26
Ada

John G. P. Barnes
John Barnes Informatics

26.1 Introduction

Ada 95 is a comprehensive high-level programming language especially suited for the professional development of large or critical programs for which correctness and robustness are major considerations. This chapter gives a brief account of the background to the development of Ada 95 (and its predecessor Ada 83), its place in the overall language scene, and then outlines some of the major features of the language.

Ada 95 is a direct descendant of, and highly compatible with, Ada 83, which was originally sponsored by the U.S. Department of Defense for use in the embedded system application area. Ada 83 became an ANSI standard in 1983 and an ISO standard in 1987.

Time and technology do not stand still and accordingly, after several years' use, it was decided that the Ada 83 language standard should be revised in the light of experience and changing requirements. One major change was that Ada was now being used for many areas of application other than the embedded systems for which it was originally designed. Much had also been learned about new programming paradigms such as object oriented programming.

A set of requirements for the revision was published in 1990. An interesting aspect of the requirements is that they cover a number of specialized application areas. It seemed likely that it would be too costly to implement a language meeting all these requirements in their entirety on every architecture. On the other hand, one of the strengths of Ada is its portability, and the last thing anyone wanted was the anarchy of uncontrolled subsets. As a consequence, Ada 95 comprises a core language plus a small number of specialized annexes. All compilers have to implement the core language and vendors can choose to implement zero, one, or more annexes according to the needs of their markets.

Having established the requirements, the language design was contracted to Intermetrics Inc. under the technical leadership of S. Tucker Taft with continued strong interaction with the user community. The new ISO standard was published on February 15, 1995 and so the revised language is called Ada 95.

26.1.1 Software Engineering

It should not be thought that Ada is just another programming language. Ada is about software engineering, and by analogy with other branches of engineering it can be seen that there are two main problems with the development of software: the need to reuse software components as much as possible and the need to establish disciplined ways of working.

As a language, Ada (and hereafter by Ada we mean Ada 95) largely solves the problem of writing reusable software components (or at least, because of its excellent ability to prescribe interfaces it provides an enabling technology in which reusable software can be written).

Many years' use of Ada 83 has shown that Ada is living up to its promise of providing a language which can reduce the cost of both the initial development of software and its later maintenance. The main advantage of Ada is simply that it is reliable. The strong typing and related features ensure that programs contain few surprises; most errors are detected at compile time and of those that remain many are detected by run-time constraints. This aspect of Ada considerably reduces the costs and risks of program development compared, for example, with C and its derivatives such as C++. Moreover an Ada compilation system includes the facilities found in separate tools such as lint and make for C. Even if Ada is seen as just another programming language, it reaches parts of the software development process that other languages do not reach.

The essence of Ada 95 is that it adds extra flexibility to the inherent reliability of Ada 83, thereby producing an outstanding language suitable for the development needs of applications well into the new millenium.

Two kinds of applications stand out where Ada is particularly relevant. The very large and the very critical. Very large applications, which inevitably have a long lifetime, require the cooperative effort of large teams. The information hiding properties of Ada, and especially the way in which integrity is maintained across compilation unit boundaries, are invaluable in enabling such developments to progress smoothly. Furthermore, if and when the requirements change and the program has to be modified, the structure, and especially the readability of Ada, enables rapid understanding of the original program even if it has been modified by a different team.

Very critical applications are those that just have to be correct otherwise people or the environment may be damaged. Obvious examples occur in avionics, railway signaling, process control, and medical applications. Such programs may not be large, but they have to be very well understood and often mathematically proven to be correct. The full flexibility of Ada is not appropriate in this case, but the intrinsic reliability of the strongly typed kernel of the language is exactly what is required. Indeed many certification agencies dictate the properties of acceptable languages and while they do not always explicitly demand a subset of Ada, nevertheless the same properties are not provided by any other practically available language.

Ada is thus very appropriate for avionics applications which embrace both the large and the critical.

26.1.2 Abstraction and Freedom

The evolution of programming languages essentially concerns the use of abstraction to hide unnecessary and harmful details of the program.

Thus "expression abstraction" in any language (such as Fortran or Pascal) hides the use of the machine registers to evaluate expressions, and "control abstraction" in Algol and Pascal hides the goto's and labels which had to be explicitly used in early versions of languages such as Fortran. A more recent advance is "data abstraction". This means separating the details of the representation of data from the abstract operations defined upon the data.

Older languages take a very simple view of data types. In all cases the data are directly described in numerical terms. Thus if the data to be manipulated are not really numerical (they could be traffic light colors) then some mapping of the abstract type must be made by the programmer into a numerical type (usually integer). This mapping is purely in the mind of the programmer and does not appear in the written program except perhaps as a comment.

Pascal introduced a certain amount of data abstraction as instanced by the enumeration type. Enumeration types allow us to talk about the traffic light colors in their own terms without having to know how they are represented in the computer.

Another form of data abstraction concerns visibility. It has long been recognized that the traditional block structure of Algol and Pascal is not adequate. For example, it is not possible in Pascal to write two procedures to operate on some common data and make the procedures accessible without also making the data directly accessible. Many languages have provided control of visibility through separate compilation; this technique is adequate for medium-sized systems, but since the separate compilation facility usually depends upon some external system, total control of visibility is not gained. Ada 83 was probably the first practical language to bring together these various forms of data abstraction.

Another language which made an important contribution to the development of data abstraction is Simula 67 with its concept of class. This leads us into the paradigm now known as Object Oriented Programming (OOP) which is currently in vogue. There seems to be no precise definition of OOP, but its essence is a flexible form of data abstraction providing the ability to define new data abstractions in terms of old ones and allowing dynamic selection of types.

All types in Ada 83 are static and thus Ada 83 is not classed as a truly Object Oriented language but as an Object Based language. However, Ada 95 includes all the essential functionality associated with OOP such as polymorphism and type extension.

We are probably too close to the current scene to achieve a proper perspective. It remains to be seen just how useful "object abstraction" actually is. Indeed it might well be that inheritance and other aspects of OOP turn out to be unsatisfactory by obscuring the details of types although not hiding them completely; this could be argued to be an abstraction leak making the problems of program maintenance harder if OOP is overused.

Another concept relevant to the design of languages is freedom. Freedom takes two forms. Freedom to do whatever we want on the one hand, and freedom from dangers and difficulties on the other. In general terms, modern society seems obsessed with freedom to do one's own thing and is less concerned with freedom from unpleasant consequences.

In terms of a programming language we need both freedoms at appropriate points. For large parts of a program we need freedom from inadvertant errors; this is best provided by a controlled framework in which the details of the machine and other parts of the system are hidden through various forms of abstraction. However, there are areas where freedom to get down to the raw hardware is vital; this especially applies to embedded applications where access to interrupt mechanisms and autonomous transfer of data are vital for proper responses to external events.

The merit of Ada is that it provides both kinds of freedom: freedom from errors by the use of abstraction and yet freedom to get at the underlying machine and other systems when necessary.

A brief survey of how Ada relates to other languages would not be complete without mention of C and C++. These have a completely different evolutionary trail than the classic Algol-Pascal-Ada route.

The origin of C can be traced back to the CPL language devised in the early 1960s. From it emerged the simple system programming language BCPL and from that B and then C. The essence of BCPL was the array and pointer model which abandoned any hope of strong typing and (with hindsight) a proper mathematical model of the mapping of the program onto a computing engine. Even the use of : = for assignment was lost in this evolution which reverted to the confusing use of = as in Fortran. About the only feature of the elegant CPL code remaining in C is the unfortunate braces {} and the associated compound statement structure which has now been abandoned by all other languages in favor of the more reliable bracketed form originally proposed by Algol 68.

C is an example of a language which almost has all the freedom to do things, but with little freedom from difficulties. Of course there is a need for a low-level systems language with functionality like C. It is, however, unfortunate that the interesting structural ideas in C++ have been grafted onto the fragile C foundation. As a consequence, although C++ has many important capabilities for data abstraction, including inheritance and polymorphism, it is all too easy to break these abstractions and create programs that violently misbehave or are exceedingly hard to understand and maintain.

The designers of Ada 95 have striven to incorporate the positive dynamic facilities of the kind found in C++ onto the firm foundation provided by Ada 83. Ada 95 is thus an important advance along the evolution of abstraction. It incorporates full object abstraction in a way that is highly reliable without incurring excessive run-time costs.

26.1.3 From Ada 83 to Ada 95

In this section (and especially for the benefit of those familiar with Ada 83), we briefly survey the main changes from Ada 83 to Ada 95. As we said above, one of the great strengths of Ada 83 is its reliability. The strong typing ensures that most errors are detected at compile time while many of those remaining are detected by various checks at run time. Moreover, the compile-time checking extends across compilation unit boundaries. This reliability aspect of Ada considerably reduces the costs and risks of program development (especially for large programs) compared with weaker languages which do not have such a rigorous model.

However, after a number of years' experience it became clear that some improvements were necessary to completely satisfy the present and the future needs of users from a whole variety of application areas. Four main areas were perceived as needing attention.

- Object oriented programming. Recent experience with other languages has shown the benefits of the object oriented paradigm. This provides much flexibility and, in particular, it enables a program to be extended without editing or recompiling existing and tested parts of it.
- Program libraries. The library mechanism is one of Ada's great strengths. Nevertheless the flat structure in Ada 83 is a hindrance to fine visibility control and to program extension without recompilation.
- Interfacing. Although Ada 83 does have facilities to enable it to interface to external systems written in other languages, these have not proved to be as flexible as they might. For example, it has been particularly awkward to program call-back mechanisms which are very useful, especially when using graphical user interfaces.
- Tasking. The Ada 83 rendezvous model provides an advanced description of many paradigms. However, it has not turned out to be entirely appropriate for shared data situations where a static monitor-like approach brings performance benefits. Furthermore, Ada 83 has a rather rigid approach to priorities and it is not easy to take advantage of recent deeper understanding of scheduling theory which has emerged since Ada was first designed.

The first three topics are really all about flexibility, and so a prime goal of the design of Ada 95 has been to give the language a more open and extensible feel without losing its inherent integrity and efficiency. In other words to get a better balance of the two forms of freedom.

The additions to Ada 95 which contribute to this more flexible feel are the extended or tagged types, the hierarchical library, and the greater ability to manipulate pointers or references. The tagged types and hierarchical library together provide very powerful tools for programming by extension.

In the case of the tasking model, the introduction of protected types allows a more efficient implementation of standard paradigms of shared data access. This brings with it the benefits of speed provided by low-level primitives such as semaphores without the risks incurred by the use of such unstructured primitives. Moreover, the clearly data oriented view brought by the protected types fits in naturally with the general spirit of the object oriented paradigm. Other improvements to the tasking model allow a more flexible response to interrupts and other changes of state.

