attempt to push an account onto a stack of circles
sa.put (c);         -- Attempt to push a circle onto a stack of accounts
c := sa.item        -- Attempt to access as a circle the top of a stack of accounts
```

This will rule out erroneous operations of the kind described earlier, such as attempting to withdraw money from a circle.

The type rule

The type rule that makes the first set of examples valid and the second invalid is intuitively clear but let us make it precise.

First the basic non-generic rule. Consider a feature declared as follows, with no use of any formal generic parameter, in a non-generic class *C*

f (*a*: *T*): *U* **is** …

This will be the Feature Application rule, page 473.

Then a call of the form *x*.*f* (*d*), appearing in an arbitrary class *B* where *x* is of type *C*, will be typewise correct if and only if: *f* is available to *B* — that is to say, generally exported, or exported selectively to a set of classes including *B*; and *d* is of type *T*. (When we bring inheritance into the picture we will also accept *d* if its type is based on a descendant of *T*.) The result of the call — there is a result since the example assumes that *f* is a function — is of type *U*.

Now assume that *C* is generic, with *G* as formal generic parameter, and has a feature

h (*a*: *G*): *G* **is** …

A call to *h* will be of the form *y*.*h* (*e*) for some entity *y* that has been declared, for some type *V*, as

y: *C* [*V*]

The counterpart of the non-generic rule is that *e* must now be of type *V* (or a compatible type in the sense of inheritance), since the corresponding formal argument *a* is declared as being of type *G*, the formal generic parameter, and in the case of *y* we may consider *G*, wherever it appears in class *C*, as a placeholder for *V*. Similarly, the result of the call will be of type *V*. The earlier examples all follow this model: a call of the form *s*.*put* (*z*) requires an argument *z* of type *POINT* if *s* is of type *STACK* [*POINT*], *INTEGER*

if *s* is of type *STACK* [*INTEGER*]; and *s.item* returns a result of type *POINT* in the first case and *INTEGER* in the second.

These examples involve features with zero or one argument, but the rule immediately extends to an arbitrary number of arguments.

Operations on entities of generic types

In a generic class *C* [*G, H, …*] consider an entity whose type is one of the formal generic parameters, for example *x* of type *G*. When the class is used by a client to declare entities, *G* may ultimately represent any type. So any operation that the routines of *C* perform on *x* must be applicable to all types. This leaves only five kinds of operation:

Uses of entities of a formal generic type

The valid uses for an entity *x* whose type *G* is a formal generic parameter are the following:

G1 • Use of *x* as left-hand side in an assignment, *x* := *y*, where the right-hand side expression *y* is also of type *G*.

G2 • Use of *x* as right-hand side of an assignment *y* := *x*, where the left-hand side entity *y* is also of type *G*.

G3 • Use of *x* in a boolean expression of the form *x* = *y* or *x* /= *y*, where *y* is also of type *G*.

G4 • Use of *x* as actual argument in a routine call corresponding to a formal argument declared of type *G*, or of type *ANY*.

G5 • Use as target of a call to a feature of *ANY*.

In particular, a creation instruction of the form !! *x* is illegal, since we know nothing about the creation procedures, if any, defined for possible actual generic parameters corresponding to *G*.

Cases G4 and G5 refer to class *ANY*. Mentioned a few times already, this class contains features that all classes will inherit. So you can be assured that whatever actual type *G* represents in a particular generic derivation will have access to them. Among the features of *ANY* are all the basic operations for copying and comparing objects: *clone*, *copy*, *equal*, *copy*, *deep_clone*, *deep_equal* and others. This means it is all right, for *x* and *y* of a formal generic type *G*, to use instructions such as

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

