Ada 95 Rationale

The Language

The Standard Libraries

January 1995

Ada 95
Rationale

The Language

The Standard Libraries

January 1995

Intermetrics, Inc.
733 Concord Ave.
Cambridge, Massachusetts 02138
(617) 661-1840

Foreword

I am delighted to write these words of introduction to this important document which accompanies the new Ada International Standard. It is most satisfying to see the efforts of all those involved with the language revision effort bearing fruit, and I am very confident that Ada 95 will prove to be a worthy successor to Ada 83, extending Ada's Software Engineering advantages to new user communities. With modern features such as Object-Oriented Programming and enhanced interfacing capabilities, Ada 95 will enable the flexible and reliable development of major applications in the coming years.

The new standard, officially ISO/IEC 8652:1995, thus marks an important milestone in Ada's history. Enormous contributions from some of the world's finest software engineers and programming language experts have gone into the revision effort, reflecting both current and anticipated user requirements. I would therefore like to take this opportunity to acknowledge the sterling efforts of all those involved in developing the new standard and in writing this rationale document.

The Ada 95 language revision was prepared by the Ada 9X Design Team based at Intermetrics, Inc., under the direction of William Carlson, Program Manager, and S. Tucker Taft, Technical Director. The Intermetrics team included Robert Duff (Oak Tree Software) and also consultants John Barnes, Ben Brosgol, and Offer Pazy.

A team led by John Goodenough (SEI) prepared the technical requirements specification for the Ada revision, based on preliminary work by the Institute for Defense Analyses under Audrey Hook to analyze the more than 750 revision requests submitted by the Ada community.

The following consultants to the Ada 9X Project contributed to the Specialized Needs Annexes: Ted Baker (SEI, Florida State Univ.), Ken Dritz (Argonne National Laboratory), Anthony Gargaro (Computer Sciences Corp.), John Goodenough (SEI), John McHugh (consultant), and Brian Wichmann (NPL: UK).

This work was regularly reviewed by the Ada 9X Distinguished Reviewers and the members of the ISO Ada 9X Rapporteur Group (XRG): Erhard Ploedereder, Chairman of DRs and XRG (University of Stuttgart: Germany); B. Bardin (Hughes); J. Barnes (consultant: UK); B. Brett (DEC); B. Brosgol (consultant); R. Brukardt (RR Software); N. Cohen (IBM); R. Dewar (NYU); G. Dismukes (Alsys [Telesoft]); A. Evans (consultant); A. Gargaro (Computer Sciences Corp.); M. Gerhardt (ESL); J. Goodenough (SEI); S. Heilbrunner (University of Salzburg: Austria); P. Hilfinger (UC/Berkeley); B. Källberg (CelsiusTech: Sweden); M. Kamrad II (Unisys); J. van Katwijk (Delft University of Technology: The Netherlands); V. Kaufman (Russia); P. Kruchten (Rational); R. Landwehr (CCI: Germany); C. Lester (Portsmouth Polytechnic: UK); L. Månsson (TELIA Research: Sweden); S. Michell (Multiprocessor Toolsmiths: Canada); M. Mills (US Air Force); D. Pogge (US Navy); K. Power (Boeing); O. Roubine (Verdix: France); A. Strohmeier (Swiss Federal Institute of Technology: Switzerland); W. Taylor (consultant: UK); J. Tokar (Tartan); E. Vasilescu (Grumman); J. Vladik (Prospeks s.r.o.: Czech Republic); S. Van Vlierberghe (OFFIS: Belgium).

Early contributors on the Ada 9X Design Team were C. Garrity, R. Hilliard, D. Rosenfeld, L. Shafer, W. White, and M. Woodger. Other valuable feedback influencing the revision process was provided by the Ada 9X Language Precision Team (Odyssey Research Associates), the Ada 9X User/Implementer Teams (AETECH, Tartan, Alsys [Telesoft]), the Ada 9X Implementation Analysis Team (New York University), and the Volunteer Reviewers group under Art Evans (consultant).

The principal authors of this rationale document were John Barnes and Ben Brosgol with valuable contributions from Ken Dritz, Offer Pazy and Brian Wichmann. Many thanks also to

Bill Taylor for providing the material upon which Appendix X is based, and to Tucker Taft and Robert Duff for their comments and review.

Very special thanks go to Virginia Castor, John Solomond and Don Reifer from the Ada Joint Program Office for supporting and sponsoring the Ada 9X Project since its inception, and to Bob Mathis, Convenor of ISO/IEC JTC1/SC22 Working Group 9, for shepherding the revision through the standardization process.

Last but not least, I want to extend my sincerest appreciation to the Ada community at large who have volunteered their time and energy on this project. Without your support we would not have been able to succeed.

Ada 95 is here. It is now up to the users, both those who have appreciated the engineering benefits of Ada 83 and others who may be frustrated with the hazards of their current techniques, to move forward and reap the benefits of this new language.

Christine M. Anderson,
Ada 9X Project Manager

Preface

Modern society is becoming very dependent upon software. Our transport systems, financial systems, medical systems and defense systems all depend to a very large degree upon software. As a consequence the safety of many human lives and much property now depends upon the reliable functioning of software. Moreover, the fall in the cost of hardware has now made possible the development of large software systems.

Ada is a programming language of special value in the development of large programs which must work reliably. This applies to most defense applications (from which background Ada evolved) and extends to many application domains. Indeed over half the Ada programs now being developed are for non-defense applications.

This document describes the rationale for Ada 95, the revised International Standard. Ada 95 increases the flexibility of Ada thus making it applicable to wider domains but retains the inherent reliability for which Ada has become noted. Important aspects of Ada 95 include

- Object Oriented Programming. Ada 95 includes full OOP facilities giving the flexibility of programming by extension which enables programs to be extended and maintained at lower cost.

- Hierarchical Libraries. The library mechanism now takes a hierarchical form which is valuable for the control and decomposition of large programs.

- Protected Objects. The tasking features of Ada are enhanced to incorporate a very efficient mechanism for multitask synchronized access to shared data. This is of special value for hard realtime systems.

These enhancements to Ada make Ada 95 an outstanding language. It adds the flexibility of languages such as C++ to the reliable Software Engineering framework provided and proven by Ada 83 over the past ten years.

Ada 95 is a natural evolution of Ada 83. The enhancements have been made without disturbing the existing investment in Ada 83 programs and programmers. Upward compatibility has been a prime goal and has been achieved within the constraints imposed by other requirements.

This document is in several parts. The first part is an Introduction to Ada 95; it presents a general discussion of the scope and objectives of Ada 95 and its major technical features. The second part contains a more detailed chapter by chapter account of the Core language. The third part covers the various Annexes which address the predefined environment and the needs of specialized application areas. Finally there are three appendices; Appendix X addresses the issue of upward compatibility with Ada 83 and shows that for normal programs the goal of compatibility has been achieved; Appendix Y summarizes the few changes since the two public drafts of the standard; Appendix Z summarizes the requirements and concludes that they have been met. This document will be of special value to program managers, team leaders and all software professionals with concern for the organized development of software.

Ada 95 deserves the attention of all members of the computing profession. It is a coherent and reliable foundation vehicle for developing the major applications of the next decade.

The Ada 9X Design Team,
January 1995

Contents

Part Two The Core Language

Part Three The Annexes

Part Four Appendices

Part One

Introduction to Ada 95

This first part is designed to give the reader a general appreciation of the overall scope and objectives of Ada 95 and its main technical features. The opening chapter describes the background to the development of the requirements leading to the new standard. The two main chapters give a technical view of the language from two different standpoints: one highlights the main new features such as type extension, the hierarchical library and the protected type and contains many illustrative examples; the other gives a complete overview of the whole language. The reader is recommended to read this first part before attempting to read the Ada 95 Reference Manual.

I Evolution of Ada 95

Ada is a modern programming language suitable for those application areas which benefit from the discipline of organized development, that is, *Software Engineering*; it is a general purpose language with special applicability to real-time and embedded systems. Ada was originally developed by an international design team in response to requirements issued by the United States Department of Defense [DoD 78].

Ada 95 is a revised version of Ada updating the 1983 ANSI Ada standard [ANSI 83] and the equivalent 1987 ISO standard [ISO 87] in accordance with ANSI and ISO procedures. (ANSI is the American National Standards Institute and ISO is the International Standards Organization.) This present document describes the overall Rationale for the revision and includes tutorial information for the new features.

I.1 The Revision Process

Although Ada was originally designed to provide a single flexible yet portable language for real-time embedded systems to meet the needs of the US DoD, its domain of application has expanded to include many other areas, such as large-scale information systems, distributed systems, scientific computation, and systems programming. Furthermore, its user base has expanded to include all major defense agencies of the Western world, the whole of the aerospace community and increasingly many areas in civil and private sectors such as telecommunications, process control and monitoring systems. Indeed, the expansion in the civil sector is such that civil applications now generate the dominant revenues of many vendors.

After some years of use and many implementations, a decision was made in 1988 to undertake a revision to Ada, leading to an updated ANSI/ISO standard. It is normal practice to automatically reconsider such standards every 5 or 10 years and to determine whether they should be abandoned, reaffirmed as they are, or updated. In the case of Ada, an update was deemed appropriate.

To understand the process it should be explained that ANSI, as the US national standards body, originally proposed that Ada become an ISO standard. It is normal practice that ANSI, as the originating body, should sponsor a revised standard as necessary. The US DoD acted as the agent of ANSI in managing the development of the revised standard. Within the US DoD, the Ada Joint Program Office (AJPO) is responsible for Ada and the Ada Board is a federal advisory committee which advises the AJPO.

The revision effort began in January 1988 when the Ada Board was asked to prepare a recommendation to the AJPO on the most appropriate standardization process to use in developing a revised Ada standard, known in the interim as Ada 9X. The recommendation [DoD 88] was delivered in September 1988 to Virginia Castor, the then Director of the Ada Joint Program Office, who subsequently established the Ada 9X Project for conducting the revision of the Ada Standard. Christine M Anderson was appointed Project Manager in October 1988. Close consultation with ISO was important to ensure that the needs of the whole world-wide Ada community (and not just the defense community) were taken into account and to ensure the timely adoption by ISO of the new standard. Accordingly a Memorandum of Understanding was established between the US DoD and ISO [ISO 90].

The Ada 9X program consists of three main phases: the determination of the Requirements for the revised language; the actual Development of the definition of the revised language; and the Transition into use from Ada 83 to Ada 9X.

The output of the Requirements Definition phase was the Ada 9X Requirements document [DoD 90], which specified the revision needs which had to be addressed. The Mapping/Revision phase defined the changes to the standard to meet those requirements; it achieved this in practice of course by defining the new standard.

The scope of the changes was guided by the overall objective of the Ada 9X effort [DoD 89a]:

> *Revise ANSI/MIL-STD-1815A-1983 to reflect current essential requirements with minimum negative impact and maximum positive impact to the Ada community.*

It is probably fair to observe that the changes which were deemed necessary are somewhat greater than might have been foreseen when the project first started in 1988. However, technology and man's understanding do not stand still and the changes now incorporated are very appropriate to the foreseen needs of a modern language for Software Engineering into the 21st Century.

I.2 The Requirements

The development of the requirements was a somewhat iterative process with a number of sub-phases. First, the community of users was invited to submit Revision Requests, secondly, these were sorted and analyzed in order to determine what was really implied by the requests, and finally, a coherent set of requirements was written.

Establishing the level of the requirements needed care. Quite naturally an individual user would often perceive a need in a manner which reflected a symptom of an underlying problem rather than the problem itself. It was very important that the requirements reflected the essence of the problem rather than what might in effect be simply one user's suggested solution to a part of a problem. One reason for this concern was to ensure that better, perhaps deeper and simpler, changes were not excluded by shallow requirements.

In some cases a complete understanding of a requirement could not be established and this led to the introduction of Study Topics. As its name implies a Study Topic described an area where something ought to be done but for some reason the feasibility of a solution was in doubt; perhaps the needs were changing rapidly or there were conflicting needs or implementing technology was unstable or it was simply that we had an incomplete understanding of some basic principles.

The general goal of the revision was thus to satisfy all of the Requirements and to satisfy as many and as much of the Study Topics as possible. Any unsatisfied Study Topics would be set aside for further consideration for a future Ada 0X (that is the next revision due perhaps around 2005).

The requirements document described 41 Requirements and 22 Study Topics. These were divided into a number of chapters which themselves form two broad groups. The first group covered topics of widespread applicability, whereas the second group addressed more specialized topics.

The first group consisted of the need to support International Character Sets (originally identified by several nations in the 1987 ISO standardization process), support for Programming Paradigms (Object Orientation was included in this category), Real-Time requirements, requirements for System Programming (such as Unsigned Integers), and General requirements. This last topic included specific matters such as error detection and general considerations of efficiency, simplicity and consistency; examples of a number of individually minor but irritating aspects of Ada 83 which fall into this category were given in an appendix.

The second, specialized, group consisted of requirements for Parallel Processing, Distributed Processing, Safety-Critical Applications, Information Systems, and Scientific and Mathematical Applications.

The breadth of these specialized requirements raised the specter of a language embracing so many application areas that it would become too costly to implement in its entirety for every architecture. On the other hand, one of the strengths of Ada is its uniformity and the last thing desired was a plethora of uncontrolled subsets. Accordingly, the Requirements document recommended the concept of a Core language plus a small number of Specialized Needs Annexes as a sensible way forward (with emphasis on keeping the number of such annexes very small). All validated compilers would have to implement the Core language and vendors could choose to implement zero, one or more annexes according to the needs of their selected market places.

The Requirements document also included general guidance on the approach to be taken to avoid unnecessary disruption to the Ada community. This covered vital matters such as the ease of implementation, object code efficiency, keeping a balance between change and need, and upward compatibility. It finally stressed that support for reliability, safety and long-term maintainability should continue to take precedence over short-term coding convenience.

Of all these requirements there was strong emphasis on the overriding goal of upward compatibility in order to preserve existing investment by the Ada community as a whole.

I.3 The Main User Needs

The Requirements reflected a number of underlying User Needs which had become apparent over the years since Ada was first defined. Apart from a small number of very specific matters such as the character set issue, the specialized areas and generally tidying up minor inconsistencies, four main areas stood out as needing attention:

- Interfacing. Ada 83 recognizes the importance of being able to interface to external systems by the provision of features such as representation clauses and pragmas. Nevertheless, it was sometimes not easy to interface certain Ada 83 programs to other language systems. A general need was felt for a more flexible approach allowing, for instance, the secure manipulation of references and the passing of procedures as parameters. An example arises when interfacing into GUI's where it is often necessary to pass a procedure as a parameter for call-back.

- Programming by Extension. This is closely allied to reusability. Although Ada's package and generic capability are an excellent foundation, nevertheless experience with the OO paradigm in other languages had shown the advantages of being able to extend a program without any modification to existing proven components. Not only does this save disturbing existing software thus eliminating the risk of introducing errors but it also reduces recompilation costs.

- Program Libraries. The Ada program library brings important benefits by extending the strong typing across the boundaries between separately compiled units. However, the flat nature of the Ada 83 library gave problems of visibility control; for example, it prevented two library packages from sharing a full view of a private type. A common consequence of the flat structure was that packages become large and monolithic. This hindered understanding and increased the cost of recompilation. A more flexible and hierarchical structure was necessary.

- Tasking. The Ada rendezvous paradigm is a useful model for the abstract description of many tasking problems. But experience had shown that a more static monitor-like approach was also desirable for common shared-data access applications. Furthermore, the Ada priority model needed revision in order to enable users to take advantage of the greater understanding of scheduling theory which had emerged since Ada 83 was defined.

The main changes incorporated into Ada 95 reflect the response to the Requirements in meeting these four key needs while at the same time meeting an overriding need for upward compatibility in order to minimize the effort and cost of transition.

I.4 The Approach

In responding to the revision requirements, the Revision team followed the inspiration of Jean Ichbiah (who led the original design team), both in remaining faithful to the principles underlying the original Ada design, and in the approach to the revision process. To quote Dr Ichbiah in his introduction to John Barnes' textbook on Ada [Barnes 82]:

> *Clearly, further progress can only come by a reappraisal of implicit assumptions underlying certain compromises. Here is the major contradiction in any design work. On the one hand, one can only reach an harmonious integration of several features by immersing oneself into the logic of the existing parts; it is only in this way that one can achieve a perfect combination. On the other hand, this perception of perfection, and the implied acceptance of certain unconscious assumptions, will prevent further progress.*

Wherever possible, enhanced functionality in Ada 95 has been achieved by seeking and eliminating such unnecessary assumptions, thereby permitting the generalization of features already in Ada, and the removal of special cases and restrictions. A careful analysis was made of Ada 83, of language study notes prepared during the original design process and of the many Ada Issues and other comments that have since been received. Based on this analysis, and on the Ada community's experience in implementing and using Ada during the past ten years, the team identified limitations that, while they were included to simplify implementations and/or to lower risk when the language was first standardized, are no longer necessary. They also drew upon the wealth of practical experience gained during the 1980's in the use of object-oriented design methods, object-oriented programming languages, and real-time programming made possible by Ada.

The resulting Ada 95 revision is upwardly compatible for virtually all existing Ada 83 applications. Most incompatibilities are restricted to pathological combinations of features that are rarely used in practice. Total upward compatibility would not have allowed the correction of certain errors and certainly would not have allowed the enhancements needed to satisfy many of the requirements. Indeed, as discussed in Appendix A in more detail, no language revision has ever achieved total upward compatibility. The careful attention to this issue in the design of Ada 95 means that the expected transition costs for existing Ada programs are anticipated to be very low indeed.