The remainder of this chapter is a brief survey of most of the key features of Ada 95. Some will be familiar to those who know Ada 83, but much will be new or appear in a new light.

26.2 Key Concepts

Ada is a large language since it addresses many important issues relevant to the programming of practical systems in the real world. It is, for instance, much larger than Pascal which, unless it extended in some way, is really only suitable for training purposes (for which it was designed) and for small

personal programs. Similarly, Ada is much larger than C although perhaps about the same size as C++. But a big difference is the stress which Ada places on integrity and readability. Some of the key issues in Ada are

- Readability—it is recognized that professional programs are read much more often than they are written. It is important therefore to avoid an overly terse notation which, although allowing a program to be written down quickly, makes it almost impossible to be read except perhaps by the original author soon after it was written.
- Strong typing—this ensures that each object has a clearly defined set of values and prevents confusion between logically distinct concepts. As a consequence, many errors are detected by the compiler which in other languages would have led to an executable but incorrect program.
- Programming in the large—mechanisms for encapsulation, separate compilation, and library management are necessary for writing portable and maintainable programs of any size.
- Exception handling—it is a fact of life that programs of consequence are rarely perfect. It is necessary to provide a means whereby a program can be constructed in a layered and partitioned way so that the consequences of unanticipated events in one part can be contained.
- Data abstraction—as mentioned earlier, extra portability and maintainability can be obtained if the details of the representation of data can be kept separate from the specifications of the logical operations on the data.
- Object oriented programming—in order to promote the reuse of tested code, the type of flexibility associated with OOP is important. Type extension (inheritance), polymorphism, and late binding are all desirable especially if achieved without loss of type integrity.
- Tasking—for many applications it is important that the program be conceived as a series of parallel activities rather than just as a single sequence of actions. Building appropriate facilities into a language rather than adding them later via calls to an operating system gives better portability and reliability.
- Generic units—in many cases the logic of part of a program is independent of the types of the values being manipulated. A mechanism is therefore necessary for the creation of related pieces of program from a single template. This is particularly useful for the creation of libraries.
- Interfacing—programs do not live in isolation and it is important to be able to communicate with systems written in other languages.

An overall theme in designing Ada was concern for the programming process. Programming is a human activity and a language should be designed to be helpful. An important aspect of this is enabling errors to be detected early in the overall process. For example, care was taken that wherever possible a single typographical error would result in a program that does not compile rather than in a program that still compiles but does the wrong thing.

26.2.1 Overall Structure

One of the most important objectives of software engineering is to be able to reuse existing pieces of a program so that the effort of writing new coding is kept to a minimum. The concept of a library of program components naturally emerges, and an important aspect of a programming language is therefore its ability to express how to use the items in a library.

Ada recognizes this situation and introduces the concept of library units. A complete Ada program is assembled as a main subprogram (itself a library unit) which calls upon the services of other library units.

The main subprogram takes the form of a procedure of an appropriate name. The service library units can be subprograms (procedures or functions) but they are more likely to be packages. A package is a group of related items such as subprograms but may contain other entities as well.

Suppose we wish to write a program to print out the square root of some number. We can expect various library units to be available to provide us with a means of computing square roots and doing input and output. Our job is merely to write a main subprogram to incorporate these services as we wish. We will suppose that the square root can be obtained by calling a function in our library whose name is `Sqrt`. We will also suppose that our library includes a package called `Simple_IO` containing various simple input-output facilities. These facilities might include procedures for reading numbers, printing numbers, printing strings of characters, and so on.

Our program might look like

```
with Sqrt, Simple_IO;
procedure Print_Root is
  use Simple_IO;
  X: Float;
begin
  Get(X);
  Put(Sqrt(X));
end Print_Root;
```

The program is written as a procedure called `Print_Root` preceded by a *with* clause giving the names of the library units which it wishes to use. Between **is** and **begin** we can write declarations, and between **begin** and **end** we write statements. Broadly speaking, declarations introduce the entities we wish to manipulate and statements indicate the sequential actions to be performed.

We have introduced a variable `X` of type `Float` which is a predefined language type. Values of this type are a set of certain floating point numbers and the declaration of `X` indicates that `X` can have values only from this set. In our example a value is assigned to `X` by calling the procedure `Get` which is in our package `Simple_IO`.

Writing

```
use Simple_IO ;
```

gives us immediate access to the facilities in the package `Simple_IO`. If we had omitted this use clause we would have had to write

```
Simple_IO.Get(X);
```

in order to indicate where `Get` was to be found.

The procedure then contains the statement

```
Put(Sqrt(X));
```

which calls the procedure `Put` in the package `Simple_IO` with a parameter which in turn is the result of calling the function `Sqrt` with the parameter `X`.

Some small-scale details should be noted. The various statements and declarations all terminate with a semicolon; this is unlike some other languages such as Pascal where semicolons are separators rather than terminators. The program contains various identifiers such as **procedure**, `Put` and `X`. These fall into two categories. A few identifiers (69 in fact) such as **procedure** and **is** are used to indicate the structure of the program; they are reserved and can be used for no other purpose. All others, such as `Put` and `X`, can be used for whatever purpose we desire. Some of these, notably `Float` in our example, have a predefined meaning but we can nevertheless reuse them if we so wish although it might be confusing to do so. For clarity we write the reserved words in lower-case bold and capitalize the others. This is purely a notational convenience; the language rules do not distinguish the two cases except when we consider the manipulation of characters themselves. Note also how the underline character is used to break up long identifiers into meaningful parts.

Finally, observe that the name of the procedure, Print_Root, is repeated between the final **end** and the terminating semicolon. This is optional but is recommended so as to clarify the overall structure, although this is obvious in a small example such as this.

Our program is very simple; it might be more useful to enable it to cater for a whole series of numbers and print out each answer on a separate line. We could stop the program somewhat arbitrarily by giving it a value of zero.

```
with Sqrt, Simple_IO;
procedure Print_Roots is
  use Simple_IO;
  X: Float;
begin
  Put("Roots of various numbers");
  New_Line(2);
  loop
    Get(X);
    exit when X = 0.0;
    Put(" Root of ");
    Put(X);
    Put(" is ");
    if X < 0.0 then
      Put(" not calculable ");
    else
      Put(Sqrt(X));
    end if;
    New_Line;
  end loop;
  New_Line;
  Put("Program finished");
  New_Line;
end Print_Roots;
```

The output has been enhanced by the calls of further procedures New_Line and Put in the package Simple_IO. A call of New_Line will output the number of new lines specified by the parameter (which is of the predefined type Integer); the procedure New_Line has been written in such a way that if no parameter is supplied then a default value of 1 is assumed. There are also calls of Put with a string as argument. This is in fact a different procedure from the one that prints the number X. The compiler knows which is which because of the different types of parameters. Having more than one procedure with the same name is known as overloading. Note also the form of the string; this is a situation where the case of the letters does matter.

Various new control structures are also introduced. The statements between **loop** and **end loop** are repeated until the condition X = 0.0 in the **exit** statement is found to be true; when this is so the loop is finished and we immediately carry on after **end loop**. We also check that X is not negative; if it is we output the message "not calculable" rather than attempting to call Sqrt . This is done by the **if** statement; if the condition between **if** and **then** is true, then the statements between **then** and **else** are executed, otherwise those between **else** and **end if** are executed.

The general bracketing structure should be observed: **loop** is matched by **end loop** and **if** by **end if**. All the control structures of Ada have this closed form, rather than the open form of Pascal and C that can lead to poorly structured and incorrect programs.

We will now consider in outline the possible general form of the function Sqrt and the package Simple_IO that we have been using. The function Sqrt will have a structure similar to that of our

main subprogram; the major difference will be the existence of parameters.

```
function Sqrt(F: Float) return Float is
  R: Float;
begin
  -- compute value of Sqrt(F) in R
  return R;
end Sqrt;
```

We see here the description of the formal parameters (in this case only one) and the type of the result. The details of the calculation are represented by the comment which starts with a double hyphen. The return statement is the means by which the result of the function is indicated. Note the distinction between a function which returns a result and is called as part of an expression, and a procedure which does not have a result and is called as a single statement.

The package `Simple_IO` will be in two parts: the specification which describes its interface to the outside world, and the body which contains the details of how it is implemented. If it just contained the procedures that we have used, its specification might be

```
package Simple_IO is
  procedure Get(F: out Float);
  procedure Put(F: in Float);
  procedure Put(S: in String);
  procedure New_Line(N: in Integer := 1);
end Simple_IO;
```

The parameter of `Get` is an **out** parameter because the effect of calling `Get` as in `Get(X);` is to transmit a value *out* from the procedure to the actual parameter `X`. The other parameters are all **in** parameters because the value goes *in* to the procedures.

Only a part of the procedures occurs in the package specification; this part is known as the procedure specification and gives just enough information to enable the procedures to be called. We see also the two overloaded specifications of `Put`, one with a parameter of type `Float` and the other with a parameter of type `String`. Finally, note how the default value of 1 for the parameter of `New_Line` is indicated.

The package body for `Simple_IO` will contain the full procedure bodies plus any other supporting material needed for their implementation and is naturally hidden from the outside user. In vague outline it might look like

```
with Ada.Text_IO;
package body Simple_IO is
     …
   procedure Get(F: out Float) is
     …
   begin
     …
   end Get;
   -- other procedures similarly
end Simple_IO;
```

The with clause shows that the implementation of the procedures in `Simple_IO` uses the more general package `Ada.Text_IO`. The notation indicates that `Text_IO` is a `child` package of the package `Ada`. It should also be noticed how the full body of `Get` repeats the procedure specification which was given in the corresponding package specification. Note that the package `Text_IO` really exists whereas `Simple_IO` is a figment of our imagination made up for the purpose of our example. We will say more about `Text_IO` in a moment.

The example in this section has briefly revealed some of the overall structure and control statements of Ada. One purpose of this section has been to stress that the idea of packages is one of the most important concepts in Ada. A program should be conceived as a number of components which provide services to and receive services from each other.

Perhaps this is an appropriate point to mention the special package `Standard` which exists in every implementation and contains the declarations of all the predefined identifiers such as `Float` and `Integer`. We can assume access to `Standard` automatically and do not have to give its name in a with clause.

