ASEET Advanced Ada Workshop

Software Engineering Institute

12-15 January 1988

Carnegie Mellon University
Pittsburgh, Pennsylvania

Sponsored by the
U.S. Department of Defense
The Advanced Ada Workshop is offered semi-annually by the ASEET Team. This document contains the tutorials for the Workshop held January 12-15, 1988 at Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. Topics are Introduction, Software Engineering, Packages, Exceptions, Tasking and Generics.
Welcome to the Software Engineering Institute. I'd like to extend my greetings and express the hope that your visit is informative, pleasant, and productive.

If this is your first contact with the SEI, you may be interested in the following background information. The Software Engineering Institute is a federally funded research and development center (FFRDC). Our organization was formed in 1984 in response to the need for advances across all phases of the software engineering process. It is operated by Carnegie Mellon University, under contract with the Department of Defense. Our main directives include bringing the ablest professional minds and the most effective technology to bear on the rapid improvement of the quality of operational software in mission-critical computer systems, exploring and disseminating technology, and establishing standards of excellence for software engineering practice.

We concentrate most of our effort on technology transition, although we are actively involved with technology generation as well. Our approach is to shift software engineering from a labor-intensive basis to a technology-intensive basis through automation based on sound models and theories, and to concentrate on technology transition throughout the managerial, professional, legal, economic, and computational facets of software engineering. Programs at the SEI provide a framework for coordinated efforts within defined areas of technology. They build a foundation to support continued improvement in an area of technology, to develop SEI expertise, and to facilitate the transition of technology and information into practice.

I hope your visit exceeds your expectation.

Sincerely,

Larry Druffel
Director

Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
(412) 268-7700
Ada Software Engineering
Education and Training (ASEET) Team
Advanced Ada Workshop
Software Engineering Institute
12-15 January 1988

Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Authors and Affiliations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sec. I</td>
<td>Introduction</td>
<td>Major Allan Kopp, Chairman, ASEET Team</td>
</tr>
<tr>
<td>Sec. II</td>
<td>Software Engineering</td>
<td>Captain Roger Beauman, Captain Michael Simpson, Keesler Technical Training Center</td>
</tr>
<tr>
<td>Sec. III</td>
<td>Packages</td>
<td>Mr. John Bailey, IDA Consultant</td>
</tr>
<tr>
<td>Sec. III</td>
<td>Exceptions</td>
<td>Major Pat Lawlis, Air Force Institute of Technology</td>
</tr>
<tr>
<td>Sec. IV</td>
<td>Tasking</td>
<td>Captain David Cook, United States Air Force Academy</td>
</tr>
<tr>
<td>Sec. V</td>
<td>Generics</td>
<td>Lieutenant Commander Lindy Moran, United States Naval Academy, Major Chuck Engle, Software Engineering Institute</td>
</tr>
</tbody>
</table>
ADVANCED Ada WORKSHOP
Software Engineering Institute
12-15 January 1988

SCHEDULE
Tuesday, 12 January 1988

8:00   Welcoming Remarks   Training Room A
8:15   Introduction        Major Allan Kopp
               AJPO Representative
9:00   Tutorial - Software Engineering   Capt. Roger Beauman-Keesler AFB
10:00  Break
10:15  Tutorial - Software Engineering   Capt. Roger Beauman - Keesler AFB
               Capt. Michael Simpson - Keesler AFB
12:00  Lunch
1:30   Tutorial - Software Engineering   Capt. Roger Beauman - Keesler AFB
               Capt. Michael Simpson - Keesler AFB
3:00   Break
3:15   Tutorial - Software Engineering   Capt. Roger Beauman - Keesler AFB
               Capt. Michael Simpson - Keesler AFB
5:00   End of Session
7:00-9:00  Birds of a Feather   AJPO activities Major Allan Kopp
WEDNESDAY, 13 JANUARY 1988

Training Room A

8:30  Tutorial - Packages  Mr. John Bailey - IDA Consultant
10:00 Break
10:15 Tutorial - Packages  Mr. John Bailey - IDA Consultant
12:00 Lunch
1:30  Tutorial - Exceptions  Major Pat Lawlis - AFIT
3:00  Break
3:15  Tutorial - Exceptions  Major Pat Lawlis - AFIT
5:00  End of Session
7:00 - 9:00  Birds of a Feather  Ada Information Clearinghouse and ASEET Materials Library

Thursday, 14 January 1988

Training Room

8:30  Tutorial - Tasking  Capt. David Cook - Air Force Academy
10:00 Break
10:15 Tutorial - Tasking  Capt. David Cook - Air Force Academy
12:00 Lunch
1:30  Tutorial - Tasking  Capt. David Cook - Air Force Academy
3:00  Break
3:15  Tutorial - Tasking  Capt. David Cook - Air Force Academy
5:00  End of Session
7:00 - 9:00  Birds of a Feather  Compilers Ada Tools SIMTEL20
Friday, 15 January 1988

Training Room A

8:30  Tutorial - Generics  LCDR Lindy Moran - US Naval Academy
      Major Chuck Engle - SEI

10:00 Break

10:15 Tutorial - Generics  LCDR Lindy Moran - US Naval Academy
      Major Chuck Engle - SEI

12:00 Lunch

1:30  Tutorial - Generics  LCDR Lindy Moran - US Naval Academy
      Major Chuck Engle - SEI

3:00  Break

3:15  Tutorial - Generics  LCDR Lindy Moran - US Naval Academy
      Major Chuck Engle - SEI

5:00  End of Session
Ada® PROGRAM

©Ada IS A REGISTERED TRADEMARK OF THE U.S. GOVERNMENT
(Ada JOINT PROGRAM OFFICE)
Ada® REQUIREMENTS DEFINED IN A SERIES OF DRAFT SPECIFICATIONS

- STRAWMAN (1975)
  - PRELIMINARY REQUIREMENTS TO EVOKE COMMENT

- WOODENMAN (1975)
  - RESULT OF A FOUR MONTH REVIEW OF STRAWMAN

- TINMAN (1976)
  - FIRST COMPLETE SET OF REQUIREMENTS BASED ON MILITARY INPUTS

- IRONMAN (1977)
  - USED AS BASIS FOR Ada® EFFORT

- STEELMAN (1978)
  - BASED ON RESULTS OF FIRST Ada® PHASE
  - CURRENT STATEMENT OF REQUIREMENTS

Ada® is a registered trademark of the U.S. Government. (Ada Joint Program Office)
STEELMAN REQUIREMENTS

- STRONG TYPING — EXPLICIT DEFINITION AND ENFORCEMENT OF CHARACTERISTICS OF DATA ELEMENTS
- ENCAPSULATION — RESTRICTS VISIBILITY AND USE OF SELECTED VARIABLES; FACILITATES BOTH TOP DOWN DEVELOPMENT AND ACCUMULATION OF REUSABLE MODULES
- GENERIC FACILITY — PROVIDES EXTENSIBILITY TO THE PROGRAMMER WITHOUT EXTENDING THE LANGUAGE
- TASKING — STRUCTURED APPROACH TO CONCURRENT PROCESSING AND INTERPROCESS COMMUNICATION
- EXCEPTION HANDLING — FACILITY FOR DEALING WITH EXCEPTIONAL SITUATIONS WHICH OCCUR DURING PROGRAM EXECUTION
- INTERRUPT HANDLING — FACILITY FOR PROCESSING INTERRUPTS AND OTHER EXTERNAL STIMULI
- NUMERIC PRECISION — MACHINE INDEPENDENT APPROACH TO INTEGERS, FIXED POINT AND FLOATING POINT
- MACHINE DEPENDENCIES — EXPLICIT DECLARATION AND ENCAPSULATION OF HARDWARE AND OPERATING SYSTEM DEPENDENCIES

Ada® is a registered trademark of the U.S. Government. (Ada Joint Program Office)
<table>
<thead>
<tr>
<th>Languages Formally Evaluated</th>
</tr>
</thead>
<tbody>
<tr>
<td>• COBOL</td>
</tr>
<tr>
<td>• FORTRAN</td>
</tr>
<tr>
<td>• PL/I</td>
</tr>
<tr>
<td>• TACPOL</td>
</tr>
<tr>
<td>• HAL/S</td>
</tr>
<tr>
<td>• CMS-2</td>
</tr>
<tr>
<td>• CS-4</td>
</tr>
<tr>
<td>• JOVIAL J3B</td>
</tr>
<tr>
<td>• ALGOL 60</td>
</tr>
<tr>
<td>• CORAL 66</td>
</tr>
<tr>
<td>• SIMULA 67</td>
</tr>
<tr>
<td>• ALGOL 68</td>
</tr>
<tr>
<td>• PASCAL</td>
</tr>
<tr>
<td>• LIS</td>
</tr>
<tr>
<td>• LTR</td>
</tr>
<tr>
<td>• JOVIAL J73</td>
</tr>
<tr>
<td>• SPL/I</td>
</tr>
<tr>
<td>• EUCLID</td>
</tr>
<tr>
<td>• MORAL</td>
</tr>
<tr>
<td>• ECL</td>
</tr>
<tr>
<td>• PDL/2</td>
</tr>
<tr>
<td>• PEARL</td>
</tr>
<tr>
<td>• RTL/2</td>
</tr>
</tbody>
</table>
STANDARDIZATION