Following the guidance of the Requirements document, Ada 95 comprises a Core language, which must be implemented in its entirety, plus several Specialized Needs Annexes which provide extended features for specific application areas. These Annexes provide standard definitions for application-specific packages, pragmas, attributes, and capacity and performance characteristics of implementations. The Annexes address the following areas: Systems Programming, Real-Time Systems, Distributed Systems, Information Systems, Numerics, and Safety and Security.

It should be noted that the Core language includes a considerably extended predefined environment covering important functionality such as mathematical functions and string handling. Much of the functionality of this predefined environment and the specialized annexes is already provided by implementations of Ada 83 but in non-standard ways; providing this within the standard will thus increase portability between implementations.

I.5 Using this Document

This document provides a description of the main features of Ada 95 and the rationale for the changes from Ada 83. It follows the inspiration of the rationale for Ada 83 [IBFW 86] by blending exposition with explanation.

It is in four parts, this first part is designed to give the reader a general appreciation for the scope and objectives of Ada 95 and its major features. Chapter II highlights the main changes — primarily the four key areas outlined in I.3. The intent is to provide a technically oriented management overview illustrating, with examples, the prime benefits of Ada 95 with respect to Ada 83. By contrast, Chapter III provides an overview of the whole language in a discursive style and introduces new terminology where appropriate.

The second and third parts take the discussion a step further. The second part covers the Core language per se; it addresses those important features not discussed in this first part and gives more detail of the rationale including alternatives that were considered and rejected. The third part similarly addresses the predefined environment and the various Specialized Needs Annexes.

Finally the fourth part contains three appendices. Appendix X summarizes the incompatibilities thereby giving guidance to existing Ada 83 programmers in order to smooth their transition to Ada 95. Appendix Y summarizes the main differences between the final International Standard (which this edition of the rationale describes) and the previous two drafts, the Draft International Standard [DIS 94] and the Committee Draft [CD 93]. Appendix Z summarizes the original requirements and briefly analyzes how they have been met by Ada 95.

It is generally assumed that the reader is familiar with Ada 83. This first part should be read before approaching the Ada 95 Reference Manual [RM95] which should then be read in parallel with Parts Two and Three. It should also be noted that the general intent of all parts of this document is to address the broad picture. The rationale for minute detail is admirably addressed in the Annotated Ada 95 Reference Manual [AARM] which is a version of [RM95] including embedded non-normative discussion mainly of interest to language experts and compiler writers.

The chapters in this first part have Roman numbers; the chapters in Parts Two and Three have numbers and letters essentially corresponding to the sections and annexes of [RM 95]. The appendices in Part Four are called X, Y and Z to avoid confusion with the annex chapters. All chapters thus have a unique identifier for ease of cross reference.

II Highlights of Ada 95

The brightest highlights of Ada 95 are its inherent reliability and its ability to provide abstraction through the package and private type. These features already exist in Ada 83 and so in a real sense Ada 83 already contains the best of Ada 95. Indeed, Ada 83 is already a very good language. However, time and technology do not stand still, and Ada 95 is designed to meet increased requirements which have arisen from three directions. These are: feedback from the use of existing paradigms; additional market requirements to match evolving hardware capability; and increased fundamental understanding which has introduced new paradigms. As we will see, Ada 95 follows on from the tradition of excellence of Ada 83 and meets these additional requirements in an outstanding manner.

One of the great strengths of Ada 83 is its reliability. The strong typing and related features ensure that programs contain few surprises; most errors are detected at compile time and of those remaining many are detected by runtime constraints. This aspect of Ada considerably reduces the costs and risks of program development compared for example with C or its derivatives such as C++.

However, Ada 83 has proved to be somewhat less flexible than might be desired in some circumstances. For example, it has not always proved straightforward to interface to non-Ada systems. Moreover, the type model coupled with the flat library mechanism can cause significant costs through the need for apparently unnecessary recompilation.

A prime goal of the design of Ada 95 has thus been to give the language a more open and extensible feel without losing the inherent integrity and efficiency of Ada 83. That is to keep the Software Engineering but allow more flexibility.

The additions in Ada 95 which contribute to this more flexible feel are type extension, the hierarchical library and the greater ability to manipulate pointers or references.

As a consequence, Ada 95 incorporates the benefits of Object Oriented languages without incurring the pervasive overheads of languages such as SmallTalk or the insecurity brought by the weak C foundation in the case of C++. Ada 95 remains a very strongly typed language but provides the prime benefits of all key aspects of the Object Oriented paradigm.

Another area of major change in Ada 95 is in the tasking model where the introduction of protected types allows a more efficient implementation of standard problems of shared data access. This brings 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.

Ada 95 also incorporates numerous other minor changes reflecting feedback from the use of existing features as well as specific new features addressing the needs of specialized applications and communities.

This chapter highlights the major new features of Ada 95 and the consequential benefits as seen by the general Ada user.

II.1 Programming by Extension

The key idea of programming by extension is the ability to declare a new type that refines an existing parent type by inheriting, modifying or adding to both the existing components and the operations of the parent type. A major goal is the reuse of existing reliable software without the need for recompilation and retesting.

Type extension in Ada 95 builds upon the existing Ada 83 concept of a derived type. In Ada 83, a derived type inherited the operations of its parent and could add new operations; however, it was not possible to add new components to the type. The whole mechanism was thus somewhat static. By contrast, in Ada 95 a derived type can also be extended to add new components. As we will see, the mechanism is much more dynamic and allows greater flexibility through late binding and polymorphism.

In Ada 95, record types may be extended on derivation provided that they are marked as tagged. Private types implemented as record types can also be marked as tagged. As the name implies, a tagged type has an associated tag. The word *tag* is familiar from Pascal where it is used to denote what in Ada is known as a discriminant; as we shall see later, the Ada 95 tag is effectively a hidden discriminant identifying the type and so the term is very appropriate.

As a very simple example suppose we wish to manipulate various kinds of geometrical objects which form some sort of hierarchy. All objects will have a position given by their *x*- and *y*-coordinates. So we can declare the root of the hierarchy as

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

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. Note that **`tagged`** is of course a new reserved word; Ada 95 has a small number (six) of such new reserved words.

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`.

Just as in Ada 83, derived types inherit the operations which "belong" to the parent type — these are called *primitive operations* in Ada 95. User-written subprograms are classed as primitive operations if they are declared in the same package specification as the type and have the type as parameter or result.

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. This would also be inherited by the type `Circle` but would be inappropriate; it would be more sensible to explicitly declare

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

which will override the inherited operation.

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

```
O: Object := (1.0, 0.5);
C: Circle := (0.0, 0.0, 12.2);
...
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 46.8);
```

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.

We now consider a more practical example which illustrates the use of tagged types to build a system as a hierarchy of types and packages. We will see how this allows the system to be extended without recompilation of the central part of the system. By way of illustration we start by showing the rigid way this had to be programmed in Ada 83 by the use of variants.

Our system represents the processing of alerts (alarms) in a ground mission control station. Alerts are of three levels of priority. Low level alerts are merely logged, medium level alerts cause a person to be assigned to deal with the problem and high level alerts cause an alarm bell to ring if the matter is not dealt with by a specified time. In addition, a message is displayed on various devices according to its priority.

First consider how this might have be done in Ada 83

```
with Calendar;
package Alert_System is

   type Priority is (Low, Medium, High);
   type Device is (Teletype, Console, Big_Screen);

   type Alert(P: Priority) is
      record
         Time_Of_Arrival: Calendar.Time;
         Message: Text;
         case P is
            when Low => null;
            when Medium | High =>
               Action_Officer: Person;
               case P is
                  when Low | Medium => null;
                  when High =>
                     Ring_Alarm_At: Calendar.Time;
               end case;
         end case;
      end record;

   procedure Display(A: in Alert; On: in Device);
   procedure Handle(A: in out Alert);
   procedure Log(A: in Alert);
   procedure Set_Alarm(A: in Alert);

end Alert_System;
```

Each alert is represented as a discriminated record with the priority as the discriminant. Perhaps surprisingly, the structure and processing depend on this discriminant in a quite complex manner. One immediate difficulty is that we are more or less obliged to use nested variants because of the rule that all the components of a record have to have different identifiers. The body of the procedure Handle might be

```
procedure Handle(A: in out Alert) is
begin
   A.Time_Of_Arrival := Calendar.Clock;
   Log(A);
   Display(A, Teletype);
   case A.P is
      when Low => null;  -- nothing special
      when Medium | High =>
```

```
            A.Action_Officer := Assign_Volunteer;
            Display(A, Console);
            case A.P is
               when Low | Medium => null;
               when High =>
                  Display(A, Big_Screen);
                  Set_Alarm(A);
            end case;
      end case;
   end Handle;
```

One problem with this approach is that the code is curiously complex due to the nested structure and consequently hard to maintain and error-prone. If we try to avoid the nested case statement then we have to repeat some of the code.

A more serious problem is that if, for example, we need to add a further alert category, perhaps an emergency alert (which would mean adding another value to the type `Priority`), then the whole system will have to be modified and recompiled. Existing reliable code will then be disturbed with the risk of subsequent errors.

In Ada 95 we can use a series of tagged types with a distinct procedure `Handle` for each one. This completely eliminates the need for case statements or variants and indeed the type `Priority` itself is no longer required because it is now inherent in the types themselves (it is implicit in the tag). The package specification now becomes

```
   with Calendar;
   package New_Alert_System is

      type Device is (Teletype, Console, Big_Screen);

      type Alert is tagged
         record
            Time_Of_Arrival: Calendar.Time;
            Message: Text;
         end record;

      procedure Display(A: in Alert; On: in Device);
      procedure Handle(A: in out Alert);
      procedure Log(A: in Alert);

      type Low_Alert is new Alert with null record;

      type Medium_Alert is new Alert with
         record
            Action_Officer: Person;
         end record;

      -- now override inherited operation
      procedure Handle(MA: in out Medium_Alert);

      type High_Alert is new Medium_Alert with
         record
            Ring_Alarm_At: Calendar.Time;
         end record;

      procedure Handle(HA: in out High_Alert);
      procedure Set_Alarm(HA: in High_Alert);

   end New_Alert_System;
```

In this formulation the variant record is replaced with the tagged type `Alert` and three types derived from it. Note that Ada 95 allows a type to be derived in the same package specification as the parent and to inherit all the primitive operations but we cannot add any new primitive operations to the parent after a type has been derived from it. This is different to Ada 83 where the operations were not derivable until after the end of the package specification. This change allows the related types to be conveniently encapsulated all in the same package.

The type `Low_Alert` is simply a copy of `Alert` (note **`with null record;`**) and could be dispensed with although it maintains equivalence with the Ada 83 version; `Low_Alert` inherits the procedure `Handle` from `Alert`. The type `Medium_Alert` extends `Alert` and provides its own procedure `Handle` thus overriding the inherited version. The type `High_Alert` further extends `Medium_Alert` and similarly provides its own procedure `Handle`. Thus instead of a single procedure `Handle` containing complex case statements the Ada 95 solution distributes the logic for handling alerts to each specific alert type without any redundancy.

Note that `Low_Alert`, `Medium_Alert` and `High_Alert` all also inherit the procedures `Display` and `Log` but without change. Finally, `High_Alert` adds the procedure `Set_Alarm` which is not used by the lower alert levels and thus it seems inappropriate to declare it for them.

The package body is as follows

```
package body New_Alert_System is

   procedure Handle(A: in out Alert) is
   begin
      A.Time_Of_Arrival := Calendar.Clock;
      Log(A);
      Display(A, Teletype);
   end Handle;

   procedure Handle(MA: in out Medium_Alert) is
   begin
      Handle(Alert(MA)); -- handle as plain alert
      MA.Action_Officer := Assign_Volunteer;
      Display(MA, Console);
   end Handle;

   procedure Handle(HA: in out High_Alert) is
   begin
      Handle(Medium_Alert(HA)); -- conversion
      Display(HA, Big_Screen);
      Set_Alarm(HA);
   end Handle;

   procedure Display(A: in Alert; On: in Device) is separate;
   procedure Log(A: in Alert) is separate;

   procedure Set_Alarm(HA: in High_Alert) is separate;

end New_Alert_System;
```

Each distinct body for `Handle` contains just the code relevant to the type and delegates additional processing back to its parent using an explicit type conversion. Note carefully that all type checking is static and so no runtime penalties are incurred with this structure (the variant checks have been avoided).

In the Ada 95 model a new alert level (perhaps `Emergency_Alert`) can be added without recompilation (and perhaps more importantly, without retesting) of the existing code.

```
with New_Alert_System;
package Emergency_Alert_System is
```

```
      type Emergency_Alert is
         new New_Alert_System.Alert with private;

      procedure Handle(EA: in out Emergency_Alert);

      procedure Display(EA: in Emergency_Alert;
                        On: in New_Alert_System.Device);

      procedure Log(EA: in Emergency_Alert);

   private
      ...
   end Emergency_Alert_System;
```

In the Ada 83 model extensive recompilation would have been necessary since the variant records would have required redefinition. Thus we see that Ada 95 truly provides Programming by Extension.

II.2 Class Wide Programming

The facilities we have seen so far have allowed us to define a new type as an extension of an existing one. We have introduced the different kinds of alerts as distinct but related types. What we also need is a means to manipulate *any* kind of alert and to process it accordingly. We do this through the introduction of the notion of class-wide types.

With each tagged type `T` there is an associated type `T'Class`. This type comprises the union of all the types in the tree of derived types rooted at `T`. The values of `T'Class` are thus the values of `T` and all its derived types. Moreover a value of any type derived from `T` can be implicitly converted to the type `T'Class`.

Thus in the case of the type `Alert` the tree of types is as shown in Figure II-1.

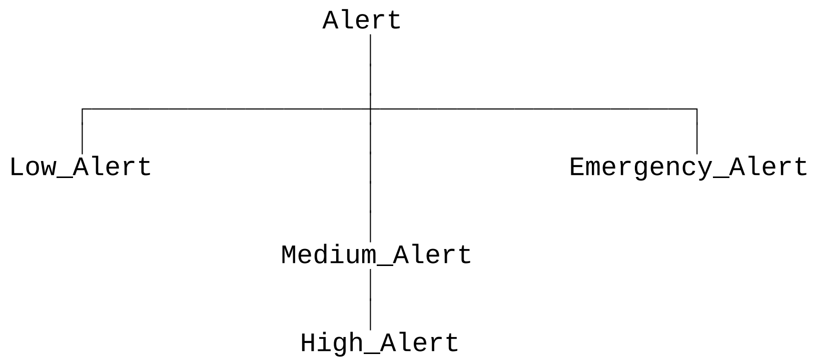

Figure II-1: A Tree of Types

A value of any of the alert types can be implicitly converted to `Alert'Class`. Note carefully that `Medium_Alert'Class` is not the same as `Alert'Class`; the former consists just of `Medium_Alert` and `High_Alert`.

Each value of a class-wide type has a tag which identifies its particular type from other types in the tree of types at runtime. It is the existence of this tag which gives rise to the term tagged types.

The type T'Class is treated as an unconstrained type; this is because we cannot possibly know how much space could be required by any value of a class-wide type because the type might be extended. As a consequence, although we can declare an object of a class-wide type we must initialize it and it is then constrained by the tag.

However, we can declare an access type referring to a class-wide type in which case the access could designate any value of the class-wide type from time to time. The use of access types is therefore a key factor in class-wide programming. Moreover, a parameter of a procedure can also be of a class-wide type. There is a strong analogy between class-wide types and other unconstrained types such as array types.

We can now continue our example by considering how we might buffer up a series of alerts and process them in sequence. The whole essence of the problem is that such a central routine cannot know of the individual types because we need it to work (without recompilation) even if we extend the system by adding a new alert type to it.

The central routine could thus take a class-wide value as its parameter so we might have

```
procedure Process_Alerts(AC: in out Alert'Class) is
   ...
begin
   ...
   Handle(AC); -- dispatch according to tag
   ...
end Process_Alerts;
```

In this case we do not know which procedure Handle to call until runtime because we do not know which specific type the alert belongs to. However, AC is of a class-wide type and so its value includes a tag indicating the specific type of the value. The choice of Handle is then determined by the value of this tag; the parameter is then implicitly converted to the appropriate specific alert type before being passed to the appropriate procedure Handle.

This runtime choice of procedure is called *dispatching* and is key to the flexibility of class-wide programming.

Before being processed, the alerts might be held on a heterogeneous queue using an access type

```
type Alert_Ptr is access Alert'Class;
```

and the central routine could manipulate the alerts directly from such a queue

```
procedure Process_Alerts is
   Next_Alert: Alert_Ptr;
begin
   ...
   Next_Alert := -- get next alert
   ...
   Handle(Next_Alert.all);  -- dispatch to appropriate Handle
   ...
end Process_Alerts;
```

In this case, the value of the object referred to by Next_Alert is of a class-wide type and so includes a tag indicating the specific type. The parameter Next_Alert.**all** is thus dereferenced, the value of the tag gives the choice of Handle and the parameter is then implicitly converted as before and then passed to the chosen procedure Handle.

Dispatching may be implemented as a simple indirect jump through a table of subprograms indexed by the primitive operations such as Handle. This is generally much more efficient than the alternative of variant records and case statements, with their attendant variant checks.

II.3 Abstract Types and Subprograms

The final topic to be introduced in this brief introduction to the Object Oriented features of Ada 95 is the concept of abstract tagged types and abstract subprograms. These are marked as abstract in their declaration. The purpose of an abstract type is to provide a common foundation upon which useful types can be built by derivation. An abstract subprogram is a sort of place holder for an operation to be provided (it does not have a body).

An abstract tagged type can have abstract primitive subprograms and these dispatch. An abstract type on its own is of little use because we cannot declare an object of an abstract type.

Upon derivation from an abstract type we can provide actual subprograms for the abstract subprograms of the parent type (and it is in this sense that we said they were place holders). If all the abstract subprograms are replaced by proper subprograms then the type need not be declared as abstract and we can then declare objects of the type in the usual way. (The rules ensure that dispatching always works.)