26.2.2 Errors and Exceptions

We introduce this topic by considering what would have happened in the example in the previous section if we had not tested for a negative value of `X` and consequently called `Sqrt` with a negative argument. Assuming that `Sqrt` has itself been written in an appropriate manner, then it clearly cannot deliver a value to be used as the parameter of `Put`. Instead an exception will be raised. The raising of an exception indicates that something unusual has happened and the normal sequence of execution is broken. In our case the exception might be `Constraint_Error` which is a predefined exception declared in the package `Standard`. If we did nothing to cope with this possibility then our program would be terminated and no doubt the Ada Run Time System will give us a message saying that our program has failed and why. We can, however, look out for an exception and take remedial action if it occurs. In fact we could replace the conditional statement

```
if X < 0.0 then
  Put("not calculable");
else
  Put(Sqrt(X));
end if;
```

by

```
begin
  Put(Sqrt(X));
exception
  when Constraint_Error =>
    Put("not calculable");
end;
```

This fragment of a program is an example of a block. If an exception is raised by the sequence of statements between **begin** and **exception**, then control immediately passes to the one or more statements following the handler for that exception, and these are obeyed instead. If there were no handler for the exception (it might be another exception such as `Storage_Error`) then control passes up the flow hierarchy until we come to an appropriate handler or fall out of the main subprogram, which then becomes terminated as we mentioned, with a message from the Run Time System.

The above example is not a good illustration of the use of exceptions since the event we are guarding against can easily be tested for directly. Nevertheless it does show the general idea of how we can look out for unexpected events and leads us into a brief consideration of errors in general.

From the linguistic viewpoint, an Ada program may be incorrect for various reasons. There are two main error categories, according to how they are detected.

- Many errors are detected by the compiler—these include simple punctuation mistakes such as leaving out a semicolon or attempting to violate the type rules such as mixing up colors and fish. In these cases the program is said to be illegal and will not be executed.
- Other errors are detected when the program is executed. An attempt to find the square root of a negative number or divide by zero are examples of such errors. In these cases an exception is raised as we have just seen, and we have an opportunity to recover from the situation.

One of the main goals in the design of Ada was to ensure that human errors fall into the first category most of the time so that their detection and correction is a straightforward matter.

26.2.3 Scalar Type Model

We have said that one of the key benefits of Ada is its strong typing. This is well illustrated by the enumeration type. Consider

```
declare
  type Color is (Red, Amber, Green);
  type Fish is (Cod, Hake, Plaice);
  X, Y: Color;
  A, B: Fish;
begin
  X := Red;              -- ok
  A := Hake;             -- ok
  B := X;                -- illegal
  …
end;
```

Here we have a block which declares two enumeration types, `Color` and `Fish`, and two variables of each type, and then performs various assignments. The declarations of the types gives the allowed values of the types. Thus the variable `X` can only take one of the three values `Red`, `Amber`, or `Green`. The fundamental rule of strong typing is that we cannot assign a value of one type to a variable of a different type. So we cannot mix up colors and fish and thus our (presumably accidental) attempt to assign the value of `X` to `B` is illegal and will be detected during compilation.

There are three enumeration types predefined in the package `Standard`. One is

```
type Boolean is (False, True);
```

which plays a fundamental role in control flow. Thus the predefined relational operators such as $<$ produce a result of this type and such a value follows **if** as we saw in the construction

```
if X < 0.0 then
```

in the example above. The other predefined enumeration types are `Character` and `Wide_Character`. The values of these types are the 8-bit ISO Latin-1 characters and the 16-bit ISO Basic Multilingual Plane characters; these types naturally play an important role in input-output. The literal values of these types include the printable characters and these are represented by placing them in single quotes thus `'X'` or `'a'` or indeed `'''`.

The other fundamental types are the numeric types. One way or another, all other data types are built out of enumeration types and numeric types. The two major classes of numeric types are the integer types and floating point types (there are also fixed-point types which are rather obscure and deserve no further mention in this brief overview). The integer types in fact are subdivided into signed integer types (such as `Integer`) and unsigned or modular types. All implementations will have the types `Integer` and `Float`. In addition, if the architecture is appropriate, an implementation may have other predefined numeric types, `Long_Integer`, `Long_Float`, `Short_Float`, and so on. There will also be specific integer types for an implementation depending upon the supported word lengths such as `Integer_16` and corresponding unsigned types such as `Unsigned_16`.

One of the problems of numeric types is how to obtain both portability and efficiency in the face of variations in machine architecture. In order to explain how this is done in Ada it is convenient to introduce the concept of a derived type. (We will deal with derived types in more detail when we come to object oriented programming).

The simplest form of derived type introduces a new type which is almost identical to an existing type except that it is logically distinct. If we write

```
type Light is new Color;
```

then Light will, like Color, be an enumeration type with literals Red, Amber and Green. However, values of the two types cannot be arbitrarily mixed since they are logically distinct. Nevertheless, in recognition of the close relationship, a value of one type can be converted to the other by explicitly using the destination type name. So we can write

```
declare
  type Light is new Color;
  C: Color;
  L: Light;
begin
  L := Amber; -- the light amber, not the color
  C := Color(L); -- explicit conversion
  …
end;
```

whereas a direct assignment

```
C := L;          -- illegal
```

would violate the strong typing rule and this violation would be detected during compilation.

Returning now to our numeric types, if we write

```
type My_Float  is new Float;
```

then My_Float will have all the operations (+, −, etc.) of Float and in general can be considered as equivalent. Now suppose we transfer the program to a different computer on which the predefined type Float is not so accurate and that Long_Float is necessary. Assuming that the program has been written using My_Float rather than Float, then replacing the declaration of My_Float by

```
type My_Float is new Long_Float;
```

is the only change necessary. We can actually do better than this by directly stating the precision that we require, thus

```
type My_Float is digits 7;
```

will cause My_Float to be based on the smallest predefined type with at least seven decimal digits of accuracy.

A similar approach is possible with integer types so that rather than using the predefined types Integer or Long_Integer, we can give the range of values required thusly:

```
type My_Integer is range 21000_000 … + 1000_000;
```

The point is that it is not good practice to use the predefined numeric types directly when writing professional programs which may need to be portable.

26.2.4 Arrays and Records

Ada naturally enables the creation of composite array and record types. Arrays may actually be declared without giving a name to the underlying type (the type is then said to be anonymous), but records always have a type name.

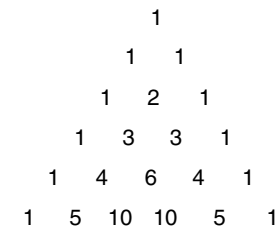

FIGURE 26.1 Pascal's triangle.

As an example of the use of arrays suppose we wish to compute the successive rows of Pascal's triangle. This is usually represented as shown in Figure 26.1. The reader will recall that the rows are the coefficients in the expansion of $(1 + x)^n$ and that a neat way of computing the values is to note that each one is the sum of the two diagonal neighbors in the row above.

Suppose that we are interested in the first 10 rows. We could declare an array to hold such a row by

```
Pascal: array (0 .. 10) of Integer;
```

and now assuming that the current values of the array `Pascal` correspond to row $n-1$, with the component Pascal(0) being 1, then the next row could be computed in a similar array `Next` by

```
Next(0) := 1;
for I in 1 .. N-1 loop
  Next(I) := Pascal(I-1) + Pascal(I);
end loop;
Next(N) := 1;
```

and then the array `Next` could be copied into the array `Pascal`.

This illustrates another form of loop statement where a controlled variable **I** takes successive values from a range; the variable is automatically declared to be of the type of the range which in this case is `Integer`. Note that the intermediate array `Next` could be avoided by iterating backwards over the array; we indicate this by writing **reverse** in front of the range, thus

```
Pascal(N) := 1;
  for I in reverse 1 .. N-1 loop
    Pascal(I) := Pascal(I-1) + Pascal(I);
end loop;
```

We can also declare arrays of several dimensions. So if we wanted to keep all the rows of the triangle we might declare

```
Pascal2: array (0 .. 10, 0 .. 10) of Integer;
```

and then the loop for computing row n would be

```
Pascal2(N, 0) := 1;
for I in 1 .. N - 1 loop
  Pascal2(N, I) := Pascal2(N−1, I−1) + Pascal2(N−1, I);
end loop;
Pascal2(N, N) := 1;
```

We have declared our arrays without giving a name to their type. We could alternatively have written

```
type Row is array (0 .. Size) of Integer;
Pascal, Next: Row;
```

where we have given the name Row to the type and then declared the two arrays Pascal and Next. There are advantages to this approach as we will see later. Incidentally, the bounds of an array do not have to be constant, they could be any computed values such as the value of some variable Size.

We conclude this brief discussion of arrays by observing that the type String is in fact an array whose components are of the enumeration type Character. Its declaration (in the package Standard) is

```
type String is array (Positive range <>) of Character;
```

and this illustrates a form of type declaration which is said to be indefinite because it does not give the bounds of the array; these have to be supplied when an object is declared:

```
Buffer: String(1 .. 80);
```

Incidentally, the identifier Positive in the declaration of the type String denotes what is known as a subtype of Integer; values of the subtype Positive are the positive integers and so the bounds of all arrays of type String must also be positive—the lower bound is of course typically 1, but need not be.

A record is an object comprising a number of named components typically of different types. We always have to give a name to a record type. If we were manipulating a number of buffers then it would be convenient to declare a record type containing the buffer and an indication of the start and finish of that part of the buffer actually containing useful data.

```
type Buffer is
  record
    Data: String(1 .. 80);
    Start, Finish: Integer;
  end record;
```

An individual buffer could then be declared by

```
My_Buffer: Buffer;
```

and the components of the buffer can then be manipulated using a dotted notation to select the individual components

```
My_Buffer.Start := 1;
My_Buffer.Finish := 3;
My_Buffer.Data(1 .. 3) :== "XYZ";
```

Note that this assigns values to the first three components of the array using a so-called slice.

Whole array and record values can be created using aggregates, which are simply a set of values in parentheses separated by commas. Thus we could assign appropriate values to Pascal and My_Buffer by

```
Pascal(0 .. 4) := (1, 4, 6, 4, 1);
My_Buffer := (('X', 'Y', 'Z', others => ' '), 1, 3);
```

where in the latter case we have in fact assigned all 80 values to the array Data and used **others** to indicate that after the three useful characters the remainder of the array is padded with spaces. Note also the nesting of parentheses.

This concludes our brief discussion on simple arrays and records. We will show how record types can be extended in a moment.

26.2.5 Access Types

The last section showed how the scalar types (numeric and enumeration types) may be composed into arrays and records. The other vital means for creating structures is through the use of access types (the Ada name for pointer types); access types allow list processing and are typically used with record types.

The explicit manipulation of pointers or references has been an important feature of most languages since Algol 68. References rather dominated Algol 68 and caused problems, and the corresponding pointer facility in Pascal is rather austere. The pointer facility in C, on the other hand, provides raw flexibility which is open to abuse and quite insecure and thus the cause of many wrong programs.

Ada provides both a high degree of reliability and considerable flexibility through access types. Ada access types must explicitly indicate the type of data to which they refer. The most general form of access types can refer to any data of the type concerned but we will restrict ourselves in this overview to those which just refer to data declared in a storage pool (the Ada term for a heap).