REFERENCE MANUAL PUBLISHED July 1980
MIL-STD 1815 DESIGNATED December 1980
ANSI CANVASS INITIATED April 1981
ANSI CANVASS COMPLETED October 1981
ANSI RECANVASS INITIATED July 1982
ANSI RECANVASS COMPLETED September 1982
ANSI/MIL-STD 1815A Ada® January 1983

Ada® is a registered trademark of the U.S. Government. (Ada Joint Program Office)
# HIGH ORDER LANGUAGE COMPARISON

## Requirements:

<table>
<thead>
<tr>
<th>Feature</th>
<th>FORTRAN</th>
<th>JOVIAL</th>
<th>PL/1</th>
<th>PASCAL</th>
<th>ADA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic Expressions</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program Structure</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Iteration Loops</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Formatted Input/Output</td>
<td>X</td>
<td>-</td>
<td>X</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>Block Structure</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Recursion</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Formal Syntax Definition</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Multi-Tasking</td>
<td></td>
<td>X</td>
<td>-</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Exception Handling</td>
<td></td>
<td>X</td>
<td>-</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Extensive Typing Mechanisms</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Pointers</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strong Typing</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Enumeration Types</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Subrange Types</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Type Declarations</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Aggregate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Derived Type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exception</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pragma</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generic Program Unit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rendezvous</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### ADA Capabilities vs Other DOD HOLS

<table>
<thead>
<tr>
<th>FOR CMS</th>
<th>J73</th>
<th>ADA MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESIGN CRITERIA</td>
<td>38</td>
<td>33</td>
</tr>
<tr>
<td>GENERAL SYNTAX</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>DATA TYPING</td>
<td>46</td>
<td>65</td>
</tr>
<tr>
<td>CONTROL</td>
<td>31</td>
<td>34</td>
</tr>
<tr>
<td>FUNCTIONS &amp; I/O</td>
<td>11</td>
<td>22</td>
</tr>
<tr>
<td>REAL TIME PROCESSING</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>OTHER TECHNIQUES</td>
<td>17</td>
<td>30</td>
</tr>
<tr>
<td>TOTALS</td>
<td>171</td>
<td>213</td>
</tr>
</tbody>
</table>
PROGRAM ELEMENT 63226F

TITLE: DOD COMMON PROGRAMMING LANGUAGE (Ada)

DOD MISSION AREA: ELECTRONIC AND PHYSICAL SCIENCES

BUDGET ACTIVITY: ADVANCED TECHNOLOGY DEVELOPMENT

DESCRIPTION:
- PART OF DOD ACTIVITY TO INTRODUCE, IMPLEMENT AND PROVIDE LIFE CYCLE SUPPORT FOR Ada
- PROVIDE RESOURCES TO MEET THOSE LANGUAGE SUPPORT REQUIREMENTS WHICH ARE COMMON TO THE VARIOUS SERVICES AND AGENCIES
- PROVIDE FOR CONFIGURATION CONTROL OF THE Ada LANGUAGE
- PROVIDE FOR STANDARDIZATION

OPR: Ada JOINT PROGRAM OFFICE (AJPO)
AJPO PROJECTS

LANGUAGE CONTROL
VALIDATION
EDUCATION & PROMOTION
Ada PROGRAMMING
SUPPORT ENVIRONMENTS
TECHNOLOGY INSERTION
LANGUAGE CONTROL

APL/360
ALGOL 60
COLA3L
JOSS
FLAP
IPL-V
MATH LAB 66
PL/I
SNOBOL 3
COMIT II
ALTRAN

FORTRAN I
FORTRAN II
FORTRAN II
FORTRAN 66
FORTRAN 78
FORTRAN V
COBOL 61
COBOL 64
COBOL 65
COBOL 68
CMS-2Y
CMS-2W

Ada

1990
1980
1970
LANGUAGE CONTROL

Ada BOARD
AMERICAN NATIONAL STANDARDS INSTITUTE (ANSI)
FEDERAL INFORMATION PROCESSING STANDARD (FIPS)
INTERNATIONAL STANDARDS ORGANIZATION (ISO)
NYU Ada ED
TRADEMARK REGISTRATION/CERTIFICATION
Ada FORMAL METHODS
Ada RATIONALE
REVISED LANGUAGE REFERENCE MANUAL
LANGUAGE CONTROL

AJPO PROGRAM SUPPORT:

- Establishing & Maintaining Ada as MILITARY, ANSI
  & FIPS standard permits explicit designation in RFPs
  & SOWs

- Trademark establishment & registration actions
  protect program offices from marketing of invalid
  vendor products

- LRM, Ada/ED and Rationale clarify standard for
  compiler implementers allowing rapid development &
  tailoring to meet program specific needs.
MILITARY USE

EMBEDDED COMPUTER SYSTEMS 1975 -

ORIGINAL TARGET USE FOR Ada INTENDED BY HOLWG

INCLUDES ALL DEFENSE SYSTEMS WHERE COMPUTERS ARE A PART OF A LARGER SYSTEM; MISSILES, A/C, SHIPS, TANKS.

EXPECTED APPLICATIONS INCLUDE:

- TRACKING, SYSTEMS SW, NAVIGATION,
- COMMAND, CONTROL & COMMUNICATIONS,
- SIMULATION, COMPUTATION

EXAMPLES: ATF, CAMP
MISSION CRITICAL COMPUTER RESOURCES 1983 -
DIRECTED BY USDRE MEMO; EFFECTIVE 1 Jul 84
INCLUDES EMBEDDED SYSTEMS AND ALL ADP
SYSTEMS WHERE THE FUNCTION, OPERATION, OR
USE INVOLVES
INTELLIGENCE, CRYPTOLOGIC, COMMAND,
CONTROL OR CRITICAL TO MISSION FULFILLMENT
EXAMPLES: MILSTAR, AFATDS

EMBEDDED COMPUTER SYSTEMS 1975 -
INFORMATION SYSTEMS 1984 -

DIRECTED BY ARMY INFORMATION SYSTEMS COMMAND, EFFECTIVE 1 OCT 1984

FIPS APPROVED OCT 1985 OPENING Ada TO AIR FORCE & NAVY INFORMATION SYSTEMS

INCLUDES ROUTINE ADMIN AND BUSINESS APPLICATIONS; E.G.,

PAYROLL, FINANCE, LOGISTICS AND PERSONNEL MANAGEMENT

EXAMPLES: ISS, AF PHASE IV

MISSION CRITICAL COMPUTER RESOURCES 1983 -

EMBEDDED COMPUTER SYSTEMS 1975 -
VALIDATION

VALIDATION APPROACH

POLICY & PROCEDURES
Ada COMPILER VALIDATION CAPABILITY
Ada VALIDATION ORGANIZATION
Ada VALIDATION FACILITY
VALIDATION

Ada VALIDATION PROCEDURES AND GUIDELINES
Ada VALIDATION ORGANIZATION
Ada VALIDATION FACILITIES - WRIGHT-PATTERSON AFB
- GSA/NBS
- GERMANY
- FRANCE
- UNITED KINGDOM
Ada COMPILER VALIDATION CAPABILITY
AUTOMATED VALIDATION TOOLS
VALIDATION

AJPO PROGRAM SUPPORT:

- Certification System with DoD Ada Validation Facility (at WPAFB) permits program office enforcement of Ada standard

- Ada Compiler Validation Procedures and Guidelines coordinates contractor and program office software acquisition/maintenance actions with Ada validation
EDUCATION AND PROMOTION
EDUCATION

- Ada SOFTWARE ENGINEERING EDUCATION AND TRAINING (ASEET) TEAM
  - ASEE ANNUAL SYMPOSIUM
  - ADVANCED Ada WORK SHOPS
  - PROFESSIONAL DEVELOPMENT BRIEFINGS
  - ASEE MATERIALS LIBRARY
  - TRAINING GUIDE FOR Ada SOFTWARE ENGINEERING PROJECTS
  - ASEE PUBLIC REPORTS