Returning now to our example of processing alerts we could reformulate this so that the root type `Alert` was just an abstract type and then build the specific types upon this. This would enable us to program and compile all the general infrastructure routines for processing all alerts such as `Process_Alerts` in the previous section without any concern at all for the individual alerts and indeed before deciding what they should contain.

The baseline package could then simply become

```
package Base_Alert_System is

   type Alert is abstract tagged null record;
   procedure Handle(A: in out Alert) is abstract;

end Base_Alert_System;
```

in which we declare the type `Alert` as a tagged null record with just the procedure `Handle` as an abstract subprogram; this does not have a body. (Note the abbreviated form for a null record declaration which saves us having to write **`record null; end record;`**)

We can now develop our alert infrastructure and then later add the normal alert system containing the three levels of alerts thus

```
with Calendar;
with Base_Alert_System;
package Normal_Alert_System is

   type Device is (Teletype, Console, Big_Screen);

   type Low_Alert is new Base_Alert_System.Alert with
      record
         Time_Of_Arrival: Calendar.Time;
         Message: Text;
      end record;

   -- now provide actual subprogram for abstract one
   procedure Handle(LA: in out Low_Alert);

   procedure Display(LA: in Low_Alert; On: in Device);
   procedure Log(LA: in Low_Alert);

   type Medium_Alert is new Low_Alert with
      record
         Action_Officer: Person;
      end record;
```

```
   procedure Handle(MA: in out Medium_Alert);

   type High_Alert is new Medium_Alert with
      record
         Ring_Alarm_At: Calendar.Time;
      end record;

   procedure Handle(HA: in out High_Alert);
   procedure Set_Alarm(HA: in High_Alert);

end Normal_Alert_System;
```

In this revised formulation we must provide a procedure `Handle` for `Low_Alert` to meet the promise of the abstract type. The procedures `Display` and `Log` now take a parameter of `Low_Alert` and the type `Medium_Alert` is more naturally derived from `Low_Alert`.

Note carefully that we did not make `Display` and `Log` abstract subprograms in the package `Base_Alert_System`. There was no need; it is only `Handle` that is required by the general infrastructure such as the procedure `Process_Alerts` and to add the others would weaken the abstraction and clutter the base level.

Corresponding changes are required to the package body; the procedure `Handle` previously applying to `Alert` now applies to `Low_Alert` and in the procedure `Handle` for `Medium_Alert` we need to change the type conversion in the call to the "parent" `Handle` which is of course now the procedure `Handle` for `Low_Alert`. The two procedures thus become

```
procedure Handle(LA: in out Low_Alert) is
begin
   LA.Time_Of_Arrival := Calendar.Clock;
   Log(LA);
   Display(LA, Teletype);
end Handle;

procedure Handle(MA: in out Medium_Alert) is
begin
   Handle(Low_Alert(MA)); -- handle as low alert
   MA.Action_Officer := Assign_Volunteer;
   Display(MA, Console);
end Handle;
```

When we now add our `Emergency_Alert` we can choose to derive this from the baseline `Alert` as before or perhaps from some other point in the tree picking up the existing facilities of one of the other levels.

II.4 Summary of Type Extension

The key points we have seen are as follows.

Ada 95 introduces the notion of tagged types. Only record (and private) types can be tagged. Values of tagged types carry a tag with them. The tag indicates the specific type. A tagged type can be extended on derivation with additional components.

Primitive operations of a type are inherited on derivation. The primitive operations are those implicitly declared, plus, in the case of a type declared in a package specification, all subprograms with a parameter or result of that type also declared in the package specification. Primitive operations can be overridden on derivation and further ones added.

A tagged type can be declared as abstract and can have abstract primitive subprograms. An abstract subprogram does not have a body but one can be provided on derivation. An abstract type provides a foundation for building specific types with some common protocol.

`T'Class` denotes the class-wide type rooted at `T`. Implicit conversion is allowed to values of `T'Class`. Objects and parameters of `T'Class` are treated as unconstrained. An appropriate access type can designate any value of `T'Class`.

Calling a primitive operation with an actual parameter of a class-wide type results in dispatching: that is the runtime selection of the operation according to the tag. This is often called late binding — a key property of Object Oriented languages.

Another term commonly encountered is polymorphism. Class-wide types are said to be polymorphic because their values are of different "shapes" (from the Greek *poly*, many, and *morphe*, form). Polymorphism is another property of Object Oriented languages.

One of the main advantages of type extension is that it can be done without recompiling and retesting an existing stable system. This is perhaps the most important overall characteristic of Object Oriented languages.

II.5 Dynamic Selection

In the previous section we mentioned late binding; this simply means that the procedure to be called is identified late in the compile-link-run process. All procedure calls were bound early in Ada 83 and this was one reason why the language felt so static; even the generic mechanism only deferred binding to instantiation which is still essentially a compile time process.

There were a number of reasons for taking such a static approach in Ada 83. There was concern for the implementation cost of dynamic binding, it was also clear that the presence of dynamic binding would reduce the provability of programs, and moreover it was felt that the introduction of generics where subprograms could be passed as parameters would cater for practical situations where formal procedure parameters were used in other languages.

However, the absence of dynamic binding in Ada 83 has been unfortunate. It is now realized that implementation costs are trivial and not necessarily pervasive; provability is not a relevant argument because we now know that in any safety-critical software where mathematical provability is a real issue, we only use a small subset of the language. And furthermore, the generic mechanism has proved not to be a sufficiently flexible alternative anyway.

We have seen how dispatching in Ada 95 is one mechanism for late binding. Another is provided by the manipulation of subprogram values through an extension of access types.

In Ada 95 an access type can refer to a subprogram; an access-to-subprogram value can be created by the `Access` attribute and a subprogram can be called indirectly by dereferencing such an access value. Thus we can write

```
type Trig_Function is access function(F: Float) return Float;
T: Trig_Function;
X, Theta: Float;
```

and `T` can then "point to" functions such as `Sin`, `Cos` and `Tan`. We can then assign an appropriate access-to-subprogram value to `T` by for example

```
T := Sin'Access;
```

and later indirectly call the subprogram currently referred to by `T` as expected

```
X := T(Theta);
```

which is really an abbreviation for

```
X := T.all(Theta);
```

Just as with many other uses of access types the `.all` is not usually required but it would be necessary if there were no parameters.

The access to subprogram mechanism can be used to program general dynamic selection and to pass subprograms as parameters. It allows program call-back to be implemented in a natural and efficient manner.

There are a number of rules which ensure that access to subprogram values cannot be misused. Conformance matching ensures that the subprogram always has the correct number and type of parameters and there are rules about accessibility that ensure that a subprogram is not called out of context. Flexibility is thus gained without loss of integrity.

Simple classic numerical algorithms can now be implemented in Ada 95 in the same way as in languages such as Fortran but with complete security. Thus an integration routine might have the following specification

```
type Integrand is access function(X: Float) return Float;

function Integrate(F: Integrand; From, To: Float;
                   Accuracy: Float := 1.0E-7) return Float;
```

and we might then write

```
Area := Integrate(Log'Access, 1.0, 2.0);
```

which will compute the area under the curve for $\log(x)$ from 1.0 to 2.0. Within the body of the function `Integrate` there will be calls of the actual subprogram passed as a parameter; this is a simple form of call-back.

A common paradigm within the process industry is to implement sequencing control through successive calls of a number of interpreter actions. A sequence compiler might interactively build an array of such actions which are then obeyed. Thus we might have

```
type Action is access procedure;
Action_Sequence: array(1 .. N) of Action;

... -- build the array

    -- and then obey it
for I in Action_Sequence'Range loop
   Action_Sequence(I).all;
end loop;
```

It is of course possible for a record (possibly private) to contain components whose types are access to subprogram types. Consider the following example of a package which provides facilities associated with the actions obtained when we press keys on our keyboard or perhaps click our mouse on a window button.

```
package Push_Buttons is

   type Button is private;

   type Button_Response is access procedure(B: in out Button);

   function Create(...) return Button;

   procedure Push(B: in out Button);

   procedure Set_Response(B: in out Button;
                          R: in Button_Response);
```

```
   procedure Default_Response(B: in out Button);

   ...

private
   type Button is
      record
         Response: Button_Response := Default_Response'Access;
         ...    -- other aspects of the button
      end record;
end Push_Buttons;
```

A button is represented as a private record containing a number of components describing properties of the button (position on screen for example). One component is an access to a procedure which is the action to be executed when the button is pushed. Note carefully that the button value is passed to this procedure as a parameter so that the procedure can obtain access to the other components of the record describing the button. The procedure `Create` fills in these other components and other functions (not shown) provide access to them. The procedure `Push` invokes the action of clicking the mouse and an appropriate default procedure is provided which warns the user if the button has not been set. The body might be as follows

```
package body Push_Buttons is

   procedure Push(B: in out Button) is
   begin
      B.Response(B);  -- indirect call
   end Push;

   procedure Set_Response(B: in out Button;
                          R: in Button_Response) is
   begin
      B.Response := R;  -- set procedure value in record
   end Set_Response;

   procedure Default_Response(B: in out Button) is
   begin
      Put("Button not set");
      Monitor.Beep;
   end Default_Response;

   ...

end Push_Buttons;
```

We can now set the specific actions we want when a button is pushed. Thus we might want some emergency action to take place when a big red button is pushed.

```
Big_Red_Button: Button;

procedure Emergency(B: in out Button) is
begin
   -- call fire brigade etc
end Emergency;

...

Set_Response(Big_Red_Button, Emergency'Access);
```

```
...

Push(Big_Red_Button);
```

The reader will realize that the access to subprogram mechanism coupled with the inheritance and dispatching facilities described earlier enable very flexible yet secure dynamic structures to be programmed.

II.6 Other Access Types

We have just seen how access types in Ada 95 have been extended to provide a means of manipulating subprogram values. Access types have also been extended to provide more flexible access to objects.

In Ada 83, access values could only refer to objects dynamically created through the allocator mechanism (using **new**). It was not possible to access objects declared in the normal way. This approach was inherited from Pascal which had similar restrictions and was a reaction against the very flexible approach adopted by Algol 68 and C which can give rise to dangerous dangling references.

However, the ability to manipulate pointers is very valuable provided the risks can be overcome. The view taken by Ada 83 has proved unnecessarily inflexible, especially when interfacing to external systems possibly written in other languages.

In Ada 95 we can declare a general access type such as

```
type Int_Ptr is access all Integer;
```

and we can then assign the "address" of any variable of type `Integer` to a variable of type `Int_Ptr` provided that the designated variable is marked as aliased. So we can write

```
IP: Int_Ptr;
I: aliased Integer;
...
IP := I'Access;
```

and we can then read and update the variable `I` through the access variable `IP`. Note once more the use of `'Access`. Note also that **aliased** is another new reserved word.

As with access to subprogram values there are rules that (at compile time) ensure that dangling references cannot arise.

A variation is that we can restrict the access to be read-only by replacing **all** in the type definition by **constant**. This allows read-only access to any variable and also to a constant thus

```
type Const_Int_Ptr is access constant Integer;
CIP: Const_Int_Ptr;
I: aliased Integer;
C: aliased constant Integer := 1815;
```

followed by

```
CIP := I'Access;  -- access to a variable, or
CIP := C'Access;  -- access to a constant
```

The type accessed by a general access type can of course be any type such as an array or record. We can thus build chains from records statically declared. Note that we can also use an allocator to generate general access values. Our chain could thus include a mixture of records from both storage mechanisms.

Finally note that the accessed object could be a component of a composite type. Thus we could point into the middle of a record (provided the component is marked as aliased). In a fast implementation of Conway's Game of Life a cell might contain access values directly referencing the component of its eight neighbors containing the counter saying whether the cell is alive or dead.

```
type Ref_Count is access constant Integer range 0 .. 1;
type Ref_Count_Array is array (Integer range <>) of Ref_Count;

type Cell is
   record
      Life_Count: aliased Integer range 0 .. 1;
      Total_Neighbor_Count: Integer range 0 .. 8;
      Neighbor_Count: Ref_Count_Array(1 .. 8);
      ...
   end record;
```

We can now link the cells together according to our model by statements such as

```
This_Cell.Neighbor_Count(1) := Cell_To_The_North.Life_Count'Access;
```

and then the heart of the computation which computes the sum of the life counts in the neighbors might be

```
C.Total_Neighbor_Count := 0;
for I in C.Neighbor_Count'Range loop
   C.Total_Neighbor_Count :=
        C.Total_Neighbor_Count + C.Neighbor_Count(I).all;
end loop;
```

Observe that we have given the type `Ref_Count` and the component `Life_Count` the same static subtypes so that they can be checked at compile time; this is not essential but avoids a run-time check that would otherwise be required if they did not statically match.

General access types can also be used to program static ragged arrays as for example a table of messages of different lengths. The key to this is that the accessed type can be unconstrained (such as `String`) and thus we can have an array of pointers to strings of different lengths. In Ada 83 we would have had to allocate all the strings dynamically using an allocator.

In conclusion we have seen how the access types of Ada 83 have been considerably enhanced in Ada 95 to allow much more flexible programming which is especially important in open systems while nevertheless retaining the inherent security missing in languages such as C and C++.

II.7 Hierarchical Libraries

One of the great strengths of Ada is the library package where the distinct specification and body decouple the user interface to a package (the specification) from its implementation (the body). This enables the details of the implementation and the clients to be recompiled separately without interference provided the specification remains stable.

However, although this works well for smallish programs it has proved cumbersome when programs become large or complex. There are two aspects of the problem: the coarse control of visibility of private types and the inability to extend without recompilation.

There are occasions when we wish to write two logically distinct packages which nevertheless share a private type. We could not do this in Ada 83. We either had to make the type not private so that both packages could see it with the unfortunate consequence that all the client packages could also see the type; this broke the abstraction. Or, on the other hand, if we wished to keep the abstraction, then we had to merge the two packages together and this resulted in a large monolithic package with increased recompilation costs. (We discount as naughty the use of tricks such as `Unchecked_Conversion` to get at the details of private types.)

The other aspect of the difficulty arose when we wished to extend an existing system by adding more facilities to it. If we add to a package specification then naturally we have to recompile it but moreover we also have to recompile all existing clients even if the additions have no impact upon them.

In Ada 95 these and other similar problems are solved by the introduction of a hierarchical library structure containing child packages and child subprograms. There are two kinds of children: public children and private children. We will just consider public children for the moment; private children are discussed in the next section.

Consider first the familiar example of a package for the manipulation of complex numbers. It might contain the private type itself plus the four standard operations and also subprograms to construct and decompose a complex number taking a cartesian view. Thus we might have

```
package Complex_Numbers is

   type Complex is private;
   function "+" (Left, Right: Complex) return Complex;
   ... -- similarly "-", "*" and "/"

   function Cartesian_To_Complex(Real, Imag: Float) return Complex;
   function Real_Part(X: Complex) return Float;
   function Imag_Part(X: Complex) return Float;

private
...
end Complex_Numbers;
```

We have deliberately not shown the completion of the private type since it is immaterial how it is implemented. Although this package gives the user a cartesian view of the type, nevertheless it certainly does not have to be implemented that way.

Some time later we might need to additionally provide a polar view by the provision of subprograms which construct and decompose a complex number from and to its polar coordinates. In Ada 83 we could only do this by adding to the existing package and this forced us to recompile all the existing clients.

In Ada 95, however, we can add a child package as follows

```
package Complex_Numbers.Polar is

   function Polar_To_Complex(R, Theta: Float) return Complex;
   function "abs" (Right: Complex) return Float;
   function Arg(X: Complex) return Float;

end Complex_Numbers.Polar;
```

and within the body of this package we can access the private type `Complex` itself.

Note the notation, a package having the name `P.Q` is a child package of its parent package `P`. We can think of the child package as being declared inside the declarative region of its parent but after the end of the specification of its parent; most of the visibility rules stem from this model. In other words the declarative region defined by the parent (which is primarily both the specification and body of the parent) also includes the space occupied by the text of the children; but it is important to realize that the children are inside that region and do not just extend it. Observe that the child does not need a with clause for the parent and that the entities of the parent are directly visible without a use clause.

In just the same way, library packages in Ada 95 can be thought of as being declared in the declarative region of the package `Standard` and after the end of its specification. Note that a child subprogram is not a primitive operation of a type declared in its parent's specification because the child is not declared in the specification but after it.

The important special visibility rule is that the private part (if any) and the body of the child have visibility of the private part of their parent. (They naturally also have visibility of the visible part.) However, the visible part of a (public) child package does not have visibility of the private part of its parent; if it did it would allow renaming and hence the export of the hidden private details to any client; this would break the abstraction of the private type (this rule does not apply to private children as explained later).

The body of the child package for our complex number example could simply be

```
package body Complex_Numbers.Polar is

   -- bodies of Polar_To_Complex etc

end Complex_Numbers.Polar;
```

In order to access the procedures of the child package the client must have a with clause for the child package. However this also implicitly provides a with clause for the parent as well thereby saving us the burden of having to write one separately. Thus we might have

```
with Complex_Numbers.Polar;
package Client is
...
```

and then within `Client` we can access the various subprograms in the usual way by writing `Complex_Numbers.Real_Part` or `Complex_Numbers.Polar.Arg` and so on.

Direct visibility can be obtained by use clauses as expected. However, a use clause for the child does not imply one for the parent; but, because of the model that the child is in the declarative region of the parent, a use clause for the parent makes the child name itself directly visible. So writing

```
with Complex_Numbers.Polar; use Complex_Numbers;
```

now allows us to refer to the subprograms as `Real_Part` and `Polar.Arg` respectively.

We could of course have added

```
use Complex_Numbers.Polar;
```

and we would then be able to refer to the subprogram `Polar.Arg` as just `Arg`.