For example, suppose we wanted to declare various buffers of the type in the previous section. We might write

```
type Buffer_Ptr is access Buffer;
Handle: Buffer_Ptr;
…
Handle := new Buffer;
```

This allocates a buffer in the storage pool and sets a reference to it into the variable `Handle`. We can then refer to the various components of the buffer indirectly using the variable `Handle`

```
Handle.Start := 1;
Handle.Finish := 3;
```

and we can refer to the complete record as `Handle.all`. Note that `Handle.Start` is strictly an abbreviation for `Handle.all.Start`.

Access types are of particular value for list processing where one record structure contains an access value to another record structure. The classic example is typified by

```
type Cell;
type Cell_Ptr is access Cell;

type Cell is
  record
    Value: Data;
    Next: Cell_Ptr;
end record;
```

The type `Cell` is a record containing a component of some type `Data` plus a component `Next` which can refer to another similar record. Note the partial declaration of the type `Cell`. This is required in the declaration of the type `Cell_Ptr` because of the inherent nature of the circularity of the declarations.

Access types can be used to refer to any type, although records are common. Access types may also be used to refer to subprograms and this is particularly important when communicating with programs in other languages.

26.2.6 Error Detection

We mentioned earlier that an overall theme in the design of Ada was concern for correctness and that errors should be detected early in the programming process. As a simple example consider a fragment of a program controlling the crossing gates on a railroad. First we have an enumeration type describing the state of a signal:

```
type Signal is (Clear, Caution, Danger);
```

and then perhaps

```
if The_Signal = Clear then
    Open_Gates;
    Start_Train;
end if;
```

It is instructive to consider how this might be written in C and then to consider the consequences of various simple programming errors. C does not have enumeration types as such, so the signal values have to be held as an integer (**int**) with code values such as 0, 1, and 2 representing the three states. This already has potential for errors because there is nothing in the C language that can prevent us from assigning a silly value such as 4 to a signal whereas it is not even possible to attempt such a thing when using the Ada enumeration type.

The corresponding text in C is

```
if (The_Signal == 0)
    {Open_Gates();
    Start_Train();
    }
```

It is interesting to consider what would happen in the two languages if we make various typographical errors. Suppose first that we accidentally type an extra semicolon at the end of the first line. The Ada program then fails to compile and the error is immediately drawn to our attention; the C program, however, still compiles and the condition is ignored (since it then controls no statements). The C program consequently always opens the gates and starts the train irrespective of the state of the signal!

Another possibility is that one of the = signs might be omitted in C. The equality then becomes an assignment and also returns the result as the argument for the test. So the program still compiles, the signal is always set clear no matter what its previous state, and of course the gates are then opened and the train is started on its perilous journey. The corresponding error in Ada might be to write : = instead of = and of course the program will then not compile.

However, many errors cannot be detected at compile time. For example using `My_Buffer` we might write

```
Index: Integer;
…
Index := 81;
…
My_Buffer.Data(Index) := 'x';
```

which attempts to write to the 81st component of the array, which does not exist. Such assignments are checked in Ada at run time and `Constraint_Error` would be raised so that the integrity of the program is not violated.

The corresponding instructions in C would undoubtedly overwrite an adjacent piece of storage and probably corrupt the value in `My_Buffer.Start`.

It often happens that variables such as `Index` can only sensibly have a certain range of values; this can be indicated by introducing a subtype

```
subtype Buffer_Index is Integer range 1 ..80;
Index: Buffer_Index;
```

or by indicating the constraint directly

```
Index: Integer range 1 .. 80;
```

This has the advantage that the attempt to assign 81 to `Index` is itself checked and prevented, so the error is detected even earlier.

The reader may feel that such checks will make the program slower. Studies have shown that provided the ranges are appropriate then the overhead should be minimal. For example, having ensured that `Index` cannot have a value out of range there is no need to apply checks to `My_Buffer.Data(Index)` as well. In any event, if we are really confident that the program is correct, the checks can be switched off for production use.

26.3 Abstraction

As mentioned earlier, abstraction in various forms seems to be the key to the development of programming languages.

We saw above how we can declare a type for the manipulation of a buffer

```
type Buffer is
  record
    Data: String(1 .. 80);
    Start: Integer;
    Finish: Integer;
  end record;
```

in which the component `Data` actually holds the characters in the buffer and `Start` and `Finish` index the ends of the part of the buffer containing useful information. We also saw how the various components might be updated and read using normal assignment.

However, such direct assignment is often unwise since the user could inadvertently set inconsistent values into the components or read nonsense components of the array. A much better approach is to create an Abstract Data Type (ADT) so that the user cannot see the internal details of the type but can only access it through various subprogram calls which define an appropriate protocol.

This can be done using a package containing a private type. Let us suppose that the protocol allows us to reload the buffer (possibly not completely full) and to read one character at a time. Consider the following

```
package Buffer_System is    -- visible part

  type Buffer is private;

  procedure Load(B: out Buffer; S: in String);
  procedure Get(B: in out Buffer; C: out Character);

private                    -- private part
   Max: constant Integer := 80;
   type Buffer is
     record
        Data: String(1 .. Max);
        Start: Integer := 1;
        Finish: Integer := 0;
     end record;

end Buffer_System;

package body Buffer_System is

    procedure Load(B: out Buffer; S: in String) is
    begin
      B.Start := 1;
      B.Finish := S'Length;
      B.Data(B.Start .. B.Finish) := S;
```

```
    end Load;

    procedure Get(B: in out Buffer; C: out Character) is
    begin
      C := B.Data(B.Start);
      B.Start := B.Start + 1;
      end Get;
  end Buffer_System;
```

With this formulation the client can only access the information in the visible part of the specification which is the bit before the word **private**. In this visible part the declaration of the type `Buffer` merely says that it is private and the full declaration then occurs in the private part. There are thus two views of the type `Buffer`; the external client just sees the partial view whereas within the package the code of the server subprograms can see the full view. The specifications of the server subprograms are naturally also declared in the visible part.

The net effect is that the user can declare and manipulate a buffer by simply writing

```
My_Buffer: Buffer;
…
Load(My_Buffer, Some_String);
…
Get(My_Buffer, A_Character);
```

but the internal structure is quite hidden. There are two advantages: one is that the user cannot inadvertently misuse the buffer and the second is that the internal structure of the private type could be rearranged if necessary; provided that the protocol is maintained, the user program will not need to be changed.

This hiding of information and consequent separation of concerns is very important and illustrates the benefit of data abstraction. The design of appropriate interface protocols is the key to the development and subsequent maintenance of large programs.

The astute reader will note that we have not bothered to ensure that the buffer is not loaded when there is still unread data in it or the string is too long to fit, nor read from when it is empty. We could rectify this by declaring our own exception called perhaps `Error` in the visible part of the specification thus

```
Error: exception;
```

and then check within Load by for example

```
if S'Length > Max or B.Start <= B.Finish then
    raise Error;
end if;
```

This causes our own exception to be raised if the buffer is overloaded.

As a minor point note the use of the constant `Max` so that the literal 80 only appears in one place. Note also the attribute `Length` which applies to any array and gives the number of its components. The upper and lower bounds of an array `S` are incidentally given by `S'First` and `S'Last`.

Another point is that the parameter `Buffer` of `Get` is marked as **in out** because the procedure both reads the initial value of `Buffer` and updates it.

Finally, note that the components `Start` and `Finish` of the record have initial values in the declaration of the record type; these ensure that when a buffer is declared these components are assigned sensible values and thereby indicate that the buffer is empty. An alternative, of course, would be to provide a procedure `Reset` but the user might forget to call it.

26.3.1 Objects and Inheritance

As its name suggests, object oriented programming (OOP) concerns the idea of programming around objects. A good example of an object in this sense is the variable `My_Buffer` of the type `Buffer` in the previous section. We conceive of the object such as a buffer as a coordinated whole and not just as the sum of its components. Indeed, the external user cannot see the components at all but can only manipulate the buffer through the various subprograms associated with the type.

Certain operations upon a type are called the primitive operations of the type. In the case of the type `Buffer` they are the subprograms declared in the package specification along with the type itself and which have parameters or a result of the type. In the case of a type such as `Integer`, the primitive operations are those such as + and − which are predefined for the type (and indeed they are declared in `Standard` along with the type `Integer` itself and so fit the same model).

Other important ideas in OOP are

- The ability to define one type in terms of another and especially as an extension of another; this is type extension,
- The ability for such a derived type to inherit the primitive operations of its parent and also to add to and replace such operations; this is inheritance,
- The ability to distinguish the specific type of an object at run time from among several related types and in particular to select an operation according to the specific type; this is (dynamic) polymorphism.

In an earlier section we showed how the type `Light` was derived from `Color` and also showed how numeric portability could be aided by deriving a numeric type such as `My_Float` from one of the predefined types. These were very simple forms of inheritance; the new types inherited the primitive operations of the parent; however, the types were not extended in any way and underneath were really the same type. The main benefit of such derivation is simply to provide a different name and thereby to distinguish the different uses of the same underlying type in order to prevent us from inadvertently using a `Light` when we meant to use a `Color`.

The more general case is where we wish to extend a type in some way and also to distinguish objects of different types at run time. The most natural form of type for the purposes of extension, of course, is a record where we can consider extension as simply the addition of further components. The other point is that if we need to distinguish the type at run time then the object must contain an indication of its type. This is provided by a hidden component called the tag. Type extension in Ada is thus naturally carried out using tagged record types.

As a simple example suppose we wish to manipulate various kinds of geometrical objects. We can imagine that the kinds of objects form a hierarchy as shown in Figure 26.2.

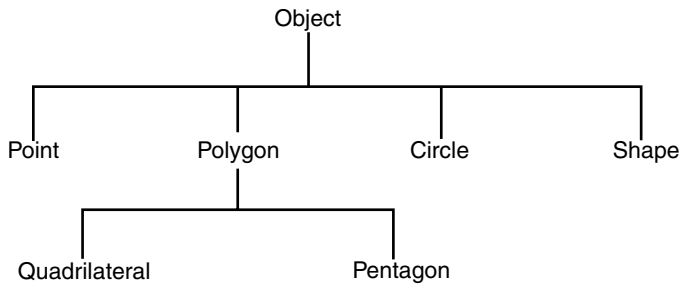

FIGURE 26.2 A hierarchy of geometrical objects.

All objects will have a position given by their *x*- and *y*-coordinates. So we declare the root of the hierarchy as

```
type Object is tagged
  record
    X_Coord: Float;
    Y_Coord: Float;
  end record;
```

Note carefully the introduction of the reserved word **tagged**. This indicates that values of the type carry a tag at run time and that the type can be extended. The other types of geometrical objects will be derived (directly or indirectly) from this type. For example we could have

```
type Circle is new Object with
  record
      Radius: Float;
  end record;
```

and the type `Circle` then has the three components `X_Coord`, `Y_Coord`, and `Radius`. It inherits the two coordinates from the type `Object` and the component `Radius` is added explicitly.