- AFCEA STUDY

- CATALOG of RESOURCES in EDUCATION for Ada SOFTWARE ENGINEERING
PROMOTION

Ada INFORMATION CLEARINGHOUSE PRODUCTS

- ON-LINE Ada-INFORMATION DIRECTORY
- CLEARINGHOUSE STAFF AVAILABLE FOR TELEPHONE QUERIES
- CATALOG OF RESOURCES FOR EDUCATION IN Ada AND SOFTWARE ENGINEERING (CREASE 4.0)
- CLEARINGHOUSE STAFF AVAILABLE FOR TELEPHONE QUERIES
- Ada BIBLIOGRAPHY VOL III
- DOCUMENT SEARCHES
- DOCUMENTS REFERENCE LIST
- SPECIAL TOPIC PACKETS
- VALIDATION COMPILER LIST
- GENERAL INFO
- Ada IMPLEMENTATIONS LIST
- EDUCATION
- VALIDATION
- HISTORICAL
- CURRENT AWARENESS
- CALENDAR OF Ada EVENTS
PROMOTION (CON’T)

RECENT ACTIVITIES

Ada INFORMATION CLEARINGHOUSE
ONGOING, NEW CONTRACT
EARLY 1987

Ada USAGE DATA BASE
INITIATED FY86, CONTINUALLY
EXPANDING

PRODUCTS & TOOLS DATABASE
INITIATED FY86, CONTINUALLY
EXPANDING

DDN SUPPORT &
PUBLIC BULLETIN BOARD
ONGOING COMMUNICATIONS
SERVICE
PROMOTION (CON'T)

AdalC MONTHLY ACTIVITY*

<table>
<thead>
<tr>
<th>Category</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>INCOMING CALLS</td>
<td>310</td>
</tr>
<tr>
<td>SPECIAL TOPIC PACKETS</td>
<td>155</td>
</tr>
<tr>
<td>INDIVIDUAL DOCUMENTS</td>
<td>140</td>
</tr>
<tr>
<td>ELECTRONIC BULLETION BOARD</td>
<td></td>
</tr>
<tr>
<td>DOCUMENTS DOWNLOADED</td>
<td>3,643</td>
</tr>
<tr>
<td>CALLS PER MONTH</td>
<td>529</td>
</tr>
<tr>
<td>NEWSLETTER (QUARTERLY)</td>
<td>4,000</td>
</tr>
</tbody>
</table>

*(average for three month period 4/87-6/87)*
AJPO PROGRAM SUPPORT:

- ASEET coordinates DoD training and education activities which support program office personnel and higher level management

- AdaIC provides up to date information to program offices on the availability of Ada technologies for use on DoD systems
COOPERATIVE ACTIVITIES RELATED TO Ada

CONGRESSIONAL INITIATIVES: NUNN AMENDMENT, SOFTWARE VALLEY

DoD PROGRAMS: Ada, STARS, SEI, VHSIC, SDI

MILITARY STANDARDS: MIL-STD-2167 & HANDBOOK, MIL-STD-2168

GOVERNMENT THRUSTS: NASA, NBS, GSA

INDUSTRY GROUPS: SIGAda, AdaJUG, AFCEA, IEEE

ACADEMIC ACTIVITIES: Ada TECHNOLOGY CENTER, EDUCATIONAL SYMPOSIUM
Ada PROGRAMMING SUPPORT ENVIRONMENTS

COMMON APSE INTERFACE SET (CAIS)

LEVEL 0

APSE [LEVEL 3]
MAPSE [LEVEL 2]
COMPILER
KAPSE FUNCTIONS [LEVEL 1]
JCL INTERPRETER
CONFIG. MGR.
LINKER/LOADER
DEBUGGER
Ada PROGRAMMING SUPPORT ENVIRONMENTS

- APSE PERFORMANCE

  EVALUATION & VALIDATION (E&V) TEAM
  Ada COMPILER BENCHMARKS
  Ada COMPILER EVALUATION CAPABILITY
  Ada RUN TIME ENVIRONMENT WORKING GROUP (ARTEWG)

- PORTABILITY OF TOOLS, APPLICATIONS & DATA BASES

  KAPSE INTERFACE TEAM (KIT)
  COMMON APSE INTERFACE SET (DoD-STD-CAIS)
  CAIS VALIDATION CAPABILITY
  CAIS OPERATIONAL DEFINITION
SOFTWARE DEVELOPMENT PRODUCTIVITY

IMPROVEMENTS WITH Ada

LANGUAGE FEATURES SUPPORTING REUSE

PUBLICLY AVAILABLE SOFTWARE INVENTORIES

LIFE CYCLE TOOLS

NETWORKED PROGRAMMER WORKSTATIONS

RAPID PROTOTYPING

EMPHASIS ON SOFTWARE ENGINEERING

REUSEABLE SOFTWARE COMPONENTS

INTEGRATED ENVIRONMENTS

OOD & OTHER METHODOLOGIES
ADA PROGRAMMING SUPPORT ENVIRONMENTS

AJPO PROGRAM SUPPORT:

- AJPO supported initial Air Force and Army APSE Developments for multiple reuse in systems acquisitions

- Prototype Ada Compiler Benchmarking system and Ada Compiler Evaluation Capability provide program offices with technologies to evaluate Ada compiler performance

- Assisting program offices in requesting vendors to correct/improve Ada Technology performance (SAC)

- Sponsoring ARTEWG to focus industry on realtime performance issues to support Ada use in embedded weapons
TECHNOLOGY INSERTION

HAVE Ada® OPPORTUNITIES BEEN KNOCKING?

Ada® IS A TRADEMARK OF THE U.S. GOVERNMENT (Ada JOINT PROGRAM OFFICE)
TECHNOLOGY INSERTION

- DoD DIRECTIVES 3405.1 AND 3405.2

- Ada TECHNOLOGY INSERTION PROGRAM (ATIP)

- NATO INITIATIVE (NUNN AMENDMENT)
DOD DIRECTIVE 3405.1 -- APRIL 2, 1987

COMPUTER PROGRAMMING LANGUAGE POLICY

'UMBRELLA' POLICY FOR ALL DOD -- (REPLACES DODI 5000.31)

ESTABLISHES LONG-RANGE GOAL OF:
TRANSITION TO THE USE OF ADA FOR ALL DOD SOFTWARE DEVELOPMENT

MANDATES ADA FOR:
INTELLIGENCE SYSTEMS -- (INCLUDES MCCR)
COMMAND & CONTROL OF MILITARY FORCES -- (INCLUDES MCCR)
SYSTEMS INTEGRAL TO A WEAPON SYSTEM -- (DODD 3405.2)
MANDATES ADA FOR ALL OTHER (MIS) APPLICATIONS EXCEPT:

WHERE ANOTHER APPROVED HOL IS MORE COST EFFECTIVE OVER THE APPLICATION'S LIFE-CYCLE

**UPDATES 'APPROVED' HOL LIST:**

<table>
<thead>
<tr>
<th>Programming Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADA</td>
</tr>
<tr>
<td>C/ATLAS</td>
</tr>
<tr>
<td>COBOL</td>
</tr>
<tr>
<td>CMS-2M</td>
</tr>
<tr>
<td>CMS-2Y</td>
</tr>
<tr>
<td>FORTRAN</td>
</tr>
<tr>
<td>JOVIAL (J73)</td>
</tr>
<tr>
<td>MINIMAL BASIC</td>
</tr>
<tr>
<td>PASCAL</td>
</tr>
<tr>
<td>SPL/1</td>
</tr>
</tbody>
</table>
DOD DIRECTIVE 3405.2
MARCH 30, 1987

USE OF ADA IN WEAPON SYSTEMS

MANDATES ADA FOR ALL SYSTEMS INTEGRAL TO WEAPON SYSTEMS,
MEANING:

PHYSICALLY A PART OF, DEDICATED TO, ESSENTIAL IN REAL TIME

USED FOR SPECIALIZED TRAINING, DIAGNOSTIC TESTING & MAINTENANCE

USED FOR SIMULATION, CALIBRATION OR RESEARCH & DEVELOPMENT

APPLIES TO ALL PHASES OF THE LIFE CYCLE AND MAJOR UPGRADES
REQUIRES USE OF:

VALIDATED COMPILERS
SOFTWARE ENGINEERING PRINCIPLES (2167/HANDBOOK)
ADA-BASED PROGRAM DESIGN LANGUAGE

REQUIRES DOD COMPONENTS TO DESIGNATE AN

ADA EXECUTIVE OFFICIAL AND AN
ADA WAIVER CONTROL OFFICER

REQUIRES COMPONENT ADA IMPLEMENTATION PLAN BY 30 AUG 87
TECHNOLOGY INSERTION (CON’T)
ATIP PROGRAM

- BREAKDOWN TECH RISK BARRIERS
  - MULTILEVEL SECURITY
  - DISTRIBUTED PROCESSING
  - COMPLEX QUERIED DBMS
  - ADVANCED ARCHITECTURES
  - ULTRA HIGH PERFORMANCE CODE
  - MAXIMUM REUSABLE COMPONENTS

