



Ada 95 in Context

Michael B. Feldman

The George Washington University

adapted and updated from Chapter 10,
*Handbook of Programming Languages, vol. 1, Object-
Oriented Languages* (Macmillan 1998).

Copyright 1997, 1999 Michael B. Feldman; All Rights
Reserved

Table of Contents

1. Introduction	1
2. Preliminaries	2
2.1 A Brief Historical Sketch of Ada 83	2
2.1.1 The Early History of Ada 83	2
2.1.2 The Name	7
2.2 From Ada 83 to Ada 95	8
2.2.1 Language Features	9
2.2.2 The Annexes	9
2.2.3 The Role of the GNU Ada 95 (GNAT) Compiler	10
2.3 Validation, or “Just Which Language Does This Compiler Compile?”	11
2.3.1 The Validation Process	12
2.3.2 An Example Test Profile	13
2.3.3 What Does It All Mean?	14
3. Ada in Today’s World	15
3.1 Ada in Use	15
3.1.1 Ada in U.S. Defense Applications	15
3.1.2 Non-defense Applications	17
3.2 Education and Ada	18
3.2.1 Ada 95 Textbooks	18
3.2.2 Ada as a Foundation Programming Language	19
3.2.3 The Ada IC “CREASE” Database	20
3.3 Ada 95 Compiler Availability	20
4. Programming	21
4.1 Examples of Basic Structure and Syntax	22
4.1.1 Types and Objects	22
4.1.2 Packages and Files	22
4.1.3 The Usual “Hello World” Program	23
4.1.4 A Less “Verbose” Coding Style	25
4.1.5 The Calendar package	25
4.1.6 Displaying the Current Date	26
4.1.7 Displaying the Date and Time	28
4.1.8 The Brevity/Clarity Tradeoff	31
4.1.9 Loops, Arrays, and Files—the Advantage of Subtypes	33
4.1.10 Type Composition: Arrays and Records	36
4.1.11 Exception Handling	39
4.2 Packages: the Ada Encapsulation Mechanism	45
4.2.1 A Root Package for this Handbook	47
4.2.2 A Package for Rational-Number Arithmetic	48

4.2.3 A Child Package for Rational Input and Output	51
4.2.4 Using the Rationals Package	52
4.2.5 Implementing the Rationals and Rationals.IO Packages	54
4.2.6 Unconstrained array types.....	58
4.2.7 Generic Packages	62
4.2.8 Using the Generic Matrix Package	65
4.2.9 Implementing the Generic Matrix Package	68
4.3 Type Extension, Inheritance, and Polymorphism	71
4.3.1 Classical Polymorphic Types: Variant Records.....	73
4.3.2 Type Extension: Ada 95 Tagged Types	74
4.3.4 Using the Instruments Hierarchy: Polymorphic Dispatch	78
4.3.5 Building a Dashboard.....	80
4.3.6 Comments on Pointers in Ada	82
4.3.7 Implementation of the Instruments Hierarchy	83
4.3.8 Dynamic Data Structures	86
4.3.9 Controlled Types and Finalization	89
4.3.10 Implementing the Linked List Package	91
4.4 Concurrent Programming.....	94
4.4.1 Why Concurrency?	94
4.4.2 A Concurrent Application.....	95
4.4.3 A Tasking-Safe Random Number Generator.....	96
4.4.5 A Tasking-Safe Screen Manager	97
4.4.6 The Main Multitask Program.....	99
4.4.7 Implementing the Random Number Generator.....	101
4.4.8 Implementing the Screen Manager	103
4.4.9 Comments on Concurrent Programming	105
5. Bibliography.....	106
5.1 World-Wide Web Resources on Ada	107
5.2 Published Books on Ada 95	107
5.3 Selected Articles and Other Publications of Interest	109

Ada 95 in Context

1. Introduction

Ada is alive and well, and might well be found in your trains, planes and automobiles, not to mention satellites, steel mills, and—if you live in Switzerland—your bank.

This chapter introduces you to the Ada programming language. The chapter is organized as follows:

- Section 2, Preliminaries: Ada history, the Ada 95 project, and Ada compiler validation
- Section 3, Ada in Today’s World: Defense and non-defense projects, Ada in education, Ada compiler availability
- Section 4, Programming in Ada: a tour of the language via annotated complete, compilable, tested examples.
- Section 5: Bibliography

The current Ada standard is Ada 95; the language of the original Ada standard is now referred to as Ada 83. Throughout this chapter, when we refer to Ada we mean Ada 95, except in the historical section. Where necessary to full understanding, we distinguish between Ada 83 and Ada 95, but we keep this distinction to a minimum.

Frequently we refer to the wealth of materials available on the World Wide Web. With the exception of published books, nearly every conceivable Ada document is available on the Web, as are freely downloadable compilers. These resources are collected at five main web sites, whose Uniform Resource Locators (URLs, or addresses) are given in the Bibliography. The sites are

- Ada Programming Language Resources for Educators and Students, sponsored by the Education Working Group of the ACM Special Interest Group on Ada (SIGAda), maintained by this author and referred to here as the "Educator site"
- Ada Information Clearinghouse (Ada IC), operated by IIT Research Institute under contract to the Ada Resource Association and referred to as the "Ada IC site". (During 1998, responsibility for this contract was "privatized" to ARA from the Ada Joint Program Office (AJPO) in the U.S. Department of Defense.)

- Public Ada Library (PAL), a very large collection of Ada documents, programs, compilers, etc., maintained by Richard Conn on the Washington University (St. Louis) Internet archive server and referred to as “the PAL”
- Home of the Brave Ada Programmers, maintained in Switzerland by Magnus Kempe and referred to as “HBAP”
- ACM Special Interest Group on Ada, referred to as “SIGAda”

In referring to specific files and other resources on the Web, we have decided to refrain from giving detailed URLs, as these have an annoying tendency to change. All the sites are organized for quick search and retrieval, and contain many references to resources on the other sites, so we deem it better just to mention the overall site name and leave it the reader to pay an electronic visit.

2. Preliminaries

This section briefly reviews the history of Ada 83 and Ada 95, and discusses the very important process of *validation* of Ada implementations.

2.1 A Brief Historical Sketch of Ada 83

This chapter is not intended as a historical document, but we hold that one can best assess a technical contribution if one acquires some understanding of the history and context of that contribution. This section therefore provides a historical sketch of Ada.

2.1.1 The Early History of Ada 83

The history of Ada 83 has been written exhaustively—and entertainingly—in a paper in the Second History of Programming Languages Conference (HOPL-II) by William Whitaker [Whitaker 1996]. Whitaker led the original High Order Language Working Group (HOLWG) effort at the U.S. Department of Defense (DoD). This group wrote the requirements for, and oversaw the competition for the design of, the language that became Ada. Whitaker’s article also makes fascinating reading for its insights into the workings of a large organization. For this early history, excerpts from Whitaker’s account serve much more effectively than could our attempt to re-word it.

To begin, we quote from Whitaker’s commentary on the reasons for desiring a common DoD language:

[T]he proposal for language commonality across DoD was extremely radical at the time and initially met almost universal opposition. In fact, it was regarded as unrealistic to expect to use a high order language for embedded systems. It may be surprising that a consensus did not mandate a common high order language for embedded systems much earlier. There are, however, a number of managerial and technical constraints that acted against this. For many DoD systems, severe timing and memory considerations were dominant, governed by real-time interaction with the exterior environment.

Because of these constraints, and restrictions in developmental cost and time scale, many systems opted for assembly language programming. This decision was influenced by past experience with poor quality compilers and the fact that an assembler routinely comes with the machine, while the compiler and its tools usually must be developed after the project has begun. The advantages of high order languages, however, were compelling, and more systems turned to them. Because of limitations of available high order languages, the programs generated often included very large portions done in assembly code and linked to an HOL structure, negating many of the HOL advantages.

Further, many systems found it convenient to produce their own high order language or some incompatible, dialect of an existing one. Since there was no general facility for control of existing languages, each systems office did the configuration control on its language and compilers and continued this for their particular dialect through the entire maintenance phase of the system. This had the effect of reducing the contractual flexibility of the government and restricting competition in maintenance and further development.

This lack of commonality negated many advantages of high order languages including transportability, sharing of tools, the development of very powerful tools of high efficiency and, in fact, not only raised the total cost of existing tools, but in some cases essentially priced them out of the market. Development projects were very poorly supported and forced to live with technology far below what should be the state-of-the-art.

The target for a major language project was to be DoD “software in the large.” This is often given a limited interpretation, namely that DoD programs are individually large, which is certainly true and drives many of the technical requirements on the language. But the fact that the DoD has hundreds of such large programs provides an

opportunity for economies of scale that are potentially much greater than the sum of individual projects. The problem is not just that of producing a subsystem of 200,000 lines of code, but of the servicing of a “system” that is all the code produced by the DoD for (by 1990) \$30 billion per year (“programming in the very large”). This path drives other requirements and properties, like machine independence which forces the validation requirement and the rejection of subsets. But these advantages can be realized only if the technology is applied with consistency over the whole of the DoD, and an even larger community if possible. So a strong position from the DoD was vital to the plan.

Once the working groups was formed, it agreed on these goals for a common DoD language:

- The language should facilitate the reduction of software costs. The costs must be reckoned on the total burden of the life cycle including maintenance, not just the cost of production or program writing.
- Transportability allows the reusing of major portions of software and tools from previous projects, and the flexibility for a system to change hardware while keeping the same software.
- The maintenance of very long lived software in an ever-changing threat situation requires responsiveness and timely flexibility.
- Reliability is an extremely severe requirement in many Defense systems and is often reflected in the high cost of extensive testing and verification procedures.
- The readability of programs produced for such long term systems use is clearly more important than coding speed, or writability.
- The general acceptability of high order languages is often determined by the efficiency and quality of the compiled code. While rapidly falling costs of hardware may make this difficult to substantiate in the abstract, each project manager will compare the efficiency of the object code produced against an absolute standard of the best possible machine language programming. Very little degradation is acceptable.

The abstract of the Whitaker article summarizes the early stages of the actual project very nicely. We have interpolated a few bits of text, in brackets.

The Department of Defense (DoD) High Order Language Commonality program began in 1975 with the goal of establishing a single high order computer programming language appropriate for DoD real-time embedded computer systems. A High Order Language Working Group (HOLWG) was chartered to formulate the DoD requirements for High Order Languages, to evaluate existing languages against those requirements, and to implement the minimal set of languages required for DoD use.

Other parts of the effort included administrative initiatives toward the eventual goal. Specifically, DoD Directive 5000.29 which provided that new defense systems should be programmed in a DoD “approved” and centrally controlled high order language and DoD Instruction 5000.31 which gave the interim defining list of approved languages. [These included Fortran, COBOL, TACPOL, CMS-2, SPL/1, Jovial J3 and Jovial J73.]

The HOLWG language requirements were widely distributed for comment throughout the military and civil communities worldwide. Each successive version of the requirements, from STRAWMAN [1975] through [WOODENMAN, TINMAN, and IRONMAN, to] STEELMAN [1978], produced a more refined definition of the proposed language. During the requirement development process, it was determined that the set of requirements generated was both necessary and sufficient for all major DoD applications (and the analogous large commercial applications). Formal evaluations were performed on dozens of existing languages.

It was concluded that no existing language could be adopted as a single common high order language for the DoD, but that a single language meeting essentially all the requirements was both feasible and desirable. Four contractors were funded [in August 1977] to produce competitive prototypes. A first-phase evaluation [starting in February 1978] reduced the designs to two, which were carried to completion. In turn, [in May 1979] a single language design [submitted by the Frenchman Jean Ichbiah and his team at CII-Honeywell Bull] was subsequently chosen. [The Reference Manual and Rationale for this design were published as a two-part set of *SIGPLAN Notices*, June 1979.]

Follow-on steps included the test and evaluation of the language, control of the language and validation of compilers. The production of compilers and a program development and tool environment were to be accomplished separately by the individual Service Components. The general requirements and expectations for the environment and the control of the language were addressed in another iterative series of documents. A language validation capability (the test code suite) and associated facilities were established to assure compliance to the language definition of compilers using the name “Ada.” The name Ada was initially protected by a DoD-owned trademark [which was relinquished, along with the copyright on the Reference Manual, in 1987].

Several refinements of the language design took place between 1979 and 1983. Here is Whitaker’s summary of those years.

Upon the completion of the development project in late 1980 ... the loose organization of the HOLWG was superseded by a DoD Ada Joint Program Office (AJPO), chartered 12 December 1980. ... This office managed the standardization processes with ANSI and ISO. One consequence of the transition was that this new organization was exclusively involved in the control and support of Ada, not in the overall DoD software problem. While initially proposed and budgeted to generate tools and applications libraries, the AJPO abandoned that role and concerned itself mainly with the control and with DoD policies.

The chief task was to continue to polish the language definition in connection with an ANSI canvass process leading to ANSI, and eventually ISO, standardization. A major challenge was to maintain close international involvement in the development and assure that national and international standards did not differ (a real possibility in the standards world of that time, but much less likely today). Through the usual Ada open process, the definition was refined to MIL-STD 1815A, and this was endorsed by ANSI in February 1983 [U.S. Dept. of Defense 1983], with updated Rationale [Ichbiah *et al.* 1987]. Ada was endorsed as ISO Standard 8652 in 1987. [The ISO standard was a one-page document incorporating the ANSI standard by reference; there were no technical changes at all.]

One final quotation from Whitaker, about the language evaluation process, is revealing:

Other languages were considered for formal evaluation, but were not included because preliminary examination led one to believe that they would not meet the requirements so were not viable candidates for the purposes of the DoD. One such language was C. At that time DARPA was working with Western Electric/Bell Labs on UNIX, contractually supporting some DARPA contractors and other government facilities using UNIX. It was the evaluation policy to have the owners provide assessments of their own languages, in addition to the contracted evaluations, so HOLWG took advantage of this connection between DARPA and Bell Labs to request their cooperation. When Bell Labs were invited to evaluate C against the DoD requirements, they said that there was no chance of C meeting the requirements of readability, safety, etc., for which we were striving, and that it should not even be on the list of evaluated languages. We recognized the truth in their observation and honored their request.

Some have claimed that Ada was “designed by a committee.” This claim is simply not true. The HOLWG represented the intended *users* of the new language. This group oversaw the design competition, but each of the four full language designs was firmly in the hands of a small team in industry. The team that won the competition consisted of Jean Ichbiah and six other members who attest to Ichbiah’s very strong leadership. It is true that the process was an open one which was unprecedented in language development, and that many in the community reviewed the design and advised on its refinement, but Ada was designed by its designers.

2.1.2 The Name

The name “Ada” honors Countess Augusta Ada Lovelace (1815-1852), a mathematician and the only legitimate daughter of poet Lord Byron. While in her twenties, she worked with Charles Babbage on his Difference Engine and thus is considered the world’s first computer programmer. The name was suggested in 1978 by Jack Cooper, a member of the HOLWG, at a meeting of the group in Paris. Whitaker, who was not in Paris at the time, is unable to confirm that the meeting took place in a Paris café. On the other hand, his article does contain verbatim copies of a wonderful series of letters in which DoD requested—and received—permission from Ada’s rightful heir, Lord Lytton, to name the language for her.

Ada’s life has been chronicled in a number of biographies, including Dorothy Stein’s 1985 work [Stein 1985]. Many Ada language enthusiasts are known to be quite emphatic in their reminders that the Ada language honors a real—and very interesting—person, and therefore is *not* to be spelled

ADA, which refers to (depending on context) the American Dental Association and the Americans with Disabilities Act. We note in passing that the military version of the Ada 83 standard carries reference number MIL-STD 1815A. This number was not chosen idly; the “inside joke” is that the real Ada was born in 1815.

2.2 From Ada 83 to Ada 95

Under ANSI practice, a standard is revisited five years after its adoption, and a process is begun to determine whether the standard should be retained as is, modified, or withdrawn. Accordingly, in 1988 the Ada standard was opened up formally for a second look. (Naturally, during the five years many informal comments had been collected.) In the words of the Ada IC flier, *Introducing Ada 95*,

... A Board of Distinguished Reviewers representing six different countries and comprising 28 world-renowned leaders in academia and industry provided oversight and evaluation of the immense input from the international community of users. Over 750 recommendations were received by individuals who were invited to submit Revision Requests ... Conferences, workshops, small-group meetings and one-on-one consultations were held with various segments of the Ada community, and advice was received from some of the world's finest software engineers and government technology leaders.

... The revision is an update of the 1987 International Organization for Standardization (ISO) release and the equivalent 1983 American National Standards Institute (ANSI) Ada standard. Drafts of the revised standard were formally considered by the ISO between September 1993 and October 1994; ballots were cast over a period of 15 months by the 22 member countries, and officially tallied on November 1, 1994.

ISO delegates accepted the revision unanimously and the revised standard reference manual was published 15 February 1995. The ISO approval made Ada 95 the first internationally standardized, fully object oriented programming (OOP) language. Ada 95 also received ANSI approval [in April 1995], following a period of public review and comment

The design of the language revisions, as well as creation of a new reference manual, was completed by Intermetrics, Inc., of Cambridge, MA.

The Intermetrics team leader was S. Tucker Taft, and the Ada 95 language design shows his imprint as vividly as Ada 83 shows that of Jean Ichbiah. As was the case with Ada 83, the Ada 95 process was an especially open one, with input solicited from any and all in the community, but Ada 95, like its predecessor, was designed by its designers, not by a “committee.”

Six or seven years may seem a long time to revise a language standard, but in fact, most other language standardization processes have taken longer, sometimes much longer. Designing a language is complex and highly specialized, and convincing a large number of organizations to approve the design and vote favorably on a national or international standard is time-consuming and requires much skill in the art of human persuasion. This is borne out by the very lengthy process of developing a standard for C++, and also the current (Fall 1997) controversies raging among those who would produce a standard for Java.

2.2.1 Language Features

Ada 95 is a smooth and upward-compatible extension of Ada 83. Revisions to Ada 83 were requested, and designed, in several key areas. We quote again from the Ada IC summary. These features will be explored in some depth in the programming section.

The capabilities of Ada 83 were enhanced for Ada 95 through the definition of a small number of new "building blocks" in three basic areas: object oriented programming, programming in the large, and real-time and parallel programming. In each case, the revision team used existing features as the basis for enhanced capabilities: the derived and universal types were the basis for object-oriented programming features; the existing notion of library units formed the basis for hierarchical name space and program partitioning; and the concepts of private types, functions, procedures, and entries provided the basis for protected record construct, supporting fast mutual exclusion, and asynchronous task communication.

2.2.2 The Annexes

One very important aspect of the Ada 95 design is that the standard is now divided into a core language—which provides all the syntactic structures—and a set of Annexes. The Annexes provide predefined types and packages (modules) as well as specifying certain implementation recommendations and requirements; no new syntax is introduced in the Annexes.

Annex A describes the rich set of standard libraries, which support text, binary, and stream input/output, command-line parameters, math functions, random number generators, character mapping and translation, dynamic strings, and other assorted capabilities. Implementations are required to provide a complete Annex A. Annex B describes Ada's standard interfaces to C, COBOL, and Fortran, including mechanisms for importing foreign-language routines to Ada programs, and exporting Ada routines to other languages. Also specified are types corresponding to those of the other languages, and storage-allocation conventions to match those of the other languages (Fortran-style array storage is a good example).

Annexes C through H are referred to as "special needs" annexes; implementations are not required to provide support for these, but must follow the standard if they do. These annexes are:

- Annex C, Systems Programming (access to machine instructions, interrupts, etc.)
- Annex D, Real-Time Systems (policies for real-time optimization of multiple tasks)
- Annex E, Distributed Systems (programs spread over multiple nodes)
- Annex F, Information Systems (exact arithmetic for decimal types, edit-directed output)
- Annex G, Numerics (complex-number types and operations, floating-point accuracy requirements)
- Annex H, Safety and Security

These annexes will be described further in the programming section.

2.2.3 The Role of the GNU Ada 95 (GNAT) Compiler

A critically important aspect of the growing acceptance of Ada 95 is the availability of a freely available compiler, known as GNAT. This Ada 95 implementation is integrated into the GNU ("GNU's Not UNIX") family of languages and uses the same back-end code generator. It supports the full Ada 95 language, including all annexes. The compiler can be downloaded from the PAL and all its mirror sites, and—as is the case with all software released under the GNU General Public License—full source code is available along with the binaries.

The GNAT project was originally located at New York University and funded by AJPO from 1993 to 1995. Starting in mid-1995, development and commercial support has continued at Ada Core Technologies (ACT), a company founded by principals of the NYU GNAT project.

GNAT has made Ada 95 accessible to a very large number of users around the world. Further, because full source code for the compiler and run-time system can be freely downloaded, Ada is once again becoming an interesting subject of academic and other research.

2.3 Validation, or "Just Which Language Does This Compiler Compile?"

One critically important aspect of the Ada project is *validation* of implementations (compilers and associated linkers and run-time support libraries). DoD desired to have, above all, a *common* language, supported by a strong standard that described the language. In that vision, hardware could be procured competitively, without concern about whether the compilers from hardware vendors A and B compiled the same source language. Similarly, once the hardware platform was determined, compilers for that platform could be procured competitively based on price and performance, because all compilers would support the same language. *Validation demonstrates the conformance of a compiler to the standard.*

Compiler validation is necessarily a *testing* process, as there is no other effective way to demonstrate standard conformance. DoD therefore commissioned the development of a set of test programs, called the Ada Compiler Validation Capability (ACVC). This test suite was made readily available to interested parties, originally on magnetic tape; with the growing use of the Internet, the ACVC was distributed by ftp.

Over the roughly ten years of Ada 83, the ACVC was, of course, updated a number of times, to take account of practical experience with the compilers, add new tests as compiler bugs were discovered, and so on. The final operative Ada 83 test suite was ACVC 1.11, containing something over 2000 distinct test segments. Various forms of the applicable regulations determined the frequency with which a validated compiler was required to be re-tested with a newer test suite.

With the advent of Ada 95, the validation process was changed somewhat to accommodate Ada 95's structure as a core language with specialized-needs annexes; vendors can now choose to submit any or all annexes for separate validation.

Since the reader is probably unfamiliar with the compiler validation process, it is useful for us to go into it in a bit of detail, to put the process in perspective. The following section, describing Ada 95 validation, is adapted directly from the Ada IC flyer, *The Ada Compiler Validation Process*.

2.3.1 The Validation Process

Historically, the validation process was carried out by the Ada certification body. This body consisted of the Ada Joint Program Office (AJPO) and the Ada Validation Facilities (AVFs).

The AJPO issued validation certificates for AVF-tested Ada implementations, registered Ada implementations that were derived by an AVF, maintained the Ada Compiler Validation Capability (ACVC), reviewed all Validation Summary Reports (VSRs), reviewed and adjudicated all disputes regarding the Ada test instrument, and maintained current operating agreements with the AVFs. The AJPO had the responsibility for establishing and maintaining a certification system for the ISO.

In order to obtain a validation certificate, the following six steps were completed by the customer and the Ada certification body:

1. A formal validation agreement between the customer and an AVF was required in order to obtain validation services.
2. Pre-validation, consisting of customer testing, submission of results to the AVF, and resolution of any test issues (e.g., a missing or incomplete result to a test) preceded the actual validation.
3. Validation testing was performed by an AVF at a site mutually agreed upon by the customer and the AVF (usually the customer's site).
4. A Declaration of Conformance was then completed and signed by the customer no later than at the time of validation testing.
5. A Validation Summary Report (VSR) was prepared by the AVF to document the validation.
6. A Validation Certificate (VC) was then issued to the customer by the authority of the AJPO for a successfully tested Ada implementation.

With Ada 95 compilers, compliance is measured only within the limits of the collection of test programs contained in the ACVC for the core language and specialized needs annexes. An Ada implementation passes a given ACVC version if 1) it processes each test of the customized test suite in accordance with criteria for individual tests and 2) the test result profile matches the passing requirements for the specific ACVC version.

Matrices displaying the test result profiles for Core Ada 95 Test Categories and Special Needs Annexes are shown in the Validation Summary Report. The information contained in the matrix is reformatted and provided in the Validated Compilers List (VCL) to allow easy access for buyers and users.

As part of the implementation of the revised DoD Ada policy described in Section 3.1.1, the administrative responsibility for validation was shifted in 1998 from AJPO to the private sector, specifically to the Ada Resource Association (ARA). The term “validation” has been changed to “conformity assessment,” but the basic technical process remains unchanged.

2.3.2 An Example Test Profile

To give the reader some flavor of the tests, we show an example of the test profile that is published in the VCL.