Child packages thus neatly solve both the problem of sharing a private type over several compilation units and the problem of extending a package without recompiling the clients. They thus provide another form of programming by extension.

A package may of course have several children. In fact with hindsight it might have been more logical to have developed our complex number package as three packages: a parent containing the private type and the four arithmetic operations and then two child packages, one giving the cartesian view and the other giving the polar view of the type. At a later date we could add yet another package providing perhaps the trigonometric functions on complex numbers and again this can be done without recompiling what has already been written and thus without the risk of introducing errors.

The extension mechanism provided by child packages fits neatly together with that provided by tagged types. Thus a child package might itself have a private part and then within that part we might derive and extend a private type from the parent package. This is illustrated by the following example which relates to the processing of widgets in a window system.

```
package XTK is
   type Widget is tagged private;
```

```
      type Widget_Access is access Widget'Class;
      ...
   private
      type Widget is tagged
         record
            Parent: Widget_Access;
            ...
         end record;
   end XTK;

   -- now extend the Widget

   package XTK.Color is
      type Colored_Widget is new Widget with private;
      ...
   private
      type Colored_Widget is new Widget with
         record
            Color: ...
         end record;
   end XTK.Color;
```

An interesting point with this construction is that clients at the parent level (those just withing `XTK`) only see the external properties common to all widgets, although by class-wide programming using `Widget_Access`, they may be manipulating colored widgets. However, a client of `XTK.Color` also has access to the external properties of colored widgets because we have made the extended type visible (although still private of course). It should be noted that in fact the private part of `XTK.Color` does not actually access the private part of `XTK` although it has visibility of it. But of course the body of `XTK.Color` undoubtedly will and that is why we need a child package.

Another example is provided by the alert system discussed in II.1. It would probably be better if the additional package concerning emergency alerts was actually a child of the main package thus

```
   package New_Alert_System.Emergency is
      type Emergency_Alert is new Alert with private;
      ...
   end New_Alert_System.Emergency;
```

The advantages are manifold. The commonality of naming makes it clear that the child is indeed just a part of the total system; this is emphasized by not needing a with clause for the parent and that the entities in the parent are immediately visible. In addition, although not required in this example, any private mechanisms in the private part of the parent would be visible to the child. The alternative structure in II.3 where the baseline used an abstract type could also be rearranged.

The benefit of just the commonality of naming is very important since it prevents the inadvertent interference between different parts of subsystems. This is used to good advantage in the arrangement of the Ada 95 predefined library as will be seen in II.13.

Finally, it is very important to realize that the child mechanism is hierarchical. Children may have children to any level so we can build a complete tree providing decomposition of facilities in a natural manner. A child may have a private part and this is then visible from its children but not its parent.

With regard to siblings a child can obviously only have visibility of a previously compiled sibling anyway. And then the normal rules apply: a child can only see the visible part of its siblings.

A parent body may access (via with clauses) and thus depend upon its children and grandchildren. A child body automatically depends upon its parent (and grandparent) and needs no with clause for them. A child body can depend upon its siblings (again via with clauses).

II.8 Private Child Units

In the previous section we introduced the concept of hierarchical child packages and showed how these allowed extension and continued privacy of private types without recompilation. However, the whole idea was based on the provision of additional facilities for the client. The specifications of the additional packages were all visible to the client.

In the development of large subsystems it often happens that we would like to decompose the system for implementation reasons but without giving any additional visibility to clients.

Ada 83 had a problem in this area which we have not yet addressed. In Ada 83 the only means at our disposal for the decomposition of a body was the subunit. However, although a subunit could be recompiled without affecting other subunits at the same level, any change to the top level body (which of course includes the stubs of the subunits) required all subunits to be recompiled.

Ada 95 also solves this problem by the provision of a form of child unit that is totally private to its parent. In order to illustrate this idea consider the following outline of an operating system.

```
package OS is
   -- parent package defines types used throughout the system
   type File_Descriptor is private;
   ...
private
   type File_Descriptor is new Integer;
end OS;

package OS.Exceptions is
   -- exceptions used throughout the system
   File_Descriptor_Error,
   File_Name_Error,
   Permission_Error: exception;
end OS.Exceptions;

with OS.Exceptions;
package OS.File_Manager is
   type File_Mode is (Read_Only, Write_Only, Read_Write);
   function Open(File_Name: String; Mode: File_Mode)
      return File_Descriptor;

   procedure Close(File: in File_Descriptor);
   ...
end OS.File_Manager;

procedure OS.Interpret(Command: String);

private package OS.Internals is
   ...
end OS.Internals;

private package OS.Internals_Debug is
   ...
end OS.Internals_Debug;
```

In this example the parent package contains the types used throughout the system. There are then three public child units, the package `OS.Exceptions` containing various exceptions, the package `OS.File_Manager` which provides file open/close routines (note the explicit with clause for its sibling `OS.Exceptions`) and a procedure `OS.Interpret` which interprets a command line passed as a parameter. (Incidentally this illustrates that a child unit can be a subprogram as well as a package. It can actually be any library unit and that includes a generic declaration and a generic instantiation.) Finally we have two private child packages called `OS.Internals` and `OS.Internals_Debug`.

A private child (distinguished by starting with the word **`private`**) can be declared at any point in the child hierarchy. The visibility rules for private children are similar to those for public children but there are two extra rules.

The first extra rule is that a private child is only visible within the subtree of the hierarchy whose root is its parent. And moreover within that tree it is not visible to the specifications of any public siblings (although it is visible to their bodies).

In our example, since the private child is a direct child of the package `OS`, the package `OS.Internals` is visible to the bodies of `OS` itself, of `OS.File_Manager` and of `OS.Interpret` (`OS.Exceptions` has no body anyway) and it is also visible to both body and specification of `OS.Internals_Debug`. But it is not visible outside `OS` and a client package certainly cannot access `OS.Internals` at all.

The other extra rule is that the visible part of the private child can access the private part of its parent. This is quite safe because it cannot export information about a private type to a client because it is not itself visible. Nor can it export information indirectly via its public siblings because, as mentioned above, it is not visible to the visible parts of their specifications but only to their private parts and bodies.

We can now safely implement our system in the package `OS.Internals` and we can create a subtree for the convenience of development and extensibility. We would then have a third level in the hierarchy containing packages such as `OS.Internals.Devices`, `OS.Internals.Access_Rights` and so on.

It might be helpful just to summarize the various visibility rules which are actually quite simple and mostly follow from the model of the child being located after the end of the specification of its parent but inside the parent's declarative region. (We use "to with" for brevity.)

- A specification never needs to with its parent; it may with a sibling except that a public child specification may not with a private sibling; it may not with its own child (it has not been compiled yet!).

- A body never needs to with its parent; it may with a sibling (private or not); it may with its own child (and grandchild...).

- The entities of the parent are accessible by simple name within a child; a use clause is not required.

- The context clause of the parent also applies to a child.

- A private child is never visible outside the tree rooted at its parent. And within that tree it is not visible to the specifications of public siblings.

- The private part and body of any child can access the private part of its parent (and grandparent...).

- In addition the visible part of a private child can also access the private part of its parent (and grandparent...).

- A with clause for a child automatically implies with clauses for all its ancestors.

- A use clause for a library unit makes the child units accessible by simple name (this only applies to child units for which there is also a with clause).

These rules may seem a bit complex but actually stem from just a few considerations of consistency. Questions regarding access to children of sibling units and other remote relatives follow by analogy with an external client viewing the appropriate subtree.

We conclude our discussion of hierarchical libraries by considering their interaction with generics. Genericity is also an important tool in the construction of subsystems and it is essential that it be usable with the child concept.

Any parent unit may have generic children but a generic parent can only have generic children. If the parent unit is not generic then a generic child may be instantiated in the usual way at any point where it is visible.

On the other hand, if the parent unit is itself generic, then a generic child can be instantiated outside the parent hierarchy provided the parent is first instantiated and the child is mentioned in a with clause; the instantiation of the child then refers to the instantiation of the parent. Note that although the original generic hierarchy consists of library units, the instantiations need not be library units.

As a simple example, we might wish to make the package `Complex_Numbers` of the previous section generic with respect to the underlying floating point type. We would write

```
generic
   type Float_Type is digits <>;
package Complex_Numbers is
...
end Complex_Numbers;

generic
package Complex_Numbers.Polar is
...
end Complex_Numbers.Polar;
```

and then the instantiations might be

```
with Complex_Numbers;
package Real_Complex_Numbers is new Complex_Numbers(Real);

with Complex_Numbers.Polar;
package Real_Complex_Numbers.Real_Polar is
                                new Real_Complex_Numbers.Polar;
```

We thus have to instantiate the generic hierarchy (or as much of it as we want) unit by unit. This avoids a number of problems that would arise with a more liberal approach but enables complete subsystems to be built in a generic manner. In the above example we chose to make the instantiated packages into a corresponding hierarchy but as mentioned they could equally have been instantiated as local packages with unrelated names. But an important point is that the instantiation of the child refers to the instantiation of the parent and not to the generic parent. This ensures that the instantiation of the child has visibility of the correct instantiation of the parent.

The reader will now appreciate that the hierarchical library system of Ada 95 provides a very powerful and convenient tool for the development of large systems from component subsystems. This is of particular value in developing bindings to systems such as POSIX in a very elegantly organized manner.

II.9 Protected Types

The rendezvous model of Ada 83 provided an advanced high level approach to task synchronization which avoided the methodological difficulties encountered by the use of low-level primitives such as semaphores and signals. As is well-known, such low-level primitives suffer from similar problems as gotos; it is obvious what they do and they are trivial to implement but in practice easy to misuse and can lead to programs which are difficult to maintain.

Unfortunately the rendezvous has not proved entirely satisfactory. It required additional tasks to manage shared data and this often led to poor performance. Moreover, in some situations, awkward race conditions arose essentially because of abstraction inversion. And from a methodological viewpoint the rendezvous is clearly control oriented and thus out-of-line with a modern object oriented approach.

In Ada 95 we introduce the concept of a protected type which encapsulates and provides synchronized access to the private data of objects of the type without the introduction of an additional task. Protected types are very similar in spirit to the shared objects of the Orca language developed by Bal, Kaashoek and Tanenbaum of Amsterdam [Bal 92].

A protected type has a distinct specification and body in a similar style to a package or task. The specification provides the access protocol and the body provides the implementation details. We can also have a single protected object by analogy with a single task.

As a simple example consider the following

```
protected Variable is
   function Read return Item;
   procedure Write(New_Value: Item);
private
   Data: Item;
end Variable;

protected body Variable is

   function Read return Item is
   begin
      return Data;
   end Read;

   procedure Write(New_Value: Item) is
   begin
      Data := New_Value;
   end Write;

end Variable;
```

The protected object `Variable` provides controlled access to the private variable `Data` of some type `Item`. The function `Read` enables us to read the current value whereas the procedure `Write` enables us to update the value. Calls use the familiar dotted notation.

```
X := Variable.Read;
...
Variable.Write(New_Value => Y);
```

Within a protected body we can have a number of subprograms and the implementation is such that (like a monitor) calls of the subprograms are mutually exclusive and thus cannot interfere with each other. A procedure in the protected body can access the private data in an arbitrary manner whereas a function is only allowed read access to the private data. The implementation is consequently permitted to perform the useful optimization of allowing multiple calls of functions at the same time.

By analogy with entries in tasks, a protected type may also have entries. The action of an entry call is provided by an entry body which has a barrier condition which must be true before the entry body can be executed. There is a strong parallel between an accept statement with a guard in a task body and an entry body with a barrier in a protected body, although, as we shall see in a moment, the timing of the evaluation of barriers is quite different to that of guards.

A good illustration of the use of barriers is given by a protected type implementing the classic bounded buffer. Consider

```
protected type Bounded_Buffer is
   entry Put(X: in Item);
   entry Get(X: out Item);
private
   A: Item_Array(1 .. Max);
   I, J: Integer range 1 .. Max := 1;
   Count: Integer range 0 .. Max := 0;
end Bounded_Buffer;

protected body Bounded_Buffer is

   entry Put(X: in Item) when Count < Max is
   begin
      A(I) := X;
      I := I mod Max + 1; Count := Count + 1;
   end Put;

   entry Get(X: out Item) when Count > 0 is
   begin
      X := A(J);
      J := J mod Max + 1; Count := Count - 1;
   end Get;

end Bounded_Buffer;
```

This provides a cyclic bounded buffer holding up to `Max` values of the type `Item` with access through the entries `Put` and `Get`. We can declare an object of the protected type and access it as expected

```
My_Buffer: Bounded_Buffer;
...
My_Buffer.Put(X);
```

The behavior of the protected type is controlled by the barriers. When an entry is called its barrier is evaluated; if the barrier is false then the call is queued in much the same way that calls on entries in tasks are queued. When `My_Buffer` is declared, the buffer is empty and so the barrier for `Put` is true whereas the barrier for `Get` is false. So initially only a call of `Put` can be executed and a task issuing a call of `Get` will be queued.

At the end of the execution of an entry body (or a procedure body) of the protected object all barriers which have queued tasks are reevaluated thus possibly permitting the processing of an entry call which had been queued on a false barrier. So at the end of the first call of `Put`, if a call of `Get` had been queued, then the barrier is reevaluated thus permitting a waiting call of `Get` to be serviced at once.

It is important to realize that there is no task associated with the buffer itself; the evaluation of barriers is effectively performed by the runtime system. Barriers are evaluated when an entry is first called and when something happens which could sensibly change the state of a barrier with a waiting task.

Thus barriers are only reevaluated at the end of an entry or procedure body and not at the end of a protected function call because a function call cannot change the state of the protected object and so is not expected to change the values of barriers. These rules ensure that a protected object can be implemented efficiently.

Note that a barrier *could* refer to a global variable; such a variable might get changed other than through a call of a protected procedure or entry — it could be changed by another task or even by a call of a protected function; such changes will thus not be acted upon promptly. The programmer needs to be aware of this and should not use global variables in barriers without due consideration.

It must be understood that the barrier protection mechanism is superimposed upon the natural mutual exclusion of the protected construct thus giving two distinct levels of protection. At the end of a protected call, already queued entries (whose barriers are now true) take precedence over other calls contending for the protected object. On the other hand, a new entry call cannot even evaluate its barrier if the protected object is busy with another call until that call (and any processible queued calls) have finished.

This has the following important consequence: if the state of a protected resource changes and there is a task waiting for the new state, then this task will gain access to the resource and be guaranteed that the state of the resource when it gets it is the same as when the decision to release the task was made. Unsatisfactory polling and race conditions are completely avoided.

Protected objects are very similar to monitors in general concept; they are passive constructions with synchronization provided by the language runtime system. However, they have a great advantage over monitors in that the protocols are described by barrier conditions (which are fairly easy to prove correct) rather than the low-level and unstructured signals internal to monitors as found in Modula.

In other words protected objects have the essential advantages of the high level guards of the rendezvous model but without the overhead of an active task.

Protected types enable very efficient implementations of various semaphore and similar paradigms. For example a counting semaphore might be implemented as follows

```
protected type Counting_Semaphore(Start_Count: Integer := 1) is
   entry Secure;
   procedure Release;
   function Count return Integer;
private
   Current_Count: Integer := Start_Count;
end Counting_Semaphore;

protected body Counting_Semaphore is

   entry Secure when Current_Count > 0 is
   begin
      Current_Count := Current_Count - 1;
   end Secure;

   procedure Release is
   begin
      Current_Count := Current_Count + 1;
   end Release;

   function Count return Integer is
   begin
      return Current_Count;
   end Count;

end Counting_Semaphore;
```

This implements the general form of Dijkstra's semaphore. It illustrates the use of all three forms of protected operations: a function, a procedure and an entry. The entry `Secure` and the procedure `Release` correspond to the P and V operations (from the Dutch Passeren and Vrijmaken) and the function `Count` gives the current value of the semaphore. This example also illustrates that a protected type can have a discriminant which is here used to provide the initial value of the semaphore or in other words the number of items of the resource being guarded by the semaphore.

It is important to note that a task type may also have a discriminant in Ada 95 and this can similarly be used to initialize a task. This can for example be used to tell a task who it is (perhaps from among an array of tasks) without introducing a special entry just for that purpose.

Our final example introduces the ability to requeue a call on another entry. It sometimes happens that a service needs to be provided in two parts and that the calling task has to be suspended after the first part until conditions are such that the second part can be done. Two entry calls are then necessary but attempts to program this in Ada 83 usually run into difficulties; race conditions can arise in the interval between the calls and there is often unnecessary visibility of the internal protocol.

The example is of a broadcast signal. Tasks wait for some event and then when it occurs all the waiting tasks are released and the event reset. The difficulty is to prevent tasks that call the wait operation after the event has occurred, but before the signal can be reset, from getting through. In other words, we must reset the signal in preference to letting new tasks through. The requeue statement allows us to program such preference control. An implementation is

```
protected Event is
   entry Wait;
   entry Signal;
private
   entry Reset;
   Occurred: Boolean := False;
end Event;

protected body Event is

   entry Wait when Occurred is
   begin
      null;                  -- note null body
   end Wait;

   entry Signal when True is  -- barrier is always true
   begin
      if Wait'Count > 0 then
         Occurred := True;
         requeue Reset;
      end if;
   end Signal;

   entry Reset when Wait'Count = 0 is
   begin
      Occurred := False;
   end Reset;

end Event;
```

Tasks indicate that they wish to wait for the event by the call

```
Event.Wait;
```

and the happening of the event is notified by some task calling

```
Event.Signal;
```

whereupon all the waiting tasks are allowed to proceed and the event is reset so that future calls of `Wait` work properly.

The `Boolean` variable `Occurred` is normally false and is only true while tasks are being released. The entry `Wait` has no body but just exists so that calling tasks can suspend themselves on its queue while waiting for `Occurred` to become true.