- DIRECT PROGRAM OFFICE ASSISTANCE
  - SIMULATION
  - AVIONICS
  - FIRE CONTROL
  - MISSILES
  - C3I
  - ELECTRONIC WARFARE

- MERIT SELECTION

- COST SHARED

- STANDARD METRICS COLLECTION, ANALYSIS AND REPORTING
NATO SWG ON APSE

PRINCIPLE POCs: Virginia L. Castor, Chair SWG on APSE
David R. Taylor, U.S. Delegate/U.S. PM

BRIEF TASK DESCRIPTION:

- The Special Working Group on Ada Programming Support Environments (SWG on APSE), consisting of representatives from Belgium, Canada, Denmark, France, Germany, Italy, Netherlands, Norway, Spain, United Kingdom, and the United States, have agreed to a memorandum of understanding (MOU) to:

  a. develop and demonstrate a group of software tools representative of a usable APSE through their initial implementation on two distinct computer architectures using an agreed interface set;

  b. develop methods and tools for the evaluation of APSEs and demonstrate this technology on the products resulting from this effort; and

  c. develop the requirements and specifications of an interface standard for APSEs, based on reviews of evolutionary interface developments to be recommended for adoption and use by NATO and the participating nations.
DEMONSTRATION OF APSE CAPABILITY

Operational Weapon Systems Computer
MOTOROLA 68020

Operational Weapon Systems Scenario
VAX/VMS

Operational Weapon Systems Scenario
IBM/MVS
Ada TECHNOLOGY INSERTION

AJPO PROGRAM SUPPORT:

- Expanding Acquisition/Maintenance Management structure to support Ada in DoD Directives and Mil-Std-2167

- Providing Direct Assistance to Program offices:
  - SDIO (FIXIT)
  - ATF (consultation)
  - 155 MM HIP (AdaTAG)
  - Flight Dynamics Lab (F-15 DEMO)
  - Phase IV (funded Ada evaluation ECP)
  - Microprocessor Evaluation - (Air Force MIS Application)
  - Assisting FAA in selecting Ada for Advanced Automated System
  - Assisting Dept of Commerce in selecting Ada for use in flexible mfg
SUMMARY
Ada PROGRAM IMPACTS
WITHIN THE DoD

RIISING MILITARY USE
STRONGER DISCIPLINE IN LANGUAGE STANDARDIZATION
WIDER COOPERATION
INCREASED AVAILABILITY OF COMPILERS & ENVIRONMENTS
IMPROVED PORTABILITY
HIGHER PRODUCTIVITY
DEMAND FOR PERFORMANCE EVALUATION
ADVANCED Ada WORKSHOP

Applied Ada Software Engineering

Capt Roger D. Beauman
Capt Michael S. Simpson

Ada Software Engineering Education and Training (ASEET) Team
Ada is a registered trademark of the U.S. government, Ada Joint Program Office
APPLIED Ada SOFTWARE ENGINEERING

* Basic Problem
  -- Projection to the 1990's
  -- A Macro Solution
* A Practical Solution
  -- Software Engineering
  -- Ada
* Software Engineering
  -- Goals
  -- Principles
* Why Ada?
  -- Features of Ada
  -- Software Engineering Applications
BASIC PROBLEM

Projection to the 1990's

* Multiprocessors - Networks and Parallel Architectures
* Distributed Databases
* Hardware Capabilities
* Software Demands
* Hardware Costs
DISTRIBUTED DATABASES

* Central Control Over Data
* Minimize Effort in Storing Data
* "The Ada Package Store"
DISTRIBUTED DATABASES

* Central Control Over Data
* Minimize Effort in Storing Data
* "The Ada Package Store"
HARDWARE CAPABILITIES

* Mainframe in a Micro
  -- Intel 80286, 80386, 80486, ???
  -- Motorola 68000, 68010, 68020, 68030, ???

* Screen Resolution
  -- Desktop Publishing, CAD/CAM

* Storage Devices
  -- 100+ MB Hard Disks
  -- Access Times - 18 ms to 40 ms

* Opens New Fields of Applications
SOFTWARE DEMANDS

* New Users with Consumer Relationships

* Non-Technical Arenas
  -- Need Guarantees
  -- Demand Reliability

* Development is the Key
  -- Design is Paramount
    --- Simplistic Operations; i.e. TV
  -- Costs of Errors
  -- Other Considerations
A MACRO SOLUTION

* Greater Use of Automation
* Higher Levels of Abstraction
* Reuseability
  -- Isolate Commonality
  -- Create Workable Abstractions
  -- Reusable Parts Library
* Rapid Prototyping
  -- Gain Insight
  -- Evaluate Design Expectations
  -- Compare Design Alternatives

A solution offered by Edward Lieblein
A PRACTICAL SOLUTION

Software Engineering Myths

* Anyone Can Be a Software Engineer
* Automated Tools = Software Engineering
* Structured Programming = Software Engineering
* Structured Analysis = Software Engineering
* Code Re-use = Software Engineering
* It Will Make Programming Obsolete
* AI Will Make It Effortless
* Fantastic Productivity Gains
* Ada = Software Engineering
SOFTWARE ENGINEERING

A PRACTICAL SOLUTION

* What Is It?
* Why Is It Needed?
* The State of the Art
* The State of the Practice
* Why Now?
CHARACTERISTICS OF DoD SOFTWARE

* Expensive
* Incorrect
* Unreliable
* Difficult to predict
* Unmaintainable
* Not reusable
FACTORS AFFECTING DoD SOFTWARE

* Ignorance of life cycle implications
* Lack of standards
* Lack of methodologies
* Inadequate support tools
* Management
* Software professionals
CHARACTERISTICS OF DoD SOFTWARE REQUIREMENTS

* Large
* Complex
* Long lived
* High reliability
* Time constraints
* Size constraints
THE FUNDAMENTAL PROBLEM

* Our inability to manage the COMPLEXITY of our software systems

* Lack of a disciplined, engineering approach
SOFTWARE ENGINEERING

THE ESTABLISHMENT AND APPLICATION OF SOUND ENGINEERING =>

* Environments

* Tools

* Methodologies

* Models

* Principles

* Concepts
SOFTWARE ENGINEERING

COMBINED WITH =>

* Standards

* Guidelines

* Practices
Throughout the life cycle of a system:

* Correct
* Modifiable
* Reliable and safe
* Efficient
* Understandable

To support computing which is:

SOFTWARE ENGINEERING
SOFTWARE ENGINEERING

* Purposes
* Concepts
* Mechanisms
* Notation
* Usage
PURPOSES

* Create software systems according to good engineering practice

* Manage elements within the software life cycle
CONCEPTS

* Derive the architecture of software systems

* Specify modules of the system
MECHANISMS

* Tools for:
  - Writing operating systems
  - Tuning software
  - Prototyping

* Techniques for:
  - Managing projects
  - Systems analysis
  - Systems design

* Standards for:
  - Coding
  - Metrics
  - Human and machine interfacing
NOTATION

* Languages for writing linguistic models

* Documentation
Usage

- Embedded systems
- Data processing
- Control
- Expert systems
- Research and development
- Decision support
- Information management
CONTENT AREAS

* Communication skills
* Software development and evolution processes
* Problem analysis and specification
* System design
* Data Engineering
* Software generation
* System quality
* Project management
* Software engineering projects
PROGRAMMING LANGUAGES AND SOFTWARE ENGINEERING

* A programming language is a software engineering tool

* A programming language EXPRESSES and EXECUTES design methodologies

* The quality of a programming language for software engineering is determined by how well it supports a design methodology and its underlying models, principles, and concepts
TRADITIONAL PROGRAMMING LANGUAGES AND SOFTWARE ENGINEERING

Programming Languages

- Were not engineered
- Have lacked the ability to express good software engineering
- Have acted to constrain software engineering
A PRACTICAL SOLUTION

Ada

Ada and Software Engineering

* They Aren't the Same Thing
* Ada Has Unique Features That Facilitates Software Engineering
* You CAN Write Bad Code in Ada
* Ada is NOT the Total Answer

USAISEC
Ada
AND
SOFTWARE ENGINEERING

Ada
- Was itself "engineered" to support software engineering
- Embodies the same concepts, principles, and models to support methodologies
- Is the best tool (programming language) for software engineering currently available