The first matrix displays the number of tests that were Passed, Not Applicable (NA), Not Supported (NS), and Withdrawn for each of seven test categories: Ada 9X Basic, Real-Time, OOP, Type Extensions in Child Units, Child Library Units, Pre-defined Language Environment, and Mixed Features. Here is a list of the test categories and a short description for each:

- *Ada 9X Basic*: This is the subset of tests from ACVC 1.11 after removal of tests not applicable to Ada 95. These tests focus on support expected from Ada for features of Ada 83 that have been updated to be compatible with revised rules.

(Note: The following subsets of tests validate features that are new to Ada 95. Each test has been allocated into exactly one of several test subsets, based upon a general categorization of Ada features used in the test. These tests are designed to reflect the features that programmers are likely to use to solve a programming problem.)

- *Real-Time*: This subset is composed of tests for the new Ada 95 features from Section 9: Tasks and Synchronization. These features include protected objects, modifications to task types, select statements, and delay alternatives.
- *OOP*: This subset of tests focuses on some necessary facilities for achieving object-oriented programming in Ada 95. Features validated include tagged types, class attributes, and abstract types and subprograms. Other Ada 95 facilities commonly used in object-oriented programs are included in subsequent subsets.
- *Type Extensions in Child Units*: Tests that focus on the interaction of the two new Ada features of type extensions of tagged types and child library units. This includes the related semantics of visibility, accessibility, and calls on primitive operations of tagged types.
- *Child Library Units*: Tests that focus on the support for the new Ada capability to provide a hierarchical organization of the compilation units

of an Ada program with the associated capabilities of granting access to the contents of private declarations and of hiding selected units within subsystems.

- *Pre-defined Language Environment*: This subset of tests include some Ada 83 facilities and some new features defined in Annex A. Annex A provides specifications for root library units for Ada, Interface, and System, character and string handling and input/output.
- *Mixed Features*: This relatively large subset of tests focuses on the interaction of Ada features that are a mixture of familiar Ada 83 and new Ada 95 features.

Here, quoted from the VCL with minor reformatting, is the actual ACVC 2.01 test profile run in March 1997, for the GNU Ada 95 (GNAT) compiler under Solaris:

Test Profile for Ada 95 Test Categories

Test Categories	Passed	NA	NS	Withdrawn	Totals
Ada 9X Basic	2817	33	0	11	2861
Real Time	51	0	0	0	51
OOP	54	0	0	0	54
Type Extensions in					
Child Units	32	0	0	2	34
Child Library Units	37	0	0	0	37
Pre-defined Language					
Environment	26	0	0	1	27
Mixed Features	180	0	0	8	188
Totals	3197	33	0	22	3252

Test Profile for Specialized Needs Annexes

Annex	Passed	NA	NS	Withdrawn	Totals
C Syst Prog	4	0	0	3	7
D Real Time Syst	26	4	0	1	31
E Dist Syst	8	0	0	0	8
F Info Systems	3	0	0	3	6
G Numerics	5	0	0	0	5
H Safety/Security	0	0	0	0	0
Totals	46	4	0	7	57

2.3.3 What Does It All Mean?

Ada compiler validation, for all its openness, has been widely misunderstood in the community at large. It is important to understand what validation does *not* demonstrate.

First, validation does not, and cannot, demonstrate that a compiler is free of bugs. All we can say is that a validated compiler has passed a large set of standard, publicly distributed, conformance tests. However, to use a maxim attributed to Edsger Dijkstra, “Testing can show only the *presence* of bugs; it cannot show the *absence* of bugs.” On the other hand, compiler developers would like their products to be as bug-free as possible, so they continue to develop tests. Compiler vendors have their own (highly proprietary) test libraries, in addition to the ACVC; these libraries (we assume) not only implement the vendor’s internal test design, but also incorporate regression suites based on the experiences of that vendor’s actual users. As with all programs of nontrivial size, as compilers mature, they exhibit fewer and fewer bugs over time.

Second, validation does not assess the *performance* (time and space resources) of a compiler. For this purpose, various benchmark suites have been developed, but running these is not a part of the validation process.

We cannot expect validation magically to suspend all the laws of large-program development. Validation cannot guarantee perfection. However, it can, and, in the Ada case, it does, contribute much to our confidence in a compiler’s adherence to the language standard. An Ada compiler compiles *Ada*, accepting and properly translating our valid programs, and rejecting our invalid ones, according to the Reference Manual, to the extent that it is humanly possible to test it. In a software industry in which languages have tended to be poorly-defined “moving targets,” with weak or nonexistent standards and no public validation process, this is real progress.

3. Ada in Today’s World

In this section we discuss the general state of Ada in use in defense and non-defense projects, Ada in education, and the general availability of Ada 95 compilers.

3.1 Ada in Use

As we know, Ada was originally commissioned by the Department of Defense. On the other hand, it was never DoD “property,” and has, from its inception in the early 1980s, been chosen for projects well outside the defense sector. Some of its earliest applications were in purely commercial management systems: a payroll system at a truck manufacturer, a job scheduling system at a printing company. The purpose of this section is to bring the reader up to date on the general state of Ada usage in the world.

3.1.1 Ada in U.S. Defense Applications; the Ada “Mandate” Policy

Defense projects are, by their very nature, not discussed in much detail in the open literature. On the other hand, enough information has come to light to lead to a conclusion that Ada has been reasonably successful in the weapons-building community.

In 1996, DoD commissioned the National Research Council to convene a task force to recommend the future course of the DoD Ada policy. The final report of this study group [National Research Council 1996] contains a chart showing 50 million lines of active Ada code (MLOC) in DoD weapons systems, with 33 MLOC of C, 5 MLOC C++, 20 MLOC Fortran, 19 MLOC CMS-2, 14 MLOC Jovial. Considering the long life cycles of DoD systems, this is a good record for Ada. It is interesting to note that the very first validated Ada 95 compiler, validated in mid-1995, was hosted on a Sun Sparc and targeted to a proprietary board used in the revised Patriot missile system.

Published reports have discussed a number of successful military management information systems written in Ada, including a recent one in Ada 95 with SQL integration. The AdaSAGE system, developed by Idaho National Engineering Laboratory, and consisting of a relational data base manager, interface builder, and much other support for secure PC-based information systems, won a “best product” award at a non-Ada object-technology conference, and is said to have many customers within DoD.

The NRC report recommended that DoD maintain its Ada requirement for “warfighting software” but eliminate the requirement for other applications, and DoD management subsequently decided to eliminate the requirement altogether, opting to embed language choice into a Software Engineering Plan Review process for each project. It will be interesting to observe the degree to which Ada continues to be chosen for U.S. military applications, in the absence of a firm requirement to choose it.

Historically, DoD has created joint program offices to nurture new technologies to maturity. The Ada Joint Program Office was created for this purpose with respect to Ada. With the adoption of the Ada 95 standard, and the fairly widespread use of stable and high-performance Ada 83 and Ada 95 compilers within and outside the defense sector, Ada was seen to have matured. There was no longer a need for a joint program office. AJPO’s responsibilities were transitioned to the military services, to permanent structures like the Defense Information Systems Agency, and to the private sector, especially the Ada Resource Association. AJPO closed its doors in 1998.

It is useful to comment on the Ada “mandate.” The term “mandate” has acquired a very negative connotation in current U.S. political life, but its use to describe DoD’s Ada policy is inappropriate. In its negative sense, “mandate” has meant the Federal government compelling action from state or local governments, the business sector, or the general public, without providing a suitable level of Federal funding for these actions. On-the-job safety regulations, air and water quality standards, requirements placed on educational institutions, are sometimes assailed in the daily press as “unfunded mandates.”

Whatever one’s opinion on Federal regulation in general, one must agree that the Ada requirement is simply not a mandate in the sense above. DoD has never required *anyone* to use Ada except vendors developing software under DoD contract, for DoD use. Whether DoD should impose such a requirement—or whether such a requirement could be consistently enforced—has been, of course, a hotly debated issue; the fact remains that the Ada requirement was a contractual requirement, nothing more.

3.1.2 Non-defense Applications

We have been tracking non-defense Ada projects for a number of years, and participate in a joint SIGAda-Ada IC effort to prepare and publish application briefs for projects whose sponsors are willing to see some publicity. Several dozen of these “success stories” are online and accessible from the Ada IC and SIGAda Educator web sites.

It is often difficult to get authoritative information for attribution. Many companies are, understandably, uncomfortable seeing details of their projects—including languages and tools used—described in print. These details are often considered trade secrets, and sometimes are deemed to be unwanted invitations to “head hunters” to approach key employees. On the other hand, the published stories, combined with inside tips from software developers, yield enough information for a useful summary of the state of Ada in the non-defense world.

We can say with confidence that software written in Ada can be found in

- on-board software in most new commercial aircraft, including the Boeing 777 with several million lines of Ada, other recent Boeing and Airbus models, the latest Russian Ilyushin and Tupolev airliners, Canadair and Embraer regional aircraft, and so on
- many countries’ latest air traffic control systems, including nations in North America, Europe, Asia, Africa, and Australia

- a number of commercial communications satellites, including Intelsat VII, NSTAR, PanAmSat, ChinaSat, and others
- much new railway signaling and control, including the French TGV, Channel Tunnel, and Paris suburban systems, as well as urban metro lines in Paris, London, Hong Kong, Cairo, Calcutta, Caracas and elsewhere
- the secondary shutdown system in a Czech nuclear power plant
- a U.S. commercially marketed power plant emission monitoring system
- a significant proportion of the U.S. military and civilian Global Positioning System (GPS) receivers, including some of the mass-produced GPS receivers used in U.S. rental cars
- a major U.S. computer-controlled steel rolling mill, an early adopter of Ada now moving to a second-generation Ada 95 control system
- major banking and financial systems including the Swiss Postbank electronics funds transfer system
- several major commercially-marketed medical-analytical systems

It goes without saying that these projects were under no U.S. DoD requirement to use Ada. Further, while many of the projects are government-sponsored (by air traffic control agencies, national railway administrations, etc.), the governments involved generally did not impose the use of Ada. It follows that Ada was chosen as the implementation language, by the companies doing the implementing. Why did they choose to go with Ada?

In some cases—avionics, for example—it can be conjectured that a company’s experience with Ada in the defense sector led them to choose it for similar commercial projects. There are other reasons as well. In the various published articles and success story briefs, a recurring theme is that Ada was chosen because of its technical characteristics. Among other attributes, strong typing and language-supported multitasking are often mentioned; also, the confidence engendered by compiler validation seems to be an important reason for choosing Ada. In Ada 95 projects, the ease with which new Ada code can interface with existing subsystems like SQL, CORBA, and the like is starting to be an important factor.

3.2 Education and Ada

In this section we briefly discuss the Ada 95 textbook literature and survey the use of Ada in college and university education.

3.2.1 Ada 95 Textbooks

At this writing, seventeen published books, at different levels but with Ada 95 as their focus have appeared since 1995. Anyone wishing to learn the language has an excellent range of books from which to choose.

The Ada 95 Reference Manual [Taft and Duff 1997] and Rationale [Barnes 1997], both available on the Web, but until recently only with difficulty in paper form, have now been formally published as books.

Four texts [Culwin 1997, English 1997, Feldman and Koffman 1996, and Skansholm 1997] introduce Ada to first-year students and other readers with no programming experience, two works [Beidler 1997, Feldman 1997] focus on data structures and algorithms, and one [Smith 1996] deals specifically with object-oriented programming. Two texts [Burns and Wellings 1995, Burns and Wellings 1997] discuss concurrency and real-time systems. An interesting work in French [Rosen 1995] discusses software engineering methods with Ada 95.

Finally, four works [Barnes 1995, Cohen 1996, Naiditch 1995, and Wheeler 1997] introduce Ada 95 to experienced programmers, and one book [Johnston 1997] introduces the language specifically to readers experienced with C or C++.

3.2.2 Ada as a Foundation Programming Language

Since 1991, this writer has tracked the colleges and universities that have adopted Ada as a "foundation language," that is, in one of the first few computing courses taken by students majoring in the field.

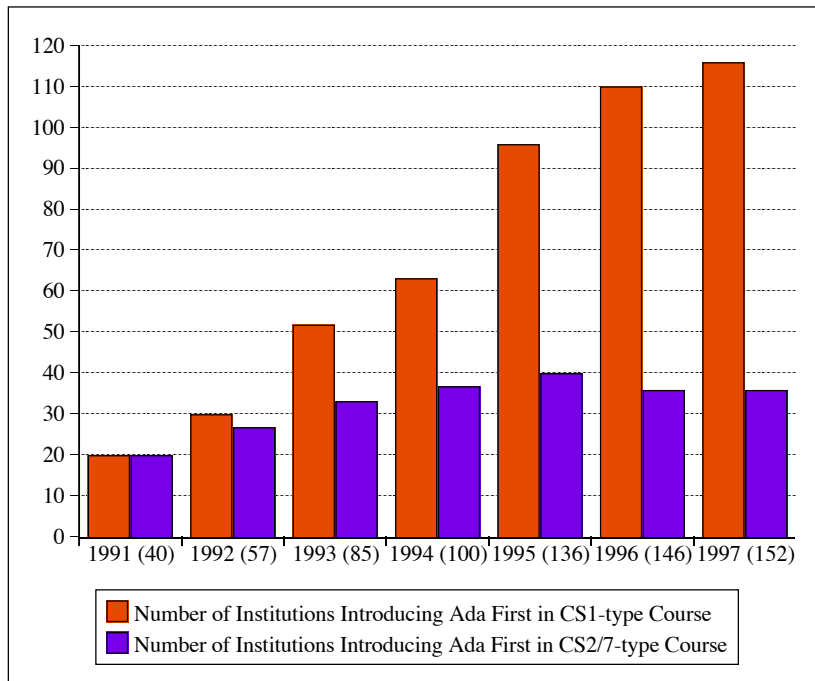
Computing curricula are far from standardized, and there is much variation from institution to institution depending upon local needs and politics. The courses we have tracked are:

- The introductory-level course taught to majors in the first year (some refer to this as “CS1”);
- The second course, which follows the “CS1” course, which is, in general, devoted mostly to algorithms and data structures (some refer to this as “CS2”);

In some institutions, one or both of the above courses are “pre-major,” so the third course is the first one taught specifically to majors. This is often a more advanced data structures course or similar.

Given the local variations, it is difficult to write a one-size-fits-all definition of the courses we follow, but we choose to keep track of precisely

these courses because they are taken by majors in either first or second year, and thus early enough to serve as a foundation upon which to build a large portion of the software curriculum. The following chart shows the trend since 1991.



3.2.3 The Ada IC “CREASE” Database

The Ada Information Clearinghouse maintains a WWW database called CREASE (Catalog of Resources for Education in Ada and Software Engineering) that keeps track of Ada-related courses, in both academic institutions and training companies. They send out occasional canvasses, follow up to ensure that their data is current, and make an electronic questionnaire available at the site. At this writing, 445 organizations are present in the database, describing 799 different courses.

3.3 Ada 95 Compiler Availability

Ada 95 compilers are readily available for all the popular computing host and target platforms and many specialized targets as well.

For example, at this writing freely downloadable ports of GNU Ada 95 (GNAT), with full source code, exist for

- Sun Solaris 2.5.1 (Sun SPARC and 80x86/Pentium)
- Sun OS 4.1.3 (Sun SPARC)
- IBM OS/2 (Warp), Windows 95 or NT, DOS, Linux (Elf object format) (all for 80x86/Pentium)
- IRIX 6.2 (SGI)
- Digital Unix 3.2 (OSF), OpenVMS 7.1 (DEC Alpha)
- AIX (PowerPC / RS6000)
- HP-UX 10.x (HP PA)
- MachTen host, MachTen and MacOS targets (Power Mac)
- Sinix 5.42 (Siemens Nixdorf)

These can be downloaded from various ftp servers, and are included on the Walnut Creek Ada CDROM.

A fully functional Aonix ObjectAda development system for Windows 95 and NT is included on a CDROM with at least two textbooks [Feldman 1996, Johnston 1997].

Other suppliers offer proprietary compilers for popular host platforms, supporting targets for these platforms as well as for various bare boards, real-time operating systems, digital signal processors, and so on. Two suppliers—Aonix and Intermetrics—support a Java Virtual Machine target, and a GNAT version targeting the JVM is under active development. At this writing the official Validated Compiler List (available on the Ada IC web site) shows over 80 validated implementations from 10 different suppliers.

4. Programming

We will introduce programming in Ada using a series of complete examples, which serve better than rules and fragments to illustrate the use of the language. Each of these programs has been compiled and tested. In each case, we have numbered the lines so that we can refer to them in the discussion; the line numbers are not part of the programs themselves, and are not entered into a source file.

The first several examples will introduce the basic structure of an Ada program; the rest will introduce more interesting concepts of program construction using modules, object-oriented programming, and concurrent programming. Each program introduces several new concepts.

4.1 Examples of Basic Structure and Syntax

Two of the most fundamental Ada concepts are the *type* and the *package*.

4.1 Types and Objects

A type consists of two sets: a set of values and a set of operations appropriate for those values. Each variable and constant in Ada has a type; variables and constants are thus holders for values. The Ada standard defines a number of standard types either in the language—for example, `Integer`, `Float`, `Boolean`, `Duration` (elapsed time), `Character`, and `String`—or in standard library packages—for example, `Time` and `Complex`. Further, the programmer can define application-specific types of various kinds (for example, enumerations, records, arrays, and tasks). Conversion among types can be done with explicit type-conversion operations; implicit type conversion does not occur in Ada.

The Ada standard requires that the compiler ensure that operations are appropriate for the values to which they are applied. Appropriateness is checked statically at compilation time, wherever possible; otherwise, run-time checks are compiled into the executable program. If a run-time check fails (for example, the program attempts to store a computed negative value in a variable declared as nonnegative), a standard *exception* is raised (in this case, `Constraint_Error`). A raised exception can be handled by the program, if code has been provided to do so; otherwise, the program terminates.

Since in Ada, each variables or constant has a state—its value—and a well-defined behavior—the set of operations defined for its type, it is entirely appropriate to refer to variables and constants as *objects*, and we shall do so in this development.

4.1.2 Packages and Files

A package encapsulates a collection of resources: type declarations, functions, procedures, variables, constants. A package generally consists of two compilation units, which developers generally choose to place in distinct source files. The package *specification* or *interface* unit gives a list of those resources intended to be directly visible to clients; the *body* or

implementation unit contains the code bodies of the procedures and functions, as well as any internal resources used by the packages but not exported to the client.

A package interface is roughly analogous to a “header file” as used in other languages, but differs in that it is required and its contents are strictly structured. The package interface can be seen as a required “contract” between a package’s developer and its user.

In other languages—the C family in particular—source programs are organized into physical files. These files have semantic meaning in the language—they create scope and visibility—so the precise contents of each file are of critical importance. In Ada, this semantic content is embodied in the *compilation unit*.

Package interfaces, package implementations, and main procedures are all compilation units; whether the units making up a program are distributed into individual files or collected together in a single file, or something in between, is irrelevant to the program semantics, and is, in fact, never discussed in the standard. Some compilers have a preferred file structure, but this can be overridden. Thus the organization of a program into physical files is a matter of one’s organizational style and does not affect the program’s meaning.

The Ada standard libraries are implemented as packages; application programs are constructed from these and application-specific or generally reusable but non-standard packages. In this major section our package examples all come from the standard libraries; in the next one we will introduce some packages of our own.

4.1.3 The Usual “Hello World” Program

Our first example is `Hello`.

Listing 1

```
1 with Ada.Text_IO;
2 procedure Hello is
3
4   -- A first very small, obligatory program
5
6 begin -- Hello
7
8   Ada.Text_IO.Put_Line
9     (Item => "This is the obligatory Hello program!");
10
11 end Hello;
```

An Ada program usually consists of a main procedure, whose statements make reference to resources that are either locally declared or imported from packages. Each package is mentioned in a `with` clause—officially called a *context clause*—at the top of the program; in line 1 of `Hello`, the standard Ada input-output library `Ada.Text_IO` is indicated.

Lines 8 and 9 use the procedure `Put_Line` to display the quoted string on the standard output file and move to a new line. We have used the full form of the call: The package name `Ada.Text_IO` precedes the call and the name of the formal parameter `Item` is associated with its actual value, using the symbol `=>`. This may seem a bit verbose; we will show some terser forms later.

The words `with`, `procedure`, `is`, `begin`, and `end` are five of Ada's *reserved words*. These are used for various syntactic purposes, and cannot be used in other contexts, e.g., as variable names. By the end of this section, you will have seen most of the reserved words.

`Put_Line`, on the other hand, is the name of a standard procedure. Procedure names are not reserved; you could write another package that provided a procedure `Put_Line`, and call this procedure in a program, as long as the compiler had enough information to resolve the potential ambiguity in the name.

The layout of a program in terms of lines is not specified by the language, nor is the case of reserved words and identifiers. A program is treated syntactically as a sequence of characters; indentation and use of blank lines and other white space is purely a matter of programmer style. The style we use here—lower-case reserved words, mixed-case identifiers, white space—is generally consistent with the Reference Manual and most textbooks, and with generally accepted practice.

The only exception regarding lines is that a *comment*—inserted for the human reader and ignored by the compiler—begins with `--` and ends at the end of the same line. This line-oriented comment style—used in Fortran and assembler, and adopted by C++—was chosen in preference to the bracketed style of Pascal and C, in order to make it obvious that a block of code has been “commented-out.” Lines 4 and 6 show comments used in two ways: an entire line, and following some Ada code.

In the examples of this section, we have chosen not to take up space with program comments, but to concentrate on numbered lines and prose explanations.

A note on context clauses: A context clause is only superficially similar to the `#include` of C and other languages. It serves two purposes:

- At compilation time, the clause provides context for the compilation: The compiler reads each package interface (in source or intermediate form, depending on the compiler) and checks the correctness of each reference made to resources in that package.
- At binding and linking time, the clause is used to bind the compiled form of the main with the compiled form of the packages; the binder and linker must ensure that the compiled form of each package is up to date.

There is no exact equivalent in Ada to the `#include`, which simply copies in the contents of another file. One could in theory use a preprocessor to get the equivalent, but it is almost never necessary to do so, because context clauses provide similar functionality with more integrity-checking.

A final point on the standard libraries: the many Ada 95 standard libraries are easy to recognize: their names all begin with `Ada`. They are in fact *child packages* of the root package `Ada`. More on child packages later.

4.1.4 A Less “Verbose” Coding Style

`Hello_Terse` shows two ways of reducing the “verbosity” of a program. As we shall see, one programmer’s “verbosity” may well be another’s “clarity.”

Listing 2

```
1 with Ada.Text_IO;
2 use Ada.Text_IO;
3 procedure Hello_Terse is
4 -- A terse version of the Hello program
5 begin
6   Put_Line ("This is the obligatory Hello program!");
7 end Hello_Terse;
```

First, we add a `use` clause (line 2), which makes all the resources of the given package directly visible. That is, we can name a resource (as in line 6) without preceding the reference with the package name. This may seem like an obvious advantage, but for longer programs using a number of packages, omitting package names tends to obscure the meaning of the statements and to lead to reader confusion. We will return to this issue.

Second, we omit the formal parameter name. This, too may seem like an obvious advantage, but for procedures with a number of parameters, using the named association allows us to supply actual parameters in any order and may also lead to clearer, less error-prone procedure calls.

4.1.5 The Calendar package

The next several examples examine the standard library package, `Ada.Calendar`, and use it while introducing several Ada data and control structures.

Here is an extract from the package interface of `Ada.Calendar`, which provides date and time services in a standard fashion. The resources listed in the interface are exported to client programs.

Listing 3

```

1 package Ada.Calendar is
2   type Time is private;
3
4   subtype Year_Number is Integer range 1901..2099;
5   subtype Month_Number is Integer range 1..12;
6   subtype Day_Number is Integer range 1..31;
7   subtype Day_Duration is Duration range 0.0..86_400.0;
8
9   function Clock return Time;
10
11  function Year (Date : Time) return Year_Number;
12  function Month (Date : Time) return Month_Number;
13  function Day (Date : Time) return Day_Number;
14  function Seconds(Date : Time) return Day_Duration;
15  ...
16 end Ada.Calendar;
```

We see that `Time` is declared (line 2) as a *private type*, which signals that its structure—its set of values— is not directly available to us. We use time values only via package operations.

Lines 4-7 define four *subtypes*. A subtype declaration creates a subset of the values of the original type; subtypes are commonly used, as here, to restrict the appropriate ranges of certain variables. The three `Integer` subtypes here have obvious meanings in the calendar; the three functions in lines 11-13 extract the corresponding components from a `Time` value. The `Duration` subtype `Day_Duration` provides the elapsed time values in a 24-hour period; the function `Seconds` returns the time elapsed since midnight of the given day.

4.1.6 Displaying the Current Date

The next example, `Show_Date`, displays the current date in the form 05 AUG 1997.