The entry `Signal` is interesting. It has a permanently true barrier and so is always processed. If there are no tasks on the queue of `Wait` (that is no tasks are waiting), then there is nothing to do and so it exits. On the other hand if there are tasks waiting then it must release them in such a way that no further tasks can get on the queue but then regain control so that it can reset the flag. It does this by requeuing itself on the entry `Reset` after setting `Occurred` to true to indicate that the event has occurred.

The semantics of requeue are such that this completes the action of `Signal`. However, remember that at the end of the body of a protected entry or procedure the barriers are reevaluated for those entries which have tasks queued. In this case there are indeed tasks on the queue for `Wait` and there is also a task on the queue for `Reset` (the task that called `Signal` in the first place); the barrier for `Wait` is now true but of course the barrier for `Reset` is false since there are still tasks on the queue for `Wait`. A waiting task is thus allowed to execute the body of `Wait` (being null this does nothing) and the task thus proceeds and then the barrier evaluation repeats. The same process continues until all the waiting tasks have gone when finally the barrier of `Reset` also becomes true. The original task which called signal now executes the body of `Reset` thus resetting `Occurred` to false so that the system is once more in its initial state. The protected object as a whole is now finally left since there are no waiting tasks on any of the barriers.

Note carefully that if any tasks had tried to call `Wait` or `Signal` while the whole process was in progress then they would not have been able to do so because the protected object as a whole was busy. This illustrates the two levels of protection and is the underlying reason why a race condition does not arise.

Another consequence of the two levels is that it still all works properly even in the face of such difficulties as timed and conditional calls and aborts. The reader may recall, for example, that by contrast, the `Count` attribute for entries in tasks cannot be relied upon in the face of timed entry calls.

A minor point to note is that the entry `Reset` is declared in the private part of the protected type and thus cannot be called from outside. Ada 95 also allows a task to have a private part containing private entries.

The above example has been used for illustration only. The astute reader will have observed that the condition is not strictly needed inside `Signal`; without it the caller will simply always requeue and then immediately be processed if there are no waiting tasks. But the condition clarifies the description. Indeed, the very astute reader might care to note that we can actually program this example in Ada 95 without using requeue at all. A more realistic classic example is the disk scheduler where a caller is requeued if the head is currently over the wrong track.

In this section we have outlined the main features of protected types. There are a number of detailed aspects that we have not covered. The general intent, however, should be clear. Protected types provide a data-oriented approach to synchronization which couples the high-level conditions (the barriers) with the efficiency of monitors. Furthermore the requeue statement provides a means of programming preference control and thus enables race conditions to be avoided.

It must be remembered, of course, that the existing task model remains; the rendezvous will continue to be a necessary approach in many circumstances of a general nature (such as for directly passing messages). But the protected object provides a better paradigm for most data-oriented situations.

II.10 Task Scheduling and Timing

A criticism of Ada 83 has been that its scheduling rules are unsatisfactory especially with regard to the rendezvous. First-in-first-out queuing on entries and the arbitrary selection from several open alternatives in a select statement lead to conflict with the normal preemptive priority rules. For example, priority inversion occurs when a high priority task is on an entry queue behind a lower priority task.

Furthermore, mode changes may require the ability to dynamically change priorities and this conflicts with the simple static model of Ada 83. In addition, advances in the design of scheduling techniques based on Rate Monotonic Scheduling prescribe a variety of techniques to be used in different circumstances according to the arrival pattern of events; see [Sha 90a] and [Klein 93].

Ada 95 allows much more freedom in the choice of priority and scheduling rules. However, because this is a specialized area (and may not be appropriate on some host architectures), the details are contained in the Real-Time Systems annex to which the reader is referred for more details.

Timing is another important aspect of scheduling and the delay statement of Ada 83 has not proved adequate in all circumstances.

For example, an attempt to wait until a specific time by a sequence such as

```
Next_Time: Time;
...
Next_Time := time_to_be_woken_up;
delay Next_Time - Clock;
```

which is intended to stop the task until the time given by the variable `Next_Time`, is not foolproof. The problem is that there is a race condition. Between calling the function `Clock` and issuing the delay statement, it is possible for the task to be preempted by a higher priority task. The result is that when the delay is finally issued, the `Duration` value will be inappropriate and the task will be delayed for too long.

This difficulty is overcome in Ada 95 by the introduction of a complementary **delay until** statement which takes a `Time` (rather than a `Duration`) as its argument. We can then simply write

```
delay until Next_Time;
```

and all will be well.

The final new tasking facility to be introduced in this section is the ability to perform an asynchronous transfer of control. This enables an activity to be abandoned if some condition arises (such as running out of time) and an alternative sequence of statements to be executed instead. This gives the capability of performing mode changes.

This could of course be programmed in Ada 83 by the introduction of an agent task and the use of the abort statement but this was a heavy solution not at all appropriate for most applications needing a mode change.

Asynchronous transfer of control is achieved by a new form of select statement which comprises two parts: an abortable part and a triggering alternative. As a simple example consider

```
select
   delay 5.0;                -- triggering alternative
   Put_Line("Calculation did not complete");
then abort
   Invert_Giant_Matrix(M);   -- abortable part
end select;
```

The general idea is that if the statements between **then abort** and **end select** do not complete before the expiry of the delay then they are abandoned and the statements following the delay executed instead. Thus if we cannot invert our large matrix in five seconds we give up and print a message.

The statement that triggers the abandonment can alternatively be an entry call instead of a delay statement. If the call returns before the computation is complete then again the computation is abandoned and any statements following the entry call are executed instead. On the other hand if the computation completes before the entry call, then the entry call is itself abandoned.

The entry call can, of course, be to a task or to a protected object as described in the previous section. Indeed, Ada 95 allows an entry call to be to a protected object or to a task in all contexts.

Other refinements to the Ada tasking model include a better description of the behavior of the abort statement and a more useful approach to shared variables by the introduction of a number of pragmas.

II.11 Generic Parameters

The generic facility in Ada 83 has proved very useful for developing reusable software particularly with regard to its type parameterization capability. However, there were a few anomalies which have been rectified in Ada 95. In addition a number of further parameter models have been added to match the object oriented facilities.

In Ada 83 the so-called contract model was broken because of the lack of distinction between constrained and unconstrained formal parameters. Thus if we had

```
generic
   type T is private;
package P is ...

package body P is
   X: T;
...
```

then in Ada 83 we could instantiate this with a type such as `Integer` which was fine. However we could also supply an unconstrained type such as `String` and this failed because when we came to declare the object `T` we found that there were no constraints and we could not declare an object as an unconstrained array. The problem was that the error was not detected through a mismatch in the instantiation mechanism but as an error in the body itself. But the whole essence of the contract model is that if the actual parameter satisfies the requirements of the formal then any body which matches the formal specification will work. The poor user might not have had access to the source of the body but nevertheless found errors reported in it despite the instantiation apparently working.

This serious violation of the contract model is repaired in Ada 95. The parameter matching rules for the example above no longer accept an unconstrained type such as `String` but require a type such as `Integer` or a constrained type or a record type with default discriminants (these are collectively known as definite types in Ada 95).

If we wish to write a generic package that will indeed accept an unconstrained type then we have to use a new form of notation as follows

```
generic
   type T(<>) is private;
package P ...
```

In this case we are not allowed to declare an (uninitialized) object of type `T` in the body; we can only use `T` in ways which do not require a constrained type. The actual parameter can now be any unconstrained type such as `String`; it could, of course, also be a constrained type.

Other new parameter models are useful for combining genericity with type extension and for writing class-wide generic packages. The formal declaration

```
type T is tagged private;
```

requires that the actual type be tagged.

We can also write

```
type T is new S;
```

or

```
type T is new S with private;
```

In both cases the actual type must be `S` or derived directly or indirectly from `S`. If we add **`with private`** then both `S` and the actual type must be tagged. (Remember the rule that all tagged types have **`tagged`** or **`with`** in their declaration.)

In all these cases we can also follow the formal type name with `(<>)` to indicate that the actual may be unconstrained (strictly, indefinite to use the terminology introduced above). Furthermore if we follow **`is`** by **`abstract`** then the actual type can also be abstract (but it need not be).

The last new kind of formal generic parameter is the formal generic package. This greatly simplifies the composition of generic packages. It allows one package to be used as a parameter to another so that a hierarchy of facilities can be created.

Examples are inevitably a bit long but consider first the following two packages in Ada 83. The first defines a private type for complex numbers and the basic operations upon them. The second builds on the first and provides various vector operations on complex numbers. The whole system is generic with respect to the underlying floating point type used for the complex numbers.

```
generic
   type Float_Type is digits <>;
package Generic_Complex_Numbers is

   type Complex is private;

   function "+" (Left, Right: Complex) return Complex;
   function "-" (Left, Right: Complex) return Complex;

   -- etc

end Generic_Complex_Numbers;

generic
   type Float_Type is digits <>;
   type Complex is private;

   with function "+" (Left, Right: Complex) return Complex is <>;
   with function "-" (Left, Right: Complex) return Complex is <>;

   -- and so on

package Generic_Complex_Vectors is

   -- types and operations on vectors

end Generic_Complex_Vectors;
```

and we can then instantiate these two packages by for example

```
package Long_Complex is
   new Generic_Complex_Numbers(Long_Float);

use Long_Complex;
```

```
package Long_Complex_Vectors is
   new Generic_Complex_Vectors(Long_Float, Complex);
```

In this Ada 83 formulation we had to pass the type `Complex` and all its operations exported from `Complex_Numbers` back into the vector package as distinct formal parameters so that we could use them in that package. The burden was somewhat reduced by using the default mechanism for the operations but this incurred the slight risk that the user might have redefined one of them with incorrect properties (it also forced us to write a use clause or lots of renamings).

This burden is completely alleviated in Ada 95 by the ability to declare generic formal packages. In the generic formal part we can write

```
with package P is new Q(<>);
```

and then the actual parameter corresponding to `P` must be any package which has been obtained by instantiating `Q` which must itself be a generic package.

Returning to our example, in Ada 95, having written `Generic_Complex_Numbers` as before, we can now write

```
with Generic_Complex_Numbers;
generic
   with package Complex_Numbers is
      new Generic_Complex_Numbers (<>);
package Generic_Complex_Vectors is

   -- as before

end Generic_Complex_Vectors;
```

where the actual package must be any instantiation of `Generic_Complex_Numbers`. Hence our previous instantiations can now be simplified and we can write

```
package Long_Complex is
   new Generic_Complex_Numbers(Long_Float);

package Long_Complex_Vectors is
   new Generic_Complex_Vectors(Long_Complex);
```

The key point is that we no longer have to import (explicitly or implicitly) the type and operators exported by the instantiation of `Generic_Complex_Numbers`. Hence the parameter list of `Generic_Complex_Vectors` is reduced to merely the one parameter which is the package `Long_Complex` obtained by the instantiation of `Generic_Complex_Numbers`. We no longer even have to pass the underlying type `Long_Float`.

Although this example has been couched in terms of a numerical application, the general approach is applicable to many examples of building a hierarchy of generic packages.

II.12 Other Improvements

We have now covered most of the major improvements which give Ada 95 so much extra power over Ada 83. But the discussion has not been complete; we have omitted important facilities such as the introduction of controlled types giving initialization, finalization and user defined assignment and the use of access discriminants to give the functionality of multiple inheritance.

There are also a number of minor changes which remove various irritations and which together make Ada 95 a major improvement within existing paradigms. We will now briefly mention the more notable of these improvements.

The attribute `T'Base` can now be used as a type mark. So if `Float_Type` is a generic formal parameter we can then declare

```
Local: Float_Type'Base;
```

and any constraints imposed by the actual parameter will not then apply to the working variable `Local`. This is important for certain numeric algorithms where we wish to be unconstrained in intermediate computations.

The underlying model for the numeric types is slightly changed by the introduction of fictitious types *root_integer* and *root_real*. This brings a number of simplifications and improvements regarding implicit type conversions and one is the removal of the notorious irritation that

```
for I in -1 .. 100 loop
```

was not allowed in Ada 83. It is allowed in Ada 95.

The rule distinguishing basic declarative items from later declarative items has been removed (this essentially said that little declarations cannot follow big declarations and was intended to prevent little ones getting lost visually but it backfired). As a consequence declarations can now be in any order. This often helps with the placing of representation clauses.

Another irritation in Ada 83 was the problem of use clauses and operators. There was a dilemma between, on the one hand, disallowing a use clause and then having to use prefix notation for operators or introduce a lot of renaming or, on the other hand, allowing a use clause so that infixed operators could be used but then allowing visibility of everything and running the risk that package name prefixes might be omitted with a consequent serious loss of readability. Many organizations have imposed a complete ban on use clauses and burdened themselves with lots of renaming. This is solved in Ada 95 by the introduction of a use type clause. If we have a package `Complex_Numbers` which declares a type `Complex` and various operators `"+"`, `"-"` and so on, we can write

```
with Complex_Numbers; use type Complex_Numbers.Complex;
```

and then within our package we can use the operators belonging to the type `Complex` in infix notation. Other identifiers in `Complex_Numbers` will still have to use the full dotted notation so we can see from which package they come. Predefined operators such as `"="` are also made directly visible by an appropriate use type clause.

Concerning `"="` the rules regarding its redefinition are now completely relaxed. It may be redefined for any type at all and need not necessarily return a result of type `Boolean`. The only remaining rule in this area is that if the redefined `"="` does return a result of type `Boolean` then a corresponding `"/="` is also implicitly declared. On the other hand, `"/="` may itself be redefined only if its result is not type `Boolean`.

The rules regarding static expressions are improved and allow further sensible expressions to be treated as static. A static expression may now contain membership tests, attributes, conversions and so on. Moreover, an expression which looks static but occurs in a context not demanding a static expression will be evaluated statically; this was surprisingly not the case in Ada 83 — an expression such as `2 + 3` was only required to be evaluated at compile time if it occurred in a context demanding a static expression. Note also that rounding of odd halves is now defined as away from zero so `Integer(1.5)` is now `2`.

A small change which will be welcomed is that a subprogram body may now be provided by renaming. This avoids tediously writing code which merely calls another subprogram. Renaming is now also allowed for generic units and a library unit may now be renamed as a library unit; these facilities will be found to be particularly useful with child units.

Another change which will bring a sigh of relief is that **out** parameters can now be read. They are treated just like a variable that happens not to be explicitly initialized; this change will save the introduction of many local variables and much frustration. A related change is that the restriction that it was not possible to declare a subprogram with **out** parameters of a limited type is also lifted.

Some restrictions regarding arrays are also relaxed. It is now possible to deduce the bounds of a variable (as well as a constant) from an appropriate initial value, such as in

```
S: String := Get_Message;  -- a function call
```

which avoids having to write the tedious

```
Message: constant String := Get_Message;
S: String(Message'Range) := Message;
```

It is also possible to use a named aggregate with an "others" component as an initial value or in an assignment. Sliding is now permitted for subprogram parameters and function results in return statements which are treated like assignment with regard to array bound matching.

There are also improvements in the treatment of discriminants. A private type can now have a discriminated type with defaults as its full type thus

```
package P is
   type T is private;
private
   type T(N: Natural := 1) is
   ...
end P;
```

Infuriatingly this was not allowed in Ada 83 although the corresponding problem with matching generic parameters was eliminated many years ago.

An important improvement in exception handlers is the ability to access information regarding the occurrence of an exception. This is done by declaring a "choice parameter" in the handler and we can then use that to get hold of, for example, the exception name for debugging purposes. We can write

```
when Event: others =>
   Put_Line("Unexpected exception: " & Exception_Name(Event));
```

where the function `Exception_Name` returns the name of the exception as a string (such as `"Constraint_Error"`). Other functions provide further useful diagnostic information regarding the cause of the exception.

An important improvement which will be a great relief to systems programmers is that the language now includes support for unsigned integer types (modular types). This provides shift and logical operations as well as modular arithmetic operations and thus enables unsigned integer values to be manipulated as sequences of bits.

Another improvement worth mentioning in this brief summary concerns library package bodies. In Ada 83 a package body was optional if it was not required by the language (for providing subprogram bodies for example). However, this rule, which was meant to be a helpful convenience, seriously misfired sometimes when a library package was recompiled and bodies which just did initialization could get inadvertently lost without any warning. In Ada 95, a library package is only allowed to have a body if it is required by language rules; the pragma `Elaborate_Body` is one way of indicating that a body is required.

Finally, in order to meet the needs of the international community, the type `Character` has been changed to the full 8-bit ISO set (Latin-1) and the type `Wide_Character` representing the 16-bit ISO Basic Multilingual Plane has been added. The type `Wide_String` is also defined by analogy.

II.13 The Predefined Library

There are many additional predefined packages in the standard library which has been restructured in order to take advantage of the facilities offered by the hierarchical library. As mentioned above, root library packages behave as children of `Standard`. There are just three such predefined child packages of `Standard`, namely `System`, `Interfaces` and `Ada` and these in turn have a number of child packages. Those of `System` are concerned with intrinsic language capability such as the control of storage. Those of `Interfaces` concern the interfaces to other languages. The remaining more general predefined packages are children of the package `Ada`.

An important reason for the new structure is that it avoids potential name conflicts with packages written by the user; thus only the names `Ada` and `Interfaces` could conflict with existing Ada 83 code. Without this structure the risk of conflict would have been high especially given the many new predefined packages in Ada 95.

The existing packages such as `Calendar`, `Unchecked_Conversion` and `Text_IO` are now child packages of `Ada`. Compatibility with Ada 83 is achieved by the use of library unit renaming (itself a new feature in Ada 95) thus

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

We will now briefly summarize the more notable packages in the predefined library in order to give the reader an appreciation of the breadth of standard facilities provided.