STANDARDS
ENVIRONMENTS

GUIDELINES
TOOLS
CONCEPTS
PRINCIPLES
MODELS
METHODOLOGIES

PRACTICES
TOOL

---
LANGUAGE DEVELOPMENT

* Requirements completed before development
* Competitive procurement used for design
* Formal planned test and evaluation phase
* Massive public commentary used
* Design team used
* Strict standardization control
SOFTWARE ENGINEERING

* Goals of Software Engineering
* Principles of Software Engineering
GOALS OF SOFTWARE ENGINEERING

★ MODIFIABILITY
★ RELIABILITY
★ EFFICIENCY
★ UNDERSTANDABILITY
PRINCIPLES OF SOFTWARE ENGINEERING

* Abstraction
* Modularity
* Localization
* Information hiding
* Completeness
* Confirmability
* Uniformity
ABSTRACTION

* The process of separating out the important parts of something while ignoring the inessential details

* Separates the "what" from the "how"

* Reduces the level of complexity

* There are levels of abstraction within a system
MODULARITY

* Purposeful structuring of a system into parts which work together

* Each part performs some smaller task of the overall system

* Can concentrate and develop parts independently as long as interfaces are defined and shared

* Can develop hierarchies of management and implementation
LOCALIZATION

* Putting things that logically belong together in the same physical place

INFORMATION HIDING

* Puts a wall around localized details

* Prevents reliance upon details and causes focus of attention to interfaces and logical properties
COMPLETENESS

* Ensuring all important parts are present
* Nothing left out

CONFIRMABILITY

* Developing parts that can be effectively tested

UNIFORMITY

* No unnecessary differences across a system
FEATURES OF Ada

* Supports Large System Development
* Supports Structured Programming
* Supports Top-Down Development
* Supports Strong Data Typing
* Supports Data Abstraction
* Supports Information Hiding and Data Encapsulation
SYSTEMS ENGINEERING

* Analyze problem
* Break into solvable parts
* Implement parts
* Test parts
* Integrate parts to form total system
* Test total system
REQUIREMENTS FOR EFFECTIVE SYSTEMS ENGINEERING

* Ability to express architecture

* Ability to define and enforce interfaces

* Ability to create independent components

* Ability to separate architecture issues from implementation issues
Overview of Important Ada Features

Readability
Program Units
Separate Compilation
Subprograms
Packages
Strong Typing
Typing Structures
Data Abstraction
Tasks
Exceptions
Generics
Low Level Features
READABILITY

* Ada was engineered with the understanding that programming is a human activity

* Features are provided that allow a maintenance person to quickly grasp the meaning of a particular program and to understand its structure

* Readability is more than just a language issue
PROGRAM UNITS

* Components of Ada which together form a working Ada software system

* Express the architecture of a system

* Define and enforce interfaces
PROGRAM UNITS

SUBPROGRAMS
Working components that perform some action

TASKS
Performs actions in parallel with other program units

PACKAGES
A mechanism for collecting entities together into logical units
PROGRAM UNITS

* Consist of two parts: specification and body

SPECIFICATION: Defines the interface between the program unit and other program units (the WHAT)

BODY: Defines the implementation of the program unit (the HOW)
PROGRAM UNITS

* The specification of the program unit is the only means of connecting program units

* The interface is enforced

* The body of a program unit is not accessible to other program units

* There is a clear distinction between architecture and implementation
SEPARATE COMPILATION

* Program units may be separately compiled
* Separate compilation is possible because of the separation of specification and body
* A system is put together by referencing the specifications of other program units
SEPARATE COMPILATION

* A program unit's specification may be compiled separately from its body.

* Realizes not only a logical distinction between architecture and implementation, but also a physical distinction.
SEPARATE COMPILATION

* Allows development of independent software components

* Currently we all but lose the human effort going into software; it is disposable

* Separate compilation allows us to reuse components and keep our investment
DISCRETE COMPONENTS

* Allow a system to be composed of black boxes

* Provide clear, understandable functions

* Black boxes can be more effectively validated and verified

* Prevalent across engineering disciplines
* Gives ability to express abstract actions

* A black box

* The basic discrete component which acts like

* Mechanism to pass data to and from the subprogram

* Contains an interface (parameter part)

* Functions

* Procedures

* A program unit that performs a particular action

SUBPROGRAMS
MAJOR FEATURES OF ADA

* Packages
* Strong Typing
* Typing Structures
* Data Abstraction

* Tasks
* Exceptions
* Generics
PACKAGES

* Definition
* Components of a Package
  -- Specification
  -- Body
* Goals and Principles of Software Engineering Supported
PACKAGES

* Program units that allow us to collect logically related entities in one physical place
* Allow the definition of reusable software components/resources
* A fundamental feature of Ada which allow a change of mindset
* An architecture-oriented feature
PACKAGES

* Place a "wall" around resources
* Export resources to users of a package
* May contain local resources hidden from the user of a package
Program Units

package ROBOT_CONTROL is
    type SPEED is range 0..100;
    type DISTANCE is range 0..500;
    type DEGREES is range 0..359;
    procedure GO_FORWARD ( HOW_FAST : in SPEED;
                            HOW_FAR : in DISTANCE );
    procedure REVERSE ( HOW_FAST : in SPEED;
                        HOW_FAR : in DISTANCE );
    procedure TURN ( HOW_MUCH : in DEGREES );
end ROBOT_CONTROL;
with ROBOT_CONTROL;

procedure DO_A_SQUARE is
begin

    ROBOT_CONTROL.GO_FORWARD( HOW_FAST => 100,
                               HOW_FAR => 20);
    ROBOT_CONTROL.TURN( 90 );
    ROBOT_CONTROL.GO_FORWARD( 100, 20 );
    ROBOT_CONTROL.TURN( 90 );
    ROBOT_CONTROL.GO_FORWARD( 100, 20 );
    ROBOT_CONTROL.TURN( 90 );
    ROBOT_CONTROL.GO_FORWARD( 100, 20 );
    ROBOT_CONTROL.TURN( 90 );

end DO_A_SQUARE;
Program Units

Package bodies

-- Define local declarations
-- Define implementation of subprograms
-- defined in specification
package body ROBOT_CONTROL is
  -- local declarations
  procedure RESET_SYSTEM is
  begin
    -- implementation
  end RESET_SYSTEM;
  procedure GO_FORWARD...is...
  procedure REVERSE...is...
  procedure TURN...is...
end ROBOT_CONTROL;
PACKAGES

DIRECTLY SUPPORT:

* Abstraction
* Information hiding
* Modularity
* Localization

* Understandability
* Efficiency
* Reliability and safety
* Modifiability
* Correctness
STRONG TYPING

* Raw Materials for Software Engineering
* Effects of Strong Typing
* Goals and Principles of Software Engineering Supported
THE RAW MATERIALS OF ENGINEERING

* All engineering disciplines shape raw materials into a finished product

* The materials and methods combine to define different disciplines
STRUCTURING RAW MATERIALS

* There is a requirement to structure raw materials
  - To quantify
  - To manage
  - To test
  - To validate

* Methods of structuring vary across disciplines
SOME RAW MATERIALS OF SOFTWARE ENGINEERING

* Binary switches
* Computer memory locations
* Data
STRONG TYPING

* Defines structure of data (mapping)
* Enforces structure of data
STRONG TYPING

* Enforces abstraction of structure on data
* Increases confidence of correctness
* Increases reliability and safety
* Promotes understandability and maintainability
Types

-- A type consists of a set of values that objects of the type may take on, and a set of operations applicable to those values

-- Ada is a strongly typed language!
* Every object must be declared of some type name
* Different type names may not be implicitly mixed
* Operations on a type must preserve the type

    AN INTEGER : INTEGER;
    A FLOAT NUMBER : FLOAT;
    ANOTHER FLOAT : FLOAT;

    A FLOAT NUMBER := ANOTHER FLOAT + AN INTEGER;
    -- illegal
**Typing Structures**

* **Discrete Data Types**
  -- Enumeration
  -- Integer

* **Real Data Types**
  -- Fixed Point (Absolute Error)
  -- Floating Point (Relative Error)

* **Composite Types**
  -- Arrays (Homogeneous)
  -- Records (Heterogeneous)

* **Dynamic Types**
  -- Access Types

* **Abstract Data Types**
  -- Private
  -- Limited Private
TYING STRUCTURES

* Variety of problems requires a variety of structuring capabilities

* Ada provides a rich variety or types
TYPING STRUCTURES IN Ada

* Discrete data
  - Enumeration
  - Integer

* Real data
  - Fixed point (absolute error)
  - Floating point (relative error)

* Composite data
  - Arrays (homogeneous)
  - Records (heterogeneous)

* Dynamic data
  - Access types
Types

--- Define a set of exact, consecutive values

Integers

-- PLANES_HEIGHT is range 0..100,000;
-- DEPTH is range 0..20,000;

begin
PLANES_HEIGHT := 10,000;
USER_DEFINED
end;

begin
PLANES_HEIGHT := 200,000; 
--- error

PLANES_HEIGHT := DIVER.DEPTH;
DIVER.DEPTH := DEPTH;

P.
Types

Enumeration

--- Define a set of ordered enumeration values
--- Used in array indexing, case statements,
--- and looping

USER DEFINED

type SUIT is (CLUBS, HEARTS, DIAMONDS, SPADES);
type COLOR is (RED, WHITE, BLUE);
type SWITCH is (OFF, ON);
type EVEN DIGITS is ('2','4','6','8');
type MIXED is (ONE,'2',THREE,'*','!',more);

where CLUBS < HEARTS < DIAMONDS < SPADES
Types

Fixed point types

-- Absolute bound on error
-- Larger error for smaller numbers (around zero)

USER DEFINED

type INCREMENT is delta 1.0/8 range 0.0 .. 1.0;

0, 1*2e-3, 2*2e-3, 4*2e-3, 5*2e-3,...