Listing 4

```

1 with Ada.Text_IO;
```

```

2 with Ada.Calendar;
3 procedure Show_Date is
4
5   Now : Ada.Calendar.Time;
6
7   type Months is
8     (Jan, Feb, Mar, Apr, May, Jun,
9      Jul, Aug, Sep, Oct, Nov, Dec);
10
11 begin
12
13   Now := Ada.Calendar.Clock;
14
15   Ada.Text_IO.Put_Line("Today is"
16     & Integer'Image(Ada.Calendar.Day(Now)) & ' '
17     & Months'Image
18     (Months'Val( Ada.Calendar.Month(Now) - 1))
19     & Integer'Image(Ada.Calendar.Year (Now)));
20
21 end Show_Date;
```

The context clause in line 2 mentions `Ada.Calendar`; line 7 declares a time variable `Now`, which, in line 13, is set to the current time by a call to `Ada.Calendar.Clock`. Line 13 is an *assignment statement*; the symbol `:=` means *assignment*; the statement

```
Now := Ada.Calendar.Clock;
```

assigns to the variable `Right_Now` the value returned by the `Clock` call.

To display the date, we must extract the month, day, and year from the time value. We extract the day in line 16, with the function call `Ada.Calendar.Day(Now)`, and extract the month and year in lines 18 and 19 with similar calls.

We choose to display the date (lines 15-19) by constructing a string from substrings concatenated together using the `&` operator. The expression (line 16)

```
Integer'Image(Ada.Calendar.Day(Now))
```

extracts the day as an integer value, then produces its *image* (`Integer'Image`) as a numeric string. The form `Integer'Image` is known as an *attribute*; each type in Ada has a set of retrievable attributes.

To display the month as a 3-letter abbreviation instead of a number 1 .. 12, we introduce an *enumeration type* in lines 5-7. An enumeration type is declared by listing—enumerating—its values. This creates an ordered set of

values, with no value appearing more than once. The values are, syntactically, either characters or (as in this case) identifiers.

Two especially useful attributes of enumeration types are `'Pos` and `'Val`. `Months'Pos(Jan)` returns a nonnegative integer representing the *position* of `Jan` in the type, or 0. `Months'Val(0)` goes the other way, returning `Jan`; `Months'Pos(May)` returns 4; `Months'Val(8)` returns `Sep`. Since `Ada.Calendar.Month` returns an integer in the range 1 .. 12, the expression

```
Months'Val(Ada.Calendar.Month(Now) - 1)
```

returns just the right enumeration value, and so

```
Months'Image(Months'Val(Ada.Calendar.Month(Now) - 1))
```

produces the desired output substring.

4.1.7 Displaying the Date and Time

The program `Show_Date_and_Time` extends the previous example to display both the time and the date.

Listing 5

```
1 with Ada.Text_IO, Ada.Integer_Text_IO;
2 with Ada.Calendar;
3 procedure Show_Date_and_Time is
4
5   type Months is
6     (Jan, Feb, Mar, Apr, May, Jun,
7      Jul, Aug, Sep, Oct, Nov, Dec);
8
9   Now      : Ada.Calendar.Time := Ada.Calendar.Clock;
10  SecsPast_0h00: Natural
11      := Natural(Ada.Calendar.Seconds(Now));
12  MinsPast_0h00: Natural := SecsPast_0h00 / 60;
13  Secs        : Natural := SecsPast_0h00 rem 60;
14  Mins        : Natural := MinsPast_0h00 rem 60;
15  Hrs         : Natural := MinsPast_0h00 / 60;
16
17 begin
18
19   Ada.Text_IO.Put("The date and time is"
20     & Integer'Image(Ada.Calendar.Day(Now)) & ' '
21     & Months'Image
22       (Months'Val(Ada.Calendar.Month(Now) - 1))
23     & Integer'Image(Ada.Calendar.Year(Now)) & ' ');
24
25   if Hrs < 10 then
```

```
26   Ada.Text_IO.Put (Item => '0');
27   end if;
28   Ada.Integer_Text_IO.Put (Item => Hrs, Width => 1);
29   Ada.Text_IO.Put (Item => ':');
30
31   if Mins < 10 then
32     Ada.Text_IO.Put (Item => '0');
33   end if;
34   Ada.Integer_Text_IO.Put (Item => Mins, Width => 1);
35   Ada.Text_IO.Put (Item => ':');
36
37   if Secs < 10 then
38     Ada.Text_IO.Put (Item => '0');
39   end if;
40   Ada.Integer_Text_IO.Put (Item => Secs, Width => 1);
41
42   Ada.Text_IO.New_Line;
43
44 end Show_Date_and_Time;
```

Using the `'Image` attribute to display a number is convenient, but it gives us little control over the format of the output value. Therefore, the context clause in line 1 mentions another standard package, `Ada.Integer_Text_IO`, which provides extended facilities for reading and writing integer values.

In line 9 we declare `Now` as before, then add five more variables. These are all given as type `Natural`, a subtype of `Integer`, declared in the standard as

```
subtype Natural is Integer range 0..Integer'Last;
```

where `Integer'Last` returns the largest positive integer supported by the type.

This is a good time to mention that the Ada standard does not specify the range of `Integer`, but requires that the range $-2^{15}+1$.. $+2^{15}-1$ (16 bits) be included. In current practice on popular computers, it is usually safe to assume that `Integer` is represented using 32 bits. For absolute certainty, Ada allows one to declare one's own numeric types and specify the required width in bits; this is an example of the way Ada provides simple default structures but allows the programmer much flexibility in overriding the defaults where necessary.

Returning to the example, each of our six declarations includes an *initial expression*, which is evaluated at run time when the declaration is *elaborated*. Note that each of these expressions is dynamic, depending on the current clock value. The expression

```
Natural (Ada.Calendar.Seconds (Now) )
```

converts the `Seconds` component from `Day_Duration` to `Natural`. `Duration` is actually a fixed-point type, not an integer one, so fractions of seconds can be represented. Converting to `Natural` truncates the value, discarding the fractional part; we are interested only in whole seconds here.

The expressions

```
SecsPast_0h00 / 60
SecsPast_0h00 rem 60
```

are integer expressions, computing the quotient and the remainder respectively. In Ada, the interpretation of arithmetic operators, including division, depends on the types of their operands. `SecsPast_0h00` (seconds since midnight) and `60` are both integer values, so the division is a truncating one.

There is no C-style promotion or implicit conversion in Ada; for example, mixed integer/real expressions are rejected by the compiler. One can use explicit type conversions to adjust the types in expressions that would otherwise be mixed. The requirement for explicit type conversion ensures that the result of an arithmetic expression is obvious to the human reader.

In our example, since the values of all the variables are computed at the time their declarations are elaborated, the statement-sequence part of the program need only display the result. Lines 19-23 display the date just as before; the rest of the program displays the time in 24-hour hh:mm:ss form. The `if` statements in lines 25-27, 31-33, and 37-39 are used to display leading zeroes where necessary, for example 09:15:08. `if` statements are *fully bracketed*, that is, the `if` must have a closing `end if`. An `if` can include a single `else` and/or one or more `elsif` clauses. Two other `if` variations are

```
if ... then
  ...
else
  ...
end if;

if ... then
  ...
elsif ... then
  ...
elsif ... then
  ...
else
```

```
...
end if;
```

Lines 28, 34, and 40 are calls to `Ada.Integer_Text_IO.Put`. `Width=>1` is a formatting parameter that requests *at least* 1 position for the output value: A value of one or more digits will be displayed properly; the effect of this width parameter is to left-adjust the value.

`Width` is a parameter that is defined with a default value, so that a call to `Put` can omit `Width`. The default width is one larger than the width of `Integer'Last`, typically 11.

We note in passing that the integer `Put` has yet another parameter, `Base`, which defaults to 10 and indicates the desired number base (bases in the range 2..16 are allowed). The call

```
Ada.Integer_Text_IO(Item => 19, Base => 8);
```

will display the octal value `8#23#` (base first, then value delimited by `#` signs); substituting `Base=>2` will display `2#10011#`. Using this syntax, it is possible to format *input* values in non-decimal bases. Again we see that Ada provides simple defaults with much capability to override these.

4.1.8 The Brevity/Clarity Tradeoff

Ada has been criticized for its apparent “verbosity.” Ada’s proponents respond that Ada was designed for the development and maintenance of programs of nontrivial size, with long life cycles. Such programs are written once, but read and modified many times over their lives, so the language definitely favors the reader over the writer, and Ada programmers tend to become conditioned to write less for their own eyes than for others’. This courtesy to the reader of a program is not considered to be a disadvantage!

On the other hand, it is possible to write Ada more tersely. Single-character variable names are legal and the language does not compel the use of white space. Also, as we observed earlier, one can add `use` clauses and employ positional parameters instead of named ones. Consider `Show_Use_Clause`, which is a terse version of `Show_Date_and_Time`. If it were a program of a few thousand lines instead of 44 lines, making reference to five or ten different packages, the reader would probably miss the explicit package and formal parameter names.

Listing 6

```
1 with Ada.Text_IO, Ada.Integer_Text_IO;
```

```

2 use Ada.Text_IO, Ada.Integer_Text_IO;
3 with Ada.Calendar;
4 use Ada.Calendar;
5 procedure Show_Use-Clause is
6
7   type Months is
8     (Jan, Feb, Mar, Apr, May, Jun,
9      Jul, Aug, Sep, Oct, Nov, Dec);
10
11   Now      : Time := Clock;
12   SecsPast_0h00: Natural := Natural(Seconds(Now));
13   MinsPast_0h00: Natural := SecsPast_0h00/60;
14   Secs      : Natural := SecsPast_0h00 rem 60;
15   Mins      : Natural := MinsPast_0h00 rem 60;
16   Hrs       : Natural := MinsPast_0h00 / 60;
17
18 begin
19
20   Put("The date and time is"
21     & Integer'Image(Day(Now)) & ' '
22     & Months'Image(Months'Val(Month(Now) - 1))
23     & Integer'Image(Year(Now)) & ' ');
24
25   if Hrs < 10 then
26     Put ('0');
27   end if;
28   Put (Hrs, 1);
29   Put (':');
30
31   if Mins < 10 then
32     Put ('0');
33   end if;
34   Put (Mins, 1);
35   Put (':');
36
37   if Secs < 10 then
38     Put ('0');
39   end if;
40   Put (Secs, 1);
41
42   New_Line;
43
44 end Show_Use-Clause;
```

Note that in this program, there are calls to three different `Put` procedures: one for single characters, one for strings, and one for integers. This is an example of *overloading*, or giving the same name to several different subprograms. Overloading is quite common in Ada; the compiler readily distinguishes the three procedures by examining the profile (order, number, and type) of the parameters.

If, by some chance, two different procedures with the same name *and* the same parameter profile were made directly visible by `use` clauses (this could happen if the procedures were in different packages), the compiler

would discover the ambiguity and give an error message. The programmer could easily correct the ambiguity, by prefixing the package names.

We have designed the examples here to explicate the language and some of its standard libraries; therefore we generally use a verbose style for maximum clarity. Occasionally, we add `use` clauses and omit parameter names, where there is no loss of clarity. Programming style issues should really be managed by each project team.

4.1.9 Loops, Arrays, and Files—the Advantage of Subtypes

Our next example produces a simple letter-frequency histogram, a horizontal bar graph that shows the number of times each alphabet letter occurs in an input file. For example, given the input

```
This is a test of an Ada 95
simple histogram program.
```

the program displays

```

a | *****
d | *
e | **
f | *
g | **
h | **
i | *****
l | *
m | ***
n | *
o | ***
p | **
r | ***
s | *****
t | ***
A | *
T | *
```

Input is taken from the standard input file, which for most operating systems defaults to the terminal keyboard. Input can be “redirected” from an external disk file if the operating system permits this; redirection is an operating system feature and lies outside the Ada standard.

Listing 7

```

1 with Ada.Text_IO;
2 procedure Histogram is
3
4   subtype UpperCase is Character range 'A'..'Z';
5   subtype LowerCase is Character range 'a'..'z';
```

```

6
7  Uppers : array(UpperCase) of Natural := (others => 0);
8  LowerCase : array(LowerCase) of Natural := (others => 0);
9
10 NextCh   : Character;
11 PlotChar : constant Character := '*';
12
13 begin
14
15   while not Ada.Text_IO.End_Of_File loop
16     while not Ada.Text_IO.End_Of_Line loop
17
18       Ada.Text_IO.Get(NextCh);
19
20       case NextCh is
21         when UpperCase =>
22           Uppers(NextCh) := Uppers(NextCh) + 1;
23         when LowerCase =>
24           LowerCase(NextCh) := LowerCase(NextCh) + 1;
25         when others =>
26           null;
27         end case;
28
29       end loop;
30       Ada.Text_IO.Skip_Line;
31     end loop;
32
33   for C in LowerCase loop
34
35     if LowerCase(C) /= 0 then
36       Ada.Text_IO.Put(Item => C & " | ");
37       for Count in 1..LowerCase(C) loop
38         Ada.Text_IO.Put(PlotChar);
39       end loop;
40       Ada.Text_IO.New_Line;
41     end if;
42
43   end loop;
44
45   for C in UpperCase loop
46
47     if Uppers(C) /= 0 then
48       Ada.Text_IO.Put(Item => C & " | ");
49       for Count in 1..Uppers(C) loop
50         Ada.Text_IO.Put(PlotChar);
51       end loop;
52       Ada.Text_IO.New_Line;
53     end if;
54
55   end loop;
56
57 end Histogram;

```

Lines 4 and 5 introduce subtypes for the upper-case and lower-case letters, respectively; lines 7 and 8 introduce two *arrays* to hold the letter frequency counts. An array is an *indexed*, or *subscripted*, collection of elements; the index set is given in parentheses. All elements of a given array

have the same type, but this type can be arbitrary, chosen for the application needs. Because all the elements have the same type, we say that an array type is *homogeneous*. In this case we need nonnegative counters for the letters, so the array elements are of type `Natural`. The initial expression `(others => 0)` initializes all elements of both arrays to 0.

Line 11 introduces the constant `PlotChar`, as the asterisk we use to plot the histogram. A constant *must* have an initial expression; once the constant is set, it cannot be changed during its lifetime (in this case, the duration of the program execution). A constant value is set at the time its declaration is elaborated; it need not be static but can be computed at that time.

Lines 15 and 16 are the opening lines of the two nested *while* loops that drive the program. `End_Of_Line` and `End_Of_File` are Boolean-returning standard functions. Line 18 is a `Get` that reads a single character from the standard input file; it is because `Get` skips over line terminators that we need to test explicitly for `End_Of_Line` in line 16. Lines 29-31 bracket the loops.

Once a character is read, the `case` statement in lines 20-27 classifies it and acts accordingly, incrementing the appropriate counter. Note the `case` form: The first two `when` clauses refer to subtype ranges `UpperCase` and `LowerCase` and the `when others` covers all other possible values of the character variable. An Ada `case` statement is safe: *All* values of the `case` expression must be covered, using an `others` clause if necessary.

Once the input file has been completely read, we draw the bar graph. Line 33 starts a `for` (counting) loop which is bracketed by the `end loop` in line 43. The meaning of the `for` is evident: it covers, in succession, each value in the subtype range of `LowerCase`.

We choose to plot bars only for those characters that actually appear in the input. The expression `LowerCase(C) /= 0` (line 35) returns true if the number of occurrences of `C` was non-zero; `/=` is Ada's "unequal" symbol. In line 37 we show a loop expression whose bounds are given explicitly instead of by naming a subtype.

The loop in lines 45-55 just repeats the logic of the earlier loop, plotting bars for the upper-case characters.

The control structures in this example show the value of using subtype ranges. Once a subtype is carefully defined with a range determined by the application, it can be used repeatedly throughout the program. This ensures that appropriate ranges are used in loops and, especially, in loops that run through arrays. Programming without subtype ranges—either in Ada or in

languages like those in the C family that do not support subtypes—often results in runtime errors due to overrunning subscripts; such errors do not occur with carefully designed subtypes.

4.1.10 Type Composition: Arrays and Records

Our next example is a miniature information systems application. A company sells five kind of gadgets, indicated by the colors Red, Blue, Green, Yellow, and Black. In a given time period, a transaction file is created in which a transaction indicates a kind of gadget and a positive (sale) or negative (return) monetary quantity. The program `Gadget_Report` summarizes the sales, giving, for each of the five gadget types, the number of transactions and the total monetary value. For example, an input transaction file containing

```
green 56.00
blue 25.00
green 2078.00
black -2065.00
```

will result in this report:

Gadget Type	Transactions	Value
RED	0	0.00
BLUE	1	25.00
GREEN	2	2134.00
YELLOW	0	0.00
BLACK	1	-2065.00

Listing 8

```
1 with Ada.Text_IO;
2 with Ada.Integer_Text_IO;
3 with Ada.Float_Text_IO;
4 procedure Gadget_Report is
5
6   subtype Money is
7     Float range -1_000_000_000.00..1_000_000_000.00;
8
9   type Gadgets is (Red, Blue, Green, Yellow, Black);
10  package Gadget_IO is
11    new Ada.Text_IO.Enumeration_IO (Enum => Gadgets);
12
13  type SummaryEntry is record
14    HowMany: Natural := 0;
15    Net     : Money := 0.00;
16  end record;
17
18  type Summary is array(Gadgets) of SummaryEntry;
19
```

```
20  TodaysSummary : Summary;
21  InputData: Ada.Text_IO.File_Type;
22  WhichKind: Gadgets;
23  Amount: Money;
24
25  begin
26
27    Ada.Text_IO.Open(File => InputData,
28      Mode => Ada.Text_IO.In_File, Name => "today.dat");
29
30    while not
31
32      Ada.Text_IO.End_of_File(File => InputData) loop
33      Gadget_IO.Get(File => InputData, Item => WhichKind);
34      Ada.Float_Text_IO.Get
35        (File => InputData, Item => Amount);
36      Ada.Text_IO.Skip_line(File => InputData);
37
38      TodaysSummary(WhichKind).HowMany :=
39        TodaysSummary(WhichKind).HowMany + 1;
40      TodaysSummary(WhichKind).Net :=
41        TodaysSummary(WhichKind).Net + Amount;
42
43    end loop;
44
45    Ada.Text_IO.Close (File => InputData);
46
47    Ada.Text_IO.Put_Line
48      (Item => "Gadget Type Transactions Value");
49    Ada.Text_IO.Put_Line
50      (Item => "-----");
51
52    for WhichKind in Gadgets loop
53      Gadget_IO.Put(Item => WhichKind, Width => 7);
54      Ada.Integer_Text_IO.Put(
55        Item => TodaysSummary(WhichKind).HowMany,
56        Width => 12);
57      Ada.Float_Text_IO.Put(
58        Item => TodaysSummary(WhichKind).Net,
59        Fore => 10, Aft => 2, Exp => 0);
60      Ada.Text_IO.New_Line;
61    end loop;
62
63  end Gadget_Report;
```

We represent our monetary type as a subtype of the predefined type `Float`. Line 3 mentions `Ada.Float_Text_IO`, which will allow us to read and write these quantities; lines 6-7 shows the subtype declaration. The underscore characters groups the digits by threes; they are just used to improve readability.

We declare `Gadgets` as an enumeration type (line 9). Lines 10-11 constitute a *generic instance* of the standard generic package `Ada.Text_IO.Enumeration_IO`. A generic package is a *template* for a package; in this case, `Enumeration_IO` is generic because we must supply

the name of an enumeration type on whose values input and output is being done. The instance `Gadget_IO` is capable of reading and writing *exactly* the five gadget values.

This generic specialization is especially advantageous in reading input values, because the input routine (`Gadget_IO.Get`) *validates* that an input token in fact belongs to the set of desired values. If the token does not belong to the desired set, the standard exception `Ada.Text_IO.Data_Error` is raised at the point of the `Get` call; the exception can be handled by program code, as we shall see in the next example.

In our reporting program, each transaction is read from the input file and the appropriate summaries—number of transactions and net income per gadget kind—are updated. The transaction can then be discarded. To define a summary entry, lines 13-16 declare a *record type*. A record type—in this case `SummaryEntry`—has fields or *components*—in this case `HowMany` and `Net`; a record type is heterogenous because in general, each component has its own type. This type declaration creates no objects; it just gives a rule for creating objects from object declarations.

To hold the gadget summaries, we now declare (line 18) an array type `Summary`. The components of the array are of type `SummaryEntry`, which illustrates *type composition*: a programmer-defined array type each of whose elements is a programmer-defined record type. Types can be composed at will in this fashion.

Now, in lines 20-23, we declare four objects (variables): an array object `TodaysSummary`, a file object `InputData`, and two variables to hold the current transaction.

Instead of reading from standard input, in this program we read from an external file whose name as seen by the operating system is `today.dat`. This external file is associated with our file object `InputData` by the `Open` call in lines 27-28. The `File` and `Name` parameters are obvious; `Mode` refers to one of the standard sequential file modes `In_File`, `Out_File`, and `Append_File` (an output mode in which writing begins at the end of an existing file). (Ada's standard library for direct access files also has an `InOut_File` mode to open a direct file for both reading and writing).

The main loop of this program (lines 30-43) is very straightforward: We read a transaction from the file and update the summaries. The expression

```
TodaysSummary(WhichKind).HowMany
```

selects the `HowMany` field of the `WhichKind` element of the array object `TodaysSummary`.

Finally, lines 45-61 close the input file and then loop through the summary array to display the results. The formatting parameters `Fore` and `Aft` (line 59) for `Ada.Float_Text_IO.Put` indicate the desired number of displayed places before and after the decimal point. A non-zero `Exp` calls for the number to be displayed in “E-notation” and gives the number of places in the exponent. Thus `Fore =>1, Aft=>2, Exp=>2` would display `-2065.00` as `-2.07E+3`. This would be fine for an engineering application but not for data processing.

4.1.11 Exception Handling; Command Parameters; Information Systems Features

Our last “inner syntax” example, `Show_Exception`, is a modification of `Gadget_Report` that illustrates exception handling, the decimal-types and edit-directed-output features of the Ada 95 Information Systems Annex (Annex F), and the standard package `Ada.Command_Line`. For the input file `today.dat` as used above, issuing the operating system command

```
show_exception today.dat
```

produces the output

```
show_exception: Input file is today.dat
Gadget Type Transactions Value
-----
RED          0          $0.00
BLUE         1          $25.00
GREEN        2       $2,134.00
YELLOW       0          $0.00
BLACK        1       $2,065.00CR
```

The command `show_exception` produces the output

```
show_exception: No input file provided.
```

A transaction containing bad data is ignored (in a real application, it would be written to an error report); for the file `bad.dat` containing a misspelled gadget type (`gren`),

```
green 56.00
blue 25.00
gren 20.00
black -20.00
```

the command `show_exception bad.dat` results in

```
show_exception: Input file is bad.dat
Ignoring Bad Transaction
Gadget Type Transactions Value
-----
RED          0          $0.00
BLUE         1          $25.00
GREEN        1          $56.00
YELLOW       0          $0.00
BLACK        1          $20.00CR
```

Listing 9

```
1 with Ada.Text_IO, Ada.Text_IO.Editing;
2 with Ada.Integer_Text_IO;
3 with Ada.Command_Line;
4 use Ada.Text_IO, Ada.Integer_Text_IO;
5 procedure Show_Exception is
6
7   type Money is delta 0.01 digits 12;
8   package Money_In is
9     new Ada.Text_IO.Decimal_IO(Num => Money);
10  package Money_Out is new
11    Ada.Text_IO.Editing.Decimal_Output (Num => Money);
12
13  type Gadgets is (Red, Blue, Green, Yellow, Black);
14  package Gadget_IO is
15    new Ada.Text_IO.Enumeration_IO (Enum => Gadgets);
16
17  type Transaction is record
18    Kind: Gadgets;
19    Amount: Money := 0.00;
20  end record;
21
22  type Database is
23    array(Integer range <>) of Transaction;
24
25  type SummaryEntry is record
26    HowMany: Natural := 0;
27    Net : Money := 0.00;
28  end record;
29
30  type Summary is array(Gadgets) of SummaryEntry;
31
32  TodaysActivity: Database(1..100);
33  TodaysSummary : Summary;
34
35  InputData: Ada.Text_IO.File_Type;
36
37  TransactionCount: Natural := 0;
38  WhichKind: Gadgets;
39
40 begin
41
42  Put(Item => Ada.Command_Line.Command_Name);
```

```
43
44  if Ada.Command_Line.Argument_Count = 0 then
45    Put_Line (Item => ": No input file provided.");
46    return;
47  else
48    Put_Line(Item => ": Input file is "
49      & Ada.Command_Line.Argument(Number => 1));
50  end if;
51
52  Open(File => InputData,
53    Mode => Ada.Text_IO.In_File,
54    Name => Ada.Command_Line.Argument(Number => 1));
55
56  while not End_of_File(File => InputData) loop
57    TransactionCount := TransactionCount + 1;
58
59  begin
60    Gadget_IO.Get(File => InputData,
61      Item => TodaysActivity(TransactionCount).Kind);
62    Money_In.Get (File => InputData,
63      Item=>TodaysActivity(TransactionCount).Amount);
64    Skip_line(File => InputData);
65  exception
66    when Constraint_Error =>
67      Put_Line(Item =>
68        "Out of range amount; ignoring transaction");
69      Skip_Line(File => InputData);
70      TransactionCount := TransactionCount - 1;
71    when Ada.Text_IO.Data_Error =>
72      Put_Line(Item =>
73        "Bad gadget or amount; ignoring transaction");
74      Skip_Line(File => InputData);
75      TransactionCount := TransactionCount - 1;
76  end;
77  end loop;
78
79  Ada.Text_IO.Close (File => InputData);
80
81  for Count in 1..TransactionCount loop
82    WhichKind := TodaysActivity(Count).Kind;
83    TodaysSummary(WhichKind).HowMany :=
84      TodaysSummary(WhichKind).HowMany + 1;
85    TodaysSummary(WhichKind).Net :=
86      TodaysSummary(WhichKind).Net
87      + TodaysActivity(Count).Amount;
88  end loop;
89
90  Ada.Text_IO.Put_Line
91    (Item => "Gadget Type Transactions Value");
92  Ada.Text_IO.Put_Line
93    (Item => "-----");
94
95  for WhichKind in Gadgets loop
96    Gadget_IO.Put(Item => WhichKind, Width => 7);
97    Put(Item => TodaysSummary(WhichKind).HowMany,
98      Width => 12);
99    Money_Out.Put(Item => TodaysSummary(WhichKind).Net,
```

```

100     Pic => Ada.Text_IO Editing.To_Picture
101     ("$$$_$$$_$$$9.99CR");
102     Ada.Text_IO.New_Line;
103   end loop;
104
105 exception
106
107   when Ada.Text_IO.Name_Error =>
108     Put_Line (Item => "File Not Found, Goodbye.");
109
110 end Show_Exception;
```

The context clause in line 1 mentions `Ada.Text_IO.Editing`. This Annex F package provides for Cobol or PL/I-style edit-directed output, which is widely used in business reports. The context clause in line 3 mentions `Ada.Command_Line`, which provides a platform-independent equivalent of C's `argc/argv` capability. Using this package, a program can read its own command-line name as well as any other command-line parameters the user has supplied.