Sometimes it is convenient to derive a new type without adding any further components. For example:

```
type Point is new Object with null record;
```

In this last case we have derived `Point` from `Object` but naturally not added any new components. However, since we are dealing with tagged types we have to explicitly add **with null record**; to indicate that we did not want any new components. This has the advantage that it is always clear from a declaration whether a type is tagged or not.

A private type can also be marked as tagged

```
type Shape is tagged private ;
```

and the full type declaration must then (ultimately) be a tagged record

```
type Shape is tagged
  record …
```

or derived from a tagged record such as `Object`. On the other hand we might wish to make visible the fact that the type `Shape` is derived from `Object` and yet keep the additional components hidden. In this case we would write

```
package Hidden_Shape is
  type Shape is new Object with private;     -- client view
  …
private
  type Shape is new Object with              -- server view
    record
      -- the private components
    end record;
end Hidden_Shape;
```

In this last case it is not necessary for the full declaration of `Shape` to be derived directly from the type `Object`. There might be a chain of intermediate derived types (it could be derived from `Circle`); all that matters is that `Shape` is ultimately derived from `Object`.

The primitive operations of a type are those declared in the same package specification as the type and that have parameters or result of the type. On derivation these operations are inherited by the new type. They can be overridden by new versions and new operations can be added and these then become primitive operations of the new type and are themselves naturally inherited by any further derived type.

Thus we might have declared a function giving the distance from the origin

```
function Distance(O: in Object) return Float is
begin
  return Sqrt(O.X_Coord**2 + O.Y_Coord**2);
end Distance;
```

The type `Circle` would then sensibly inherit this function. If however, we were concerned with the area of an object then we might start with

```
function Area(O: in Object) return Float is
begin
  return 0.0;
end Area;
```

which returns zero since a raw object has no area. The abstract concept of an area applies also to a circle and so it is appropriate that a function `Area` be defined for the type `Circle`. However, to inherit the function from the type `Object` is clearly inappropriate and so we explicitly declare

```
function Area(C: in Circle) return Float is
begin
  return Pi*C.Radius**2;
end Area;
```

which then overrides the inherited operation. We can perhaps summarize these ideas by saying that the specification is always inherited whereas the implementation may be inherited but can be replaced.

It is possible to convert a value from the type `Circle` to `Object` and vice versa. From circle to object is straightforward, we simply write

```
O: Object := (1.0, 0.5);
C: Circle := (0.0, 0.0, 34.7);
…

O := Object(C);
```

which effectively ignores the third component. However, conversion in the other direction requires the provision of a value for the extra component and this is done by an extension aggregate, thus:

```
C := (O with 41.2);
```

where the expression `O` is extended after **with** by the values of the extra components written just as in a normal aggregate. In this case we only had to give a value for the radius.

It is important to remember that the primitive operations are those declared in the same package as the type. Thus the type `Object` and the functions `Distance` and `Area` might be in a package `Objects`. And then a package Shapes might contain the types `Circle` and `Point` and a type `Triangle` with sides `A`, `B`, and `C` and appropriate functions returning the area. The overall structure would then be

```
package Objects is
  type Object is tagged
    record
      X_Coord: Float;
      Y_Coord: Float;
    end record;
```

```
  function Distance(O: Object) return Float;
  function Area(O: Object) return Float;
end Objects;

package body Objects is
  function Distance(O: Object) return Float is
  begin
    return Sqrt(O.X_Coord**2 + O.Y_Coord**2);
  end Distance;

  function Area(O: Object) return Float is
  begin
    return 0.0;
  end Area;
end Objects;

with Objects; use Objects;
package Shapes is
  type Point is new Object with null record;

  type Circle is new Object with
    record
      Radius: Float;
    end record;

  function Area(C: Circle) return Float;

  type Triangle is new Object with
    record
      A, B, C: Float;
    end record;

  function Area(T: Triangle) return Float;
end Shapes;

package body Shapes is
  function Area(C: Circle) return Float is
  begin
    return Pi *C.Radius**2;
  end Area;

  function Area(T: Triangle) return Float is
    S: constant Float:= 0.5*(T.A + T.B + T.C);
  begin
    return Sqrt(S *(S - T.A)*(S - T.B)*(S - T.C));
  end Area;
end Shapes;
```

Note that we can put the use clause for `Objects` immediately after the **with** clause.

26.3.2 Classes and Polymorphism

In the last section we showed how to declare a hierarchy of types derived from the type `Object`. We saw how on derivation further components and operations could be added and that operations could be replaced.

However, it is very important to note that an operation cannot be taken away nor can a component be removed. As a consequence we are guaranteed that all the types derived from a common ancestor will have all the components and operations of that ancestor.

So in the case of the type `Object`, all types in the hierarchy derived from `Object` will have the common components such as their coordinates and the common operations such as `Distance` and `Area`. Since they have these common properties it is natural that we should be able to manipulate a value of any type in the hierarchy without knowing exactly which type it is, provided that we only use the common properties. Such general manipulation is done through the concept of a class.

Ada carefully distinguishes between the set of types such as `Object` plus all its derivatives on the one hand, and an individual type such as `Object` itself on the other hand. A set of such types is known as a class. Associated with each class is a type called the class-wide type which for the set rooted at `Object` is denoted by `Object'Class`. The type `Object` is referred to as a specific type when we need to distinguish it from a class-wide type.

We can of course have subclasses. For example, `Polygon'Class` represents the set of all types derived from and including `Polygon`. This is a subset of the class `Object'Class`. All the properties of `Object'Class` will also apply to `Polygon'Class` but not vice versa. For example, although we have not shown it, the type `Polygon` will presumably contain a component giving the length of the sides. Such a component will belong to all types of the class `Polygon'Class` but not to `Object'Class`.

As a simple example of the use of a class-wide type consider the following function

```
function Moment(OC: Object'Class) return Float is
begin
  return OC.X_Coord * Area(OC);
end Moment;
```

Those who recall their school mechanics will remember that the moment of a force about a fulcrum is the product of the weight multiplied by the distance from the fulcrum. So in our example, taking the *x*-axis as being horizontal, the moment of a geometrical object about the origin is the *x*-coordinate multiplied by the weight which we can take as being proportional to the area (and for simplicity we have assumed is just the area).

This function has a formal parameter of the class-wide type `Object'Class`. This means it can be called with an actual parameter whose type is any specific type in the class comprising the set of all types derived from `Object`. Thus we could write

```
C: Circle …
M: Float;
…
M := Moment(C);
```

Within the function `Moment` we can naturally refer to the specific object as `OC`. Since we know that the object must be of a specific type in the `Object` class, we are guaranteed that it will have a component `OC.X_Coord`. Similarly we are guaranteed that the function `Area` will exist for the type of the object since it is a primitive operation of the type `Object` and will have been inherited by (and possibly overridden for) every type derived from `Object`. So the appropriate function `Area` is called and the result multiplied by the *x*-coordinate and returned as the result of the function `Moment`.

Note carefully that the particular function `Area` to be called is not known until the program executes. The choice depends upon the specific type of the parameter and this is determined by the tag of the object passed as actual parameter; remember that the tag is a sort of hidden component of the tagged type. This selection of the particular subprogram according to the tag is known as *dispatching* and is a vital aspect of the dynamic behavior provided by polymorphism.

Dispatching only occurs when the actual parameter is of a class-wide type; if we call `Area` with an object of a specific type such as `C` of type `Circle`, then the choice is made at compile time. Dispatching is often called late binding because the call is only bound to the called subprogram late in the compile-link-execute

process. The binding to a call of Area with the parameter C of type Circle is called static binding because the subprogram to be called is determined at compile time.

Observe that the function Moment is not a primitive operation of any type; it is just an operation of Object'Class and it happens that a value of any specific type derived from Object can be implicitly converted to the class-wide type. Class wide types do not have primitive operations and so no inheritance is involved.

It is interesting to consider what would have happened if we had written

```
function Moment(O: Object) return Float is
begin
  return O.X_Coord * Area(O);
end Moment;
```

where the formal parameter is of the specific type Object. This always returns zero because the function Area for an Object always returns zero. If this function Moment were declared in the same package as Object then it would be a primitive operation of Object and thus inherited by the type Circle. However, the internal call would still be to the function Area for the type Object and not to the type Circle and so the answer would still be zero. This is because the binding is static and inheritance simply passes on the same code. The code mechanically works on a Circle because it only uses the Object part of the circle (we say it sees the Object view of the Circle); but unfortunately it is not what we want. Of course, we could override the inherited operation by writing

```
function Moment(C: Circle) return Float is
begin
  return C.X_Coord*Area(C);
end Moment;
```

but this is tedious and causes unnecessary duplication of similar code. The proper approach for such general situations is to use the original class-wide version with its internal dispatching; this can be shared by all types without duplication and always calls the appropriate function Area.

A major advantage of using a class-wide operation such as Moment is that a system using it can be written, compiled, and tested without knowing all the specific types to which it is to be applied. Moreover, we can then add further types to the system without recompilation of the existing tested system.

For example we could add a further type

```
type Pentagon is new Object with…
function Area(P: Pentagon) return Float;
…
Star: Pentagon := …
…
Put("Moment of star is");
Put(Moment(Star));
```

and then the old existing tried and tested Moment will call the new Area for the Pentagon without being recompiled. (It will of course have to be relinked.)

This works because of the mechanism used for dynamic binding; the essence of the idea is that the class-wide code has dynamic links into the new code and this is accessed via the tag of the type. This creates a very flexible and extensible interface ideal for building up a system from reusable components.

One problem with class-wide types is that we cannot know how much space might be occupied by an arbitrary object of the type because the type might be extended. So although we can declare an object of a class-wide type it has to be initialized and thereafter that object can only be of the specific type of that initial value. Note that a formal parameter of a class-wide type such as in Moment is allowed because the space is provided by the actual parameter.

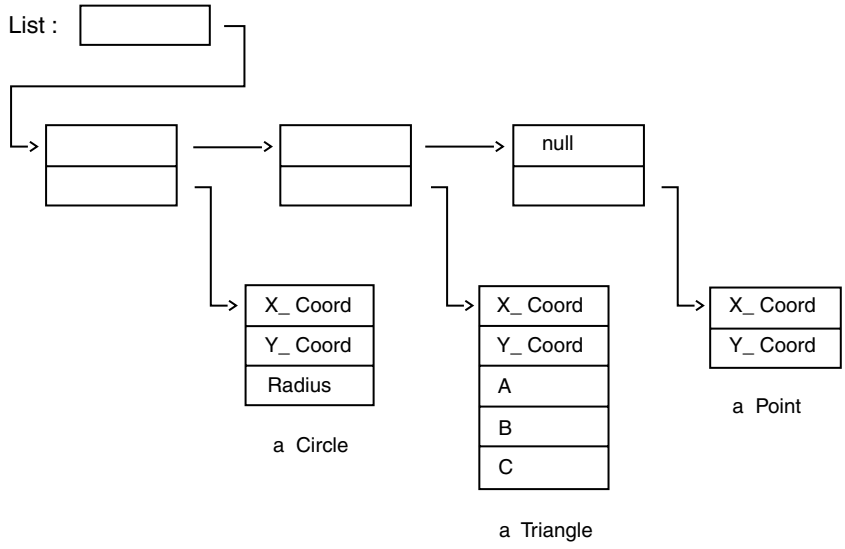

FIGURE 26.3 A chain of objects.