PREDEFINED

DURATION --> (Used for "delay" statements)
Types

Floating point types

--- Relative bound of error
--- Defined in terms of significant digits
--- More accurate at smaller numbers, less at larger

USER DEFINED

type NUMBERS is digits 3 range 0.0 .. 20_000;

0.001, 0.002, 0.003...999.0,1000.0,1001.0...,10000.0,10100.0

PREDEFINED

FLOAT
Arrays

TYPE HOURS IS RANGE 0..40;

TYPE DAYS IS (SUN, MON, TUE, WED, THU, FRI, SAT);

TYPE WORK_HOURS IS ARRAY (DAYS) OF HOURS;

MY_HOURS : WORK_HOURS := (0, 8, 8, 7, 6, 1, 0);
Types
Records

UNDISCRIMINATED

type DAYS is (MON,TUE,WED,THU,FRI,SAT,SUN);
type DAY is range 1..31;
type MONTH is (JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,
   SEP,OCT,NOV,DEC);
type YEAR is range 0..2085;
type DATE is record
   DAY_OF_WEEK : DAYS;
   DAY_NUMBER : DAY;
   MONTH_NAME : MONTH;
   YEAR_NUMBER : YEAR;
end record;
TODAY : DATE;
begin
TODAY.DAY_OF_WEEK := TUE;
TODAY.DAY_NUMBER := 26;
TODAY.MONTH_NAME := NOV;

TODAY

<table>
<thead>
<tr>
<th>DAY_OF_WEEK</th>
<th>TUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAY_NUMBER</td>
<td>26</td>
</tr>
<tr>
<td>MONTH_NAME</td>
<td>NOV</td>
</tr>
<tr>
<td>YEAR_NUMBER</td>
<td>1985</td>
</tr>
</tbody>
</table>
DATA ABSTRACTION

* Definition

* Goals and Principles of Software Engineering Supported

* Baskin-Robbins Ice Cream Example
DATA ABSTRACTION

* Combines primitive raw materials to form higher level structures

* Levels of abstraction

* Enforces an abstraction on a higher level structure

* Prohibits use of implementation details

* Promotes understandability

* Promotes modifiability
DATA ABSTRACTION AND PRIVATE TYPES

* Private types directly implement data abstraction

* Directly implement information hiding
package B_R is

    type NUMBERS is range 0 .. 99;

    procedure TAKE ( A_NUMBER : out NUMBERS );

    function NOW_SERVING return NUMBERS;

    procedure SERVE ( NUMBER : NUMBERS );

end B_R;
with B_R; use B_R;
procedure ICE_CREAM is
  YOUR_NUMBER : NUMBERS;
begin
  TAKE ( YOUR_NUMBER );
  loop
    if NOW_SERVING = YOUR_NUMBER then
      SERVE ( YOUR_NUMBER );
      exit;
    end if;
  end loop;
end ICE_CREAM;
with B_R; use B_R;
procedure ICE_CREAM is

    YOUR_NUMBER : NUMBERS;

begin
    TAKE ( YOUR_NUMBER );
    loop
        if NOW_SERVING = YOUR_NUMBER then
            SERVE ( YOUR_NUMBER );
            exit;
        else
            YOUR_NUMBER := YOUR_NUMBER - 1;
        end if;
    end loop;

end ICE_CREAM;
package B_R is

    type NUMBERS is private;

    procedure TAKE ( A_NUMBER : out NUMBERS );
    function NOW_SERVING return NUMBERS;
    procedure SERVE ( NUMBER : in NUMBERS );

    private

    type NUMBERS is range 0..99;

end B_R;
with B_R; use B_R;
procedure ICE_CREAM is
    YOUR_NUMBER : NUMBERS;
begin
    TAKE ( YOUR_NUMBER );
    loop
        if NOW_SERVING = YOUR_NUMBER then
            SERVE ( YOUR_NUMBER );
            exit;
        else
            YOUR_NUMBER := NOW_SERVING;
        end if;
    end loop;
end ICE_CREAM;
package B_R is

  type NUMBERS is limited private;

  procedure TAKE ( A_NUMBER : out NUMBERS );
  function NOW_SERVING return NUMBERS;
  procedure SERVE ( NUMBER : in NUMBERS );
  function "=" ( LEFT, RIGHT : in NUMBERS ) return BOOLEAN;

private

  type NUMBERS is range 0..99;

end B_R;
with B_R; use B_R;
procedure ICE_CREAM is

    YOUR_NUMBER : NUMBERS;
    procedure GO_TO_DQ is separate;

begin
    TAKE ( YOUR_NUMBER );
    loop
        if NOW_SERVING = YOUR_NUMBER then
            SERVE ( YOUR_NUMBER );
            exit;
        else
            GO_TO_DQ;
            exit;
        end if;
    end loop;
end ICE_CREAM;
TASKS

* Definition

* Goals and Principles of Software Engineering Supported

* Example
TASKS

* Program unit that acts in parallel with other entities

* Directly implements those parts of embedded systems which act in parallel

* Takes advantage of move toward parallel hardware architectures
  - Fault tolerance
  - Distributed systems

* Eliminates need to introduce additional complexity into a system
Tasks

procedure SENSOR_CONTROLLER is

  function OUT_OF_LIMITS return BOOLEAN;
  procedure SOUND_ALARM;

  task MONITOR_SENSOR; -- specification
  task body MONITOR_SENSOR is -- body
  begin
    loop
      if OUT_OF_LIMITS then
        SOUND_ALARM;
      end if;
    end loop;
  end MONITOR_SENSOR;

  function OUT_OF_LIMITS return BOOLEAN is separate;
  procedure SOUND_ALARM is separate;
  begin
    null; -- Task is activated here
  end SENSOR_CONTROLLER;
Tasks

--- a basic task with no communication

with TEXT_IO; use TEXT_IO;
procedure COUNT_NUMBERS is
  package INT_IO is new INTEGER_IO (INTEGER);
  use INT_IO;
  task COUNT_SMALL;
  task COUNT_LARGE;

  task body COUNT_SMALL is
    begin
      for INDEX in -100..0 loop
        PUT(INDEX);
        NEW_LINE;
      end loop;
    end COUNT_SMALL;

  task body COUNT_LARGE is
    begin
      for INDEX in 0..100 loop
        PUT(INDEX);
        NEW_LINE;
      end loop;
    end COUNT_LARGE;

    begin
      null;  -- tasks are started here
    end COUNT_NUMBERS;
EXCEPTIONS

* Definition
* Goals and Principles of Software Engineering Supported
* Types of Exceptions in Ada
  -- Pre-defined Exceptions
  -- User-defined Exceptions
* Example
SOFTWARE RELIABILITY AND SAFETY

* Errors will occur
  - Hardware
  - Software

* Real time systems must be able to operate in a degraded mode

* Reliability and safety must be engineered into a system

* Traditional languages lack specific features for dealing with errors and exceptional situations
EXCEPTIONS

* Deal specifically with errors and exceptional situations

* When an exception is raised processing is suspended and control is passed to an appropriate exception handler
  - Try again
  - Fix error
  - Propogate exception

* Increase reliability

* Reduce complexity
Exceptions

--- Real time systems must have the ability to handle error situations to be reliable

--- Exceptions deal with exceptional situations
Exceptions

--- When an exception situation occurs, the exception is said to be "raised"