In the original `Gadget_Report` above, our monetary type was a subtype of `Float`. This is not the best way to represent money, because `Float` is just an approximation of the real numbers and there is always a danger of accumulated round-off errors. For monetary quantities it is better to use a decimal type which stores quantities exactly to within a given accuracy. Accordingly, we write (line 7)

```
type Money is delta 0.01 digits 12;
```

which specifies a new numeric type with 12 decimal digits of representation, accurate to the nearest 0.01. This gives us, according to the standard, a range of $\pm 10^{10} - 0.01$ or $\pm 999,999,999.99$. Lines 8-11 create instances of the two generic package `Ada.Text_IO.Decimal_IO` (which we will use for input) and `Ada.Text_IO.Editing` (which we will use for edit-directed output).

In `Gadget_Report` we discarded each transaction after reading it. Here we choose to retain all the transactions in an array, so that, for example, we could extend the program to support queries. Accordingly, we declare (lines 17-20) a transaction record type, and (lines 22-23) an array type we shall call `Database`. This is an example of an *unconstrained array type*; the range expression

```
Integer range <>
```

indicates that the bounds of each object will be *some* integer subrange, but that we choose *not* to specify the array bounds in the type declaration, preferring instead to pin down the bounds in our object declarations. This would allow us to declare a number of array objects, each with different bounds. In this example, we declare just one array object, namely (line 32)

```
Today'sActivity: Database(1..100);
```

which allocates one array with bounds 1..100.

Moving now to the program statements, we retrieve and display the program name (as actually seen in the file system) using (line 42) `Ada.Command_Line.Command_Name`, then (lines 44-50) ascertain that the user has supplied an input file, using `Ada.Command_Line.Argument_Count` (similar to `argc`). If a file is supplied, we open it, using `Ada.Command_Line.Argument` (similar to `argv`) to retrieve its name.

Suppose the file name is incorrect, so that `Open` cannot find the proper file and thus fails. `Ada.Text_IO` will raise an exception, generally `Ada.Text_IO.Name_Error`. At this point, control passes to the *exception handler* for `Name_Error`, if the programmer has coded one; otherwise, control passes to the run-time system and the program terminates.

Syntactically, exception handlers in Ada are associated with "frames," that is, with `begin-end` blocks. The structure of a `begin-end` block is

```
begin
  --sequence of statements
[exception
  --one or more exception handlers]
end
```

where the brackets indicate that the exception handler part is optional. `exception` is a reserved word.

Our example contains two exception handler sections. The one governing the file-open operation begins at line 105. In this simple program, if the file-open fails, there is no point in continuing execution, so we associate the handler with the main `begin-end` pair. If the file-open fails, control immediately passes here. There is only one handler here, for `Name_Error`. The handler code displays a message, then control passes out of the frame. In this case, since the frame is the main frame, the program terminates.

If the file can be opened successfully, the main `while` loop (lines 56-77) reads transactions from the file, storing them this time in the array `TodayActivity`. For example,

```
Gadget_IO.Get(File => InputData,
             Item =>
             TodayActivity(TransactionCount).Kind);
```

reads a gadget kind from the file, storing it in the `Kind` field of the transaction record at `TodayActivity(TransactionCount)`.

Now suppose that in a given transaction, the first token does not represent a valid gadget, or the monetary amount is ill-formed or out of range. This condition results in either `Ada.Text_IO.Date_Error` or `Constraint_Error` being raised. The invalid transaction should not be recorded, but the program should continue to process other transactions. We need to handle the exception locally, and therefore provide a `begin/end` frame (lines 59-76) with its own exception handler part (lines 65-75).

If either of the `Get` calls should raise an exception, the statement containing the call is abandoned, control immediately passes to this handler section, and the relevant handler (the appropriate `when` clause) is selected. The appropriate actions are taken, then control passes out of this frame, namely, just below the `end` at line 76. Now control—still in the `while` loop—passes back to the top of this loop to read the next transaction.

Language-provided exception-handling models are of two varieties *resumption* or *termination*. In a resumption model, control passes from a handler directly back to the failed statement. Ada uses the *termination* model. The designers observed that it is often fruitless and unreliable to *automatically* try to resume a failed statement; in any case, the programmer can provide for resumption using loops and local exception handlers, much as we have done here.

As another example of this, an Ada idiom commonly used for interactive input is:

```
loop
  begin
    Get(...);
    exit;
  exception
    when ...
    when ...
  end;
end loop;
```

Here the exception handler protects just the one `Get`. If the call succeeds, the `exit` statement passes control to just below the `end loop`. If the `Get` fails on an exception, the handlers recover, perhaps writing a message to the interactive user, after which control passes below the `end`, back around the loop for another attempt.

Completing our example, lines 81-88 loop through the transaction database and create the summary array, then lines 90-103 display the results. The statement

```
Money_Out.Put(Item => TodaySummary(WhichKind).Net,
              Pic => Ada.Text_IO.Editing.To_Picture
              ("$$_$$$_$$9.99CR"));
```

display the monetary quantity in edited form, using a parameter `Pic` of type `Ada.Text_IO.Editing.Picture` to provide the desired format. These picture strings are similar to those in COBOL and in spreadsheet programs. In this case we have indicated a floating dollar sign, commas to separate groups of digits, and CR to indicate a negative monetary value. Thus `-2065.00` displays as `$2,065.00CR`.

As in other systems used to program financial reports, Ada picture strings have many rules and much flexibility. Edited formats need not even be programmed statically. Because a picture is, syntactically, just a string, it need not be supplied as a literal, but can be constructed “on the fly” depending on other conditions, and passed to `Put` at the last moment.

This is the last example of the brief Ada programs designed to illustrate types and objects, the “inner syntax” of the language, and a number of the standard library packages. In the next section we show how programs are constructed from packages.

4.2 Packages: the Ada Encapsulation Mechanism

In this section we will illustrate the use of packages with some of our own. Before doing so, it is useful to examine a standard library package, namely the full interface of `Ada.Calendar`, taken verbatim from the Reference Manual, because it is an especially good example of careful design of appropriate operations. Also, many Ada packages—standard and application specific—follow the general structure of `Ada.Calendar`.

Listing 10

```
1 package Ada.Calendar is
2
3   type Time is private;
```

```

4
5 subtype Year_Number is Integer range 1901 .. 2099;
6 subtype Month_Number is Integer range 1 .. 12;
7 subtype Day_Number is Integer range 1 .. 31;
8 subtype Day_Duration is Duration range 0.0 .. 86_400.0;
9
10 function Clock return Time;
11
12 function Year (Date: Time) return Year_Number;
13 function Month (Date: Time) return Month_Number;
14 function Day (Date: Time) return Day_Number;
15 function Seconds(Date: Time) return Day_Duration;
16
17 procedure Split (Date : in Time;
18                 Year : out Year_Number;
19                 Month : out Month_Number;
20                 Day : out Day_Number;
21                 Seconds: out Day_Duration);
22
23 function Time_Of(Year : Year_Number;
24                 Month : Month_Number;
25                 Day : Day_Number;
26                 Seconds: Day_Duration := 0.0)
27   return Time;
28
29 function "+" (Left: Time; Right: Duration) return Time;
30 function "+" (Left: Duration; Right: Time) return Time;
31 function "-" (Left: Time; Right: Duration) return Time;
32 function "-" (Left: Time; Right: Time) return Duration;
33
34 function "<" (Left, Right: Time) return Boolean;
35 function "<=" (Left, Right: Time) return Boolean;
36 function ">" (Left, Right: Time) return Boolean;
37 function ">=" (Left, Right: Time) return Boolean;
38
39 Time_Error: exception;
40
41 private
42   ... -- not specified by the language
43 end Ada.Calendar;
```

We have seen lines 1-15 earlier and will not belabor them. Lines 17-20 define a procedure that accepts a `Time` value and returns, in four out parameters, the four components that the functions in lines 12-15 return separately. The four functions and this procedure are, in object-oriented design, generally called *selector* operations because they return selected information from an object—in this case a `Time` object—without modifying the original object.

Lines 23-27 provide a *constructor* function that goes the other way: Given the four component values, `Time_Of` produces a time. The `Seconds` parameter (line 26) is provided with a default value of 0.0. If a calling program does not supply an actual parameter for this formal parameter, the default will be used. If `T` is a `Time` variable, then

```
T := Ada.Calendar.Time_Of(7, 29, 1997);
```

returns a value corresponding to midnight at the start of July 29, 1997. On the other hand,

```
T := Ada.Calendar.Time_Of(6, 31, 1997);
```

is invalid because June has 30 days, and

```
T := Ada.Calendar.Time_Of(2, 29, 1997);
```

is invalid because 1997 is not a leap year. In both of these cases, `Ada.Calendar.Time_Error` (as defined in line 39) will be raised at the point of each call; it is up to the calling program to handle the exception. We will return shortly to package-defined exceptions.

Lines 29-32 define *infix operators* for `Time` values. Note that there is no operator to add two times—it is physically meaningless to add 2 o'clock to 3 o'clock. On the other hand, adding a `Duration` and a `Time` is meaningful—`Duration` represents *elapsed* time and 2 o'clock plus 15 minutes gives 2:15. Note that we need one operator for the `Time` value on the left, and another for the `Duration` value on the left.

Finally, we see (lines 34-37) comparison operators for `Time` values. `Time` is ordered, so comparing two times is meaningful.

This quick look at `Ada.Calendar` has introduced constructor and selector operations, and Ada's style for defining new infix operators. We will consider these issues in more detail below in introducing some of our own packages.

4.2.1 A Root Package for this Handbook

First we introduce our structure and naming convention. By analogy with the standard library structure in which all standard packages are descendants ("children" or "grandchildren") of a root package `Ada`, in this handbook all our packages are descendants of a root package `HB`:

Listing 11

```

1 package HB is
2   -- Root package for Handbook examples
3 end HB;
```

This package interface provides no resources; it exists purely to serve as the root. Because this interface is empty, there is no need for us to provide an implementation.

Ada does not require us to use a root package. We choose to do so in conformity with a growing trend in the Ada literature, in which the provider of a given collection of packages uses a root package to provide a distinctive top-level name for all packages in the collection. In this way the user of different collections can tell the origin of a given package; further, the likelihood of duplicated package names is minimized.

4.2.2 A Package for Rational-Number Arithmetic

Our first package example is `HB.Rationals`, which provides a type and a set of programmer-defined operations on rational numbers (“fractions”). The Ada standard provides for integer, fixed-point, and floating-point scalar arithmetic, and Annex G, the standard numerics annex, provides for complex numbers and arithmetic. On the other hand, Ada provides no predefined support for rationals, which have many applications.

A rational value has two components, the numerator and the denominator. Here is the package interface for `HB.Rationals`.

Listing 12

```

1 package HB.Rationals is
2
3   type Rational is private;
4
5   ZeroDenominator: exception;
6
7   function "/" (X, Y: Integer) return Rational;
8   -- constructor:
9   -- Pre :   X and Y are defined
10  -- Post:   returns a rational number
11  --   If Y > 0, returns X/Y
12  --   If Y < 0, returns (-X)/(-Y)
13  -- Raises: ZeroDenominator if Y = 0
14
15  function Numer (R : Rational) return Integer;
16  function Denom (R : Rational) return Positive;
17  -- selectors:
18  -- Pre: R is defined
19  -- Post: return numerator and denominator respectively
20
21  function "=" (R1, R2 : Rational) return Boolean;
22  function "<" (R1, R2 : Rational) return Boolean;
23  function "<=" (R1, R2 : Rational) return Boolean;
24  function ">" (R1, R2 : Rational) return Boolean;
25  function ">=" (R1, R2 : Rational) return Boolean;
```

```

26  -- comparators:
27  -- Pre : R1 and R2 are defined
28  -- Post: return the obvious Boolean results
29
30  function "+"(R: Rational) return Rational;
31  function "-"(R: Rational) return Rational;
32  function "ABS"(R: Rational) return Rational;
33  function "+"(R1, R2 : Rational) return Rational;
34  function "-"(R1, R2 : Rational) return Rational;
35  function "*" (R1, R2 : Rational) return Rational;
36  function "/"(R1, R2 : Rational) return Rational;
37  -- Pre : R, R1 and R2 are defined
38  -- Post: return the obvious arithmetic results
39
40  private
41
42  type Rational is record
43     Numerator : Integer := 0;
44     Denominator: Positive := 1;
45  end record;
46
47  end HB.Rationals;
```

Line 3 gives a *partial view* of a type `Rational`: the type is declared as `private` but its structure is not. We make the type `private` for encapsulation purposes: The structural details of a `private` type are not visible to client programs, and the only operations predefined on `private` types are assignment and equality/inequality tests.

Using a `private` type gives us the flexibility to choose—and later change—the internal structure without requiring any changes to client programs. Further, we can provide a complete set of operations on our type, with no worry that a client will be able to use any additional operations. This gives us control over the abstraction provided by the package.

We mention in passing that Ada also provides for `limited private` types, which have no predefined operations at all, not even assignment and equality test. That is, the programmer of a package providing a `limited private` type is assured that a client program can do *nothing* with objects of the type, except those operations explicitly provided in the package. This is useful in situations where, for example, it is necessary or desirable to prohibit clients from copying one object to another. The type `Ada.Text_IO.File_Type` is a predefined `limited private` type.

The *full view* of the `private` type `Rational` is given in the `private` part of the interface (lines 40-46). We see that a rational is represented as a record, that its numerator can be an arbitrary integer but that its denominator is positive, and that each rational quantity is initialized. Human readers of the package interface can obviously see this full view, but client programs

cannot. A client program therefore cannot refer directly to the numerator and denominator fields, but must use package-provided operations to do so.

Line 5 defines a package-specific exception `ZeroDenominator`; we shall see later how this is raised if a client tries to assign 0 to the denominator of a rational.

The declaration (line 7)

```
function "/" (X, Y: Integer) return Rational;
```

declares an *operator* `/`, which clients can use to construct a rational. This operator allows us to write, in client programs, statements like

```
R: Rational := 1/3;
```

Defining, in this fashion, additional meanings for the existing operator symbols is called *operator overloading*. Ada allows us to overload any of the existing operator symbols and prohibits us from defining any new operator symbols.

In this interface we have documented the various groups of operations with comments indicating preconditions and postconditions. These lie outside Ada but are increasingly commonly used in Ada and other languages to give concise and consistent descriptions of sets of operations. The precondition (line 9)

```
-- Pre : X and Y are defined
```

indicates that the behavior of this operation can be assured *only* if the client program refrains from passing it uninitialized variables. In Ada, variables are not initialized by default and there is no requirement that the programmer provide initialization expressions in declarations. Moreover, there is, in general, no assured compile-time or run-time check that variables are initialized. The precondition is therefore very important in Ada: As in most programming languages, initialization is a programmer responsibility.

The postcondition

```
-- Post: returns a rational number
-- If Y > 0, returns X/Y
-- If Y < 0, returns (-X)/(-Y)
-- Raises: ZeroDenominator if Y = 0
```

describes the behavior of the `/` operator in succinct terms. Lines 15-38 declare a full set of operations on rational quantities; their intent is obvious from their declarations and postconditions.

Line 21

```
function "=" (R1, R2 : Rational) return Boolean;
```

merits a bit of discussion. The equality-test operation is predefined in Ada for most types, including `private` ones. Here we *override* the predefined equality test because it may give incorrect results. Predefined equality compares its two operands bitwise; any disagreement yields a `False` result. But algebraically, $1/3 = 2/6 = 24/72$, etc. Ada's predefined equality would return `False` if applied to $1/3$ and $2/6$. We need our own equality test, so we provide it in the package interface and deliver it in the package body we will examine shortly.

We defined all our operators as *functions* because, in fact, mathematically an operator *is* just a function with certain restrictions on its arguments. Indeed, the standard defines the operators on the predefined types using just this function syntax.

4.2.3 A Child Package for Rational Input and Output

To provide clients with an easy way to input and display rationals, we provide a *child package* `HB.Rationals.IO`, in which `Get` and `Put` operations are provided for the standard and named input and output files. There is no requirement that this package be a child of `Rationals`; we simply chose to make it so.

Listing 13

```
1 with Ada.Text_IO;
2 use Ada.Text_IO;
3 package HB.Rationals.IO IS
4
5   procedure Get (Item : out Rational);
6   procedure Get
7     (File: in File_Type; Item : out Rational);
8
9   procedure Put (Item : in Rational);
10  procedure Put
11    (File: in File_Type; Item : in Rational);
12
13 end HB.Rationals.IO;
```

In this package interface, note how the various procedure parameters are specified. In writing Ada procedures, we choose a *mode* for each parameter.

That is, we declare each parameter as `in`, `out`, or `in out`; in the absence of a mode specifier, `in` is specified by default.

Ada separates the intended use of parameters from the mechanism by which they are passed. An `in` parameter is treated as a constant, or read-only value, within the body of the procedure. Since no code within the procedure can attempt to modify an `in` parameter—this is, of course, checked by the compiler—it is of no concern whether the parameter is passed by copy or by reference.

In fact, the Ada standard requires that scalar parameters be passed by value; an `in out` scalar is copied from actual to formal at the start of a procedure execution, and the possibly new value is copied back from formal to actual at procedure completion. On the other hand, composite (record and array) parameters can be legally passed either by copy or by reference; most compilers pass them by reference.

Parameters or *functions* must be `in`; programmers generally omit the explicit `in` from function parameter declarations, and we have omitted it in our packages here.

4.2.4 Using the Rationals Package

Before we investigate the implementations of our rational-number packages, let us examine a typical interactive client program. A session with this program might be

```
A = 1/3
B = -2/4
Enter rational number C > 2/5
Enter rational number D > 3/4

E = A + B is -2/12
A + E * B is 60/144
B's numerator is -2
```

Results like $-2/12$ and $60/144$ make it obvious that nothing in this package is reducing rationals to lowest terms. A good rationals package should do so; we have chosen to omit it in the interest of brevity. It would not be difficult to add to our `"/"` operator.

Listing 14

```
1 with Ada.Text_IO, Ada.Integer_Text_IO;
2 with HB.Rationals, HB.Rationals.IO;
3 use HB.Rationals;
4 procedure Show_Package is
```

```
5
6   A, B, C, D, E: Rational;
7
8 begin
9
10  A := 1/3;
11  B := 2/(-4);
12  Ada.Text_IO.Put(Item => "A = ");
13  HB.Rationals.IO.Put(Item => A);
14  Ada.Text_IO.New_Line;
15  Ada.Text_IO.Put(Item => "B = ");
16  HB.Rationals.IO.Put(Item => B);
17  Ada.Text_IO.New_Line;
18
19  Ada.Text_IO.Put(Item => "Enter rational number C > ");
20  HB.Rationals.IO.Get(Item => C);
21  Ada.Text_IO.Put(Item => "Enter rational number D > ");
22  HB.Rationals.IO.Get(Item => D);
23  Ada.Text_IO.New_Line;
24
25  E := A + B;
26  Ada.Text_IO.Put(Item => "E = A + B is ");
27  HB.Rationals.IO.Put(Item => E);
28  Ada.Text_IO.New_Line;
29
30  Ada.Text_IO.Put(Item => "A + E * B is ");
31  HB.Rationals.IO.Put(Item => A + E * B);
32  Ada.Text_IO.New_Line;
33
34  Ada.Text_IO.Put(Item => "B's numerator is ");
35  Ada.Integer_Text_IO.Put
36    (Item => Numer(B), Width => 1);
37  Ada.Text_IO.New_Line;
38
39 exception
40 when ZeroDenominator =>
41   Ada.Text_IO.Put_Line(Item =>
42     "Zero not allowed in denominator; bye-bye.");
43 end Show_Package;
```

Lines 1-3 provide the expected context clauses, and a `use` for `Rationals`. The `use` allows us to use the rational operators in infix form; without a `use`, we would have to say, e.g., in line 10,

```
A := Rationals."/"(1,3);
```

which is unpleasant. The rest of the program illustrates some simple rational arithmetic; note line 25, for example, which looks like an ordinary arithmetic assignment, but in fact all the operations are rational ones.

In analyzing `E := A + B`, the compiler can determine that the `+` is intended to be that for rationals, because its two operands (`A` and `B`) are of type `Rational`, as is the return type (`E`). Similarly in line 31, the expression

$A + E * B$ is a rational one, because all three operands are, as is the desired `Item` sent to `HB.Rationals.IO.Put`. Determining which of several like-named operators is called *overload resolution*; the compiler can resolve the overloads if and only if it has enough information to do so.

If `X` is `Integer` and `Y` is `Rational`, the expression `X + Y` will result in a compilation error. This is simply because there is no `+` defined for this combination of operands; we could add one to the package if we chose to do so; its line in the interface would be

```
function "+"(X: Integer; R: Rational) return Rational;
```

Indeed, nothing prevents us from writing a package providing other mixed operations, e.g.,

```
function "+"(X: Integer; F: Float) return Float;
```

This observation reveals that while mixed operations are not *predefined* in Ada, we could add these to the language via a suitable (large) collection of overloaded operators. Doing so would not be desirable, though: The compiler would have no trouble resolving all the overloads in a program that used these mixed expressions, but the human reader would very quickly get confused. *Operator overloading is very convenient when used in moderation.*

4.2.5 Implementing the Rationals and Rationals.IO Packages

Now let us examine the implementation of our rationals package. In Ada the implementation is called a `package body`, as line 1 shows.