Another similar restriction is that we cannot have an array of class-wide components (even if initialized) because the components might be of different specific types of different sizes and thus impossible to index efficiently.

One consequence of these necessary restrictions is that it is very natural to use access types with object oriented programming since there is no problem with pointing to objects of different sizes at different times. Thus, suppose we wanted to manipulate a series of geometrical objects; it is very natural to declare these in free storage as required. They can then be chained together on a list for processing. Consider

```
type Pointer is access Object'Class;
type Cell;
type Cell_Ptr is access Cell;

type Cell is
  record
    Next: Cell_Ptr;
    Element: Pointer;
  end record;
```

which enables us to create cells which can be linked together; each cell has an element which is a pointer to any geometrical object.

We can now imagine that a number of objects have been created and linked together to form a list as in Figure 26.3. We assume that this chain is accessed through a variable called `List` of the type `Cell_Ptr`.

We can now easily process the objects on the list and might, for example, compute the total moment of the set of objects by calling the following function

```
function Total_Moment(The_List: Cell_Ptr) return Float is
  Local: Cell_Ptr := The_List;
  Result: Float := 0.0;
begin
  loop
    if Local = null then          -- end of list
      return Result;
```

```
      end if;
      Result := Result + Moment(Local.Element.all);
      Local := Local.Next;
    end loop;
  end Total_Moment;
```

We conclude this brief survey of the OOP facilities in Ada by considering abstract types. It is sometimes the case that we would like to declare a type as the foundation for a class of types with certain common properties, but without allowing objects of the original type to be declared. For example, we probably would not want to declare an object of the raw type `Object`. If we wanted an object without any area then it would be appropriate to declare a `Point`. Moreover, the function Area for the type `Object` is dubious since it usually has to be overridden anyway. But it is important to be able to ensure that all types derived from `Object` do have an Area so that the dispatching in the function `Moment` always works. We can achieve this by writing

```
package Objects is
  type Object is abstract tagged
    record
      X_Coord: Float;
      Y_Coord: Float;
    end record;

  function Distance(O: in Object) return Float;
  function Area(O: in Object) return Float is abstract;
end Objects;

package body Objects is
  function Distance(O: Object) return Float is
  begin
    return Sqrt(O.X_Coord ** 2 + O.Y_Coord ** 2);
  end Distance;
end Objects;
```

In this formulation the type `Object` and the function `Area` are marked as abstract. It is illegal to declare an object of an abstract type, and an abstract subprogram has no body and so cannot be called. On deriving a concrete (that is, nonabstract) type from an abstract type any abstract inherited operations must be overridden by concrete operations. Note that we have declared the function `Distance` as not abstract; this is largely because we know that it will be appropriate anyway.

This approach has a number of advantages; we cannot declare a raw `Object` by mistake, we cannot inherit the silly function `Area`, and we cannot make the mistake of declaring the function `Moment` for the specific type `Object` (because it would then contain a call of the abstract function `Area`).

But despite the type being abstract, we can declare the function `Moment` for the class-wide type `Object'Class`. This always works because of the rule that we cannot declare an object of an abstract type; any actual object passed as parameter must be of a concrete type and will have an appropriate function `Area` to which it can dispatch.

26.3.3 Genericity

We have seen how class-wide types provide us with dynamic polymorphism. This means that we can manipulate several types with a single construction and that the specific type is determined dynamically, that is at run time. In this section we introduce the complementary concept of static polymorphism, where again we have a single construction for the manipulation of several types but in this case the choice of type is made statically at compile time.

An important objective of Software Engineering is to reuse existing software components. However, the strong typing model of Ada (even with class-wide types) sometimes gets in the way unless we have a method of writing software components which can be used for various different types. For example, the program to do a sort is largely independent of what it is sorting — all it needs is a rule for comparing the values to be sorted.

So we need a means of writing pieces of software which can be parameterized as required for different types. In Ada this is done by the generic mechanism. We can make a package or subprogram generic with respect to one or more parameters which can include types. Such a generic unit provides a template from which we can create genuine packages and subprograms by so-called instantiation. The full details of Ada generics are quite extensive but the following brief introduction will illustrate the ideas.

The standard package for the input and output of floating point values in text form is generic with respect to the actual floating type. This is because we want a single package to cope with all the possible floating types, such as the underlying machine types `Float` and `Long_Float` as well as the portable type `My_Float`. Its specification is

```
generic
  type Num is digits <>;
package Float_IO is
  …
  procedure Get(Item: out Num; …);
  procedure Put(Item: in Num; …);
  …
end Float_IO;
```

where we have omitted various details relating to the format. The one generic parameter is Num and the notation **digits** <> indicates that it must be a floating point type and echoes the declaration of `My_Float` using **digits** 7.

In order to create an actual package to manipulate values of the type `My_Float`, we write

```
package My_Float_IO is new Float_IO(My_Float);
```

which creates a package with the name `My_Float_IO` where the formal type `Num` has been replaced throughout with our actual type `My_Float`. As a consequence, procedures `Get` and `Put` taking parameters of the type `My_Float` are created and we can then call these as required.

The kind of parameterization provided by genericity is similar but rather different to that provided through class-wide types. In both cases the parameterization is over a related set of types with a set of common properties. Such a related set of types is termed a class.

A common form of class is a derivation class where all the types are derived from a common ancestor such as the type `Object`. Tagged derivation classes form the basis of dynamic polymorphism as we have seen.

But there are also broader forms of class such as the set of all floating-point types which are allowed as actual parameters for the generic package `Float_IO`. The parameters of generic units use these broader classes. For example, a very broad class is the set of all types having assignment. Such a class would be a suitable basis for writing a generic sort routine.

The two forms of polymorphism work together; a common form of generic package is one which takes as a parameter a type from a tagged derivation class. This may be used to provide important capabilities such as multiple inheritance.

26.3.4 Object Oriented Terminology

It is perhaps convenient at this point to compare the Ada terminology with that used by other object oriented (OO) languages such as Smalltalk and C++.

About the only term in common is inheritance. Ada 83 has always had inheritance although not type extension, which only occurs with tagged record types. Untagged record types are called structs in some languages. Ada actually uses the term object to denote variables and constants in general, whereas in the OO sense an object is an instance of an Abstract Data Type (ADT).

Many languages use *class* to denote what Ada calls a specific tagged type (or more strictly, an ADT consisting of a tagged type plus its primitive operations). The reason for this difference is that Ada uses the word class exclusively to refer to a group of related types. Ada classes are not just those groups of types related by derivation, but also groups with broader correspondence as used for generic parameter matching.

The Ada approach clarifies the distinction between the group of types and a single type and strengthens that clarification by introducing class-wide types as such. Some languages use the term class for both specific types and the group of types, with much resulting confusion in terms of description but also in understanding the behavior of the program and keeping track of the real nature of an object.

Primitive operations of Ada tagged types are often called methods or virtual functions. The call of a primitive operation in Ada is bound statically or dynamically according to whether the parameter is of a specific type or class wide. The rules in C++ are more complex and depend upon whether the parameter is a pointer and also whether the call is prefixed by its class name. In Ada, an operation with class-wide formal parameters is always bound statically although it applies to all types of the class.

Dispatching is the Ada term for calling a primitive operation with dynamic binding, and indeed subprogram calls through access types are also a form of dynamic binding.

Abstract types correspond to abstract class in many languages. Abstract subprograms are pure virtual member functions in C++. Note that Eiffel uses the term deferred rather than abstract.

Ancestor or parent type and descendant or derived type become superclass and subclass. The Ada concept of subtype (a type with a constrained set of values) has no correspondence in other languages which do not have range checks, and has no relationship to subclass. A subtype can never have more values than its base type, whereas a descendant type (subclass to other languages) can never have fewer values than its parent type.

Generic units are templates; in Ada they carry type checking with them, whereas many languages treat templates as raw macros.

Another important point is that many languages have no encapsulation mechanism other than so-called classes, whereas Ada has the package largely unrelated to type extension and inheritance and the private type. The effect of private and protected operations in C++ is provided in Ada by a combination of private types and child packages; the latter are kinds of friends.

26.3.5 Tasking

No survey of abstraction in Ada would be complete without a brief mention of tasking. It is often necessary to write a program as a set of parallel activities rather than just as one sequential program.

Most programming languages do not address this issue at all. Some argue that the underlying operating system provides the necessary mechanisms and that they are therefore unnecessary in a programming language. Such arguments do not stand up to careful examination for two main reasons:

- Built-in syntactic constructions provide a degree of reliability which cannot be obtained through a series of individual operating system calls.
- General operating systems do not provide the degree of control and timing required by many applications.

An Ada program can thus be written as a series of interacting tasks. There are two main ways in which tasks can communicate: directly by sending messages to each other and indirectly by accessing shared data. Direct communication between Ada tasks is achieved by one task calling an entry in another task. The calling (client) task waits while the called (server) task executes an accept statement in response to the call; the two tasks are closely coupled during this interaction, which is called a rendezvous. Controlled access to shared data is vital in tasking applications if interference is to be avoided. For example, returning

to the character buffer example, it would be a disaster if a task started to read the buffer while another task was updating it with further information since the component `B.Start` could be changed and a component of the `Data` array read by `Get` before the buffer had been correctly updated by `Load`. Ada prevents such interference by a construction known as a protected object.