The package `Ada` itself is simply

```
package Ada is
   pragma Pure(Ada);  -- as white as driven snow!
end Ada;
```

where the pragma indicates that `Ada` has no variable state; (this concept is important for sharing in distributed systems).

Input-output is provided by the existing packages `Ada.Text_IO`, `Ada.Sequential_IO`, `Ada.Direct_IO` and `Ada.IO_Exceptions` plus a number of new packages. The package `Ada.Wide_Text_IO` is identical to `Text_IO` except that it handles the types `Wide_Character` and `Wide_String`. General stream input-output is provided by `Ada.Streams` and `Ada.Streams.Stream_IO`; these enable heterogeneous files of arbitrary types to be manipulated (remember that `Direct_IO` and `Sequential_IO` manipulate files whose items are all of the same type). The package `Ada.Text_IO.Text_Streams` gives access to the stream associated with `Text_IO`; this allows mixed binary and text input-output and the use of the standard files with streams.

There are also nongeneric versions of the packages `Text_IO.Integer_IO` and `Text_IO.Float_IO` for the predefined types such as `Integer` and `Float`. Their names are `Ada.Integer_Text_IO` and `Ada.Float_Text_IO` and so on for other predefined types; there are also corresponding wide versions. These nongeneric packages will be found useful for training and overcome the need to teach genericity on day one of every Ada course.

The package `Ada.Characters.Handling` provides classification and conversion functions for characters. Examples are `Is_Letter` which returns an appropriate Boolean value and `To_Wide_Character` which converts a character to the corresponding wide character. The package `Ada.Characters.Latin_1` contains named constants in a similar style to `Standard.ASCII` which is now obsolescent.

General string handling is provided by the package `Ada.Strings`. Three different forms of string are handled by packages `Strings.Fixed`, `Strings.Bounded` and `Strings.Unbounded`. In addition, packages such as `Strings.Wide_Fixed` perform similar operations on wide strings.

Extensive mathematical facilities are provided by the package `Ada.Numerics`. This parent package is just

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

and includes the generic child package `Ada.Numerics.Generic_Elementary_Functions` which is similar to the corresponding standard ISO/IEC 11430:1994 for Ada 83 [ISO 94a]. There are also nongeneric versions such as `Ada.Numerics.Elementary_Functions` for the predefined types `Float` and so on. Facilities for manipulating complex types and complex elementary functions are provided by other child packages defined in the Numerics annex.

The package `Ada.Numerics.Float_Random` enables the user to produce streams of pseudo-random floating point numbers with ease. There is also a generic package `Ada.Numerics.Discrete_Random` which provides for streams of discrete values (both integer and enumeration types).

The package `Ada.Exceptions` defines facilities for manipulating exception occurrences such as the function `Exception_Name` mentioned above.

The package `System` has child packages `System.Storage_Elements` and `System.Storage_Pools` which are concerned with storage allocation.

The package `Interfaces` has child packages `Interfaces.C`, `Interfaces.COBOL` and `Interfaces.Fortran` which provide facilities for interfacing to programs in those languages. It also contains declarations of hardware supported numeric types. Implementations are encouraged to add further child packages for interfacing to other languages.

II.14 The Specialized Needs Annexes

There are six Specialized Needs annexes. In this summary we cannot go into detail but their content covers the following topics:

Systems Programming

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

Real-Time Systems

> As mentioned above this annex addresses various scheduling and priority issues including setting priorities dynamically, scheduling algorithms and entry queue protocols. It also includes detailed requirements on the abort statement for single and multiple processor systems and a monotonic time package (as distinct from `Calendar` which might go backwards because of time-zone or daylight-saving changes).

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 environment task. This annex defines two forms of partitions and inter-partition communication using statically and dynamically bound remote subprogram calls.

Information Systems

The core language extends fixed point types to include basic support for decimal types. This annex defines a number of packages providing detailed facilities for manipulating decimal values and conversion to external format using picture strings.

Numerics

This annex addresses the special needs of the numeric community. One significant change is the basis for model numbers. These are no longer described in the core language but in this annex. Moreover, model numbers in 95 are essentially what were called safe numbers in Ada 83 and the old model numbers and the term safe numbers have been abandoned. Having both safe and model numbers did not bring benefit commensurate with the complexity and confusion thereby introduced. This annex also includes packages for manipulating complex numbers.

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.

II.15 Conclusion

The discussion in this chapter has been designed to give the reader a general feel for the scope of Ada 95 and some of the detail. Although we have not addressed all the many improvements that Ada 95 provides, nevertheless, it will be clear that Ada 95 is an outstanding language.

III Overview of the Ada Language

The previous chapter introduced the new highlights of Ada 95. In contrast, this chapter gives an overview of the whole Ada language showing the overall framework within which the new features fit and thus also illustrates how Ada 95 is a natural evolutionary enhancement of Ada 83.

Ada is a modern algorithmic language with the usual control structures, and with the ability to define types and subprograms. It also serves the need for modularity, whereby data, types and subprograms can be packaged. It treats modularity in the physical sense as well, with a facility to support separate compilation.

In addition to these aspects, the language supports real-time programming, with facilities to define the invocation, synchronization, and timing of parallel tasks. It also supports systems programming, with facilities that allow access to system-dependent properties, and precise control over the representation of data.

The fact that the limited and predominantly upward compatible enhancements incorported in Ada 95 allow it to support state-of-the-art programming in the nineties and beyond, reconfirms the validity of Ada's underlying principles, and is a proof of the excellent foundation provided by the original design.

III.1 Objects, Types, Classes and Operations

This describes two fundamental concepts of Ada: *types*, which determine a set of values with associated operations, and *objects*, which are instances of those types. Objects hold *values*. *Variables* are objects whose values can be changed; *constants* are objects whose values cannot be changed.

III.1.1 Objects and Their Types

Every object has an associated *type*. The type determines a set of possible values that the object can contain, and the operations that can be applied to it. Users write *declarations* to define new types and objects.

Ada is a block-structured language in which the scope of declarations, including object and type declarations, is *static*. Static scoping means that the visibility of names does not depend on the input data when the program is run, but only on the textual structure of the program. Static properties such as visibility can be changed only by modifying and recompiling the source code.

Objects are created when the executing program enters the scope where they are declared (*elaboration*); they are deleted when the execution leaves that scope (*finalization*). In addition, *allocators* are executable operations that create objects dynamically. An allocator produces an access value (a value of an *access type*), which provides access to the dynamically created object. An access value is said to *designate* an object. Access objects are only allowed to designate objects of the type specified by the access type. Access types correspond to pointer types or references in other programming languages.

A *type*, together with a (possibly null) *constraint*, forms a *subtype*. User-defined subtypes constrain the values of the subtype to a subset of the values of the type. Subtype constraints are useful for run-time error detection, because they show the programmer's intent. Subtypes also allow an optimizing compiler to make more efficient use of hardware resources, because they give the compiler more information about the behavior of the program.

User-defined types provide a finer classification of objects than the predefined types, and hence greater assurance that operations are applied to only those objects for which the operations are meaningful.

III.1.2 Types, Classes and Views

Types in Ada can be categorized in a number of different ways. There are elementary types, which cannot be decomposed further, and composite types which, as the term implies, are composed of a number of components. The most important form of composite type is the record which comprises a number of named components themselves of arbitrary and possibly different types.

Records in Ada 95 are generalized to be extensible and form the basis for object-oriented programming. Such extensible record types are known as tagged types; values of such types include a tag denoting the type which is used at runtime to distinguish between different types. Record types not marked as tagged may not be extended and correspond to the record types of Ada 83.

New types may be formed by derivation from any existing type which is then known as the parent type. A derived type inherits the components and primitive operations of the parent type. In the case of deriving from a tagged record type, new components can be added thereby extending the type. In all cases new operations can be added and existing operations replaced.

The set of types derived directly or indirectly from a specific type, together with that type form a *derivation class*. The types in a class share certain properties (they all have the components of the common ancestor or root type for example) and this may be exploited in a number of ways. Treating the types in a class interchangeably by taking advantage of such common properties is termed polymorphism.

There are two means of using polymorphism in Ada. *Static polymorphism* is provided through the generic parameter mechanism whereby a generic unit may at compile time be instantiated with any type from a class of types. *Dynamic polymorphism* is provided through the use of so-called class-wide types and the distinction is then made at runtime on the basis of the value of a tag.

A class-wide type is declared implicitly whenever a tagged record type is defined. The set of values of the class-wide type is the union of the sets of values of all the types of the class. Values of class-wide types are distinguished at runtime by the value of the tag giving class-wide programming or dynamic polymorphism. The class-wide type associated with a tagged record type `T` is denoted by the attribute `T'Class`. Objects and operations may be defined for such class-wide types in the usual way.

As well as derivation classes, Ada also groups types into a number of predefined classes with common operations. This aids the description of the language and, moreover, the common properties of certain of these predefined classes may be exploited through the generic mechanism.

A broad hierarchical classification of Ada types is illustrated in Figure III-1. The following summary of the various types gives their key properties.

- A type is either an elementary type or a composite type. Elementary types cannot be decomposed further whereas the composite types have an inner structure. The elementary types can be further categorized into the scalar types and access types. The composite types comprise familiar array and record types plus the protected and task types which are concerned with multitasking.

- Scalar types are themselves subdivided into the discrete types and the real types. The discrete types have certain important common properties; for example, they may be used to index array types. The discrete types are the enumeration types and the integer types.

The integer types in turn comprise signed integer types and modular (unsigned) types. The real types comprise the other forms of numeric types.

- An enumeration type defines an ordered set of distinct enumeration literals, for example a list of states or an alphabet of characters. The enumeration types `Boolean`, `Character` (the 8-bit ISO standard character set) and `Wide_Character` (the 16-bit ISO standard character set) are predefined.

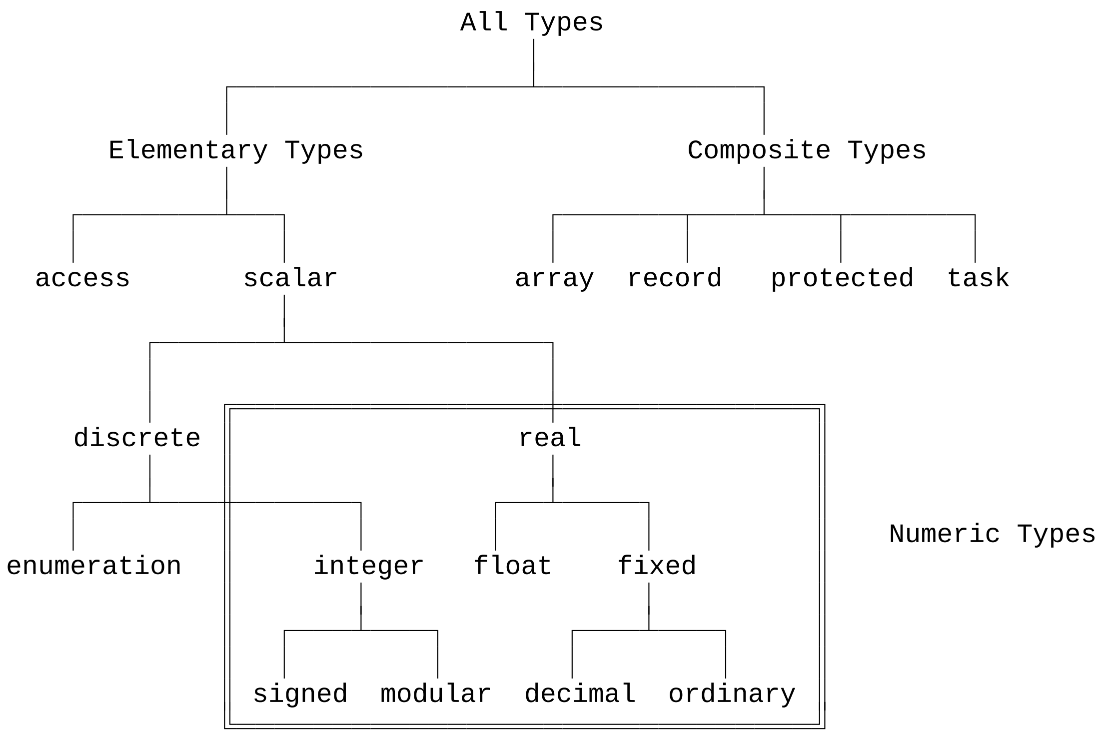

Figure III-1: Ada Type Hierarchy

- The *numeric* types do not exactly fit the hierarchy as presented, but there are certain properties that are common to all numeric types (such as the availability of arithmetic operations), so we have indicated this by a double box surrounding the numeric types. Numeric types provide a means of performing approximate or exact numerical computations. Approximate computations may be performed using either fixed point types with absolute error bounds, or floating point types with relative error bounds. Exact computations may be performed with either integer types, which denote sets of consecutive integers, or with decimal fixed point types. The numeric types `Integer`, `Float` and `Duration` are predefined.

- Access types, the remaining form of elementary type, allow the construction of linked data structures. The value of an access type is, in essence, a pointer to an object of another type, the accessed type. The accessed type may be any type. In particular the accessed type may be a class-wide type thereby allowing the construction of heterogeneous linked data structures. Access types may be used to designate objects created by allocators, declared objects (provided they are marked as aliased) and subprograms.

- Array types allow definitions of composite objects with indexable components all of the same subtype. Array types may be of one or more dimensions. The types of the indexes must be discrete. The array types `String` and `Wide_String` are predefined.

- Record types are composite types with named components not necessarily of the same type. Record types may be tagged or untagged. A tagged record type may be extended upon derivation and gives rise to a class-wide type which forms the basis for dynamic polymorphism. A tagged record type may also be marked abstract in which case no objects of the type may be declared; an abstract type may have abstract subprograms (these have no body). Abstract types and subprograms form a foundation from which concrete types may be derived.

- Protected types are composite types that provide synchronized access to their inner components via a number of protected operations. Objects of protected types are passive and do not have a distinct thread of control; the mutual exclusion is provided automatically.

- Task types are composite types which are used to define active units of processing. Each object of a task type has its own thread of control.

Record, protected and task types may be parameterized by special components called *discriminants*. A discriminant may be either of a discrete type or of an access type. A discriminant of a discrete type may be used to control the structure or size of an object. More generally, a discriminant of an access type may be used to parameterize a type with a reference to an object of another type. A discriminant may also be used in the initialization of an object of a protected or task type.

Ada provides a special syntax for defining new types within the various categories as illustrated by the following examples.

```
type Display_Color is   -- an enumeration type
   (Red, Orange, Yellow, Green, Blue, Violet);

type Color_Mask is   -- an array type
   array (Display_Color) of Boolean;

type Money is   -- a decimal fixed type
   delta 0.01 digits 18;

type Payment is   -- a record type
   record
      Amount: Money;
      Due_Date: Date;
      Paid: Boolean;
   end record;

task type Device is   -- a task type
   entry Reset;
end Device;

type Dev is access Device;   -- an access type

protected type Semaphore is   -- a protected type
   procedure Release;
   entry Seize;
private
   Mutex: Boolean := False;
end Semaphore;
```

The following example illustrates a tagged type and type extension. A tagged type declaration takes the form

```
type Animal is tagged
   record
      Species: Species_Name;
      Weight: Grams;
   end record;
```

and we may then declare

```
function Image(A: Animal) return String;
   -- Returns a human-readable identification of an Animal
```

The type `Animal` could then be extended as follows

```
type Mammal is new Animal with
   record
      Hair_Color: Color_Enum;
   end record;
```

and a corresponding

```
function Image(M: Mammal) return String;
   -- Returns a human-readable identification of a Mammal
```

The type `Mammal` has all the components of `Animal` and adds an additional component to describe the color of the mammal's hair. The process of extension and refinement could continue with other types such as `Reptile` and `Primate` leading to a tree-structured hierarchy of classes as depicted in Figure III-2.

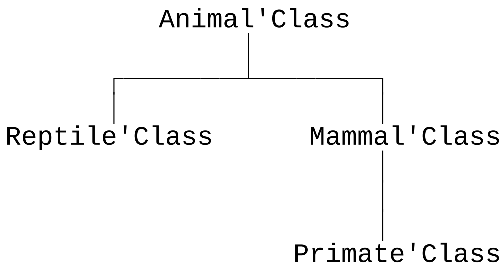

Figure III-2: A Derivation Class Hierarchy

It is important to observe that we have shown the hierarchy in terms of the class-wide types such as `Mammal'Class` rather than the specific types such as `Mammal`. This is to emphasize the fact that the type `Mammal` is not a subtype of `Animal`; `Mammal` and `Animal` are distinct types and values of one type cannot be directly assigned to objects of the other type. However, a value can be converted to an ancestor type (any type if untagged) of the same derivation class by using a type conversion; a value of a descendant tagged type can be formed by using an extension aggregate.

There are a number of other important concepts concerning types regarding the visibility of their components and operations which may be described as providing different *views* of a type.

For example, a *private* type is a type defined in a package whereby the full details of the implementation of the type are only visible inside the body and private part of the package. Private types are of fundamental importance in providing data abstraction.

Furthermore, a *limited* type is a type for which certain operations such as predefined assignment are not available. A type may be inherently *limited* (such as a task type or a protected type or a record type marked explicitly as limited) or it may be that just a certain view is limited. Thus a private type may be declared as limited and the full type as seen by the body need not be limited.

III.1.3 Operations and Overloading

There is a set of operations associated with each type. An operation is associated with a type if it takes parameters of the type, or returns a result of the type. Some operations are implicitly provided when a type is defined; others are explicitly declared by the user.

A set of operations is implicitly provided for each type declaration. Which operations are implicitly provided depends on the type, and may include any of the following

- *Syntactic constructs,* such as assignment, component selection, literals and attributes. These use special syntax specific to the operation.