```
x.copy (y)
x := clone (y)
if equal (x, y) then …
```

Ignoring *ANY*, case G4 permits a call *a.f* (*x*) in a generic class *C* [*G*] if *f* takes a formal argument of type *G*. In particular *a* could be of type *D* [*G*], where *D* is another generic class, declared as *D* [*G*] with a feature *f* that takes an argument of type *G*, here

To check creek-clarity do exercise E10.3, page 331.

denoting *D*'s own formal generic parameter. (If the preceding sentence does not immediately make sense, please read it once more and it will, I hope, soon seem as clear as a mountain creek!)

Types and classes

We have learned to view the class, the central notion in object technology, as the product of the corporate merger between the module and type concepts. Until we had genericity, we could say that every class is a module and is also a type.

With genericity, the second of these statements is not literally true any more, although the nuance will be small. A generic class declared as *C* [*G*] is, rather than a type, a type pattern covering an infinite set of possible types; you can obtain any one of these by providing an actual generic parameter — itself a type — corresponding to *G*.

This yields a more general and flexible notion. But for what we gain in power we have to pay a small price in simplicity: only through a small abuse of language can we continue talking, if *x* is declared of type *T*, about "the features of *T*" or "the clients of *T*"; other than a class, *T* may now be a generically derived type *C* [*U*] for some generic class *C* and some type *U*. Of course there is still a class involved — class *C* —, which is why the abuse of language is acceptable.

When we need to be rigorous the terminology is the following. Any type *T* is associated with a class, the **base class** of *T*, so that it is always correct to talk about the features or clients of *T*'s base class. If *T* is a non-generic class, then it is its own base class. If *T* is a generic derivation of the form *C* [*U*, …], then the base class of *T* is *C*.

"ANCHORED DECLARATION", 16.7, page 598.

The notion of base class will again be useful when we introduce yet another kind of type, also (like all others in the O-O approach) based on classes, but indirectly: anchored types.

10.4 ARRAYS

As a conclusion to this discussion it is useful to take a look at a very useful example of container class: *ARRAY*, which represents one-dimensional arrays.

Arrays as objects

The notion of array is usually part of a programming language's definition. But with object technology we do not need to burden the notation with special predefined constructs: an array is just a container object, an instance of a class which we may call *ARRAY*.

A better version of the class, relying on assertions, appears in "Arrays revisited", page 373.

ARRAY is a good example of generic class. Here is a first outline:

```
indexing
        description: "Sequences of values, all of the same type or of a conforming one, %
                    %accessible through integer indices in a contiguous interval"
class ARRAY [G] creation
        make
```

```
feature
    make (minindex, maxindex: INTEGER) is
            -- Allocate array with bounds minindex and maxindex
            -- (empty if minindex > maxindex)
        do … end
    lower, upper, count: INTEGER
            -- Minimum and maximum legal index; array size.
    put (v: G; i: INTEGER) is
            -- Assign v to the entry of index i
        do … end
    infix "@", item (i: INTEGER): G is
            -- Entry of index i
        do … end
end -- class ARRAY
```

To create an array of bounds *m* and *n*, with *a* declared of type *ARRAY* [*T*] for some type *T*, you will execute the creation instruction

```
!! a.make (m, n)
```

To set the value of an array element you will use procedure *put*: the call *a.put* (*x*, *i*) sets the value of the *i*-th element to *x*. To access the value of an element you will use function *item* (the synonym **infix** "@" will be explained shortly), as in

```
x := a.item (i)
```

Here is a sketch of how you might use the class from a client:

```
pa: ARRAY [POINT]; p1: POINT; i, j: INTEGER
…
!! pa.make (–32, 101)      -- Allocate array with the bounds shown.
pa.put (p1, i)             -- Assign p1 to entry of index i.
…
p1 := pa.item (j)          -- Assign to p1 the value of entry of index j.
```

In conventional (say Pascal) notation, you would write