--- What happens then, depends on the presence or absence of an exception handler

begin
    loop
        GET ( A_NUMBER );
        NEW_LINE;
        PUT("The number is");
        PUT ( A_NUMBER );
        NEW_LINE;
    end loop;
end GET_NUMBERS;
Exceptions

begin
  loop
    begin
      begin
        GET ( A_NUMBER );
        NEW_LINE;
        PUT ( "The number is ");
        PUT ( A_NUMBER );
        NEW_LINE;
      exception
        when DATA_ERROR => PUT_LINE("Bad number, try again");
      end;
    end loop;
  end begin GET_NUMBERS;
GENERICS

* Definition
* Goals and Principles of Software Engineering Supported
* Example of Generic Unit Use
GENERICS

* A generic is a tailorable template for a program unit

* Increases reusable software component capability by an order of magnitude
GENERICS

* Reduce size of program text

* Reduce need to reinvent the wheel

* Increase reliability by allowing reuse of known reliable components
Generics

procedure INTEGER_SWAP (FIRST_INTEGER, SECOND_INTEGER)
in out INTEGER)
is

 TEMP : INTEGER;
 TEMP := FIRST_INTEGER;
 FIRST_INTEGER := SECOND_INTEGER;
 SECOND_INTEGER := TEMP;

begin

end INTEGER_SWAP;

Generics

generic

type ELEMENT is private;
procedure SWAP (ITEM_1,ITEM_2:in out ELEMENT);

procedure SWAP (ITEM_1,ITEM_2:in out ELEMENT) is

    TEMP : ELEMENT;
begin
    TEMP := ITEM_1;
    ITEM_1 := ITEM_2;
    ITEM_2 := TEMP;

end SWAP;
Generics

with SWAP;

procedure EXAMPLE is
  procedure INTEGER_SWAP is new SWAP(INTEGER);
  procedure CHARACTER_SWAP is new SWAP(CARACTER);

  NUM_1, NUM_2 : INTEGER;
  CHAR_1, CHAR_2 : CHARACTER;

begin
  NUM_1 := 10;
  NUM_2 := 25;
  INTEGER_SWAP(NUM_1, NUM_2);
  CHAR_1 := 'A';
  CHAR_2 := 'S';
  CHARACTER_SWAP(CHAR_1, CHAR_2);
end EXAMPLE;
SUMMARY

* Basic Problem
  -- Projection to the 1990's
  -- A Macro Solution
* A Practical Solution
  -- Software Engineering
  -- Ada
* Software Engineering
  -- Goals
  -- Principles
* Why Ada?
  -- Features of Ada
  -- Software Engineering Applications
PACKAGES

DIRECTLY SUPPORTS:

* ABSTRACTION
* MODULARITY
* LOCALIZATION
* INFORMATION HIDING
  UNIFORMITY
* COMPLETENESS
* CONFIRMABILITY

* MODIFIABILITY
* RELIABILITY
* EFFICIENCY
* UNDERSTANDABILITY
TYPING

DIRECTLY SUPPORTS:

ABSTRACTION
MODULARITY
LOCALIZATION
INFORMATION HIDING
UNIFORMITY

* COMPLETENESS
* CONFIRMABILITY

* MODIFIABILITY
* RELIABILITY
* EFFICIENCY
* UNDERSTANDABILITY
DATA ABSTRACTION

DIRECTLY SUPPORTS:

* ABSTRACTION
* MODULARITY
* LOCALIZATION
* INFORMATION HIDING
* COMPLETENESS
* CONFIRMABILITY
* MODIFIABILITY
* RELIABILITY
* EFFICIENCY
* UNDERSTANDABILITY
EXCEPTIONS

DIRECTLY SUPPORTS:

ABSTRACTION
* MODULARITY
* LOCALIZATION
  INFORMATION HIDING
* UNIFORMITY
* COMPLETENESS
* CONFIRMABILITY

MODIFIABILITY
* RELIABILITY
* EFFICIENCY
* UNDERSTANDABILITY
TASKS

DIRECTLY SUPPORTS:

* ABSTRACTION
* MODULARITY
* LOCALIZATION
* INFORMATION HIDING
* UNIFORMITY
* COMPLETENESS
* CONFIRMABILITY

* MODIFIABILITY
* RELIABILITY
* EFFICIENCY
* UNDERSTANDABILITY
GENERICS

DIRECTLY SUPPORTS:

* ABSTRACTION
* MODULARITY
* LOCALIZATION
  INFORMATION HIDING
* UNIFORMITY
* COMPLETENESS
* CONFIRMABILITY

* MODIFIABILITY
* RELIABILITY
* EFFICIENCY
* UNDERSTANDABILITY
Tutorial on Ada® Exceptions

by

Major Patricia K. Lawlis

lawlis%asu@csnet-relay

Air Force Institute of Technology (AFIT)
and
Arizona State University (ASU)

13 January 1988

© Ada is a registered trademark of the U. S. Government - Ada Joint Program Office
References


Outline

=> Overview

- Naming an exception
- Creating an exception handler
- Raising an exception
- Handling exceptions
- Turning off exception checking
- Tasking exceptions
- More examples
- Summary
Overview

• What is an exception

• Ada exceptions

• Comparison
  - the American way
  - using exceptions
What Is an Exception

• A run time error

• An unusual or unexpected condition

• A condition requiring special attention

• Other than normal processing

• An important feature for debugging

• A critical feature for operational software
Ada Exceptions

- An exception has a name
  - may be predefined
  - may be declared

- The exception is raised
  - may be raised implicitly by run time system
  - may be raised explicitly by raise statement

- The exception is handled
  - exception handler may be placed in any frame*
  - exception propagates until handler is found
  - if no handler anywhere, process aborts

* executable part surrounded by begin - end
package Stack_Package is

    type Stack_Type is limited private;

    procedure Push (Stack : in out Stack_Type;
                        Element : in   Element_Type;
                        Overflow_Flag : out BOOLEAN);

end Stack_Package;

with TEXT_IO;
with Stack_Package; use Stack_Package;
procedure Flag_Waving is

    Stack : Stack_Type;
    Element : Element_Type;
    Flag    : BOOLEAN;

begin

    Push (Stack, Element, Flag);
    if Flag then
        TEXT_IO.PUT ("Stack overflow");
    end if;

end Flag_Waving;
package Stack_Package is

    type Stack_Type is limited private;
    Stack_Overflow,
    Stack_Underflow : exception;

    procedure Push (Stack in out Stack_Type;
                    Element : in   Element_Type);
                    -- may raise Stack_Overflow

    end Stack_Package;

with TEXT_IO;
with Stack_Package; use Stack_Package;
procedure More_Natural is

    Stack : Stack_Type;
    Element : Element_Type;

    begin

        Push (Stack, Element);

        exception
            when Stack_Overflow =>
                TEXT_IO.PUT ("Stack overflow");

    end More_Natural;
Outline

• Overview

=> Naming an exception

• Creating an exception handler

• Raising an exception

• Handling exceptions

• Turning off exception checking

• Tasking exceptions

• More examples

• Summary
Naming an Exception

- Predefined exceptions
- Declaring exceptions
- I/O exceptions
Predefined Exceptions

• In package STANDARD (also see chap 11 of LRM)

• CONSTRAINT_ERROR
  
  violation of range, index, or discriminant constraint...

• NUMERIC_ERROR
  
  execution of a predefined numeric operation cannot deliver a correct result

• PROGRAM_ERROR
  
  attempt to access a program unit which has not yet been elaborated...

• STORAGE_ERROR
  
  storage allocation is exceeded...

• TASKING_ERROR
  
  exception arising during intertask communication
Declaring Exceptions

exception_declaration ::= identifier_list : exception;

- Exception may be declared anywhere an object declaration is appropriate

- However, exception is not an object
  - may not be used as subprogram parameter, record or array component
  - has same scope as an object, but its effect may extend beyond its scope

Example:

procedure Calculation is

    Singular : exception;
    Overflow, Underflow : exception;

begin
    ...
end Calculation;
I/O Exceptions

• Exceptions relating to file processing

• In predefined library unit IO_EXCEPTIONS
  (also see chap 14 of LRM)

• TEXT_IO, DIRECT_IO, and SEQUENTIAL_IO with it

package IO_EXCEPTIONS is

  NAME_ERROR : exception;   --attempt to use
  USE_ERROR : exception;     --invalid operation
  STATUS_ERROR : exception;  --attempt to read
  MODE_ERROR : exception;    --beyond end of file
  DEVICE_ERROR : exception;  --attempt to input
  END_ERROR : exception;     --wrong type
  DATA_ERROR : exception;    --for text processing
  LAYOUT_ERROR : exception;

end IO_EXCEPTIONS;
Outline

- Overview

- Naming an exception

=> Creating an exception handler

- Raising an exception

- Handling exceptions

- Turning off exception checking

- Tasking exceptions

- More examples

- Summary
Creating an Exception Handler

- Defining an exception handler
- Restrictions
- Handler example
Defining an Exception Handler