Listing 15

```
1 package body HB.Rationals is
2
3   function "/" (X, Y : Integer) return Rational is
4     begin
5       if Y = 0 then
6         raise ZeroDenominator;
7       elsif X = 0 then
8         return (Numerator => 0, Denominator => 1);
9       elsif Y > 0 then
10        return (Numerator => X, Denominator => Y);
11      else
12        return (Numerator => -X, Denominator => -Y);
13      end if;
14    end "/";
15
```

```
16 function Numer (R : Rational) return Integer is
17 begin
18   return R.Numerator;
19 end Numer;
20
21 function Denom (R : Rational) return Positive is
22 begin
23   return R.Denominator;
24 end Denom;
25
26 function "=" (R1, R2 : Rational) return Boolean is
27 begin
28   return Numer(R1)*Denom(R2) = Numer(R2)*Denom(R1);
29 end "=";
30
31 function "<" (R1, R2 : Rational) return Boolean is
32 begin
33   return Numer(R1)*Denom(R2) < Numer(R2)*Denom(R1);
34 end "<";
35
36 function ">" (R1, R2 : Rational) return Boolean is
37 begin
38   return Numer(R1)*Denom(R2) > Numer(R2)*Denom(R1);
39 end ">";
40
41 function "<=" (R1, R2 : Rational) return Boolean is
42 begin
43   return Numer(R1)*Denom(R2) <= Numer(R2)*Denom(R1);
44 end "<=";
45
46 function ">=" (R1, R2 : Rational) return Boolean is
47 begin
48   return Numer(R1)*Denom(R2) >= Numer(R2)*Denom(R1);
49 end ">=";
50
51 function "+"(R : Rational) return Rational is
52 begin
53   return R;
54 end "+";
55
56 function "-"(R : Rational) return Rational is
57 begin
58   return (-Numer(R)) / Denom(R);
59 end "-";
60
61 function "abs"(R : Rational) return Rational is
62 begin
63   return (abs Numer(R)) / Denom(R);
64 end "abs";
65
66 function "+"(R1, R2 : Rational) return Rational is
67 begin
68   return (Numerator =>
69     Numer(R1)*Denom(R2) + Numer(R2)*Denom(R1),
70     Denominator => Denom(R1)*Denom(R2));
71 end "+";
72
```

```

73 function "-"(R1, R2 : Rational) return Rational is
74 begin
75     return (Numerator =>
76             Numer(R1)*Denom(R2) - Numer(R2)*Denom(R1),
77             Denominator => Denom(R1)*Denom(R2));
78 end "-";
79
80 function "*" (R1, R2 : Rational) return Rational is
81 begin
82     return (Numerator => Numer(R1)*Numer(R2),
83             Denominator => Denom(R1)*Denom(R2));
84 end "*";
85
86 function "/"(R1, R2 : Rational) return Rational is
87 begin
88     return (Denominator => Numer(R1)*Numer(R2),
89             Numerator => Denom(R1)*Denom(R2));
90 end "/";
91
92 end HB.Rationals;

```

Lines 3-14 implement the "/" operator. This is simply a matter of checking the various cases. In the first case (line 6), an attempt to store 0 in the denominator, we

```
raise ZeroDenominator;
```

Since there is no exception handler in this function, the exception is *propagated* to the point where the client called the function. If the client provides a nearby handler, control is passed to it; otherwise, control passes to the next higher level in the call chain, and so forth until either the exception is handled somewhere in the application, or it passes beyond the main program to the run-time system.

It is quite common in Ada to declare exceptions in package interfaces, then raise them in package implementations. This sort of exception responds to a client program's violation of the abstraction provided by the package—in this case, the mathematical prohibition against 0 in the denominator of a rational number. The package cannot be expected to “clean up” a violation like this: The client caused the problem and the client is responsible for solving it. Ada's exception-propagation rules encourage programmers to design such that responsibility for anomalies and violations is passed to, and handled by, just the right layer of the overall program.

The other cases in "/" are straightforward. We note (line 10) the return statement

```
return (Numerator => X, Denominator => Y);
```

in which the parenthesized expression is an *aggregate* expression, returning a record to the calling program, with its fields set as indicated.

This statement illustrates that in Ada there is no restriction on the return type of a function; specifically, a function may return a record or even an array, as the abstraction requires. This is a significant advantage over languages that require pointers to data structures to be explicitly passed to, and returned from, functions. Since in Ada composite types are generally passed by reference, pointers are indeed involved, but the compiler, not the programmer, is responsible for them.

Lines 16-24 implement Numer and Denom; lines 26-29 implement our equality test. The statement

```
return Numer(R1)*Denom(R2) = Numer(R2)*Denom(R1);
```

returns True or False depending upon whether the two cross-products are equal. Recalling that $2/6 = 3/9$, algebraically, will confirm the correctness of the boolean cross-product expression.

We note that it is neither necessary nor legal to overload "/="; overloading "=" automatically overloads "/=".

Lines 31-49 give the bodies of the other four comparison operators; lines 51-64 implement the three monadic (unary) arithmetic operators, and lines 66-90 implement the dyadic (binary) arithmetic operators. These bodies are all straightforward.

We complete our rationals system by showing the implementation of HB.Rationals.IO; these four procedure bodies are straightforward. We note only that the Get for standard input is implemented by a simple call (line 19) to the other Get for a named file, using Ada.Text_IO.Standard_Input as the file name.

Listing 16

```

1 with Ada.Text_IO, Ada.Integer_Text_IO;
2 use  Ada.Text_IO, Ada.Integer_Text_IO;
3 package body HB.Rationals.IO is
4
5     procedure Get
6         (File: in File_Type; Item : out Rational) is
7         N: Integer;
8         D: Integer;
9         Dummy: Character; -- dummy character to hold the "/"
10    begin
11        Get(File => File, Item => N);
12        Get (File => File, Item => Dummy);

```

```

13   Get(File => File, Item => D);
14   Item := N/D;
15   end Get;
16
17   procedure Get (Item : out Rational) is
18   begin
19     Get(File => Standard_Input, Item => Item);
20   end Get;
21
22   procedure Put
23     (File: in File_Type; Item : in Rational) is
24   begin
25     Put(File => File, Item => Numer(Item), Width => 1);
26     Put(File => File, Item => '/');
27     Put(File => File, Item => Denom(Item), Width => 1);
28   end Put;
29
30   procedure Put (Item : in Rational) is
31   begin
32     Put(File => Standard_Output, Item => Item);
33   end Put;
34
35 end HB.Rationals.IO;

```

4.2.6 Unconstrained array types; Vectors

Our next example gives a basic package for manipulating mathematical vectors.

Listing 17

```

1 package HB.Vectors is
2
3   type Vector is array(Integer range <>) of Float;
4
5   Bounds_Error : exception;
6
7   function "*" (K : Float; Right : Vector) return Vector;
8   -- Pre: K and Right are defined
9   -- Post: scales the vector by the scalar
10  --   Result(i) := K * Right(i)
11
12  function "*" (Left, Right : Vector) return Float;
13  -- Pre: Left and Right are defined
14  -- Post: returns the inner product of Left and Right
15  -- Raises: Bounds_Error if Left and Right
16  --   have different numbers of elements
17
18  function "+" (Left, Right : Vector) return Vector;
19  -- Pre: Left and Right are defined
20  -- Post: returns the sum of Left and Right
21  --   Result(i) := Left(i) + Right(i)
22  --   Result has Left's bounds
23  -- Raises: Bounds_Error if Left and Right
24  --   have different numbers of elements
25

```

```

26 end HB.Vectors;

```

In line 3, a vector is defined as an array of `Float` values. Recall from our earlier gadgets example that the expression `Integer range <>` indicates an unconstrained array type. This means that each vector's bounds will be an `Integer` subrange and that these bounds are set not in the type declaration but in the object declaration. For example,

```

V: Vector (1..100);
W: Vector (25..40);
Z: Vector (-5..10);

```

are all legal object declarations. Each object is now said to be *constrained*; its bounds (and the space allocated for it) are set for its lifetime.

Under Ada's rules two such constrained objects are assignment-compatible if they have the same *length*, that is, the same number of elements; thus the bounds do not have to be equal. Since `W` and `Z` have the same length (16 elements),

```

W := Z;
Z := W;

```

are both valid assignments. Moreover, we can work with *slices*, that is, subarrays. For example,

```

V (5..10) := W (32..37);

```

is valid because both lengths are 6. The slice bounds don't have to be static; they can be computed as the program executes, so slicing turns out to give us considerable expressiveness in array-handling algorithms.

Ada's predefined `String` type is, in fact, an unconstrained array type:

```

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

```

so slicing operations operate as substring operations. The standard library packages `Ada.Strings`, `Ada.Strings.Bounded` and `Ada.Strings.Unbounded` provide full sets of operations on bounded and unbounded strings of varying length, built on top of the basic support for strings as unconstrained arrays. For brevity's sake, we will not go into more detail on the string operations.

In `HB.Vectors`, we provide an exception `Bounds_Error` and several overloaded operators. The *sum* `+` of two equal-length vectors is a vector of that same length, in which each element is the sum of the corresponding

elements in the operand vectors. The *inner product* "*" of two equal-length vectors with `Float` elements is a `Float` value, the accumulated sum of all the pairwise products of the operand vectors (a pairwise product is the product of the first elements of the two vectors, or of the second elements, and so on).

As the postconditions show, this exception is raised if the client violates the vector abstraction, that is, attempts to add or multiply vectors of different lengths. If we have

```
V: Vector (1..10);
W: Vector (11..20);
X: Vector (6..11);
Z: Vector (26..35);
```

then

```
W := V + Z;
```

is valid but

```
W := X + Z;
```

is not and the exception will be raised.

This, and the vector arithmetic operations, are shown in the package implementation.

Listing 18

```
1 package body HB.Vectors is
2
3   function "*"
4     (K : Float; Right : Vector) return Vector is
5     Result: Vector(Right'Range);
6   begin
7     for I in Right'Range loop
8       Result(I) := K * Right(I);
9     end loop;
10    return Result;
11  end "*";
12
13  function "+" (Left, Right : Vector) return Vector is
14    Result : Vector(Left'Range);
15  begin
16    if Left'Length /= Right'Length then
17      raise Bounds_Error;
18    else
19      for I in Left'range loop
20        Result(I) :=
```

```
21      Left(I) + Right(I - Left'First + Right'First);
22    end loop;
23    return Result;
24  end if;
25  end "+";
26
27  function "*" (Left, Right : Vector) return Float is
28    Sum : Float := 0.0;
29  begin
30    if Left'Length /= Right'Length then
31      raise Bounds_Error;
32    else
33      for I in Left'Range loop
34        Sum := Sum +
35          Left(I) * Right(I - Left'First + Right'First);
36      end loop;
37      return Sum;
38    end if;
39  end "*";
40
41 end HB.Vectors;
```

The implementation makes heavy use of unconstrained array attributes, as is seen in the first function, which *scales* the vector by multiplying each of its elements by the same `Float` value. Line 5

```
Result: Vector(Right'Range);
```

creates a temporary result vector `Result` whose bounds are given by `Right'Range`. Equivalently we could say

```
Result: Vector(Right'First..Right'Last);
```

`Right'First` is the value of the first *index* or *subscript* of `Right`, not the first *element* value. This line, then, “sizes” the result vector to be the same as the input vector. Similarly, the loop in lines 7-9 is also bounded by `Right'Range`, and thus multiplies each vector element by the `Float` value `K`.

The operator “+” (lines 13-25) first “sizes” a result vector (line 14) as above, choosing `Left`’s bounds as the ones to use. Because the operation makes mathematical sense only if the two vectors have the same length, we check this in lines 16-17, raising the exception if the lengths disagree.

If the vectors are conformable (the lengths agree), we compute the sum by looping through both vectors. Since the *bounds* will not, in general, be the same, we loop using `Left`’s bounds, and adjust the index into `Right` accordingly, selecting

```
Left(I) + Right(I - Left'First + Right'First)
```

The inner product "*" (lines 27-39) is similar to the sum, but this time we multiply the elements pairwise, accumulating them in a scalar `Result`.

As this package has shown, unconstrained arrays and their attributes provide a concise way to operate on arrays whose bounds are unknown at compilation time. This gives us great expressive power to write general-purpose mathematical algorithms that can operate on vectors and matrices of any size, without knowing their specific bounds.

4.2.7 Generic Packages; Matrices

`HB.Vectors` shows the power of unconstrained array types, but only in a single dimension. Also, the vector element type is specified as `Float`, which makes the package insufficiently general. In the next example, we extend `HB.Vectors` in two ways: We illustrate multi-dimensional unconstrained array types and we show a *generic* or *template* package for vector and matrix operations that allows instantiation for an arbitrary element type. Here is the interface for `HB.Matrices_Generic`.

Listing 19

```

1 generic
2
3   type  ElementType is private;
4   Zero: ElementType;
5   Unity: ElementType;
6
7   with function
8     "+"(Left, Right: ElementType) return ElementType;
9   with function
10    "*" (Left, Right: ElementType) return ElementType;
11
12 package HB.Matrices_Generic is
13
14   type Vector is array(Integer range <>) of ElementType;
15   type Matrix is
16     array (Integer range <>, Integer range <>)
17       of ElementType;
18
19   Bounds_Error: exception;
20
21   function "*"
22     (K: ElementType; Right: Vector) return Vector;
23   function "*" (Left, Right: Vector) return ElementType;
24   function "+" (Left, Right: Vector) return Vector;
25
26   function "*"
27     (K: ElementType; Right: Matrix) return Matrix;
28   function Transpose (Left: Matrix) return Matrix;
```

```

29   function "+" (Left, Right: Matrix) return Matrix;
30   function "*" (Left, Right: Matrix) return Matrix;
31
32   function "*"
33     (Left: Vector; Right: Matrix) return Vector;
34   function "*"
35     (Left: Matrix; Right: Vector) return Vector;
36
37 end HB.Matrices_Generic;
```

Lines 1-11 give the generic part of a package template. Syntactically, this generic part precedes an otherwise ordinary-looking package interface. Lines 3-10 give the *generic formal parameters*. Just as the caller of an ordinary procedure or function must supply actual parameters to match that subprogram's formal parameters, so a client of a generic template must supply actual parameters for each of the generic formals.

Line 3 declares a generic formal *type* parameter. Syntactically, this looks similar to an ordinary type declaration, but it is not; it declares a *type parameter*. Ada provides for different categories of generic formal parameters, such as integer types, float types, enumeration types, constrained and unconstrained array types, and so on. Each parameter category has its own bit of syntax for describing the category.

In our case, the parameter is a *private* type parameter. This does not mean that the actual *must* be *private*, but rather that the actual may be *any* type, *including* a *private* one. This excludes from consideration only *limited* and *tagged* types (which we have not introduced yet), and thus allows great flexibility in the kinds of types that can serve as matrix elements. We could instantiate with predefined scalar types like `Integer`, or `Float`, or `Boolean` (there are many applications for boolean matrices), or even programmer-defined private composite types like `Rational`.

To understand the remaining formal parameters, first recall from `HB.Vectors` that the inner product operation first initializes a `Sum` variable to `0.0`, then, as it loops through the vectors, carries out `Float addition` and `multiplication` on the vector elements and `Sum`. In the generic package, the element type is no longer `Float`, but an arbitrary type to be supplied by the client. This means that `0.0` no longer is a correct zero value. In fact, each possible element type has its own zero: `0` (`Integer`), `0/1` (`Rational`), `False` (`Boolean`), and so on. In return for the ability to instantiate a matrix package for any type, the client gains the responsibility to tell us (line 4) what the zero is for that type. Similarly, we ask the client to supply a unity value (line 5) for the type.

Furthermore, because any programmer-defined type can be used as an element, it is not self-evident that addition and multiplication are defined for this type. We therefore list formal parameters for these operations (lines 7-10) and the client must supply the corresponding actuals. To instantiate for `Float`, one would write

```
package Float_Matrices is new HB.Matrices_Generic
(ElementType => Float,
 Zero       => 0.0,
 Unity     => 1.0,
 "+"      => "+",
 "*"      => "*");
```

and to instantiate for `Boolean` (say, for a graph-theory application that represents graphs as `Boolean` matrices) one would write

```
package Boolean_Matrices is new HB.Matrices_Generic
(ElementType => Boolean,
 Zero       => False,
 Unity     => True,
 "+"      => "or",
 "*"      => "and");
```

Returning to the package interface, lines 14-17 declare two types, `Vector` and `Matrix`. The second declaration shows a matrix as a two-dimensional unconstrained array. This lets us declare objects like

```
M: Matrix(1..3, 25..100);
```

which we view as having three rows and 76 columns.

The `Matrix` type raises two related issues. First, a two-dimensional (2-d) array is *not* a 1-d array of 1-d arrays (as it would be in Pascal or C, for example); it is a different type entirely. Second, the Ada standard does *not* specify a storage structure for multidimensional arrays; an implementer could choose row-major, or column-major, or another scheme entirely (such as a tree structure). This is an advantage, because the storage structure can be tailored to the underlying computer's memory structure. Generally, a programmer need not be concerned about the storage structure.

On the other hand, a program using the Ada 95 interface to Fortran (as described in Annex B of the standard), in order, for example, to call pre-existing Fortran subprograms, we could attach to a matrix type a compiler directive—`pragma` in Ada—of the form

```
pragma Convention(Fortran, Matrix);
```

indicating that all objects of the Ada type `Matrix` must be stored to match the column-major convention used by the related Fortran compiler.

Lines 19-24 are essentially repeated from `HB.Vectors`. Lines 26-27 declare a scaling operation for matrices; line 28 declares a matrix transposition operation in which rows are interchanged with columns, line 29 gives a matrix sum (similar to the vector sum) and line 30 provides the matrix product. In this last operation, given matrices `Left`, `Right`, and `Result`, for a given row `R` and column `C` of `Result`, `Result(R,C)` is the vector inner product of row `R` of `Left` and column `C` of `Right`. The conformability condition is that `Left` must have as many columns as `Right` has rows; `Result` has the number of rows of `Left` and the number of columns of `Right`.

Finally, lines 32-35 provide “mixed” vector/matrix operations. In the first “*”, `Left` is a row vector; multiplying it by the matrix `Right` produces another vector. This is similar to a matrix product operation in which `Left` has only one row. Similarly, the second “*” operation provides for multiplying a matrix `Left` by a vector `Right`.

A comment on package design: That we choose to export *two* types from this package shows clearly that the Ada package is an encapsulation mechanism which gives us flexibility to export one or more types—or, indeed, not to export any types at all, as in `HB.Rationals.IO`. This is a clearly different style from “class”-oriented languages in which the type and the encapsulation mechanisms are the same. Naturally, there is debate about the superiority of one style over another; however, neither side can prove its case because these are just different syntactic mechanisms—matters of preference, really—for reaching similar program-design goals.

4.2.8 Using the Generic Matrix Package

Before examining the implementation of `HB.Matrices_Generic`, let's see how it might be used. `Show_Matrices` is a sample client in which the matrix element type is `Rational`.

Listing 20

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with HB.Rationals; use HB.Rationals;
3 with HB.Rationals.IO; use HB.Rationals.IO;
4 with HB.Matrices_Generic;
5 procedure Show_Matrices is
6
7   package Rational_Matrices is new HB.Matrices_Generic
8     (ElementType => Rational,
```

```

9     Zero      => 0/1,
10    Unity     => 1/1,
11    "+" => HB.Rationals."+",
12    "*" => HB.Rationals.*";
13    use Rational_Matrices;
14
15    procedure Put(Item: in Vector) is
16    begin
17        for Element in Item'Range loop
18            Put(Item(Element));
19            Put(" ");
20        end loop;
21        New_Line(Spacing => 2);
22    end Put;
23
24    procedure Put(Item: in Matrix) is
25    begin
26        for Row in Item'Range(1) loop
27            for Col in Item'Range(2) loop
28                Put(Item(Row, Col));
29                Put(" ");
30            end loop;
31            New_Line;
32        end loop;
33        New_Line;
34    end Put;
35
36    V: Vector(1..3) := (1/3, 2/3, 0/1);
37    M: Matrix(1..3, 1..4) := ((0/1, 1/3, 1/2, 1/4),
38                            (1/1, 1/2, 1/2, 0/1),
39                            (1/4, 2/3, 1/4, 1/5));
40
41    begin
42
43        Put_Line("V ="); Put(V);
44        Put_Line("M ="); Put(M);
45        Put_Line("1/2 * M ="); Put(1/2 * M);
46        Put_Line("V * M ="); Put(V * M);
47
48        begin
49            Put_Line("M * V ="); Put(M * V);
50            Put_Line("Exception should have been raised");
51        exception
52            when Bounds_Error =>
53                Put_Line("Exception properly raised");
54        end;
55
56    end Show_Matrices;

```

This program is in “terse” style, with `use` clauses supplied and named parameters used only where extra clarity is needed. Lines 7-12 create an instance `Rational_Matrices` of the generic template. In line 11, the association

```
"+" => HB.Rationals."+"
```

is written verbosely, just for clarity’s sake. Indeed, the instantiation could have been written (tersely but more opaquely) as

```

package Rational_Matrices is
    new HB.Matrices_Generic (Rational, 0/1, 1/1, "+",
    "*");

```

The instance is now a use-able package, so we can supply a `use` clause (line 13). This instance gives us the types `Rational_Matrices.Vector` and `Rational_Matrices.Matrix`; the `use` lets us refer to these just as `Vector` and `Matrix`.

Lines 15-22 is a procedure to display the elements of a vector. There is nothing new here except (line 21) the use of the `Spacing` parameter to `New_Line`, which has the obvious meaning of “move the cursor to the next line, then move it again.”

Lines 24-34 show a procedure to display a matrix, row by row. Note in line 26 the attribute `Item'Range(1)`, which means “the range of `Item`’s first dimension, and in line 27 the attribute `Item'Range(2)`, which means “the range of `Item`’s second dimension. Here again we see the power of attributes to work with arrays whose bounds are unknown at compilation time.

Lines 36-39 declare a vector and matrix, both initialized with aggregates of rational quantities. The 2-d aggregate initializes `M` row by row. Lines 43-46 display the vector and the matrix, followed by the matrix scaled by `1/2`, then by the product of the vector and the matrix.

Lines 48-54 are used to illustrate a frame style that is commonly used in writing a program designed to test a package. The product `M*V` is mathematically impossible, because `M` has 4 columns but there column vector `V` has only 3 columns. `Bounds_Error` will be raised to signal this abstraction violation, if and only if the product operation is correctly implemented in the package.

The actual output of this program should be

```

V =
1/3  2/3  0/1

M =
0/1  1/3  1/2  1/4
1/1  1/2  1/2  0/1
1/4  2/3  1/4  1/5

1/2 * M =
0/2  1/6  1/4  1/8

```

```

1/2  1/4  1/4  0/2
1/8  2/6  1/8  1/10

```

```

V * M =
24/36  72/162  72/144  15/180

```

```

M * V =
Exception properly raised

```

4.2.9 Implementing the Generic Matrix Package

From our exploration of `HB.Vectors` and `Show_Matrices`, the package implementation is easy to understand.

Listing 21

```

1 package body HB.Matrices_Generic is
2
3   function "*"
4     (K : ElementType; Right : Vector) return Vector is
5     Result: Vector(Right'Range);
6   begin
7     for I in Right'Range loop
8       Result(I) := K * Right(I);
9     end loop;
10    return Result;
11  end "*";
12
13  function "+" (Left, Right : Vector) return Vector is
14  Result : Vector(Left'Range);
15  begin
16    if Left'Length /= Right'Length then
17      raise Bounds_Error;
18    else
19      for I in Left'Range loop
20        Result(I) :=
21          Left(I) + Right(I - Left'First + Right'First);
22      end loop;
23      return Result;
24    end if;
25  end "+";
26
27  function "*"
28    (Left, Right : Vector) return ElementType is
29  Sum : ElementType := Zero;
30  begin
31    if Left'Length /= Right'Length then
32      raise Bounds_Error;
33    else
34      for I in Left'Range loop
35        Sum := Sum +
36          Left(I) * Right(I - Left'First + Right'First);
37      end loop;
38      return Sum;
39    end if;

```

```

40  end "*";
41
42  function "*"
43    (K : ElementType; Right : Matrix) return Matrix is
44  Result: Matrix(Right'Range(1), Right'Range(2));
45  begin
46    for I in Right'Range(1) loop
47      for J in Right'Range(2) loop
48        Result(I,J) := K * Right(I,J);
49      end loop;
50    end loop;
51    return Result;
52  end "*";
53
54  function Transpose (Left: Matrix) return Matrix is
55  Result: Matrix(Left'Range(2), Left'Range(1));
56  begin
57    for I in Result'Range(1) loop
58      for J in Result'Range(2) loop
59        Result(I,J) := Left(J,I);
60      end loop;
61    end loop;
62    return Result;
63  end Transpose;
64
65  function "+" (Left, Right : Matrix) return Matrix is
66  Result : Matrix(Left'Range(1), Left'Range(2));
67  begin
68    if Left'Length(1) /= Right'Length(1) or
69       Left'Length(2) /= Right'Length(2) then
70      raise Bounds_Error;
71    else
72      for I in Left'Range(1) loop
73        for J in Left'Range(2) loop
74          Result(I, J) := Left(I, J) +
75            Right(I - Left'First(1) + Right'First(1),
76              J - Left'First(2) + Right'First(2));
77        end loop;
78      end loop;
79      return Result;
80    end if;
81  end "+";
82
83  function "*" (Left, Right : Matrix) return Matrix is
84  Result: Matrix(Left'Range(1), Right'Range(2)) :=
85    (others => (others => Zero));
86  begin
87    if Left'Length(2) /= Right'Length(1) then
88      raise Bounds_Error;
89    else
90      for RowL in Left'Range(1) loop
91        for ColR in Right'Range(2) loop
92          for ColL in Left'Range(2) loop
93            Result(RowL, ColR) := Result(RowL, ColR) +
94              Left(RowL, ColL) *
95              Right((ColL-Left'First(2))+Right'First(1),
96                ColR);

```