```
pa [i] := p1      for   pa.put (i, p1)
p1 := pa [i]      for   p1 := pa.item (i)
```

Array properties

A few observations on the preceding class:

- Similar classes exist for arrays with more dimensions: *ARRAY2* etc.
- Feature *count* may be implemented as either an attribute or a function, since it satisfies *count = upper – lower+1*. This is expressed in the actual class by an invariant, as explained in the next chapter.

- More generally, assertion techniques will allow us to associate precise consistency conditions with *put* and *item*, expressing that calls are only valid if the index *i* is between *lower* and *upper*.

The idea of describing arrays as objects and *ARRAY* as a class is a good example of the unifying and simplifying power of object technology, which helps us narrow down the notation (the design or programming language) to the bare essentials and reduce the number of special-purpose constructs. Here an array is simply viewed as an example of a container structure, with its own access method represented by features *put* and *item*.

Since *ARRAY* is a normal class, it can fully participate in what an earlier chapter called the object-oriented games; in particular other classes can inherit from it. A class *ARRAYED_LIST* describing the implementation of the abstract notion of list by arrays can be a descendant of both *LIST* and *ARRAY*. We will study many such constructions.

As soon as we learn about assertions we will take this unifying approach even further; thanks to preconditions, we will be able to handle through the normal concepts of the object-oriented method one more problem traditionally thought to require special-purpose mechanisms: run-time bounds checking (monitoring array accesses to enforce the rule that all indices must lie between the bounds).

Efficiency considerations

The fear may arise that all this elegance and simplicity could cause performance to take a hit. One of the primary reasons developers use arrays in traditional approaches is that the basic operations — accessing or modifying an array element known through its index — are fast. Are we now going to pay the price of a routine call every time we use *item* or *put*?

We do not need to. That *ARRAY* looks to the unsuspecting developer as a normal class does not prevent the compiler from cheating — from relying on some insider information. This information enables the compiler to detect calls to *item* and *put* and hijack them so as to generate exactly the same code that a Fortran, Pascal or C compiler would produce for equivalent instructions as shown above (*p1* := *pa* [*i*] and *pa* [*i*] := *p1* in Pascal syntax). So the developer will gain the best of both worlds: the uniformity, generality, simplicity, and ease of use of the O-O solution; and the performance of the traditional solution.

> The compiler's job is not trivial. As will be clear in the study of inheritance, it is possible for a descendant of class *ARRAY* to redefine any feature of the class, and such redefinitions may be called indirectly through dynamic binding. So compilers must perform a thorough analysis to check that the replacement is indeed correct. Today's compilers from ISE and other companies can indeed, for a typical array-intensive computation typical of large scientific software, generate code whose efficiency matches that of hand-written C or Fortran code.

An infix synonym

Class *ARRAY* provides the opportunity to introduce a small facility that, although not directly related to the other topics of this chapter, will be useful in practice. The declaration of feature *item* actually reads

infix "@", *item* (*i*: *INTEGER*): *G* **is** …

This introduces two feature names **infix** "@" and *item* as synonyms, that is to say as denoting the same feature, given by the declaration that follows. In general, a feature declaration of the form

a, *b*, *c*, … "Feature description"

is considered as an abbreviation for a sequence of declarations of the form

a "Feature description"
b "Feature description"
c "Feature description"
…

all for the same "Feature description". This is applicable to attributes (where the "Feature description" is of the form : *some_type*) as well as routines (where it reads **is** *routine_body*).

The benefit in this example is that you have a simpler notation for array access. Although consistent with the access mechanisms for other data structures, the notation *a*.*item* (*i*) is more wordy than the traditional *a* [*i*] found, with some variants, in Pascal, C, Fortran and so on. By defining **infix** "@"as a synonym, you can actually beat traditional languages at their own terseness game by writing an array element as *a* @ *i* (the supreme dream: undercutting — by one keystroke — even C!). Note again that this is not a special language mechanism but the straightforward application of a general O-O concept, operator features, combined here with the notion of synonym.

The notion of infix feature was introduced in "Operator features", page 187.

10.5 THE COST OF GENERICITY

As always, we need to make sure that the object-oriented techniques that we introduce for reusability, extendibility and reliability do not imply a performance overhead. The question has just been raised and answered for arrays; but we need to examine it for the genericity mechanism at large. How much will genericity cost?

The concern arises in particular because of the experience of C++, where genericity (known as the *template* mechanism) was a late addition to the language, causing performance difficulties. It appears that some compiler implementations take the idea of parameterization literally, generating a different copy of the class features for each actual generic parameter! As a consequence the literature warns C++ programmers of the dangers of using templates too generously:

> *Template instantiation time is already an issue for some C++ users… If a user creates a List<int>, a List<String>, a List<Widget>, and a List<Blidget> (where Widget and Blidget are user-defined classes), and calls head, tail, and*

From: Martin Carroll & Margaret Ellis, "Reducing Instantiation Time", in "C++ Report", vol. 6, no. 5, July-August 1994, pages 14, 16 and 64. List<T> would be LIST [T] in the notation of this book.