  For example, an attribute takes the form `X'Attr`, where `X` is the name of an entity, and `Attr` is the name of an attribute of that entity. Thus, `Integer'First` yields the lower bound of the type `Integer`.

- *Predefined operators*. These are taken from this set of 21 operator symbols

  - Logical operators: **`and or xor`**
  - Relational operators: `= /= < <= > >=`
  - Binary adding operators: `+ - &`
  - Unary adding operators: `+ -`
  - Multiplying operators: `* /` **`mod rem`**
  - Highest precedence operators: `**` **`abs not`**

- *Enumeration literals*.

- *Derived operations*, which are inherited by a derived type from its parent type (as explained below).

The *explicitly declared* operations of the type are the *subprograms* that are explicitly declared to take a parameter of the type, or to return a result of the type. There are two kinds of subprograms: *procedures* and *functions*. A procedure defines a sequence of actions; a procedure call is a statement that invokes those actions. A function is like a procedure, but also returns a result; a function call is an expression that produces the result when evaluated. Subprogram calls may be indirect, through an access value.

Procedures may take parameters of mode **`in`**, which allows reading of the parameters, mode **`out`**, which allows writing of the parameters, and mode **`in out`**, which allows both reading and writing of the parameters. In Ada 95, a parameter of mode **`out`** behaves like a variable and both reading and writing are allowed; however, it is not initialized by the actual parameter. All function parameters are of mode **`in`**. An **`in`** parameter may be an *access parameter*. Within the subprogram, an access parameter may be dereferenced to allow reading and writing of the designated object.

The *primitive operations* of a type are the implicitly provided operations of the type, and, for a type immediately declared within a package specification, those subprograms of the type that are also explicitly declared immediately within the same package specification. A derived type *inherits* the primitive operations of its parent type, which may then be overridden.

The name of an explicitly declared subprogram is a *designator*, that is, an identifier or an operator symbol (see the list of operator symbols above). Operators are a syntactic convenience; the operator notation is always equivalent to a function call. For example, `X + 1` is equivalent to `"+"(X, 1)`. When defining new types, users can supply their own implementations for any or all of these operators, and use the new operators in expressions in the same way that predefined operators are used.

The benefits of being able to use the operator symbol `"+"` to refer to the addition function for every numeric type are apparent. The alternative of having to create a unique name for each type's addition function would be cumbersome at best.

The operators are just one example of *overloading*, where a designator (e.g., `"+"`) is used as the name for more than one entity. Another example of overloading occurs whenever a new type is derived from an existing one. The new type inherits operations from the parent type, including the designators by which the operations are named; these designators are overloaded on the old and new types. Finally, the user may introduce overloading simply by defining several subprograms that have the same name. Overloaded subprograms must be distinguishable by their parameter and result types.

The user can also provide new meanings for existing operators. For example, a new meaning of `"="` can be provided for all types (only for limited types in Ada 83).

Ada compilers must determine a unique meaning for every designator in a program. The process of making this determination is called *overload resolution*. The compiler uses the context in which the designator is used, including the parameter and result types of subprograms, to perform the overload resolution. If a designator cannot be resolved to a single meaning, then the program is illegal; such ambiguities can be avoided by specifying the types of subexpressions explicitly.

Many attributes are defined by the language. Attributes denote various properties, often defined for various classes of Ada's type hierarchy. In some cases, attributes are *user-specifiable* via an *attribute definition* clause, allowing users to specify a property of an entity that would otherwise be chosen by default. For example, the ability to read and write values of a type from an external medium is provided by the operations `T'Read` and `T'Write`. By writing

```
type Matrix is ...
for Matrix'Read use My_Matrix_Reader;
for Matrix'Write use My_Matrix_Writer;
```

the predefined operations are overridden by the user's own subprograms.

III.1.4 Class Wide Types and Dispatching

Associated with each tagged type `T` is a class-wide type `T'Class`. Class-wide types have no operations of their own. However, users may define explicit operations on class-wide types. For example

```
procedure Print(A: in Animal'Class);
   -- Print human-readable information about an Animal
```

The procedure `Print` may be applied to any object within the class of animals described above.

A programmer can define several operations having the same name, even though each operation has a different implementation. The ability to give distinct operations the same name can be used to indicate that these operations have similar, or related, semantics. When the intended operation can be determined at compile time, based on its parameter and result types, overloading of subprogram names is used. For example, the predefined package `Text_IO` contains many operations called `Put`, all of which write a value of some type to a file. The implementation of `Put` is different for different types.

Dispatching provides run-time selection of the proper implementation in situations where the type of an argument to an operation cannot be determined until the program is executed, and in fact might be different each time the operation is invoked.

Ada 95 provides dispatching on the primitive operations of tagged types. When a primitive operation of a tagged type is called with an actual parameter of a class-wide type, the appropriate implementation is chosen based on the tag of the actual value. This choice is made at run time and represents the essence of dynamic polymorphism. (Note that, in some cases, the tag can be determined at compile time; this is simply regarded as an optimization.)

Continuing the example from above, we demonstrate both overloading and dispatching

```
procedure Print(S: in String);
   -- Print a string

procedure Print(A: in Animal'Class) is
   -- Print information on an animal
begin
   Print(Image(A));
end Print;
```

The `Print` operation is overloaded. One version is defined for `String` and a second is defined for `Animal'Class`. The call to `Print` within the second version resolves at compile time to the version of `Print` defined on `String` (because `Image` returns a `String`); no dispatching is involved. On the other hand, `Image` (see example in III.1.2) is indeed a dispatching operation: depending on the tag of `A` the version of `Image` associated with `Animal` or `Mammal` etc, will be called and this choice is made at runtime.

III.1.5 Abstraction and Static Evaluation

The emphasis on high performance in Ada applications, and the requirement to support interfacing to special hardware devices, mean that Ada programmers must be able to engineer the low-level mapping of algorithms and data structures onto physical hardware. On the other hand, to build large systems, programmers must operate at a high level of abstraction and compose systems from understandable building blocks. The Ada type system and facilities for separate compilation are ideally suited to reconciling these seemingly conflicting requirements.

Ada's support for static checking and evaluation make it a powerful tool, both for the abstract specification of algorithms, and for low-level systems programming and the coding of hardware-dependent algorithms. By *static*, we mean computations whose results can be determined by analyzing the source code without knowing the values of input data or any other environmental parameters that can change between executions of the program.

Ada requires static type checking. The "scope" (applicability or lifetime) of declarations is determined by the source code. Careful attention is given to when the sizes of objects are determined. Some objects' sizes are static, and other objects' sizes are not known until run time, but are fixed when the objects are created. The size of an object is only allowed to change during program execution if the object's size depends on discriminant values, and the discriminants have a default value. Other variable-size data structures can be created using dynamically allocated objects and access types.

Ada supports users who want to express their algorithms at the abstract level and depend on the compiler to choose efficient implementations, as well as users who need to specify implementation details but also want to declare the associated abstractions to the compiler to facilitate checking during both initial development and maintenance.

III.2 Statements, Expressions and Elaboration

Statements are *executed* at run time to cause an action to occur. *Expressions* are *evaluated* at run time to produce a value of some type. *Names* are also evaluated at run time in the general case; names refer to objects (containing values) or to other entities such as subprograms and types. *Declarations* are *elaborated* at run time to produce a new entity with a given name.

Many expressions and subtype constraints are statically known. Indeed, the Ada compiler is required to evaluate certain expressions and subtypes at compile time. For example, it is common that all information about a declaration is known at compile time; in such cases, the run-time elaboration need not actually execute any machine code. The language defines a mechanism that allows Ada compilers to *preelaborate* certain kinds of units; i.e., the actual actions needed to do the elaboration are done once before the program is ever run instead of many times, each time it is run.

III.2.1 Declarative Parts

Several constructs of the language contain a *declarative part* followed by a *sequence of statements*. For example, a procedure body takes the form

```
procedure P(...) is
   I: Integer := 1;  -- this is the declarative part
   ...
begin
   ...               -- this is the statement sequence
   I := I * 2;
   ...
end P;
```

The execution of the procedure body first elaborates all of the declarations given in the declarative part in the order given. It then executes the sequence of statements in the order given (unless a transfer of control causes execution to go somewhere other than to the next statement in the sequence).

The effect of elaborating the declarations is to cause the declared entities to come into existence, and to perform other declaration-specific actions. For example, the elaboration of a variable declaration may initialize the variable to the value of some expression. Often, such expressions are evaluated at compile time. However, if the declarations contain non-static expressions, then the elaboration will need to evaluate those expressions at run-time.

Controlled types allow programmers a means to define what happens to objects at the beginning and end of their lifetimes. For such types, the programmer may define an *initialization* operation, to be automatically invoked when an object of the type is elaborated, and a *finalization* operation to be automatically invoked when the object becomes inaccessible. (Declared objects become inaccessible when their scope is left. Objects created by allocators become inaccessible when `Unchecked_Deallocation` is called, or when the scope of the access type is left.) Controlled types provide a means to reliably program dynamic data structures, prevent storage leakage, and leave resources in a consistent state.

III.2.2 Assignments and Control Structures

An *assignment statement* causes the value of a variable to be replaced by that of an expression of the same type. Assignment is normally performed by a simple bit copy of the value provided by the expression. However, in the case of nonlimited controlled types, assignment can be redefined by the user.

Case statements and *if statements* allow selection of an enclosed sequence of statements based on the value of an expression. The *loop statement* allows an enclosed sequence of statements to be executed repeatedly, as directed by an *iteration scheme*, or until an *exit statement* is encountered. A *goto statement* transfers control to a place marked with a *label*. Additional control mechanisms associated with multitasking and exception handling are discussed below (see III.4 and III.5).

III.2.3 Expressions

Expressions may appear in many contexts, within both declarations and statements. Expressions are similar to expressions in most programming languages: they may refer to variables, constants and literals, and they may use any of the value-returning operations described in III.1.3. An expression produces a value. Every expression has a type that is known at compile time.

III.3 System Construction

Ada was designed specifically to support the construction of large, complex, software systems. Therefore, it must allow the composition of programs from small, understandable building blocks, while still allowing programmers to engineer the low-level mapping of algorithms and data structures onto physical hardware. Ada provides support for modern software development techniques with the following capabilities

- Packaging, the grouping together of logically related entities into packages.
- Information hiding, where the programmer defines the interface of a program unit for the users of that unit, and separately defines the implementation of the unit.
- Object-oriented programming, where objects can be defined in terms of preexisting objects with possible extension, overload resolution can be used to select operations at compile time, and dispatching can be used to select operations at run time.
- Construction of software systems from large numbers of separately compiled units stored in a library, with full compile-time checking of interfaces between separately compiled units.
- Class-wide programming.
- Construction of mixed language systems, that is programs written in more than one programming language.

The following subsections describe this support in more detail.

III.3.1 Program Units

Ada programs are composed of the following kinds of *program units*

- Subprograms — functions and procedures.
- Packages — groups of logically related entities.
- Generic units — parameterized templates for subprograms and packages.

- Tasks — active entities that may run in parallel with each other.
- Protected objects — passive entities that protect data shared by multiple tasks.

Program units may be nested within each other, in the same way as in other block-structured languages. Furthermore, they may be separately compiled.

As we shall see later, packages at the so-called library level may have child units. Ada has a hierarchical structure both at the external level of compilation and internal to a program unit.

Each program unit may be given in two parts: The *specification* defines the interface between the unit (the "server") and its users ("clients"). The *body* defines the implementation of the unit; users do not depend on the implementation details.

For packages, the specification consists of the *visible part* and the *private part*. The visible part defines the logical interface to the package. The private part defines the physical interface to the package, which is needed to generate efficient code, but has no effect on the logical properties of the entities exported by the package. Thus, the private part may be thought of as part of the implementation of the package, although it is syntactically part of the specification in order to ease the generation of efficient code.

The various parts of a package take the following form

```
 -- this is a package specification:
package Example is
   -- this is the visible part
   -- declarations of exported entities appear here
   type Counter is private;
   procedure Reset(C: in out Counter);
   procedure Increment(C: in out Counter);
   --
private
   -- this is the private part
   -- declarations appearing here are not exported
   type Counter is range 0 .. Max;
end Example;

 -- this is the corresponding package body:
package body Example is

   -- implementations of exported subprograms appear here
   -- entities that are used only in the implementation
   -- are also declared here
   Zero: constant Counter := 0;
     -- declaration of constant only used in the body

   procedure Reset(C: in out Counter) is
   begin
      C := Zero;
   end Reset;

   procedure Increment(C: in out Counter) is
   begin
      C := C + 1;
   end Increment;

end Example;
```

Tasks and protected objects may also have a private part.

III.3.2 Private Types and Information Hiding

Packages support information hiding in the sense that users of the package cannot depend on the implementation details that appear in the package body. *Private types* provide additional information-hiding capability. By declaring a type and its operations in the visible part of a package specification, the user can create a new abstract data type.

When a type is declared in a package specification, its implementation details may be hidden by declaring the type to be *private*. The implementation details are given later, as a *full type declaration* in the private part of the package.

The user of a private type is not allowed to use information about the full type. Users may declare objects of the private type, use the assignment and equality operations, and use any operations declared as subprograms in the visible part of the package. The private type declaration may allow users to refer to discriminants of the type, or it may keep them hidden. A private type may also be declared as *limited*, in which case even assignment and predefined equality operations are not available (although the programmer may define an equality operation and export it from the package where the type is declared).

In the private part, in the body of the package, and in the appropriate parts of child library units (see III.3.6), the type is not private: all operations of the type may be used in these places. For example, if the full type declaration declares an array type, then outside users of the type are not allowed to index array components, because these implementation details are hidden. However, code in the package body *is* allowed the complete set of array operations, because it can see the full type declaration.

III.3.3 Object Oriented Programming

Modern software development practices call for building programs from reusable parts, and for extending existing systems. Ada supports such practices through object oriented programming features. The basic principle of object oriented programming is to be able to define one part of a program as an extension of another, pre-existing, part. The basic building blocks of object-oriented programming were discussed in III.1.

Abstract data types may be defined in Ada using packages and private types. Types may be extended by adding new packages, deriving from existing types, and adding new operations to derived types. For non-tagged types, such extension is "static" in the sense that the compiler determines which operations apply to which objects according to the typing and overloading rules. For tagged types, however, operation selection is determined at run time, using the tag carried by each such object. This allows easy extension of existing types. To add a new tagged type, the programmer derives from an existing tagged parent type, possibly adding new record components. The programmer may override existing operations with new implementations. Then, all calls to the existing operation will automatically call the new operation in the appropriate cases; there is no need to change or even to recompile such pre-existing code. For tagged types, it is possible to write class-wide operations by defining subprograms that take parameters of a class-wide type. Class-wide programming allows the programmer to avoid redundancy in cases where an operation makes sense for all types in a class, and where the implementation of that operation is essentially the same for all types in the class.

III.3.4 Generic Units

Generic program units allow parameterization of program units. The parameters can be types and subprograms as well as objects. A normal (non-generic) program unit is produced by *instantiating* a generic unit; the normal program unit is said to be an *instance* of the generic unit. An instance of a generic package is a package; an instance of a generic subprogram is a subprogram.

The instance is a copy of the generic unit, with *actual parameters* substituted for *generic formal parameters*. Generic units may be implemented by actually generating new code for each instance, or by sharing the code for multiple instances, and passing information about the parameters at run time.

An example of a generic package is a generic linked list package that works for any element type. The data type of the elements would be passed in as a parameter. The algorithms for manipulating the lists are independent of the actual element type. Instances of the generic package would support linked lists with a particular element type. If the element type is a tagged class-wide type, then heterogeneous lists can be created, containing elements of any type in the class. Generic formal derived types permit generic units to be developed for derivation classes. Generic formal packages allow a generic unit to be parameterized by an instance of another generic package.

III.3.5 Separate Compilation

Ada allows the specifications and bodies of program units to be separately compiled. A separately compiled piece of code is called a *compilation unit*. The Ada compiler provides the same level of compile-time checking across compilation units as it does within a single compilation unit. For example, in a procedure call, the actual parameters must match the types declared for the formal parameters. This rule is checked whether the procedure declaration and the procedure call are in the same compilation unit, or different compilation units.

Ada compilers work within the context of a *program environment*, which contains information about the compilation units in a program. This information is used in part to check rules across compilation unit boundaries. There are rules about order of compilation that ensure that the compiler always has enough information to check all the rules. For example, a specification must be within the environment before all units that have visibility to names declared in that specification can be compiled.

III.3.6 Library Units

A program environment contains information concerning a collection of *library units*. Library units may be packages, subprograms, or generic units.

Package library units may have *child library units*. Thus, an entire hierarchy of library units may be created: the root of each tree is called a *root library unit*; the tree contains the root library unit, plus all of its children and their descendants.

A library unit specification and its body are compilation units; that is, they may be compiled separately.

Visibility among library units is achieved using *context clauses*; a compilation unit can see a particular library unit if it names that library unit in a context clause. Both root library units and child units may be named in a context clause. In addition, the child library units of a parent can see the parent, including the parent's private declarations, and the body of a unit always has visibility into its specification.

Child units may be used to reduce recompilation costs. Apart from dependencies created by context clauses, the immediate children of a given unit may be recompiled in any order. Therefore, if an existing library unit is extended by adding a child unit, the existing unit need not be recompiled; adding a child is accomplished without changing the source code of the parent. More importantly, other units that depend on the existing parent unit will not need to be recompiled.

The root library units are considered to be children of package `Standard`: context clauses and compilation ordering rules work the same way. Thus, child units are a straightforward generalization of Ada 83 library units.