```

97     end loop;
98     end loop;
99     end loop;
100    end if;
101    return Result;
102  end "*";
103
104  function "*"
105    (Left: Vector; Right: Matrix) return Vector is
106    Result: Vector(Right'Range(2)) := (others => Zero);
107  begin
108    if Left'Length /= Right'Length(1) then
109      raise Bounds_Error;
110    else
111      for C in Right'Range(2) loop
112        for R in Right'Range(1) loop
113          Result(C) := Result(C) + Right(R, C) *
114            Left(Left'First + (R-Right'First(1)));
115        end loop;
116      end loop;
117    end if;
118    return Result;
119  end "*";
120
121  function "*"
122    (Left: Matrix; Right: Vector) return Vector is
123    Result: Vector(Left'Range(1)) := (others => Zero);
124  begin
125    if Left'Length(2) /= Right'Length then
126      raise Bounds_Error;
127    else
128      for R in Left'Range(1) loop
129        for C in Left'Range(2) loop
130          Result(R) := Result(R) + Left(R, C) *
131            Right(Right'First + (C-Left'First(2)));
132        end loop;
133      end loop;
134    end if;
135    return Result;
136  end "*";
137
138 end HB.Matrices_Generic;

```

We will not belabor the algorithms embodied in the various operations; these are well-known and straightforward algorithms from matrix algebra. From the Ada perspective, we point out the declaration (lines 84-85)

```

Result: Matrix(Left'Range(1), Right'Range(2)) :=
  (others => (others => Zero));

```

which “sizes” a result matrix in terms of the `Left` and `Right` matrix operands, and further initializes all elements of this matrix to `Zero`, the parameter representing the zero of the element type. This aggregate initialization is very concise, and can be implemented very efficiently

because the programmer leaves it up to the compiler just *how* all the elements will be initialized at execution time.

Finally, lines 87-88 give the conformability condition for matrix multiplication: `Left` must have as many columns as `Right` has rows.

4.3 Type Extension, Inheritance, and Polymorphism

Even in its original 1983-standard form, Ada contained most of the important ingredients for object-oriented software development.

First, packages, together with `private` types, provide a strong *encapsulation* mechanism that gives the programmer tight control over the visibility, state, and behavior of objects.

Further, two kinds of *polymorphism* are provided: The `generic` package facility gives the ability to parametrize packages with respect to types and operations, and overloading allows one to use the same name for a number of different operations with different parameter profiles.

Finally, *inheritance* is provided through *type derivation*. In Ada 83 one can write, for example,

```

package Credit_Cards is
  type Credit_Card is private;
  procedure P1 (X: in Credit_Card; ...);
  procedure P2 (X: in out Credit_Card; ...);
  ...
private
  type Credit_Card is...
end Credit_Cards;

```

and then *derive* new types such as

```

type Personal_Card is new Credit_Card;
type Business_Card is new Credit_Card;

```

These three types form a type hierarchy. `Personal_Card` and `Business_Card` inherit the structural details of `Credit_Card` and also the operations `P1` and `P2`. The three types are not compatible with each other but can be explicitly converted, so that if we have

```

C: Credit_Card;
P: Personal_Card;
B: Business_Card;

```

then

```

C := B;

```

```
P := B;
B := C;
```

are all disallowed, but

```
C := Credit_Card(B);
P := Personal_Card(B);
B := Business_Card(P);
```

are all valid. Furthermore,

```
P1(C);
P1(P);
P1(B);
```

are all valid and the compiler can determine which one to dispatch for each object.

The common characteristic in generics, overloading, and Ada 83 type derivation is that given an object and a set of similarly-named operations, the operation to be executed can be determined essentially at compilation time.

Why, then, did the Ada 83 designers not go the rest of the distance and provide for the kind of inheritance and run-time polymorphism we have come to expect for object-oriented programming (OOP)? There is considerable folklore about the discussions at the time, but not much available written record on this issue. It is a matter of fact, though, that Ichbiah and his team were well aware of, and considered carefully, the costs and benefits of these features. The Ada 83 Rationale says just this on the subject (in the introduction to Chapter 9):

Facilities for modularization have appeared in many languages. Some of them—such as Simula, Clu, and Alphard—provide dynamic facilities which may entail large run-time overhead. The facility provided in Ada is more static—in the spirit of previous solutions offered in Lis, Euclid, Mesa, and Modula. At the same time it retains the best aspects of solutions in earlier languages such as Fortran and Jovial.

We conclude that given the slow speed and memory constraints of the computers of that period, the main concern was about run-time costs. The Ada 83 designers—who did most of their work in the late 1970s—provided the “hooks” for full OOP but left it to the second generation of designers—who produced Ada 95 beginning in 1988, when far more powerful hardware was common and other OO languages had become popular—to implement it fully.

The Ada 95 designers recognized the strength of the existing Ada 83 type declaration and derivation facilities, and so determined that extending the

type facility—not, for example, attempting to build C++-style classes by providing a “package type”—was the proper way to provide for inheritance. Indeed, in this way a large increase in expressive power for OOP is achieved with remarkably little additional syntax.

4.3.1 Classical Polymorphic Types: Variant Records

Consider a typical Ada application in the software controlling the instrument cluster in an automobile. There are several kinds of instruments; all have some common features but each is also different from the others. A classical solution—in Ada 83 and earlier languages—is to represent the instrument type as a *variant record*, sometimes called a *discriminated record* or, more formally, *discriminated union*. In Ada, given some basic types

```
subtype Speeds is Integer range 0 .. 85; -- mph
subtype Percent is Integer range 0 .. 100;
type InstrumentKinds is (Speedometer, Gauge,
Graphic_Gauge);
```

the variant record type might look like

```
type Instrument (Kind: InstrumentKinds) is record
  Name: String(1 .. 14) := " ";
  case Kind is
    when Speedometer =>
      SpeedValue: Speeds;
    when Gauge =>
      GaugeValue: Percent;
    when Graphic_Gauge =>
      Reading: Percent;
      Size : Integer:= 20;
      Fill : Character:= '*';
      Empty: Character:= '.';
  end case;
end record;
```

The phrase (Kind: InstrumentKinds) defines a special record component called the *discriminant*; in other languages such a field is called a *tag*. The discriminant serves to parametrize the type; a typical variable declaration might be

```
S: Instrument(Kind => Speedometer);
```

which *constrains* S to that variant. If we wished to allow an instrument variable to contain different instruments at different times—to be *unconstrained* or *mutable*—we could provide a default discriminant value in the type declaration, say,

```
type Instrument (Kind: InstrumentKinds := Gauge) is
  record...
```

in which case we could declare

```
I: Instrument;
```

whose value is *initially* be a gauge but could change over time.

In Ada, a variant type declaration *must* include a discriminant component; “free union” types without discriminants, such as those in C or Pascal do not exist. Further, Ada rules—whose details we will not belabor here—ensure that in a mutable variant object, the value of the discriminant is always consistent with the actual values in the variant part.

The variant parts are specified with a `case` construct, and operations on variant objects typically use `case` statements to manipulate the variant parts. Variant records represent yet another kind of static polymorphism, in that mutable objects can effectively change their type over time and operations can be written that select among the various type possibilities.

The inconvenience in using a variant record type is that if, in the future of the program, a new variant must be added, it must be added to the original type declaration. Worse, each operation that manipulates objects of the type must be explicitly modified with a new `case` choice to account for the new variant. In many applications this additional maintenance is quite acceptable; further, the static solution is appealing to developers of real-time systems in which *predictability* of execution time and space overhead is of paramount concern.

For full OOP, we are interested precisely in being able to extend a variant type ad infinitum without imposing any changes to existing code. In other words, we are interested in inheritance and dynamic polymorphism. In Ada 95, this is provided by a syntactically simple extension of the type system.

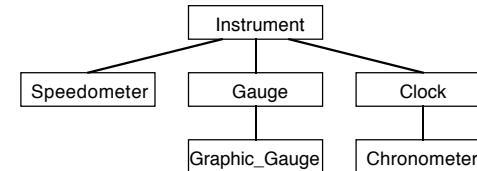
4.3.2 Type Extension: Ada 95 Tagged Types

Support in Ada 95 for full inheritance and polymorphism is done using *type extension*, building on the existing type, type derivation, and package capabilities of Ada 83. We introduce this in the context of the automobile instrument cluster mentioned in the last section. By the end of this discussion we will display a dashboard such as

```
Speed      : 45 Miles per Hour
Fuel       : 60 %
```

```
Water      : <*****.....>
Oil        : <*****.....>
Time       : 12:15:00
Chronometer : <<79976>>
```

using the following type hierarchy:



Recall that in the variant-record solution above, adding a new variant required adding a new `case` choice to every operation on objects of the variant type. Using type extension, we can add new variants without touching existing code. `HB.Instruments` is the interface of a package exporting a root type `Instrument`.

Listing 22

```
1 package HB.Instruments is
2
3   type Instrument is abstract tagged record
4     Name: String(1 .. 14) := (others => ' ');
5   end record;
6
7   procedure Set_Name(I: in out Instrument; S: String);
8   procedure Display_Value(I: Instrument);
9
10 end HB.Instruments;
```

The reserved word `tagged` is the key to type extension. Adding it to a record type declaration causes the declaration to function as the root of a type hierarchy. We can derive new types from it, as in the earlier `Credit_Card` example, but now we can also add components to the derived type.

Primitive operations of the parent type—essentially those procedures and functions declared in the same package interface, just after the parent type—are inherited by the child type but can be *overridden* by similar operations with the same name but with formal parameters of the parent type replaced by those of the child type. In this example, `Set_Name` and `Display_Value` are primitive operations of the type `Instrument`.

The reserved word `abstract` mentioned in the type declaration indicates that this type serves simply as a root for derivations; that is, we cannot declare any objects of the type. For an `abstract` type, it is possible

to declare `abstract` operations; an abstract operation has no function or procedure body, and is used simply to serve as a root operation for inheritance. An ordinary primitive operation *can* be overridden; an abstract primitive operation *must* be overridden.

Now the child package `HB.Instruments.Basic` provides a set of three instrument kinds, all derived from `Instrument`.

Listing 23

```

1 package HB.Instruments.Basic is
2
3   subtype Speeds is Integer range 0 .. 85; -- mph
4
5   type Speedometer is new Instrument with record
6     Value: Speeds;
7   end record;
8
9   procedure Set_Value(S: in out Speedometer; V: Speeds);
10  procedure Display_Value(S: Speedometer);
11
12  subtype Percent is Integer range 0 .. 100;
13
14  type Gauge is new Instrument with record
15    Value: Percent;
16  end record;
17
18  procedure Display_Value(G: Gauge);
19
20  type Graphic_Gauge is new Gauge with record
21    Size : Integer:= 20;
22    Fill : Character:= '*';
23    Empty: Character:= '.';
24  end record;
25
26  procedure Display_Value(G: Graphic_Gauge);
27
28 end HB.Instruments.Basic;
```

Lines 5-7 declare a new type `Speedometer`, derived from `Instrument`. Each `Speedometer` object has the name component of `Instrument`, but also an additional component `Value`, specific to the new type. `Speedometer` is not `abstract`; we can declare objects of this type, which of course was just the intent.

The new type has a total of three operations, each in a different category:

- `Set_Name`, which it inherits unchanged from its parent. That is, we have implicitly declared a new operation

```

procedure Set_Name(I: in out Instrument; S: in
String);
```

- `Set_Value`, a new operation specific to `Speedometer`
- `Display_Value`, a `Speedometer`-specific operation that overrides the corresponding `Instrument` one

Similarly, the type `Gauge`, another extension of `Instrument`, is declared (lines 12-18) with one overriding operation and one inherited one. Now `Graphic_Gauge` is declared (lines 20-24) as an extension of `Gauge`. We now have a three-level hierarchy of types, namely `Instrument`, `Gauge`, and `Graphic_Gauge`.

`HB.Instruments.Clocks` introduces two more instrument types.

Listing 24

```

1 package HB.Instruments.Clocks is
2
3   subtype Sixty is Integer range 0 .. 60;
4   subtype Twenty_Four is Integer range 0 .. 24;
5
6   type Clock is new Instrument with record
7     Seconds : Sixty := 0;
8     Minutes : Sixty := 0;
9     Hours : Twenty_Four := 0;
10  end record;
11
12  procedure Display_Value (C : Clock);
13  procedure Init (C : in out Clock;
14                H : Twenty_Four := 0;
15                M, S : Sixty := 0);
16
17  procedure Increment(C: in out Clock; Inc: Integer :=1);
18
19  type Chronometer is new Clock with null record;
20
21  procedure Display_Value (C : Chronometer);
22
23 end HB.Instruments.Clocks;
```

`Clock` extends `Instrument`. Its operations are `Set_Name` (inherited from `Instrument`), `Display_Value` (overrides the one inherited from `Instrument`), `Init` (new primitive), and `Increment` (new primitive).

Now `Chronometer` extends `Clock`. The construct with `null record` satisfies a rule that a type derived from a tagged type *must* have an extension, even if no new components are added.

`Chronometer` has the following operations:

- `Set_Name`, inherited from `Clock` but declared higher, with `Instrument`
- `Display_Value`, overriding the one inherited from `Clock`

- `Init` and `Increment`, inherited from `Clock`.

Our type hierarchy is

That a type has three kinds of primitive operations—inherited from somewhere above it, overriding inherited ones, and new ones—is not unique to Ada, but inherent in the nature of OOP with inheritance, sometimes called “classification-first design.”

For large “industrial-strength” type hierarchies, where each object can have many applicable operations, this can lead to real confusion for programmers and for the readers of their programs. For both groups, an important question is “Given an object of type `T`, exactly which operations are applicable to it?” Keeping a mental list of these operations is difficult: Some operations—new and overriding—are declared in the same place with the type, but inherited operations could have been declared *anywhere* above this type in the hierarchy. In comparison, using classical packages—sometimes called “composition-first design”—results in *all* operations of a given type being declared with the type.

The software industry has recognized that here, as almost everywhere in computing, there is no “free lunch” in OOP: there is an important tradeoff between the flexibility of classification-first design and the ease of understanding and maintainability in composition-first design. Systems of nontrivial size will probably use both paradigms. Ada supports both paradigms equally well, so the language need not drive the design.

One might ask why Ada requires the use of `tagged` to indicate an extensible type, that is, why all types—or, at least, all record types—are not potentially extensible. The answer is that extensible types require additional overhead: space for an internal tag, time to distinguish at run-time which of several (potentially many) derived types a given value has, and so forth. Requiring `tagged` supports a key Ada principle: “you don’t have to pay for features you don’t use.”

If a type is *not* `tagged`, there will *not* be the associated overhead. The compiler will not generate such overhead; the human developer is assured that there will be none. This is very important to developers of real-time systems, to whom knowledge of sources of potential run-time overhead is critical. Finally, compiling an Ada 83 program with an Ada 95 compiler will produce no new overhead; an Ada 83 program cannot have `tagged` types because these do not exist in Ada 83!

4.3.4 Using the Instruments Hierarchy: Polymorphic Dispatch

Before examining the implementations of the instruments packages, let us see how they might be used. Specifically, the next few listings show how to build a linked list of instruments, which will constitute a dashboard. First we give a package interface `HB.Instruments.Aux` (for “auxiliary”).

Listing 25

```

1 with HB.Instruments.Basic, HB.Instruments.Clocks;
2 use HB.Instruments.Basic, HB.Instruments.Clocks;
3 package HB.Instruments.Aux is
4
5   type InstrumentPointer is access all Instrument'Class;
6
7   Speed : aliased Speedometer;
8   Fuel : aliased Gauge;
9   Oil, Water : aliased Graphic_Gauge;
10  Time : aliased Clock;
11  Chrono : aliased Chronometer;
12
13  SpeedPointer: InstrumentPointer := Speed'Access;
14  FuelPointer: InstrumentPointer := Fuel'Access;
15  OilPointer: InstrumentPointer := Oil'Access;
16  WaterPointer: InstrumentPointer := Water'Access;
17  TimePointer: InstrumentPointer := Time'Access;
18  ChronoPointer: InstrumentPointer := Chrono'Access;
19
20  procedure Display (P: InstrumentPointer);
21
22 end HB.Instruments.Aux;
```

This package declares a *general access type* `InstrumentPointer` that can designate (“point to”) instrument objects. The reserved word `all` (in this context) indicates that an object of type `InstrumentPointer` can designate either a statically declared instrument or one whose space is heap-allocated; if `all` were omitted, only heap-allocated objects could be designated.

The designated type `Instrument'Class` consists of the entire type hierarchy of which `Instrument` is the root. An `InstrumentPointer` object can thus be made to point to an object of type `Speedometer`, `Gauge`, `Graphic_Gauge`, `Clock`, or `Chronometer`, or of any additional types that may be derived from any of these in the future. (Recall that `Instrument` itself is `abstract`, so no `Instrument` objects can exist.)

Lines 7-11 declare some objects in the class `Instrument'Class`. The reserved word `aliased` indicates that these statically declared objects may be designated by access objects; omitting `aliased` would disallow such designation. Lines 13-18 declare some access objects. Each is initialized with a “pointer” to its associated instrument, using the `'Access` attribute.

The procedure `Display` takes an access value as its parameter; its implementation is

Listing 26

```

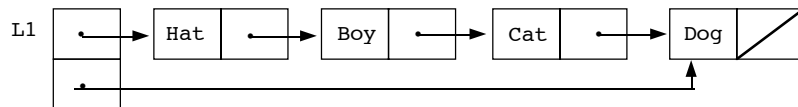
1 package body HB.Instruments.Aux is
2
3   procedure Display (P: InstrumentPointer) is
4   begin
5     Display_Value (P.all);
6   end Display;
7
8 end HB.Instruments.Aux;
```

In line 5, the expression `P.all` dereferences `P`—that is, finds the object designated by `P`. The type of `P.all` can be anything in `Instrument'Class`, and can change each time `Display` is called. Moreover, each type in this class has its own `Display_Value`. Every time line 5 is executed, the actual type of `P.all`, at that instant, determines which of many `Display_Value` operations is called. This is an example of *dynamic or run-time polymorphism*. The Ada term for this is *dispatching*; line 5 is a *dispatching operation*.

4.3.5 Building a Dashboard

We build a dashboard as a linked list of instruments. This linked list is *heterogeneous*; it is a “container” consisting of nodes, each of whose contained object can be a different instrument. Indeed, sometime in the future we could define more instruments, and our list structure must be flexible enough to contain them as well.

We shall use an instance of a generic list package `HB.Lists_Generic`, which supports creating lists like this one:



This is an interesting package that illustrates several important concepts; we defer detailed discussion of it until we ha

Listing 27

```

1 with HB.Lists_Generic;
2 with HB.Instruments.Aux;
3 package HB.Dashboards is new HB.Lists_Generic
4   (ElementType => HB.Instruments.Aux.InstrumentPointer,
```

```

5   DisplayElement => HB.Instruments.Aux.Display);
```

`HB.Dashboards` is an instance of `HB.Lists_Generic`. The generic requires a client to supply two parameters. The first, `ElementType`, must specify the type of contained objects; the second, a procedure parameter `DisplayElement`, must specify how to display an element of type `ElementType`.

The instantiated package provides a type `List` and several operations. Of these, we will use `AddToEnd`—which, given a list and a contained value, adds a new node containing that value to the tail of the list—and `Display`—which walks through a list node-by-node, displaying all the contained objects.

We now have all the pieces with which to build and display a dashboard; our main program, `Show_Dashboard`, accomplishes our task.

Listing 28

```

1 with HB.Instruments.Basic; use HB.Instruments.Basic;
2 with HB.Instruments.Clocks; use HB.Instruments.Clocks;
3 with HB.Instruments.Aux; use HB.Instruments.Aux;
4 with HB.Dashboards; use HB.Dashboards;
5 procedure Show_Dashboard is
6
7   Dashboard : List;
8
9 begin
10
11   Set_Name (Speed, "Speed");
12   Set_Name (Fuel, "Fuel");
13   Set_Name (Water, "Water");
14   Set_Name (Oil, "Oil");
15   Set_Name (Time, "Time");
16   Set_Name (Chrono, "Chronometer");
17
18   Speed.Value := 45; -- mph
19   Fuel.Value := 60; -- %
20   Water.Value := 80; -- %
21   Oil.Value := 30; -- %
22   Init (Time, 12, 15, 00);
23   Init (Chrono, 22, 12, 56);
24
25   AddToEnd (Dashboard, SpeedPointer);
26   AddToEnd (Dashboard, FuelPointer);
27   AddToEnd (Dashboard, WaterPointer);
28   AddToEnd (Dashboard, OilPointer);
29   AddToEnd (Dashboard, TimePointer);
30   AddToEnd (Dashboard, ChronoPointer);
31
32   Display (Dashboard);
33
34 end Show_Dashboard;
```

The context clauses do not include the root package `HB.Instruments` because this program makes no direct reference to anything there.

Line 7 declares a list variable; the type `List` is provided by `HB.Dashboards`. The variables `Speed`, `Fuel`, etc., and their associated pointers `SpeedPointer`, `FuelPointer`, etc., were declared in `HB.Instruments.Aux`. Lines 11-16 set the various display names; lines 18-23 set various values. Lines 25-30 build the dashboard by adding each of the pointer objects to the list. Finally, line 32 displays the entire dashboard as promised at the start of this discussion. The output, once again, is

```
Speed       : 45 Miles per Hour
Fuel        : 60 %
Water       : <*****.....>
Oil         : <*****.....>
Time        : 12:15:00
Chronometer : <<79976>>
```

4.3.6 Comments on Pointers in Ada

In building our heterogeneous list, why must the nodes contain pointers to the instrument objects and not the objects themselves? That is, why must we instantiate with `InstrumentPointer` and not something more like `Instrument'Class`? The answer relates to storage allocation. A list node must have a component indicating its contained object; space must be allocated for this object, but how much? We do not know, and cannot know, the space required for each type in `Instrument'Class`, because `Instrument` is an extensible type that allows types derived from it to grow indefinitely. On the other hand, pointers are the same size regardless of the size of their designated objects. Ada could have been designed so that these pointers were created implicitly, by the run-time system, but the designers chose to require explicit pointers, to accommodate real-time system designers who prefer not to have pointers generated “behind their backs.”

Why are Ada pointers called “access objects?” In other languages, “pointer” has been directly associated with “address,” and also the term acquired a negative reputation because pointers are so easily abused. In Ada, an access value has no direct relationship to an address. Access values and addresses may or may not have the same internal form; this is intentionally left up to the implementer. Further, no arithmetic operations are predefined on access values. This minimizes the likelihood that pointers will point to inappropriate memory locations.

Another very important characteristic of Ada pointers is that each pointer object is guaranteed by the standard to have the initial value `Null`, a special

value that denotes an “empty” pointer. An execution-time attempt to dereference a `Null` pointer results in `Constraint_Error` being raised. Automatic initialization of pointers ensures that uninitialized pointers containing “garbage” cannot be used to access unpredictable blocks of storage. This makes programs that use pointers inherently safer, and also significantly easier to write and debug.

Finally, Ada rules require that any object accessed by `'Access` have a lifetime at least as long as that of the corresponding general access type. This rule prevents a possible dangling pointer. For example, consider this short main program.

Listing 29

```
1 with HB.Instruments; use HB.Instruments;
2 with HB.Instruments.Basic; use HB.Instruments.Basic;
3 procedure Main is
4   type Pointer is access all Speedometer;
5   Pointer1: Pointer;
6 begin
7   declare
8     Pointer2: Pointer;
9     S: aliased Speedometer;
10  begin
11    Pointer2 := S'Access;
12    Pointer1 := Pointer2;
13  end;
14  Pointer1.all.Value := 50;
15 end Main;
```

In our exception handling example, we observed that one can create frames—`begin/end` blocks—within a program unit. In line 7, `declare` opens an inner frame that allows us to declare local entities whose lifetimes begin when their declarations are elaborated and end when control passes through the end of the frame. Local objects are typically stack-allocated, so their space is reclaimed at the end of the frame.

In this example, we have a big problem: `Pointer2` and `S` are allocated and reclaimed in the inner frame, but `Pointer1` is not. This means that in line 14, `Pointer1` is “dangling” and the statement will try to write a value into space that has already been reclaimed. Dangling pointers are usually not this obvious, and can be horribly difficult to debug. Luckily, Ada programs like this are invalid. The compiler rejects programs that use the `'Access` attribute at a level deeper than the one in which the corresponding general access type is declared; this simple rule guarantees that no pointer of this kind can outlive its designated value.

4.3.7 Implementation of the Instruments Hierarchy

We return now to the implementations of the various instrument packages.

Listing 30

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 package body HB.Instruments is
3
4   procedure Set_Name(I: in out Instrument; S: String) is
5   begin
6     I.Name (1..S'Length) := S;
7   end Set_Name;
8
9   procedure Display_Value(I: Instrument) is
10  begin
11    New_Line;
12    Put(I.Name);
13    Put(": ");
14  end Display_Value;
15
16 end HB.Instruments;
```

The body of `HB.Instruments` is straightforward and contains nothing new. Note that `Display_Value` just displays the string name of the instrument. Next, consider the body of `HB.Instruments.Basic`.