> *insert on all four objects, then each of these functions will be instantiated* [in the sense of generically derived] *four times. A widely useful class such as List might be instantiated in user programs with many different types, causing many functions to be instantiated. Thus, a significant amount of code might be generated for the* [features of] the *List template* [class].

The authors of this advice (both with respected C++ expertise from the original AT&T group, one of them co-author of the official C++ reference [Ellis 1990]) go on proposing various techniques for avoiding template derivation. But developers should of course be protected from such concerns. Genericity should not imply code duplication; it is possible, with appropriate language design and a good compiler, to generate a single target code for any generic class, so that all of the following will be small or zero:

- Effect on compilation time.
- Effect on the size of the generated code.
- Effect on execution time.
- Effect on execution space.

When working in such an environment, you can use the full power of genericity without any fear of unpleasant effects on either compile-time or at run-time performance.

10.6 DISCUSSION: NOT DONE YET

The presentation of genericity has introduced the basic ideas. But, as you may have noticed, it leaves two important questions unanswered.

First, in our effort to guarantee type safety, we may have erred on the conservative side. We will be prevented from pushing a bank account onto a *STACK* [*CIRCLE*], or a point onto a *STACK* [*ACCOUNT*]. This is what we want: it is hard to imagine what kind of application — other than general-purpose utilities such as a database management system — would need to handle a stack containing both points and bank accounts. But what about a graphics application asking for a stack that contains a few circles, a few rectangles, a few points? This request seems quite reasonable, and we cannot accommodate it; the type system defined so far will reject the call *figure_stack*.*put* (*that_point*) if *figure_stack* has been declared of type *STACK* [*FIGURE*] and *that_point* of any type other than *FIGURE*. We can give a name to such structures: **polymorphic data structures**. The challenge will be to support them without renouncing the benefits of type safety.

Second, our generic parameters represent arbitrary types. This is fine for stacks and arrays, since any object is by essence “stackable” and storable into an array. But when we come to structures such as vectors, we will want to be able to add two vectors, requiring that we can also add two vector elements; and if we want to define a hash table class, we will need the certainty that a hash function is applicable to every table element. Such a form of genericity, whereby the formal generic parameter does not any more stand for an arbitrary type, but represents a type guaranteed to offer certain operations, will be called **constrained genericity**.

For both of these problems, the object-oriented method will provide simple and elegant solutions, both based on combining genericity with inheritance.

10.7 KEY CONCEPTS INTRODUCED IN THIS CHAPTER

- Classes may have formal generic parameters representing types.
- Generic classes serve to describe general container data structures, implemented in the same way regardless of the elements they contain.
- Genericity is only needed in a typed language, to ensure statically checkable type safety.
- A client of a generic class must provide actual types for the formal parameters.
- The only permitted operations on an entity whose type is a formal generic parameter are operations applicable to every type. The entity may serve as left- or right-hand side of an assignment, actual routine argument, or operand of an equality or inequality test. It may also participate in universally applicable features such as cloning and object equality testing.
- The notion of array can be covered by a generic library class, without any specific language mechanism but also without any loss in run-time performance.
- More flexible advanced uses of genericity — polymorphic data structures, constrained genericity — require the introduction of inheritance.

10.8 BIBLIOGRAPHICAL NOTES

An early language supporting genericity was LPG [Bert 1983]. Ada made the concept widely known through its generic package mechanism.

For references on Ada see chapter 33.

Genericity has also been introduced in formal specification languages such as Z, CLEAR and OBJ-2, to which references appear in the chapter on abstract data types. The generic mechanism described here was derived from the mechanism introduced in an early version of Z [Abrial 1980] [Abrial 1980a] and extended in M [M 1985b].

Page 160.

Aside from the notation of this book, one of the first object-oriented languages to offer genericity was DEC's Trellis language [Schaffert 1986].

EXERCISES

E10.1 Constrained genericity

This exercise is a little peculiar since it asks you a question to which a detailed answer appears later in the book. Its aim is to get you thinking about the proper language structures, and compare your answer to what will be introduced later. It will only be worthwhile if you are new to this problem and have not yet seen the object-oriented solution. Familiarity with how the problem is handled in other approaches, notably Ada, may be helpful but is not required.

The question is about constrained genericity, a need that was presented in the discussion section. Devise a language mechanism, compatible with the spirit of the object-oriented approach and with the notations seen so far, that will address constrained genericity by enabling the author of a generic class to specify that valid actual generic parameters must possess certain operations.

E10.2 Two-dimensional arrays

Using class *ARRAY* both as inspiration and as basis for the implementation, write a generic class *ARRAY2* describing two-dimensional arrays.

E10.3 Using your own formal generic parameter as someone else's actual

Construct an example in which a routine of a generic class *C* [*G*] calls a routine declared in another generic class *D* [*G*] as taking a formal argument of type *G*.