As an example consider the following

```
package Root is
   -- specification of a root library unit
   ...
end Root;

------------------------------

package Root.Child is
   -- specification of a child library unit
   ...
end Root.Child;

------------------------------

package body Root.Child is
   -- body of the child library unit
   ...
end Root.Child;

------------------------------

private package Root.Local_Definitions is
   -- a private child package specification
   ...
end Root.Local_Definitions;

------------------------------

package body Root is
    -- body of the root library unit
   ...
end Root;
```

The lines in the above example indicate the separate compilation units; they may be submitted to the compiler separately. Note that the child library units are clearly distinguishable by their expanded names (based on the parent's name). The example also shows a *private child* package — a private child unit is visible only within the hierarchy of units rooted at its parent.

Sometimes, the body of a library unit becomes very large, because it contains one or more nested bodies. In such cases, Ada allows the nested bodies to be separately compiled as *subunits*. The nested body is replaced by a *body stub*. The subunit, which is given separately, must name its parent unit. Visibility within the subunit is as if it had appeared at the place where its body stub occurs. Subunits also support an incremental style of top-down development, because a unit may be compiled with one or more body stubs — allowing the development of those bodies to be deferred.

III.3.7 Program Composition

An executable software system is known in Ada as a *program*. A program is composed of one or more compilation units.

A program may be divided into separate *partitions*, which may represent separate address spaces. Implementations may provide mechanisms for user-defined inter-partition communication. The Distributed Systems annex defines a minimal standard interface for such communication. Partitions are intended to support distributed processing, as explained in the annex. Of course, many programs will not be partitioned; such programs consist of a single partition.

To build a partition, the user identifies a set of library units to be included. The partition consists of those library units, plus other library units depended on by the named units. The Ada implementation automatically constructs this set of units before run time.

Each partition has an *environment task*, which is provided automatically by the Ada implementation. A partition may have a *main subprogram*, which must be a library unit subprogram. The environment task elaborates all of the library units that are part of the partition, and their bodies, in an appropriate order, and then calls the main subprogram, if any. The library units and the main program may create other tasks. Thus, an executing partition may contain a hierarchy of tasks, rooted at the environment task.

III.3.8 Interfacing to Other Languages

Large programs are often composed of parts written in several languages. Ada supports this by allowing inter-language subprogram calls, in both directions, and inter-language variable references, in both directions. The user specifies these interfaces using pragmas.

III.4 Multitasking

Ada tasking provides a structured approach to concurrent processing under the control of an Ada run-time system, which provides services such as scheduling and synchronization. This describes tasks and the methods that are used for synchronizing task execution and for communicating between tasks.

Tasking is intended to support tightly coupled systems in which the communication mechanisms may be implemented in terms of shared memory. Distributed processing, where the processors are loosely coupled, is addressed in the Distributed Systems annex.

III.4.1 Tasks

Tasks are entities whose execution may proceed in parallel. A task has a thread of control. Different tasks proceed independently, except at points where they synchronize.

If there is a sufficient number of processors, then all tasks may execute in parallel. Usually, however, there are more tasks than processors; in this case, the tasks will time-share the existing processors, and the execution of multiple tasks will be interleaved on the same processor.

A task is an object of a task type. There may be more than one object of a given task type. All objects of a given task type have the same entries (interface), and share the same code (body). As a result they all execute the same algorithm. Different task objects of the same type may be parameterized using discriminants. Task types are inherently limited types; assignment and equality operations are forbidden.

Task objects are created in the same ways as other objects: they may be declared by an object declaration, or created dynamically using an allocator. Tasks may be nested within other program units, in the same manner as subprograms and packages.

All tasks created by a given declarative part or allocator are activated in parallel. This means that they can logically start running in parallel with each other. The task that created these tasks waits until they have all finished elaborating their declarative parts; it then continues running in parallel with the tasks it created.

Every task has a *master*, which is the task, subprogram, block statement, or accept statement which contains the declaration of the task object (or an access type designating the task type, in some circumstances). The task is said to *depend* on its master. The task executing the master is called the *parent* task. Before leaving a master, the parent task waits for all dependent tasks. When all of those have been terminated, or are ready to terminate, the parent task proceeds. Tasks may be terminated prematurely with the *abort statement*.

III.4.2 Communication and Synchronization

For multiple tasks to cooperate, there must be mechanisms that allow the tasks to communicate and to synchronize their execution. Synchronization and communication usually go hand-in-hand. Ada tasks synchronize and communicate in the following situations

- Tasks synchronize during activation and termination (as just explained).
- Protected objects provide synchronized access to shared data.
- Rendezvous are used for synchronous communication between a pair of tasks.
- Finally, unprotected access to shared variables is allowed, but requires a disciplined protocol to be followed by the communicating tasks.

This flexibility allows the user to choose the appropriate synchronization and communication mechanisms for the problem at hand. They are depicted in Table III-1.

Ada feature	*Synchronization*	*Communication*
Task Creation	(not needed)	Creator initializes discriminants of new task
Task Activation	Creator waits for tasks being activated	Activation failure might be reported
Task Termination	Master waits for children	(none)
Rendezvous	Entry caller and acceptor wait for each other	Entry parameters are passed between the entry caller and the acceptor
Protected Object	Mutual exclusion during data access; queued waiting for entry barriers	Tasks communicate indirectly by reading and writing the components of protected objects
Unprotected Shared Variables	User-defined, low-level synchronization	Reading and writing of shared variables

Table III-1: Summary of Communication and Synchronization

III.4.3 Protected Objects

Protected types are used to synchronize access to shared data. A protected type may contain components in a private part. Moreover, a protected type may also contain functions, procedures, and entries — the *protected operations* of the protected type. The data being shared is declared either as components of the protected type, or as global variables, possibly designated by the components of the protected type. Protected types are inherently limited.

Calls to the protected operations are synchronized as follows. Protected functions provide shared read-only access to the shared data. Multiple tasks may execute protected functions at the same time. Protected procedures and entries provide exclusive read/write access to the shared data. If any task is executing a protected procedure or entry, then no other tasks are allowed to execute any protected operation at the same time; if they try, they must wait.

Protected objects provide a safe and efficient method of synchronizing shared data access. They are safe, because they perform the necessary synchronization operations automatically, and because all synchronizing operations are collected together syntactically. (This is in contrast to lower-level mechanisms such as semaphores, where the user of the shared data must remember to lock and unlock the semaphore.) They are efficient, because their intended implementation is close to the hardware: such as spin-locks in multiprocessor systems.

III.4.4 Protected Operations and Entries

Protected types may export functions, procedures, and entries as described above. Tasks may export entries. All of these operations are called using similar syntax: `OBJ.OP(...)`, where `OBJ` is the name of the task or protected object, `OP` is the name of the operation, and `(...)` represents any actual parameters. Information is passed back and forth using **`in`**, **`in out`**, **`out`** parameters. It is the responsibility of the programmer to ensure that operations of protected objects execute for a bounded and short period of time.

A client task which calls an entry of a server task or protected object may be blocked and placed on a queue. When a server task *accepts* the entry call from a client task, we say that the two tasks are in *rendezvous*. At the beginning and the end of the rendezvous, data may be exchanged via parameters. When the rendezvous is over, the two tasks each continue execution in parallel.

Entries of protected objects are controlled by barrier expressions. When a task calls the entry, it can execute the operation immediately if the barrier expression is true; otherwise, the caller is placed on a queue until the barrier has become true. Protected functions and procedures do not have barrier expressions and, therefore, calls on them need not be queued.

From within a rendezvous or the entry body of a protected type it is possible to complete the interaction by requeuing on a further entry; this avoids race conditions which might occur with two quite distinct interactions.

Entries may also be declared in the private part of a task or protected object and thus not visible to external clients. Such entries may be called by internal tasks or by requeuing.

III.4.5 Select Statements

Select statements are used to specify that a task is willing to wait for any of a number of alternative events. Select statements take various forms.

Select statements used by a server task may contain

- One or more *accept alternatives*, which indicate that the task is willing to accept one of several entries; that is, the task waits for another task to call one of those entries.

- An optional *terminate alternative*, which allows a server task to specify that it is willing to terminate if there is no more work to do (i.e., all other tasks that depend on the same master are either terminated or are waiting at terminate alternatives).

Select statements used by a client task may contain

- One *entry call alternative*, which indicates that the task is waiting for an entry to be executed. That is, the task waits either for the server task to accept the entry, or for the protected object barrier to become true.

Both of these forms of select statement can also contain, either

- A *delay alternative*, which allows a task to specify an action to be taken if communication is not started within a given period of time, or

- An *else part*, which allows a task to specify an action to be taken if communication is not immediately possible.

Whichever alternative becomes available first is chosen. Each alternative specifies an action to be executed if and when the alternative is chosen.

The final form of select statement provides for an asynchronous transfer of control; it contains

- A *triggering alternative*, which may be a delay alternative or an entry call followed by a sequence of statements, and

- An *abortable part*, which is aborted if the triggering alternative completes before the abortable part itself completes. Control is then asynchronously transferred from somewhere in the abortable part to the statements of the triggering alternative. If the abortable part completes before being aborted, then the triggering alternative is cancelled and execution continues at the next statement after the select statement.

III.4.6 Timing

Ada provides features for measuring real time. A task may read the clock to find out what time it is. A task may delay for a certain period of time, or until a specific time.

As mentioned above, a delay alternative may be used to provide a time-out for communication or as a triggering event for initiating an asynchronous transfer of control.

III.4.7 Scheduling

Ada separates the concepts of synchronization and scheduling. Synchronization operations determine when tasks must block, and when they are ready to run. Scheduling is the method of allocating processors to ready tasks. The default scheduling policy is defined by the language. The Real-Time Systems annex defines another, priority-based, scheduling policy, based on a dispatching model. Finally, implementations are allowed to add their own policies, which can be specified by pragmas.

III.5 Exception Handling

Most programs need to recover gracefully from errors that occur during execution. Exception handling allows programs to handle such error situations without ceasing to operate.

An *exception* is used to name a particular kind of error situation. Some exceptions are predefined by the language; others may be defined by the user. Exceptions are declared in the same way as other entities

```
Buffer_Full_Error: exception;
```

This exception might be used to represent the situation of a program trying to insert data into a buffer which is already full.

When the exceptional situation happens, the exception is *raised*. Language-defined exceptions are raised for errors in using predefined features of the language. These exceptions correspond to run-time errors in other languages. For example, the language-defined exception `Constraint_Error` is raised when a subtype constraint is violated at run time. User-defined exceptions are raised by the *raise* statement. To continue the `Buffer_Full_Error` example above, the implementation of the buffer data type might contain statements such as

```
if Buffer_Index > Max_Buffer_Size then
   raise Buffer_Full_Error;
end if;
```

Subprograms, package bodies, block statements, task bodies, entry bodies, and accept statements may have *exception handlers*. An exception handler specifies an action that should be performed when a particular exception is raised. When an exception is raised, the execution of the current sequence of statements is abandoned, and control is transferred to the exception handler, if there is one. Thus, the action of the exception handler replaces the rest of the execution of the sequence of statements that caused the error condition. If there is no exception handler in a particular scope, then the exception is propagated to the calling scope. If the exception is propagated all the way out to the scope of the environment task, then execution of the program is abandoned; this is similar to the way in which program execution is abandoned in other languages when run-time errors are detected.

The following example shows a block with two exception handlers

```
begin
   ...
exception
   when Buffer_Full_Error =>
      Reset_Buffer;
   when Error: others =>
      Put_Line("Unexpected exception raised:");
      Put_Line(Ada.Exceptions.Exception_Information(Error));
end;
```

The handlers recognize two situations: if `Buffer_Full_Error` is raised, the buffer is reset. If any other exception is raised, information about that exception is printed. For many applications, it is useful to get such information about an exception when it occurs. A handler may have a choice parameter (`Error` in the example above) of type `Exception_Occurrence`. The predefined function `Exception_Information` takes a parameter of this type and returns a `String`, providing information (including the name) about the exception. These and other related facilities are defined in a child of package `Ada`.

III.6 Low Level Programming

Although the majority of program text can be written in a machine-independent manner, most large software systems contain small portions that need to depend on low-level machine characteristics. Ada allows such dependence, while still allowing the high-level aspects of the algorithms and data structures to be described in an abstract manner. Many of an implementation's machine-specific characteristics are accessible through the package `System`. This defines storage-related types, an `Address` type, and relational and arithmetic operations on addresses.

III.6.1 Pragmas

A *pragma* is used to convey information to the compiler; it is similar to a compiler directive supported by other languages. A pragma begins with the reserved word **pragma**, an identifier which is the name of the pragma, and optionally one or more arguments.

Some pragmas are defined by the language. For example, pragma `Inline` indicates to the compiler that the code for a subprogram is to be expanded inline at each call whenever possible. Most pragmas apply to a single object, type, or program unit. *Configuration pragmas* are used to specify partition-wide or program-wide options.

Implementations may provide additional pragmas, as long as they do not syntactically conflict with existing ones or use reserved words. Unrecognized pragmas have no effect on a program, but their presence must be signaled with a warning message.

III.6.2 Specifying Representations

Normally, the programmer lets the Ada compiler choose the most efficient way of representing objects. However, Ada also provides *representation clauses*, which allow the user to specify the representation of an individual object, or of all objects of a type. Other representation clauses apply to program units.

The programmer may need to specify that the representation matches the representation used by some hardware or software external to the Ada program, in order to interface to that external entity. Or, the programmer may wish to specify a more efficient representation of certain objects in cases where the compiler does not have enough information to determine the best (most efficient) representation. In either case, data types and objects are first declared in the normal manner, giving their logical properties. Later in the same declarative part, the programmer gives the representation clauses.

In addition to representation clauses, the language defines certain pragmas that control aspects of representation. Implementations may provide additional representation pragmas.

There are predefined attributes that allow users to query aspects of representation. These are useful when the programmer needs to write code that depends upon the representation, although the user might not need to control the representation.

In the absence of representation clauses or pragmas, the compiler is free to choose any representation.

III.6.3 Unprotected Shared Variables

In Ada, variables may be shared among tasks according to the normal visibility rules: if two tasks can see the name of the same variable, then they may use that variable as shared data. However, it is up to the programmer to properly synchronize access to these shared variables. In most cases, data sharing can be achieved more safely with protected objects; unprotected shared variables are primarily used in low-level systems programming. Ada allows the user to specify certain aspects of memory allocation and code generation that may affect synchronization by specifying variables as *volatile* or *atomic*.

III.6.4 Unchecked Programming

Ada provides features for bypassing certain language restrictions. These features are unchecked; it is the programmer's responsibility to make sure that they do not violate the assumptions of the rest of the program. For example, there are mechanisms for manipulating access types that might leave dangling pointers, and there is a mechanism for converting data from one type to another, bypassing the type-checking rules.

The generic function `Unchecked_Deallocation` frees storage allocated by an allocator. It is unchecked in the sense that it can leave dangling pointers.

The generic function `Unchecked_Conversion` converts data from one type to another, bypassing all type-checking rules. The conversion is done simply by reinterpreting the bit pattern as a value of the target type; no conversion actually happens at run time (except possibly bit padding or truncation).

The attribute `P'Unchecked_Access` returns a typed access value to any aliased object of the appropriate type, bypassing the accessibility checking rules (but not the type rules).

III.7 Standard Library

All implementations provide a standard library of various packages. This includes the predefined package `Standard` which defines the predefined types such as `Integer`, and the package `System` which defines various entities relating to the implementation.

The standard library also includes packages `Ada` and `Interfaces`. The package `Ada` includes child packages for computing elementary functions and generating random numbers as well as child packages for string handling and character manipulation. The package `Interfaces` defines facilities for interfacing to other languages.

III.7.1 Input Output

Input-output capabilities are provided in Ada by predefined packages and generic packages.

- *Sequential files* present a logical view of files as sequences of elements. Successive read or write operations to a sequential file result in the transfer of consecutive elements. The generic package `Sequential_IO` may be instantiated for any type to provide operations for creating, opening, closing, deleting, reading and writing sequential files. All elements of a `Sequential_IO` file are of the same type, although if the type is a tagged class-wide type, the elements may have different tags.

- *Direct files* present a logical view of files as indexed sets of elements. The index allows elements to be read or written at any position within a file. The generic package `Direct_IO` provides operations similar to `Sequential_IO`. In addition, `Direct_IO` provides operations for determining the current position and size of a file, and setting the position in the file, in terms of element numbers.

- The `Text_IO` package provides facilities for input and output in a human-readable form.

- A *stream* presents a logical view of a file (or other external medium such as a buffer or network channel) as a sequence of stream elements. Stream input and output are predefined through attributes for all nonlimited types. Users may override these default attributes and the stream operations.

III.8 Application Specific Facilities

Previous sections of this Overview have focused on the Core of the Ada language. Implementations may provide additional features, not by extending the language itself, but by providing specialized packages and implementation-defined pragmas and attributes. In order to encourage uniformity among implementations, without restricting functionality, the *Specialized Needs Annexes* define standards for such additional functionality for specific application areas. Implementations are not required to support all of these features. For example, an implementation specifically targeted to embedded machines might support the application-specific features for Real-Time Systems, but not the application-specific features for Information Systems.

The application areas discussed in the Annexes are

- Systems Programming, including access to machine operations, interrupts, elaboration control, low-level shared variables, and task identification facilities.
- Real-Time Systems, including priorities, queuing and scheduling policies, monotonic time, delay accuracy, immediate abort and a simple tasking model.
- Distributed Systems, including a model for Ada program distribution into partitions and inter-partition communication.
- Information Systems, including detailed support for decimal types and picture formatting.
- Numerics, including a model of real arithmetic plus packages for complex numbers.
- Safety and Security, including pragmas relating to the proof of correctness of programs.

III.9 Summary

The goal of this chapter has been to provide a broad overview of the whole of the Ada language. It also demonstrates that the changes to the language represent a natural extension to the original design of Ada. As a consequence, the incompatibilities between Ada 83 and Ada 95 are minimal. Those of practical significance are described in detail in Appendix X.