Listing 31

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
3 package body HB.Instruments.Basic is
4
5   procedure Display_Value(S: Speedometer) is
6   begin
7     Display_Value(Instrument (S));
8     Put(S.Value, 1);
9     Put(" Miles per Hour");
10  end Display_Value;
11
12  procedure Set_Value
13  (S: in out Speedometer; V: Speeds) is
14  begin
15    S.Value := V;
16  end Set_Value;
17
18  procedure Display_Value(G: Gauge) is
19  begin
20    Display_Value(Instrument (G));
21    Put(G.Value, 1);
22    Put(" %");
23  end Display_Value;
24
25  procedure Display_Value(G: Graphic_Gauge) is
26    Lg: constant Integer := G.Size * G.Value / 100;
27    S1: constant String(1 .. Lg) := (others => G.Fill);
```

```

28    S2: constant String(Lg + 1 .. G.Size)
29      := (others => G.Empty);
30  begin
31    Display_Value(Instrument (G));
32    Put('<');
33    Put(S1);
34    Put(S2);
35    Put('>');
36  end Display_Value;
37
38 end HB.Instruments.Basic;
```

The code here is entirely familiar except for one new detail. In line 7,

```
Display_Value(Instrument (S));
```

calls the `Display_Value` defined for `Instrument`. To make this happen, its actual parameter is converted to an `Instrument` value. What happens to the extension component that turned `Instrument` into `Speedometer`? It is effectively stripped off in this *up-conversion* process. In programming in languages supporting OOP, up-conversions are common and down-conversions are disallowed. In our package, lines 20 and 31 do similar up-conversions to `Instrument`.

Given the above discussion, `HB.Instruments.Clocks` contains much detail but no new concepts at all; it is included just for completeness.

Listing 32

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
3 package body HB.Instruments.Clocks is
4
5   procedure Display_Value(C: Clock) is
6   begin
7     Display_Value(Instrument (C));
8     if C.Hours < 10 then
9       Put('0');
10    end if;
11    Put(C.Hours, 1);
12    Put(":");
13    if C.Minutes < 10 then
14      Put('0');
15    end if;
16    Put(C.Minutes, 1);
17    Put(":");
18    if C.Seconds < 10 then
19      Put('0');
20    end if;
21    Put(C.Seconds, 1);
22  end Display_Value;
23
```

```

24 procedure Increment
25   (C: in out Clock; Inc: Integer:=1) is
26   nInc: Integer;
27 begin
28   C.Seconds :=(C.Seconds + Inc) mod 60;
29   nInc :=(C.Seconds + Inc) / 60;
30   C.Minutes :=(C.Minutes + nInc) mod 60;
31   nInc :=(C.Minutes + nInc) / 60;
32   C.Hours :=(C.Hours + nInc) mod 24;
33 end Increment;
34
35 procedure Init(C: in out Clock;
36   H: Twenty_Four := 0;
37   M, S: Sixty := 0) is
38 begin
39   C.Seconds := S;
40   C.Minutes := M;
41   C.Hours := H;
42 end Init;
43
44 procedure Display_Value(C: Chronometer) is
45 V: Integer;
46 begin
47   Display_Value(Instrument (C));
48   V := C.Seconds + C.Minutes * 60 + C.Hours * 3600;
49   Put("<<");
50   Put(V, 1);
51   Put(">>");
52 end Display_Value;
53
54 end HB.Instruments.Clocks;

```

4.3.8 Dynamic Data Structures

The dashboard example made use of a generic linked list package, `HB.Lists_Generic`, detailed discussion of which was deferred until this point. Here is the interface of `Lists_Generic`.

Listing 33

```

1 with Ada.Finalization;
2 generic
3   type ElementType is private;
4   with procedure DisplayElement (Item: in ElementType);
5 package HB.Lists_Generic is
6
7   type List is private;
8
9   procedure MakeEmpty (L : in out List);
10  procedure AddToEnd
11    (L: in out List; Element: in ElementType);
12  function "="(L1, L2: List) return Boolean;
13  procedure Display (L: in List);
14
15 private

```

```

16
17 type ListNode;
18 type ListPtr is access ListNode;
19 type ListNode is record
20   Element: ElementType;
21   Next: ListPtr;
22 end record;
23
24 type List is new Ada.Finalization.Controlled
25 with record
26   Head: ListPtr;
27   Tail: ListPtr;
28 end record;
29
30 procedure Initialize (L : in out List);
31 procedure Finalize (L: in out List);
32 procedure Adjust (L : in out List);
33
34 end HB.Lists_Generic;

```

Like most modern linked-structure systems, this package depends on the language support for dynamic storage allocation. Ada provides for such allocation; the term *storage pool* is used to refer to the “heap” from which blocks of memory are dynamically drawn. Ada also provides for deallocation of individual blocks; a deallocated block is returned to the storage pool.

The Ada standard allows, but does not require, *automatic* deallocation, sometimes called “garbage collection.” In simple terms, garbage collection is not required because developers of real-time systems feel strongly that this would lead to unpredictable run-time overhead and therefore prefer that programmers control their own storage reclamation. As we shall see shortly, Ada does provide useful mechanisms to make it easier for programmers to do this reliably.

We will return shortly to the meaning of the context clause in line 1. Meanwhile, lines 3 and 4 give, as usual, the generic formal parameters, actuals for which an instantiating client is required to supply. The first parameter is `ElementType`, which describes the kind of object to be contained in each node. Recall from the earlier matrices discussion that in this context, `private` indicates that any type, including a `private` one, but not including a `limited` or `tagged` one, can be supplied to “match” `ElementType` (in the dashboard example, we used `HB.Instruments.Aux.InstrumentPointer`). The second parameter is `DisplayElement`, which describes how to display an object of type `ElementType` (in the dashboard case, we used `HB.Instruments.Aux.Display`).

Now the public part of the interface provides a `private` type `List` and four operations, just enough to be illustrative. Additional operations would be straightforward linked-list manipulations; we choose here to omit them for brevity's sake.

`MakeEmpty` removes all the nodes in a list and returns them explicitly to the storage pool. In the absence of garbage collection, one cannot simply disconnect a linked list from its header; all the lists's nodes would remain allocated but inaccessible.

`AddToEnd` is straightforward; given an element, this operation just adds a node containing the element to the tail of the list. `Display` is equally straightforward; it iterates through the list, calling `DisplayElement` at each node.

Before discussing "=", let us examine the data structures in the `private` section of the interface. First, lines 19-22 describe the node type as a record with an element and a forward link to the next node. This link is of type `NextPtr`. Since Ada requires types to be declared before they are used, line 18

```
type ListPtr is access ListNode;
```

declares the pointer type. But now `ListNode` seems to be used before *it* is declared. We therefore declare it as an incomplete ("forward") declaration in line 17. This three-part declaration is the conventional Ada idiom for declaring linked structures.

Now lines 24-28 declare the list structure itself as a header block, a pair of pointers designating the first and last nodes, respectively, of the list (we will explain shortly why it is an extension of a tagged type `Ada.Finalization.Controlled`).

The overloaded operator "=" is interesting. Given lists `L1` and `L2`, this is a "deep" equality test that returns `True` if and only if the *contents* of the lists are equal, that is, if the element in each node of `L1` is equal to the element in the corresponding node of `L2`. Recall from the `Rationals` discussion that equality-test (and inequality test) are *predefined* for `private` types, so this "=" overrides the predefined one.

All we can expect of predefined equality is that it will compare these header blocks. If the two header blocks are equal, they are pointing to the same list. If they are unequal, predefined equality will return `False`, of

course, but the two lists might still be equal in the "deep" sense we require. We need our own "=" operator to provide this deep testing.

4.3.9 Controlled Types and Finalization

We hinted above that Ada provides a mechanism that facilitates writing an application-specific garbage-collection scheme. Three activities must be supported:

- initialization of an object just after its elaboration
- finalization of the object just before its destruction
- user-defined assignment

This support is embodied in the standard library package `Ada.Finalization`.

Listing 34

```
1 package Ada.Finalization is
2
3   type Controlled is abstract tagged private;
4   procedure Initialize(Object : in out Controlled);
5   procedure Adjust   (Object : in out Controlled);
6   procedure Finalize (Object : in out Controlled);
7
8   type Limited_Controlled is
9     abstract tagged limited private;
10  procedure Initialize
11    (Object : in out Limited_Controlled);
12  procedure Finalize
13    (Object : in out Limited_Controlled);
14 private
15   ... -- not specified by the language
16 end Ada.Finalization;
```

The type `Ada.Finalization.Controlled` is

- `abstract`—it is a root type; clients cannot declare objects
- `tagged`—it is extensible; its operations are inherited by types derived from it
- `private`—its internal structure cannot be directly accessed by a client

The operations `Initialize`, `Adjust`, and `Finalize` can be overridden for a derived type. They are not `abstract` operations, though, so they need not be overridden. In this case, the root operations are essentially "no-ops" that have no discernible effect.

How do these operations work? Suppose we have

```

type MyType is new Ada.Finalization.Controlled
  with some_extension

declare
  Object: MyType;
begin
  Object := some_expression;
  ...
end;
```

When control passes into the `declare` block, the declaration of `Object` is elaborated. Since `MyType` is derived from `Controlled`, its (inherited or overriding) `Initialize` operation is now automatically called.

Now control passes to the assignment statement. Three actions occur:

1. `Finalize(Object)` is automatically called to “clean up” `Object` before copying a new value into it
4. the expression is evaluated and the result copied into `Object`, as usual in an assignment
3. `Adjust(Object)` is automatically called

Finally, when control passes out of the block, `Object` will go out of scope and will be destroyed (typically, popped off the system stack). Before this happens, `Finalize(Object)` is automatically called.

Suppose `MyType` represents a linked list. Deriving it from `Controlled` allows us to develop our own garbage collection in terms of overriding operations.

- `Finalize` walks through the linked list, returning nodes one by one to the storage pool. This means that whenever a list variable goes out of scope, the entire list is reclaimed.
- `Adjust` provides a “deep copy.” Given `L1` and `L2`, the assignment `L1:=L2` will first deallocate all the nodes of `L1` (because `Finalize(L1)` is called automatically). The assignment itself does a “shallow copy” operation, that is, it just copies the header block of `L2` to `L1`, but our overriding `Adjust` procedure will copy its elements into newly allocated and linked nodes.

`Ada.Finalization` provides a clean and easy-to-understand equivalent of the constructor and destructor operations of other languages, and moreover provides user-defined assignment in a fashion that is well-integrated with the other operations.

It is now easy to understand the rest of the interface of `HB.Lists_Generic`. The type `List` is derived from `Ada.Finalization.Controlled`. We declare three overriding operations—putting them in the `private` part ensures that a client cannot call them directly.

4.3.10 Implementing the Linked List Package

Now let us examine the body of `HB.Lists_Generic`, which, given the preceding discussion, will be mostly straightforward.

Listing 35

```

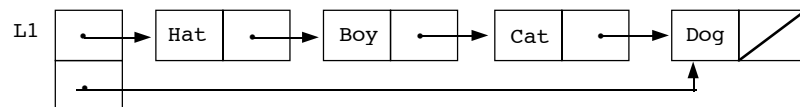
1 with Ada.Unchecked_Deallocation;
2 package body HB.Lists_Generic is
3
4   procedure Display(L: in List) is
5     Current: ListPtr := L.Head;
6   begin
7     while Current /= Null loop
8       DisplayElement(Current.Element);
9       Current := Current.Next;
10    end loop;
11  end Display;
12
13  procedure AddToEnd
14    (L: in out List; Element: in ElementType) is
15  begin
16    if L.Tail = Null then
17      L.Tail := new ListNode'(Element, Null);
18      L.Head := L.Tail;
19    else
20      L.Tail.Next := new ListNode'(Element, Null);
21      L.Tail := L.Tail.Next;
22    end if;
23  end AddToEnd;
24
25  function "="(L1, L2: List) return Boolean is
26    Current1: ListPtr := L1.Head;
27    Current2: ListPtr := L2.Head;
28  begin
29    while Current1 /= Null and Current2 /= Null loop
30      if Current1.Element /= Current2.Element then
31        return False;
32      end if;
33      Current1 := Current1.Next;
34      Current2 := Current2.Next;
35    end loop;
36    return (Current1 = Null and Current2 = Null);
37  end "=";
38
39  procedure Adjust (L: in out List) is
40    TempList: List;
41    Current: ListPtr;
```

```

42 begin
43   Current := L.Head;
44   while Current /= Null loop
45     AddToEnd (TempList, Current.Element);
46     Current := Current.Next;
47   end loop;
48   L.Head := TempList.Head;
49   L.Tail := TempList.Tail;
50   TempList.Head := Null;
51   TempList.Tail := Null;
52 end Adjust;
53
54 procedure Dispose is
55   new Ada.Unchecked_Deallocation
56     (Object => ListNode, Name => ListPtr);
57
58 procedure Finalize (L: in out List) is
59   Current: ListPtr := L.Head;
60   Leading: ListPtr;
61 begin
62   while Current /= Null loop
63     Leading := Current.Next;
64     Dispose(Current);
65     Current := Leading;
66   end loop;
67   L.Head := Null;
68   L.Tail := Null;
69 end Finalize;
70
71 procedure MakeEmpty (L: in out List) is
72 begin
73   Finalize(L);
74 end MakeEmpty;
75
76 procedure Initialize (L: in out List) is
77 begin
78   Finalize (L);
79 end Initialize;
80
81 end HB.Lists_Generic;

```

Refer again to the linked list



The context clause in line 1 mentions `Ada.Unchecked_Deallocation`. We will explain this shortly. Now given a list like the one above, it is easy to see that the procedure in lines 4-11 declares a temporary pointer `Current`, initialized to the first node in the list. The `while` loop, in traditional linked-list programming style, walks down the list, calling `DisplayElement` to display each element in turn.

In `AddToEnd` (lines 13-23), we allocate a new node, store the element in it, and connect it to the list. If the list is initially empty, the new node is the first and only one, and

```
L.Tail := new ListNode'(Element, Null);
```

stores in `L.Tail` a pointer to a new (heap-allocated) node with `Element` and `Null` in the corresponding fields. (Strictly speaking, the `Null` could have been omitted, because that field is already `Null`; we include it for emphasis.) If the list already contains some nodes, then we connect the new node to the end of the list and change the `Tail` pointer accordingly (lines 20-21).

The overloaded `"="` operator (lines 25-37) simply moves two temporary pointers down the two lists in parallel, returning `False` to the calling program if, at a given point in both lists, their elements disagree. The loop iterates until the end of one or both lists is reached. The two lists are equal (line 36) if and only both ends are reached at the same time.

Lines 39-52 give the implementation of our overriding `Adjust` procedure. Since `Adjust` has only one parameter `L`, we make `Adjust` work as a deep copy operation by copying `L` into a temporary list `TempList`, using `AddToEnd` in a straightforward `while` loop, then we copy the head and tail pointers of `TempList` back into `L`. In this manner we return in `L` a list that is distinct from the one received in `L`. Now suppose a client program executes

```
L1 := L2;
```

Because `List` is a controlled type, `L1` is first finalized. Then the header block (head and tail pointers) of `L2` are copied into `L1`. At this point, our `Adjust` is called. The client now has, in `L1`, a complete but distinct copy of `L2`, as desired.

Lines 54-56 instantiate the generic procedure `Ada.Unchecked_Deallocation`, to produce a procedure `Dispose`. If `P` and `Q` are of type `ListPtr`, then executing

```
Q := P;
Dispose(P);
```

causes these standard actions:

1. `P` is `Null` after the `Dispose` call

4. If `P` was already `Null`, the `Dispose` call has no effect
3. The storage designated by `P` is deallocated. Actually reclaiming the storage is recommended but not required by the standard; practical implementations will, of course, reclaim the storage.

Since `P` is now `Null`, an attempt to dereference `P` raises `Constraint_Error` (as discussed earlier). On the other hand, setting `P` to `Null` does not—cannot—magically set `Q` to `Null`, so `Q` is now a dangling pointer, and the behavior of an attempt to dereference it is *erroneous*. The Ada standard defines erroneous execution as one whose effect is not predictable. In this case, the standard cannot require that dangling pointers be detected at execution time; in general, the overhead to do so would be unacceptable. It is for this reason that the generic procedure is called `Unchecked_Deallocation`; it cannot be expected to check for dangling pointers.

In `HB.Lists_Generic`, our `Dispose` instance is called in our overriding `Finalize` (lines 58-69), in which two temporary pointers, `Current` and `Leading`, are used to walk down the list, calling `Dispose` on each node in turn.

Finally, our exported operation `MakeEmpty`, and the overriding operation `Initialize`, are both implemented (lines 71-79) as simple calls to `Finalize`.

Our final set of examples illustrates some of Ada's concurrent programming constructs.

4.4 Concurrent Programming

Before examining a concurrent programming example, it is useful to consider the background against which concurrent programming was introduced in Ada 83.

There is still no universally-accepted terminology to describe this subdiscipline; various authors disagree on the proper use of *concurrency*, *parallelism*, *multitasking*, and other related terms. This is not the place in which to enter at length into this debate. Here we choose to understand the term *concurrent program* as a program in which *several things can happen—or at least appear to happen—simultaneously*. This usage allows us to focus on program structures without undue concern for either the underlying operating system, or hardware configuration, or intended application.

4.4.1 Why Concurrency?

Section 9 of the Steelman Report is entitled *Parallel Processing*. It is interesting to read several paragraphs of that section:

9A. *Parallel Processing*. It shall be possible to define parallel processes. Processes (i.e., activation instances of such a definition) may be initiated at any point within the scope of the definition. Each process (activation) must have a name. It shall not be possible to exit the scope of a process name unless the process is terminated (or uninitiated).

9B. *Parallel Process Implementation*. The parallel processing facility shall be designed to minimize execution time and space. Processes shall have consistent semantics whether implemented on multicomputers, multiprocessors, or with interleaved execution on a single processor.

9H. *Passing Data*. It shall be possible to pass data between processes that do not share variables. It shall be possible to delay such data transfers until both the sending and receiving processes have requested the transfer.

Recall that Steelman was a requirements document, setting out the desired capabilities of a new programming language. It is clear from the above paragraphs that this language was intended to support concurrent processing with constructs that did not depend on either a particular operating system or, indeed, a particular hardware configuration. Steelman is, in fact, using the term “parallel” in the sense in which we defined “concurrent” above. To the Steelman authors, it was of critical importance to be able to develop sophisticated concurrent programs, with long life-cycles, that could be recompiled with minimal change for new generations of competitively-procured hardware, running new underlying operating systems.

Language-level concurrency was (and to some, still is) a controversial requirement; many argued (and some still do) that concurrency is best handled via straightforward calls to operating-system processes or threads. Opinions aside, it is a fact that Ada's concurrency constructs derive directly from the Steelman requirements. The Ada designers studied all the research on concurrent programming, and developed a practical model, as an extended version of the constructs proposed in such lab models as Dijkstra's guarded commands [Dijkstra 1975] and Hoare's monitors [Hoare 1974] and Communicating Sequential Processes (CSP) [Hoare 1978].

4.4.2 A Concurrent Application

Consider a program containing three independent subtasks, A, B, and C, which are required to execute (or, on a uniprocessor machine, appear to execute) simultaneously. Each subtask executes a given number of cycles; during each cycle, each program rests (idles or “sleeps”) a random number of seconds. We view the terminal screen as divided vertically into three equal-width “windows”; as each subtask reaches a “nap” period, it displays its state in its own “window.”

Since different subtasks are sleeping for different time periods, each subtask must control its own “window” and be able to write into it without interference from the other subtasks. The output from a typical execution might be

```
A nap 7 secs      B nap 9 secs      C nap 5 secs
A nap 10 secs     B nap 1 secs     C nap 3 secs
A nap 1 secs      B nap 1 secs     C nap 1 secs
A nap 6 secs      B nap 10 secs    C nap 7 secs
A nap 1 secs      B nap 3 secs
                  B nap 10 secs
                  B nap 1 secs
```

This program is skeletal, without much real functionality; its purpose is to illustrate the Ada concurrency model concisely. The model is very rich, providing a number of constructs we cannot touch on in the space available here.

We will construct our program using two packages and a main procedure; we will present the package interfaces and main procedure first, then examine the package bodies.

4.4.3 A Tasking-Safe Random Number Generator

The “naps” taken by our three subtasks are to be of random duration. The package `HB.Random_Task` provides a random number facility that is “concurrency-safe,” that is, it ensures that the random number generator cannot be simultaneously called by more than one subtask. This prevents the seed of the generator from being corrupted by a second call arriving in the midst of a computation. The interface of this package is

Listing 36

```
1 package HB.Random_Task is
2
3   subtype RandomRange is Natural range 1..10;
4
5   task Randomizer is
```

```
6   entry GiveNumber (Result: out RandomRange);
7   end Randomizer;
8
9 end HB.Random_Task;
```

Lines 3-5 give the interface to a *task* `Randomizer`. The task is Ada’s construct for a concurrent subtask. A task cannot be a compilation unit; it must be declared within a package or main unit. A task has an interface and an implementation (body); the task can provide services by means of *entries*, which are called by other tasks. An entry declaration is syntactically similar to a procedure declaration; an entry call is syntactically similar to a procedure call, but its behavior is different. Specifically, the Ada standard ensures that multiple calls to a given task’s entries are executed one at a time. Given a variable, in some other task,

```
R: HB.Random_Task.RandomRange;
```

the entry call, in that other task,

```
HB.Random_Task.Randomizer.GiveNumber(Result=>R);
```

will deposit in R a random number in the range 1..10. Moreover, this call is guaranteed to complete before another is allowed to begin. We will see how this is done when we examine the body of `HB.Random_Task`.

4.4.5 A Tasking-Safe Screen Manager

Each of our subtasks must be able to write to its own “window” on the screen. Our second package is `HB.Screen`. This package provides a “tasking-safe” manager for the terminal screen, which we assume has ANSI or VT100 screen-control characteristics.

Listing 37

```
1 package HB.Screen is
2
3   ScreenHeight : constant Integer := 24;
4   ScreenWidth  : constant Integer := 80;
5
6   subtype Height is Integer range 1..ScreenHeight;
7   subtype Width  is Integer range 1..ScreenWidth;
8
9   type Position is record
10    Row   : Height := 1;
11    Column: Width  := 1;
12  end record;
13
14  task Manager is
```

```

15   entry Beep;
16   entry ClearScreen;
17   entry Write (Item: in String; Where: in Position);
18   end Manager;
19
20 end HB.Screen;

```

The package exports a task `Manager` with entries providing `Beep`, `ClearScreen`, and `Write` services. Why is this task necessary?

To write a given string at a given row/column position requires two operations: positioning the cursor at the correct position, and writing the string there. Using ANSI standard screen-control commands, positioning the cursor to row 15, column 35 requires us to send, to the terminal, the string

```
ESC [ 15 ; 35 f
```

where `ESC` is the “escape” character and the blanks are included for clarity here but are not sent. This is a total of eight characters; moreover, this control string must be immediately followed by the string to be displayed.

It is fundamental to concurrent programming that wherever multiple processes (threads, tasks) are running and writing to a shared resource (in this case, the standard output file), it is possible that one of the processes will be interrupted (pre-empted, swapped out) and one of its sibling processes will gain control. If this interruption happens to occur in the midst of a “transaction” to the screen, that is, the command followed by the string, the process gaining control might start another screen transaction. Since the terminal itself has no idea that this has occurred, it interprets the incoming character stream as best it can; this is likely to result in a chaotic mess on the screen because half-completed commands can be randomly intermixed with displayed strings. For example, here is the output from an execution of a similar program without this protection:

```

1;21fA nap 7 secs41fB nap 7 secsC nap 4 secs
                                     C nap 4
secs[2;1f21fA nap 4 secsB nap 7
A nap 1 secs      B nap 8 secs      C nap 1 secs
A nap 4 secs      B nap 9 secs      C nap 6 secs
A nap 8 secs      B nap 7 secs
                                     B nap 5 secs
                                     B nap 2 secs

```

We cannot allow this chaos to occur, so we must provide *mutual exclusion* that allows at most one task at a time to send a screen transaction, and ensures that this transaction will be completed before another one is begun. Implementing the screen services as entries of a task provides the

desired mutual exclusion; we will return to the details after examining the main program.