The general syntactic form of both tasks and protected objects is similar to that of a package. They have a specification part prescribing the interface, a private part containing hidden details of the interface, and a body stating what they actually do. The general client-server model can thus be expressed as

```
task Server is
  entry Some_Service(Formal: Data);
end;

task body Server is
begin
  …
  accept Some_Service(Formal: Data) do
    -- statements providing the service
  …
  end;
  …
end Server;

task Client;

task body Client is
begin
  …
  Server.Some_Service(Actual);
  …
end Client;
```

A good example of the form of a protected object is given by the buffer example, which could be rewritten as follows

```
protected type Buffer(Max: Integer) is         -- visible part
  procedure Load(S: in String);
  procedure Get(C: out Character);
private                                        -- private part
  Data: String(1 .. Max);
  Start: Integer := 1;
  Finish: Integer := 0;
end Buffer;

protected body Buffer is

  procedure Load(S: in String) is
  begin
    Start := 1;
    Finish := S'Length;
    Data(Start .. Finish) := S;
  end Load;
```

```
  procedure Get(C: out Character) is
  begin
    C := Data(Start);
    Start := Start + 1;
  end Get;

end Buffer;
```

This construction uses a slightly different style to the package. It is a type in its own right whereas the package exported the type. As a consequence, the calls of the procedures do not need to pass explicitly the parameter referring to the buffer and, moreover, within their bodies the references to the private data are naturally taken to refer to the current instance. Note also that the type is parameterized by the discriminant Max and so we can supply the actual size of a particular buffer when it is declared. Statements using the protected type might thus look like

```
B: Buffer(80);
…
B.Load(Some_String);
…
B.Get(A_Character);
```

Although this formulation prevents disastrous interference between several clients, nevertheless it does not prevent a call of Load from overwriting unread data. As before we could insert tests and raise an exception. But the proper approach is to cause the tasks to wait if circumstances are not appropriate. This is done through the use of entries and barriers. The protected type might then be

```
protected type Buffer(Max: Integer) is
  entry Load(S: in String);
  entry Get(C: out Character);
private
  Data: String(1 .. Max);
  Start: Integer := 1;
  Finish: Integer := 0;
end Buffer;

protected body Buffer is

  entry Load(S: in String) when Start > Finish is
  begin
    Start := 1;
    Finish := S'Length;
    Data(Start .. Finish) := S;
  end Load;

  entry Get(C: out Character) when Start <= Finish is
  begin
    C := Data(Start);
    Start := Start + 1;
  end Get;

end Buffer;
```

In this formulation, the procedures are replaced by entries and each entry body has a barrier condition. A task calling an entry is queued if the barrier is false and is only allowed to proceed when the barrier becomes true. This construction is very efficient because task switching is minimized.

26.4 Programs and Libraries

A complete program is put together out of various separately compiled units. In developing a very large program it is inevitable that it will be conceived as a number of subsystems, themselves each composed from a number of separately compiled units.

We see at once the risk of name clashes between the various parts of the total. It would be all too easy for the designers of different parts of the system to reuse popular package names such as `Error_Messages` or `Debug_Info`, and so on. In order to overcome this and other related problems, Ada has a hierarchical naming scheme at the library level. Thus, a package `Parent` may have a child package with the name `Parent.Child`.

We immediately see that if our total system breaks down into a number of major parts such as acquisition, analysis, and report, then name clashes will be avoided if it is mapped into three corresponding library packages plus appropriate child units. There is then no risk of a clash between `Analysis.Debug_Info` and `Report.Debug_Info` because the names are quite distinct.

The naming is hierarchical and can continue to any depth. A good example of the use of this hierarchical naming scheme is found in the standard libraries which are provided by every implementation of Ada.

We have already mentioned the package `Standard` which is an intrinsic part of the language. All library units can be considered to be children of `Standard` and it should never be necessary to explicitly mention `Standard` at all.

In order to reduce the risk of clashes with the user's own names, the predefined library comprises just three packages, each of which has a number of children. The three packages are `System`, which is concerned with the control of storage and similar implementation matters; `Interfaces`, which is concerned with interfaces to other languages and the intrinsic hardware types; and finally `Ada` which contains the bulk of the predefined library.

In this overview we will briefly survey the main package `Ada`. The package `Ada` itself is simply

```
package Ada is
  pragma Pure(Ada);
end Ada;
```

and the various predefined units are children of `Ada`. The pragma indicates that `Ada` has no variable state (this concept is important for sharing in distributed systems, a topic well outside the scope of this discussion).

Important child packages of `Ada` are

`Numerics`; this contains the mathematical library providing the various elementary functions, random number generators, and facilities for complex numbers.

`Characters`; this contains various packages for classifying and manipulating characters as well as the names of all the characters in the Latin-1 set.

`Strings`; this contains packages for the manipulation of strings of various kinds: fixed length, bounded, and unbounded.

`Text_IO`, `Sequential_IO`, and `Direct_IO`; these and other packages provide a variety of input-output facilities.

Those familiar with Ada 83 will note that the predefined library units of Ada 83 have now become child units of `Ada`. Compatibility is achieved because of the predefined renamings of these child units as library units such as

```
with Ada.Text_IO;
package Text_IO renames Ada.Text_IO;
```

Most library units are packages and it is easy to think that all library units must be packages; indeed only a package can have child units. Of course, the main subprogram is a library subprogram and any library package can have child subprograms. A library unit can also be a generic package or subprogram and even an instantiation of a generic package or subprogram; this latter fact is often overlooked.

We have introduced the hierarchical library as simply a naming mechanism. It also has important information hiding and sharing properties. An example is that a child unit can access the information in the private part of its parent; but of course other units cannot see into the private part of a package. This and related facilities enable a group of units to share private information while keeping the information hidden from external clients.

It should also be noted that a child package does not need to have a *with* clause or *use* clause for its parent; this emphasizes the close relationship of the hierarchical structure and parallels the fact that we never need a *with* clause for `Standard` because all units are children of `Standard`.

26.4.1 Input-Output

The Ada language is defined in such a way that all input and output is performed in terms of other language features. There are no special intrinsic features just for input and output. In fact input-output is just a service required by a program and so is provided by one or more Ada packages. This approach runs the attendant risk that different implementations will provide different packages and program portability will be compromised. In order to avoid this, the language defines certain standard packages that will be available in all implementations. Other, more elaborate, packages may be appropriate to special circumstances and the language does not prevent this. Indeed very simple packages such as our purely illustrative `Simple_IO` may also be appropriate. We will now briefly describe how to use some of the features for the input and output of simple text.

Text input-output is performed through the use of the standard package `Ada.Text_IO`. Unless we specify otherwise, all communication will be through two standard files, — one for input and one for output — and we will assume that (as is likely the case for most implementations) these are such that input is from the keyboard and output is to the screen. The full details of `Text_IO` cannot be described here but if we restrict ourselves to just a few useful facilities it looks a bit like

```
with Ada.IO_Exceptions;
package Ada.Text_IO is
  type Count is …          -- an integer type
  …
  procedure New_Line(Spacing: in Count := 1);
  …
  procedure Get(Item: out Character);
  procedure Put(Item: in Character);
  procedure Put(Item: in String);
  …
  -- the package Float_IO outlined above
  -- plus a similar package Integer_IO
  …
end Ada.Text_IO;
```

Note first that this package commences with a *with* clause for the package `Ada.IO_Exceptions`. This further package contains the declaration of a number of exceptions relating to a variety of things which can go wrong with input-output. For most programs the most likely problem to arise is probably `Data_Error`, which would occur, for example, if we tried to read in a number from the keyboard but then accidentally typed in something which was not a number at all or was in the wrong format.

The next thing to note is the outline declaration of the type Count. This is an integer type having similar properties to the type Integer and almost inevitably with the same implementation (just as the type My_Integer might be based on Integer). The parameter of New_Line is of the type Count rather than plain Integer, although since the parameter will typically be a literal such as 2 (or be omitted so that the default of 1 applies) this will not be particularly evident.

A single character can be output by, for example:

```
Put('A');
```

and a string of characters by

```
Put("This Is a string of characters");
```

A value of the type My_Float can be output in various formats. But first we have to instantiate the package Float_IO mentioned above, and which is declared inside Ada.Text_IO. Having done that, we can call Put with a single parameter, the value of type My_Float to be output, in which case a standard default format is used, or we can add further parameters controlling the format.

If we do not supply any format parameters then an exponent notation is used with 7 significant digits — 1 before the point and 6 after (the 7 matches the precision given in the declaration of My_Float). There is also a leading space or minus sign. The exponent consists of the letter E followed by the exponent sign (+ or −) and then a two digit decimal exponent.

We can override the default by providing three further parameters which give, respectively, the number of characters before the point, the number of characters after the point, and the number of characters after E. However, there is still only one digit before the point. If we do not want exponent notation then we simply specify the last parameter as zero and then get normal decimal notation.

The effect is shown by the following statements with the output given as a comment. For clarity, the output is surrounded by quotes and s designates a space; in reality there are no quotes and spaces are spaces.

```
Put(12.34);                    -- "s1.234000E+01"
Put(12.34, 3, 4, 2);           -- "ss1.2340E+1"
Put(12.34, 3, 4, 0);           -- "s12.3400"
```

The output of values of integer types follows a similar pattern. In this case we similarly instantiate the generic package Integer_IO inside Ada.Text_IO which applies to all integer types with the particular type such as My_Integer.

We can then call Put with a single parameter, the value of type My_Integer, in which case a standard default field is used, or we can add a further parameter specifying the field. The default field is the smallest that will accommodate all values of the type My_Integer allowing for a leading minus sign. Thus, for the range of My_Integer, the default field is 8. It should be noticed that if we specify a field which is too small then it is expanded as necessary. So

```
Put(123);              -- "sssss123"
Put(123, 4);           -- "s123"
Put(123, 0);           -- "123"
```

Simple text input is similarly performed by a call of Get with a parameter that must be a variable of the appropriate type.

A call of Get with a floating or integer parameter will expect us to type in an appropriate number at the keyboard; this must have a decimal point if the parameter is of a floating type. It should also be noted that leading blanks (spaces) and newlines are skipped. A call of Get with a parameter of type Character will read the very next character, and this can be neatly used for controlling the flow of an

interactive program, thus

```
C: Character;
…
Put("Do you want to stop?  Answer Y if so.");
Get(C);
if C ='Y' then
    …
```

For simple programs that do not have to be portable, the effort of doing the instantiations is not necesssary if we just use the predefined types Integer and Float, since the predefined library contains nongeneric versions with the names `Ada.Integer_Text_IO` and `Ada.Float_Text_IO`, respectively. So all we need to write is

```
use Ada.Integer_Text_IO, Ada.Float_Text_IO;
```

and then we can call `Put` and `Get` without more ado.

26.4.2 Numeric Library

The numeric library comprises the package `Ada.Numerics` plus a number of child packages. The package `Ada.Numerics` is as follows:

```
package Ada.Numerics is
  pragma Pure(Numerics);
  Argument_Error: exception;
  Pi: constant := 3.14159_26535_ …;
  e : constant := 2.71828_18284_ …;
end Ada.Numerics;
```

This contains the exception `Argument_Error` which is raised if something is wrong with the argument of a numeric function (such as attempting to take the square root of a negative number) and the two useful constants Pi and e.

One child package of `Ada.Numerics` provides the familiar elementary functions such as `Sqrt` and is another illustration of the use of the generic mechanism. Its specification is

```
generic
  type Float_Type is digits <>;
package Ada.Numerics.Generic_Elementary_Functions is
  function Sqrt(X: Float_Type'Base) return Float_Type'Base;
  …   -- and so on
end;
```

Again, there is a single generic parameter giving the floating type. In order to call the function `Sqrt` we must first instantiate the generic package much as we did for `Float_IO`, thus (assuming appropriate *with* and *use* clauses)

```
package My_Elementary_Functions is
  new Generic_Elementary_Functions(My_Float);
use My_Elementary_Functions;
```

and we can then write a call of `Sqrt` directly.