A “real” program would normally use a window manager of some sort for this window-like display style. Window managers are, of course, generally provided by platform-dependent application program interfaces (APIs); it is interesting to note that APIs are not necessarily concurrency-safe, and an API-using concurrent program—in Ada or another language—may well have to provide its own safety mechanisms. We recently developed some multitasking Ada programs using the Apple Macintosh Toolbox API [Feldman 1997b]; the window manager and event handler design required us to develop Ada-level safety mechanisms very similar to the one here.

4.4.6 The Main Multitask Program

Here is the main program `Show_Tasks`.

Listing 38

```

1 with Ada.Text_IO;
2 with HB.Random_Task;
3 with HB.Screen;
4 procedure Show_Tasks is
5
6   task type SimpleTask (Message: Character;
7                         HowMany: HB.Screen.Height;
8                         Column: HB.Screen.Width) is
9     entry StartRunning;
10  end SimpleTask;
11
12  task body SimpleTask is
13    Nap: HB.Random_Task.RandomRange;
14  begin
15    accept StartRunning;
16    for Count in 1..HowMany loop
17      HB.Random_Task.Randomizer.GiveNumber(Result=>Nap);
18      HB.Screen.Manager.Write(
19        Where => (Row => Count, Column => Column),
20        Item => Message & " nap"
21          & Natural'Image(Nap) & " secs");
22    delay Duration(Nap);
23    end loop;
24  end SimpleTask;
25
26  Task_A: SimpleTask
27    (Message => 'A', HowMany => 5, Column => 1);
28  Task_B: SimpleTask
29    (Message => 'B', HowMany => 7, Column => 21);
30  Task_C: SimpleTask
31    (Message => 'C', HowMany => 4, Column => 41);
32
33 begin -- Show_Tasks

```

```

34
35  HB.Screen.Manager.ClearScreen;
36  Task_B.StartRunning;
37  Task_A.StartRunning;
38  Task_C.StartRunning;
39
40 end Show_Tasks;

```

In this main unit, we declare our three subtasks. We could as easily have enclosed these in a package, but chose to illustrate tasks declared in a main program. Lines 6-10 declare a *task type* `SimpleTask`. This implies that we can declare task *objects*; in fact, we do so in this program, in lines 26-31. Ada tasks are, in fact, a form of active concurrent objects.

Each task declared in a program is *activated* (starts running) just after control reaches the `begin` of the block in which it is declared; the order of activation is not defined by the language. Each task declared in a package is activated when that package is elaborated, that is, just before control passes to the main procedure. Here, as each task of type `SimpleTask` is activated, the actual parameters (strictly speaking, discriminant values) for `Message`, `HowMany`, and `Column` are passed to it. `Message` indicates the constant message to be displayed in each cycle, `HowMany` indicates the number of cycles this task is to run, and `Column` indicates the screen column in which to display its message.

Since an Ada task is activated implicitly based on the enclosing program's block structure, it is sometimes necessary to inhibit the task from doing any work until this "start button" is "pressed" by an entry call. To illustrate how this is done, `SimpleTask` also provides one entry, `StartRunning`.

Lines 12-24 show the body of the task type. When task objects are declared, each one has (in effect) a copy of this code. Each object may be mapped to an OS-level thread, but the precise mapping depends, of course, on the nature of the underlying OS support. The point is that the application programmer generally has no need to worry about this; the Ada implementation takes care of it just as it takes care of all the other platform dependencies like memory structures, floating-point arithmetic, and so on.

Let us choose one task object arbitrarily and follow the execution expressed in the task body. Line 15 says

```
accept StartRunning;
```

Upon reaching an `accept`, a task *waits* at the `accept` until the corresponding `entry` is called by another task. The task is put into a

suspended or *blocked* state, which allows other tasks sharing the same CPU to run. The corresponding entry calls are issued here in lines 36-38. If the "start button" is never "pressed", the task will wait forever. We will show in the package bodies how to guard against this eventuality.

Lines 16-23 are a simple `for` loop that runs for the desired number of cycles, according to the task's `HowMany` parameter. The task calls for a random number (line 17), then formats and displays a line via the screen manager (lines 18-21). Finally, line 22,

```
delay Duration(Nap);
```

causes the task to "sleep" (suspend, block) for the desired number of seconds. The type conversion `Duration(Nap)` is required because the delay period must be of type `Duration`. Note that the nap length—delay period—is recomputed in each cycle, based on the random number. We have used a relative `delay` here, that is, the `delay` is relative to the current time. Ada also provides an absolute `delay until`, whose argument is a `Time` value.

It is important to understand that the Ada standard provides that a task becomes *ready* at the expiration of a delay period. This means that the task is not blocked and will start running again when it gains control of a CPU. The standard cannot guarantee that a CPU will be available at precisely the right instant; this obviously depends on contention with other tasks in the program, or, in a multiprogramming environment, on contention with unrelated programs.

This is a simple example of the general concurrent-programming problem of ensuring that a cyclic process meets its deadline. In real-time system design, this is an important subdiscipline. It is certainly helpful to have clear and platform-independent concurrency constructs in the coding language, but these do not—cannot—substitute for careful analysis and design.

4.4.7 Implementing the Random Number Generator

Let us examine the body of `HB.Random_Task`.

Listing 39

```

1 with Ada.Numerics.Discrete_Random;
2 package body HB.Random_Task is
3
4   task body Randomizer is
5     package RandomTen is

```

```

6     new Ada.Numerics.Discrete_Random
7     (Result_Subtype => RandomRange);
8     G: RandomTen.Generator;
9     begin
10    RandomTen.Reset(Gen => G);
11    loop
12    select
13    accept GiveNumber (Result: out RandomRange) do
14    Result := RandomTen.Random(Gen => G);
15    end GiveNumber;
16    or
17    terminate;
18    end select;
19    end loop;
20 end Randomizer;
21
22 end HB.Random_Task;

```

The context clause mentions `Ada.Numerics.Discrete_Random`. This standard generic library package can be instantiated for any integer or enumeration type or subtype; one could use it, for example, to produce random coin flips of the enumeration type (`Heads`, `Tails`). Here we instantiate it (lines 6-7) for our `RandomRange` of 1..10.

Using the random number generator requires declaring a generator variable, as we declare `G` in line 8. The type `Generator` is `limited private`; as such, it has no operations except those provided by the package. A generator variable retains the current seed of the number generator, so we can run multiple random number sequences—each with its own generator variable—using the same instance of the package.

The standard provides three ways to initialize a pseudo-random sequence.

1. If we do nothing at all, the sequence is initialized with an unknown value that is the same for each initialization. This gives a repeatable pseudo-random sequence.
4. If we call `Reset` as in line 10, the sequence is initialized with a value that depends on the current time of day. The pseudo-random sequence is then effectively random, because we cannot predict this value, which will obviously be different in successive `Reset` calls.
3. If we call `Reset` with two parameters, a generator variable and an integer value, we can make the sequence repeatable by using the same integer value in successive `Reset` calls.

If we had written lines 11-19 as

```

loop
  accept GiveNumber (Result: out RandomRange) do
    Result := RandomTen.Random(Gen => G);

```

```

end GiveNumber;
end loop;

```

the task would loop repeatedly, waiting at the `accept` in each cycle, until another task called `GiveNumber`. The `accept/do/end` frame is called a *rendezvous*; while this task is executing the rendezvous code, the calling task is blocked and waiting for the result. Once the rendezvous is completed, both tasks become ready and each can run again when it gains control of a CPU.

The Ada runtime system provides a FIFO queue for each entry of each task. If several other tasks call the entry (quasi)simultaneously—for example, another call arrives while the first caller’s rendezvous is being executed—the calls are queued in this entry queue. *One* call is accepted per cycle, so the queued caller is handled in the next cycle.

The simple loop above has no way to terminate, so if, at a given point, no further calls arrive, the task “hangs,” waiting at the `accept` for a call that will never come. The `select` statement that enclose our `accept` (lines 12-18) provides two *select alternatives*. The behavior of this strange-looking construct is that calls are accepted as long as they continue to arrive. However, *if* no calls are pending on the entry, *and* the other tasks in the program are ready to terminate, then this task will terminate as well.

The `terminate` alternative provides a kind of graceful implicit termination of a task. A task will also terminate if control passes to the end of its body, so we could use program logic to force this task out of its main loop. For example, we could write a finite one:

```

while the time is earlier than 5 PM loop
  ...
end loop;

```

and the task would terminate just after 5 PM. The disadvantage here is that we would strand any pending calls. Other strategies are also possible; the programmer has substantial flexibility here.

4.4.8 Implementing the Screen Manager

Finally, we present the body of `HB.Screen`.

Listing 40

```

1 with Ada.Text_IO, Ada.Integer_Text_IO;
2 use Ada.Text_IO, Ada.Integer_Text_IO;
3 package body HB.Screen is
4
5   task body Manager is

```

```

6  begin
7    loop
8      select
9        accept Beep do
10         Put (Item => ASCII.BEL);
11         Ada.Text_IO.Flush;
12       end Beep;
13      or
14        accept ClearScreen do
15         Put (Item => ASCII.ESC & "[2J");
16         Ada.Text_IO.Flush;
17       end ClearScreen;
18      or
19        accept Write
20         (Item: in String; Where: in Position) do
21         Put (Item => ASCII.ESC & '[');
22         Put (Item => Where.Row, Width => 1);
23         Put (Item => ',');
24         Put (Item => Where.Column, Width => 1);
25         Put (Item => 'f');
26         Put (Item => Item);
27         Ada.Text_IO.Flush;
28       end Write;
29      or
30        terminate;
31      end select;
32    end loop;
33  end Manager;
34 end HB.Screen;

```

The task body for `Manager` is similar to that of the random number generator. It is intended to operate as a background task, similar to a device driver, and therefore is allowed to activate It has a main loop, inside of which is a `select` statement. This statement is an example of nondeterministic selection, and is based on the concurrent programming research of the 1970s, especially CSP and its later implementation in Occam.

In each cycle of the main loop, control reaches the `select`. If no calls are pending on any of the three entries, the task then waits at the `select` for one of these entries to be called, and responds to the one that arrives first, entering a rendezvous with the calling task. The `Put` statements in the rendezvous blocks are generally obvious; `Ada.Text_IO.Flush` is used to send the OS output buffer contents directly to the terminal. As before, the calling task blocks during the rendezvous, and furthermore `Manager` can take no other action until the rendezvous completes. This ensures that the screen transaction can be fully sent to the terminal before another begins, and prevents the messed-up screen we described earlier.

Once the rendezvous completes, the `select` is satisfied and `Manager` loops back for another cycle. Suppose now that several calls have arrived

from different tasks, so that at least two of the entry queues have callers pending. In this case, `Manager` makes an arbitrary selection of one of the queues, and then enters a rendezvous with the caller at the head of that queue. This arbitrariness—the algorithm for it is not defined by the Ada standard—makes the selection nondeterministic.

Generally, this nondeterminism is desirable; where it is not, the programmer can use Boolean conditions—so-called “guard” clauses—on the various `accept` alternatives to gain more control. For example, a priority selection can be imposed by guarding one of the queues so that it will not be selected unless another queue is empty. This *guarded command* idea was also developed in the laboratory concurrent languages of the 1970s.

The `select` statement here also has a `terminate` alternative, which acts as it did in the random-number example. If no calls are pending on any of the entry queues, and if the other tasks are ready to terminate, `Manager` will terminate as well.

It is also possible to set a timer as one of the alternatives, to trigger an action if no other call has occurred within the time-out period. There are other possibilities as well; the `select` is a “feature-rich” construct.

Ada 95 provides another, complementary, concurrency construct called the *protected type*. Protected objects have behavior similar to tasks in that mutual exclusion is automatically provided for their operations. However, protected objects are less general than tasks. They were designed not as general-purpose concurrency structures, but rather to support the special case of providing very efficient mutual exclusion on relatively simple objects, for high-performance real-time situations. In these special cases, protected operations can be implemented much more efficiently than general rendezvous-based communication. Space does not permit us to go into more detail on these special structures.

4.4.9 Comments on Concurrent Programming

It is a fact that even today, there is no effective, working standard, across today’s operating systems, for processes or threads. Even within the UNIX family, syntactic and semantic differences persist between threads libraries. Given this state of affairs, it is far more efficient and reliable for a small group compiler experts, working with a well-designed platform-independent model, to tailor a compiler and runtime system to a particular hardware/OS platform than for a large community of application programmers to deal explicitly with OS-specific concurrency constructs in every program.

Ada 83 compilers and runtimes were targeted to dozens of different platforms, from small embedded processors to supercomputers, workstation networks, and personal computers. Ada 95 compilers and runtimes support nearly as many targets already. It is not difficult to write a single concurrent program—with no “conditional compilation” for different platforms—that will compile and execute properly on all available platforms.

Indeed, this writer developed an interesting implementation of the concurrent-programming classic Dining Philosophers [Feldman 92] that compiled and ran—without changing a character of source code—on 26 different compiler/OS combinations, using compilers from 12 different vendors. An Ada 95 version of this program is included in the GNAT examples library; it compiles and runs without change using ObjectAda on Windows and also on the 12 GNAT-supported platforms. The GNAT runtime systems on these platforms support Ada-level concurrency using whatever underlying OS thread support is available, or a threads emulator if nothing suitable exists at the OS level.

Concurrent programming is an abstraction method, not just an implementation method for real-time systems. It is often said that the world is concurrent, so modeling the world is best done with concurrent programming. Often, performance of concurrent programs is adequate using straightforward, platform-independent constructs. Programs with real-time performance constraints might require some platform-dependent performance tuning, perhaps with the help of facilities provided by the Ada Real-Time Annex; here the maxim applies that “it is much easier to make a correct program fast than to make a fast program correct.”

As is the case with all newly-designed language structures, in immature Ada 83 compilers the performance of Ada tasking left much to be desired, and caused many developers to abandon tasking for more familiar OS-level operations. Over time, however, compilers and runtime systems have reached maturity and performance of Ada-level concurrency compares favorably with platform-specific calls. As the word spreads on this, developers are taking a second look, and beginning to realize just how forward-looking the concurrency model was and how effective it can be.

If Steelman had failed to require concurrency, and the resulting language had therefore failed to support it, the notion of a common language, in which programs could be written with a high degree of platform independence, would have been significantly undermined. The language would have left developers to write *precisely* the interesting and complex code—the concurrent code—in highly platform-dependent terms. With the current emphasis on multitasking in personal-computer operating systems, and the

difficulty many programmers have in coding programs for it, Ada’s higher-level constructs have much to offer.

5. Bibliography

This selected bibliography is divided in three sections: Ada web sites, Ada books, and other articles and publications of interest. A textbook bibliography containing brief reviews, and other reference lists, are available on the Web; where texts have their own associated web sites, links to these are provided in the various online bibliographies.

5.1 World-Wide Web Resources on Ada

There are five main sites on the World-Wide Web intended for use by those interested in Ada. All these sites mirror, or link to, most of the files on the others, but each site is somewhat different from the others in its emphasis and organizational structure.

ACM Special Interest Group on Ada (SIGAda). The major professional organization for those interested in Ada. This is a membership organization, with very attractive rates for students. <http://www.acm.org/sigada>

Ada Information Clearinghouse. Formerly the “official” U.S. Government-sponsored Ada site and now operated by the Ada Resource Association, a trade organization made up of vendors of Ada products. A good place to look for Ada manuals and other documents. Most of this site is copied to the PAL, but this one is less crowded. <http://www.adaresource.org>

Ada Programming Language Resources for Educators and Students. This author maintains this SIGAda-sponsored site which collects the sets of links most useful to educators and students in an easy-to-use fashion. <http://www.acm.org/sigada/education>

Home of the Brave Ada Programmers. A World Wide Web page for anyone interested in Ada. Contains links to the other Internet sites. <http://www.adahome.com>

Public Ada Library (PAL). A comprehensive collection of freely available Ada documents, compilers, tutorials, and source code libraries. Nearly everything Ada-related on the Internet is mirrored to this site. <http://wuarchive.wustl.edu/languages/ada>

5.2 Published Books on Ada 95

At this writing, eighteen books, all published since 1995, are available, for various audiences.

Barnes, J. (ed.) [1997]: *Ada 95 Rationale*. Springer-Verlag, (ISBN 3-540-63143-7) The Ada 95 Rationale is the companion to the Reference Manual (language standard); it introduces Ada 95 and its attractive new features and explains the rationale behind them. It should be studied in parallel with the Ada 95 Reference Manual. Various electronic forms are also available; a searchable HTML version is on the Web.

Barnes, J. G. P. [1998]: *Programming In Ada 95, 2nd ed.* Addison-Wesley, (ISBN 0-201-34293-6) The latest in a series of very popular Ada texts by this author. Barnes covers the whole language well and very readably, with a fine sense of humor.

Beidler, J. [1997]: *Data Structures and Algorithms: An Object-Oriented Approach Using Ada 95*. Springer-Verlag, (ISBN 0-387-94834-1) An interesting approach to this subject; its special strength is the development in parallel of two libraries of software components, one generics-based, the other inheritance-based.

Ben-Ari, M. [1998]: *Ada for Software Engineers*. John Wiley, (ISBN 0-471-97912-0) An excellent introduction to Ada for professionals with experience in software development. There are some very good, fully-coded examples here. The text is shipped with a multiplatform CDROM, containing a wealth of documentation and other references, as well as GNU Ada 95 development systems for Windows 95/98/NT and a number of UNIX variants.

Burns, A., and A. Wellings [1997]: *Real-Time Systems and Programming Languages, 2nd ed.* Addison-Wesley, (ISBN 0-201-40365-X) An excellent text on the issues in designing real-time systems. Ada 95 is emphasized, but other languages such as occam and various C dialects are also considered.

Burns, A., and A. Wellings [1998]: *Concurrency in Ada, 2nd ed.* Cambridge University Press, (ISBN 0-521-62911-X) A readable and very complete text on concurrent programming and real-time systems in Ada 95.

Cohen, N. [1996]: *Ada as a Second Language, (2nd ed.)* McGraw-Hill, (ISBN 0-07-011607-5) An encyclopedic work, over 1100 pages long. Its strength is in its thorough, exhaustive coverage of the language and its realistic examples.

Culwin, F. [1997]: *Ada: A Developmental Approach (2nd edition)*. Prentice Hall, (ISBN 0-13-264680-3) A text for students without previous programming experience. The author emphasizes design issues as well as programming.

English, J. [1997]: *Ada 95: The Craft of Object-Oriented Programming*. Prentice Hall, (ISBN 0-13-230350-7) This book introduces Ada-as a first language, using an example-driven approach that gradually develops small programs into large case studies.

Feldman, M. B. [1997]: *Software Construction and Data Structures with Ada 95*. Addison-Wesley, (ISBN 0-201-88795-9) An undergraduate text focusing on algorithms and data structures with a definite software-engineering flavor and a heavy emphasis on developing generic and polymorphic software components.

Feldman, M. B., and E. B. Koffman [1999]: *Ada 95: Problem Solving and Program Design, 3rd ed.* Addison-Wesley, (ISBN 0-201-36123-X.) A text that introduces Ada 95 to readers with no previous programming experience in any language. The text is shipped with a multiplatform CDROM, containing a wealth of documentation and other references, as well as GNU Ada 95 development systems for DOS, Linux, MacOS, and Windows 95/98/NT.

Johnston, S. [1997]: *Ada 95 for C and C++ Programmers*. Addison Wesley, 1997 (ISBN 0-201-40363-3) The correspondences between C/C++ idioms and Ada 95 ones are described; both the core language and the annexes are presented. Shipped with the Aonix ObjectAda CDROM.

Naiditch, David J. [1995]: *Rendezvous with Ada 95*. John Wiley and Sons, (ISBN 0-471-01276-9) A very readable, often humorous, survey of Ada 95.

Rosen, J-P. [1995] *Méthodes de Génie Logiciel avec Ada 95* (Software Engineering Methods with Ada 95) (in French). InterEditions, Paris, (ISBN 2-7296-0569-X) Introduces Ada 95 in the context of several important software engineering methodologies.

Skansholm, J. [1997]: *Ada from the Beginning, 3rd ed.* Addison-Wesley (ISBN 0-201-40376-5) This book was one of the first to use Ada with CS1-style pedagogy. There are excellent sections on the idiosyncracies of interactive I/O (a problem in all languages), and a sufficient number of fully-worked examples to satisfy students.

Smith, M. A. [1996]: *Object-Oriented Software in Ada 95*. International Thomson Computer Press, (ISBN 1-85032-185-X) For those interested in

pursuing object-oriented programming with Ada 95, this book can serve as an excellent followup to the present work.

Taft, T. and R.A. Duff (eds.) [1997]: *Ada 95 Reference Manual*. Springer-Verlag. (ISBN 3-540-63144-5) The Ada 95 Reference Manual completely documents the Ada 95 standard and thus is an indispensable working companion for anybody using Ada 95 professionally or learning the language systematically. Various electronic forms are also available; a searchable HTML version is on the Web.

Wheeler, D.A. [1997]: *Ada 95: The Lovelace Tutorial*. Springer-Verlag, 1997 (ISBN 0-387-94801-5) This book, based on a very successful worldwide web tutorial, introduces the basic elements of Ada 95 to those who already know another programming language.

5.3 Selected Articles and Other Publications of Interest

Dijkstra, E. W. [1975]: “Guarded Commands, Nondeterminacy, and Formal Derivation of Programs,” *Communications of the ACM* **18**, 8, pp. 453-457. Introduces much of the theory used in the Ada `select` statement.

Feldman [1992]: “Portable Dining Philosophers: a Movable Feast of Concurrency and Software Engineering,” *Proc. 23rd ACM-SIGCSE Technical Symposium on Computer Science Education*. Also on the Web at the author’s site.

Feldman, M.B. [1997a]: *Ada as a Foundation Programming Language*. This is a frequently updated document reporting the use of Ada in first-year programming courses. Available on the Web.

Feldman, M.B. [1997b]: “An Ada 95 Sort Race Construction Set,” *Proc. AdaEurope '97*. Also available on the Web at the author’s site.

Gutttag, J. V., E. Horowitz, and D. R. Musser [1978]: “Abstract Data Types and Software Validation,” *Communications of the ACM*, **21**:12, pp. 1048-1064. One of the early papers on abstract data types.

Hoare, C. A. R. [1974]: “Monitors: An Operating System Structuring Concept,” *Communications of the ACM* **17**:10, pp. 549-557. Introduces much of the theory behind Ada’s protected types.

Hoare, C. A. R. [1978]: “Communicating Sequential Processes.” *Communications of the ACM* **21**:8 (1978), pp 666-677. Introduces much of the theory behind rendezvous.

HOLWG [1978]: *Steelman*. This is the famous requirements document for a new defense programming language, which led to Ada 83. The previous versions of this document were named, in sequence, STRAWMAN, WOODENMAN, TINMAN, and IRONMAN. This document was never accessibly published but is now available on the Web.

Ichbiah, J., *et al.* [1979a]: *Preliminary Ada Reference Manual, SIGPLAN Notices*, Vol. 14, No. 6A, June 1979.

Ichbiah, J., *et al.* [1979b]: *Rationale for the Design of the Ada Programming Language, SIGPLAN Notices*, Vol. 14, No. 6B, June 1979.

Ichbiah, J., *et al.* [1987]: *Rationale for the Design of the Ada Programming Language*, Honeywell. Available on the Web at Ada IC.

Liskov, B. H., and S. N. Zilles [1977]: “Abstraction Mechanisms in CLU,” *Communications of the ACM*, **20**:8, pp. 564-576. Specification vs. implementation.

National Research Council [1996]: *Ada and Beyond: Software Policies for the Department of Defense*. Available from NRC; also on the web at the Ada IC site.

Stein, D. [1985]: *Ada: A Life and a Legacy*. MIT Press (ISBN 0-262-19242-X). A biography of the “real” Ada, after whom the language is named.

Stroustrup, B. [1982]: “Classes: an Abstract Data Type Facility for the C Language,” *SIGPLAN Notices*, **17**:1, pp. 354-356. A very early article on what became C++.

U.S. Dept. of Defense [1983]: ANSI/MIL-STD-1815A-1983, Reference Manual for the Ada Programming Language. This is the definition of the language now called Ada 83. Available on the Web at Ada IC.

Whitaker, W. [1996]: “Ada—the Project: the DoD High Order Language Working Group,” *History of Programming Languages—II*, Addison-Wesley (1996, ISBN 0-201-89502-1). This definitive paper, written for the 1993 Second History of Programming Languages Conference (HOPL-II), is also on the Web.