It should be noted that there is a nongeneric version for the type `Float` with the name `Elementary_Functions` for the convenience of those using the predefined type `Float`.

A little point to note is that the parameter and result of `Sqrt` is written as `Float_Type'Base`; the reason for this is to avoid unnecessary range checks.

We emphasize that the exception Ada.Numerics.Argument_Error is raised if the parameter of a function such as Sqrt is unacceptable. This contrasts with our hypothetical function Sqrt introduced earlier, which we assumed raised the predefined exception Constraint_Error when given a negative parameter. It is generally better to declare and raise our own exceptions rather than use the predefined ones.

Two other important child packages are those for the generation of random numbers. One returns a value of the type Float within the range 0 to 1 and the other returns a random value of a discrete type (a discrete type is an integer type or an enumeration type). We will look superficially at the latter; its specification is as follows

```
generic
  type Result_Subtype is (<>);
package Ada.Numerics.Discrete_Random is
  type Generator is limited private;
  function Random(Gen: Generator) return Result_Subtype;
  … --  plus other facilities
end Ada.Numerics.Discrete_Random;
```

This introduces a number of new points. The most important is the form of the generic formal parameter which indicates that the actual type must be a discrete type. The pattern echoes that of an enumeration type in much the same way as that for the floating generic parameter in Generic_Elementary_Functions echoed the declaration of a floating type. Thus we see that the discrete types are another example of a class of types.

A small point is that the type Generator is declared as limited. This simply means that assignment is not available for the type (or at least not for the partial view as seen by the client).

The random number generator is used as in the following fragment which illustrates the simulation of tosses of a coin

```
use Ada.Numerics;
type Coin is (Heads, Tails);
package Random_Coin is new Discrete_Random(Coin);
use Random_Coin;
G: Generator;
C: Coin;
loop
  C := Random(G);
  …
end loop;
…
```

Having declared the type Coin we then instantiate the generic package. We then declare a generator and use it as the parameter of successive calls of Random. The generator technique enables us to declare several generators and thus run several independent random sequences at the same time.

26.4.3 Running a Program

We are now in a position to put together a complete program using the proper input-output facilities. As an example, we will rewrite the procedure Print_Roots and also use the standard mathematical library. For simplicity we will first use the predefined type Float. The program becomes

```
with Ada.Text_IO;
with Ada.Float_Text_IO;
with Ada.Numerics.Elementary_Functions;
procedure Print_Roots is
  use Ada.Text_IO;
```

```
  use Ada.Float_Text_IO;
  use Ada.Numerics.Elementary_Functions;

  X: Float;
begin
  Put("Roots of various numbers");

  … -- and so on as before

end Print_Roots;
```

Note that we can put the *use* clauses inside the procedure `Print_Roots` as shown or we can place them immediately after the *with* clauses.

If we want to write a portable version, then the general approach would be

```
with Ada.Text_IO;
with Ada.Numerics.Generic_Elementary_Functions;
procedure Print_Roots is
   type My_Float is digits 7;
   package My_Float_IO is new Ada.Text_IO.Float_IO(My_Float);
   use My_Float_IO;
   package My_Elementary_Functions is
      new Ada.Numerics.Generic_Elementary_Functions(My_Float);
   use My_Elementary_Functions;

   X: My_Float;
begin
    Put("Roots of various numbers");

  … -- and so on as before

end Print_Roots;
```

To have to write all that introductory stuff each time is rather a burden, so we will put it in a standard package of our own and then compile it once so that it is permanently in our program library and can then be accessed without more ado. We include the type `My_Integer` as well, and write

```
with Ada.Text_IO;
with Ada.Numerics.Generic_Elementary_Functions;
package Etc is
  type My_Float is digits 7;
  type My_Integer is range -1000_000 .. +1000_000;
  package My_Float_IO is new Ada.Text_IO.Float_IO(My_Float);
  package My_Integer_IO is new Ada.Text_IO.Integer_IO(My_Integer);

  package My_Elementary_Functions is
      new Ada.Numerics.Generic_Elementary_Functions(My_Float);
end Etc;
```

and having compiled `Etc` our typical program can look like

```
with Ada.Text_IO, Etc;
use Ada.Text_IO, Etc;
procedure Program is
  use My_Float_IO, My_Integer_IO, My_Elementary_Functions;
  …
  …
end Program;
```

An alternative approach, rather than declaring everything in the one package Etc, is to first compile a tiny package just containing the types My_Float and My_Integer and then to compile the various instantiations as individual library packages (remember that a library unit can be just an instantiation).

```
package My_Numerics is
  type My_Float is digits 7;
  type My_Integer is range -1000_000 .. +1000_000;
end My_Numerics;

with My_Numerics; use My_Numerics;
with Ada.Text_IO;
package My_Float_IO is new Ada.Text_IO.Float_IO(My_Float);

with My_Numerics; use My_Numerics;
with Ada.Text_IO;
package My_Integer_IO is new Ada.Text_IO.Integer_IO(My_Integer);

with My_Numerics; use My_Numerics;
with Ada.Numerics.Generic_Elementary_Functions;
package My_Elementary_Functions is
  new Ada.Numerics.Generic_Elementary_Functions(My_Float);
```

With this approach we only need to include (via with clauses) the particular packages as required and our program is thus likely to be smaller if we do not need them all. We could even arrange the packages as an appropriate hierarchy, thus:

```
with Ada.Text_IO;
package My_Numerics.My_Float_IO is
    new Ada.Text_IO.Float_IO(My_Float);

with Ada.Text_IO;
package My_Numerics.My_Integer_IO is
    new Ada.Text_IO.Integer_IO(My_Integer);

with Ada.Numerics.Generic_Elementary_Functions;
package My_Numerics.My_Elementary_Functions is
    new Ada.Numerics.Generic_Elementary_Functions(My_Float);
```

One important matter remains to be addressed, and that is how to build a complete program. A major benefit of Ada is that consistency is maintained between separately compiled units so that the integrity of strong typing is preserved across compilation unit boundaries. It is therefore illegal to build a program out of inconsistent units. The exact means whereby this is achieved will depend upon the implementation.

A related issue is the order in which units are compiled. This is dictated by the idea of dependency. There are three main causes of dependency:

- A body depends upon the corresponding specification
- A child depends upon the specification of its parent
- A unit depends upon the specifications of those it mentions in a *with* clause

The key rule is that a unit can only be compiled if all those units on which it depends are present in the library environment. An important consequence is that although a specification and body can be compiled separately, the specification must always be present before the body can be compiled. Similarly if a unit is modified, then typically all those units that depend on it will also have to be recompiled in order to preserve consistency of the total program.

This brings us to the end of our brief survey of the main features of Ada. We have in fact encountered most of the main concepts, although very skimpily in some cases.

Important topics not discussed are

- Type parameterization through the use of discriminants. This is particularly important in OOP for enabling one type to be parameterized by another and provides the capabilities of multiple inheritance.
- More general access types. These provide the flexibility to reference objects on the stack as well as the heap. However, the rules are carefully designed to avoid "dangling references" without undue loss of flexibility. Access to subprogram types are important for interaction with software written in other languages and especially for programming call-back.
- Limited types. These are types which cannot be assigned. They are particularly helpful for modeling objects in the real world which by their nature are never copied. They are also useful for controlling resources.
- Controlled types. These are forms of tagged types which have automatic initialization and finalization. They enable the reliable programming of servers that keep track of resources used by clients in a foolproof manner. Controlled types also provide the capability for user-defined assignment.

It should also be noted that we have not discussed the specialized annexes at all. These provide important capabilities in a variety of areas. Very briefly, their contents are as follows:

Systems Programming. This covers a number of low-level features such as in-line machine instructions, interrupt handling, shared variable access, task identification, and per-task attributes. This annex is a prior requirement for the Real-Time Systems annex.

Real-Time Systems. This annex addresses various scheduling and priority issues including setting priorities dynamically, scheduling algorithms, and entry queue protocols. It also includes detailed requirements on a monotonic time package `Ada.Real_Time` (as distinct from the core package `Calendar` which might go backwards because of time zone or daylight-saving changes). There are also suggested tasking restrictions which might be appropriate for the development of very efficient run time systems for specialized applications.

Distributed Systems. The core language introduces the idea of a partition whereby one coherent "program" is distributed over a number of partitions each with its own main task. This annex defines active and passive partitions and interpartition communication using statically and dynamically bound remote subprogram calls.

Information Systems. The core language includes basic support for decimal types. This annex defines additional attributes and a number of packages providing detailed facilities for manipulating decimal values and conversion to and from external formats using picture strings.

Numerics. This annex addresses the special needs of the numeric community. It defines generic packages for the definition and manipulation of complex numbers. It also defines the accuracy requirements for numerics.

Safety and Security. This annex addresses restrictions on the use of the language and requirements of compilation systems for programs to be used in safety-critical and related applications where program security is vital.

As the reader will appreciate, many of these topics are important for avionics applications. The provision of these annexes ensures that these specialized capabilites are provided by all relevant implementations in a standard manner.

References

Barnes, J. (Ed.), 1997, *Ada 95 Rationale,* LNCS 1247, Springer Verlag.

Barnes, J.G.P., 1998. *Programming in Ada 95*, 2nd editiion, Addison-Wesley, Reading, MA.

Intermetrics Inc., 1995. *Annotated Ada 95 Reference Manual*, Intermetrics Inc., Burlington, MA.

International Organization for Standardization, 1995. *Information Technology — Programming Languages — Ada. Ada Reference Manual*. ISO/IEC 8652:1995(E).

Taft, S. T. and Duff, R.A. (Eds.), 1997, *Ada 95 Reference Manual,* LNCS 1246, Springer Verlag.

Further Information

The official definition of Ada 95 is the *Reference Manual for the Ada Programming Language* [ISO, 1995]. This has been reprinted [Taft and Duff, 1997]. An annotated version containing detailed explanatory information is also available [Intermetrics, 1995]. There is also an accompanying *Rationale* document giving a broad overview of the language and especially the reasons for the changes from Ada 83 [Barnes, 1997].

This chapter is based on a somewhat condensed form of Chapters 1 to 4 of *Programming in Ada 95* by the author and published by Addison-Wesley [Barnes, 1998]. The author is grateful to Addison-Wesley for their permission to use the material presented here. *Programming in Ada 95* is a comprehensive description of the core of Ada 95 with many illustrative examples plus exercises and answers. It extends to 23 chapters and includes full coverage of the predefined library, an overview of the annexes, and a CD containing an Ada compiler and a number of complete example programs.