- Exception condition is "caught" and "handled" by an exception handler

- Exception handler may appear at the end of any frame (block, subprogram, package or task body)

```
begin
    ...
    exception
        -- exception handler(s)
    end;
```

- Form similar to case statement

```
exception_handler ::= 
    when exception_choice { | exception_choice} =>
        sequence_of_statements

exception_choice ::= exception_name | others
```
Restrictions

• Exception handlers must be at the end of a frame

• Nothing but exception handlers may lie between exception and end of frame

• A handler may name any visible exception declared or predefined

• A handler includes a sequence of statements
  - response to exception condition

• A handler for others may be used
  - must be the last handler in the frame
  - handles all exceptions not listed in previous handlers of the frame (including those not in scope of visibility)
  - can be the only handler in the frame
procedure Whatever is

    Problem.Condition : exception;

begin

    ...

exception

    when Problem.Condition =>
        Fix.lt;

    when CONSTRAINT_ERROR =>
        Report.lt;

    when others =>
        Punt;

end Whatever;
Outline

- Overview

- Naming an exception

- Creating an exception handler

=> Raising an exception

- Handling exceptions

- Turning off exception checking

- Tasking exceptions

- More examples

- Summary
Raising an Exception

- Elaboration and execution exceptions
- How exceptions are raised
- Effects of raising an exception
- Raising example
Elaboration and Execution Exceptions

- Elaboration exceptions occur when declarations are being elaborated
  - after a unit is "called"
  - before execution of the unit begins
  - can only be predefined exceptions

- Execution exceptions occur during execution of a frame

- Elaboration exceptions can also be considered as execution exceptions
  - depending on viewpoint
  - can consider as part of the execution of the last executable statement making the call to the unit being elaborated
    - this helps with understanding the consistency of the rules for exception handling
How Exceptions are Raised

- Implicitly by run time system
  - predefined exceptions

- Explicitly by `raise` statement

  ```
  raise_statement ::= raise [exception_name];
  ```

  - the name of the exception must be visible at the point of the raise statement
  - a raise statement without an exception name is allowed only within an exception handler
Effects of Raising an Exception

(1) Control transfers to exception handler at end of frame being **executed** (if handler exists)

(2) Exception is lowered

(3) Sequence of statements in exception handler is executed

(4) Control passes to end of frame

- If frame does not contain an appropriate exception handler, the exception is propagated - effectively skipping steps 1 thru 3 and going straight to step 4
procedure Whatever is

    Problem_Condition : exception;
    Real_Bad_Condition : exception;

begin

    ... 
    if Problem_Arises then 
        raise Problem_Condition; 
    end if;
    ...
    if Serious_Problem then 
        raise Real_Bad_Condition;
    end if;
    ...
    exception

        when Problem_Condition => 
            Fix_It;
        ...

        when CONSTRAINT_ERROR => 
            Report_It;
        ...

        when others => 
            Punt;
       ...
end Whatever;
Outline

• Overview

• Naming an exception

• Creating an exception handler

• Raising an exception

=> Handling exceptions

• Turning off exception checking

• Tasking exceptions

• More examples

• Summary
Handling Exceptions

- How exception handling can be useful
- Which exception handler is used
- Sequence of statements in exception handler
- Propagation
- Propagation example
How Exception Handling Can Be Useful

- Normal processing could continue if
  - cause of exception condition can be "repaired"
  - alternative approach can be used
  - operation can be retried

- Degraded processing could be better than termination
  - for example, safety-critical systems

- If termination is necessary, "clean-up" can be done first
Which Exception Handler Is Used

- When exception is raised, system looks for an exception handler at the end of the frame being executed

- If exception is raised during elaboration of the declarative part of a unit (unit is not yet ready to execute)
  - elaboration is abandoned and control goes to the end of the unit with the exception still raised
  - exception part of the unit is not searched for an appropriate handler
  - effectively, the calling unit will be searched for an appropriate handler
    -- consistent with execution viewpoint
  - if elaboration of library unit, program execution is abandoned
    -- all library units are elaborated with the main program

- If exception is raised in exception handler
  - handler may contain block(s) with handler(s)
  - if not handled locally within handler, control goes to end of frame with exception raised
Propagation

- Occurs if no handler exists in frame where execution exception is raised
- Always occurs if elaboration exception is raised
- Also occurs if `raise` statement is used in handler
- Exception is propagated dynamically
  - propagates from subprogram to unit calling it (not necessarily unit containing its declaration)
  - this can result in propagation outside its scope
  - task propagation follows same principle, but a little more complicated
- Propagation continues until
  - an appropriate handler is found
  - exception propagates to main program (still with no handler) and program execution is abandoned
procedure Do_Nothing is
  begin
    Calls_It;
  exception
    when others => Fix_Everything;
end Do_Nothing;

procedure Has_It is
  Some_Problem : exception;
  begin
    ... raise Some_Problem;
    ... exception
      when Some_Problem =>
        Clean_Up;
        raise;
  end Has_It;

procedure Calls_It is
  begin
    ... Has_It;
    ... end Calls_It;
end Calls_It;
Outline

- Overview

- Naming an exception

- Creating an exception handler

- Raising an exception

- Handling exceptions

=> Turning off exception checking

- Tasking exceptions

- More examples

- Summary
Turning Off Exception Checking

- Overhead vs efficiency
- Pragma SUPPRESS
- Check identifiers
Overhead vs Efficiency

• Exception checking imposes run time overhead
  - interactive applications will never notice
  - real-time applications have legitimate concerns
    but must not sacrifice system safety

• When efficiency counts
  - first, make program work (using good design)
  - be sure possible problems are covered by exception handlers
  - check if efficient enough - stop if it is
    - if not, study execution profile
      -- eliminate bottlenecks
      -- improve algorithm
      -- avoid "cute" tricks
    - check if efficient enough - stop if it is
  - if not, trade-offs may be necessary
  - some exception checks may be expendable since debugging is done
    - however, every suppressed check poses new possibilities for problems
      -- must re-examine possible problems
      -- must re-examine exception handlers
  - always keep in mind
    -- problems will happen
    -- critical applications must be able to deal with these problems
Improving the design is far better - and easier in the long run - than suppressing checks

Moral
Pragma SUPPRESS

- Only allowed immediately within a declarative part or immediately within a package specification

pragma SUPPRESS (identifier [, [ ON => ] name]);

- identifier is that of the check to be omitted (next slide lists identifiers)

- name is that of an object, type, or unit for which the check is to be suppressed

  -- if no name is given, it applies to the remaining declarative region

- An implementation is free to ignore the suppress directive for any check which may be impossible or too costly to suppress

Example:

pragma SUPPRESS (INDEX_CHECK, ON => Index);
Check Identifiers

- These identifiers are explained in more detail in chap 11 of the LRM

- Check identifiers for suppression of CONSTRAINT_ERROR checks
  
  ACCESS_CHECK
  DISCRIMINANT_CHECK
  INDEX_CHECK
  LENGTH_CHECK
  RANGE_CHECK

- Check identifiers for suppression of NUMERIC_ERROR checks
  
  DIVISION_CHECK
  OVERFLOW_CHECK

- Check identifier for suppression of PROGRAM_ERROR checks
  
  ELABORATION_CHECK

- Check identifier for suppression of STORAGE_ERROR check
  
  STORAGE_CHECK
Outline

- Overview

- Naming an exception

- Creating an exception handler

- Raising an exception

- Handling exceptions

- Turning off exception checking

=> Tasking exceptions

- More examples

- Summary
Tasking Exceptions

- Exception handling is trickier for tasks
- Exceptions during task communication
- Tasking example
Exception Handling Is Trickier for Tasks

- Rules are not really different, just more involved
  - local exceptions handled the same within frames

If exception is raised

- during elaboration of task declarations
  - the exception TASKING_ERROR will be raised at the point of task activation (becomes execution exception in enclosing subprogram)
  - the task will be marked completed

- during execution of task body (and not resolved there)
  - task is completed
  - exception is not propagated

- during task rendezvous
  - this is the really tricky part
Exceptions During Task Communication

- If the called task terminates abnormally
  exception TASKING_ERROR is raised in calling task at the point of the entry call

- If an entry call is made for entry of a task that becomes completed before accepting the entry
  exception TASKING_ERROR is raised in calling task at the point of the entry call

- If the calling task terminates abnormally
  no exception propagates to the called task

- If an exception is raised in called task within an accept (and not handled there locally)
  the same exception is raised in the calling task at the point of the entry call
  (even if exception is later handled outside of the accept in the called task)
procedure Critical_Code is

Failure : exception;
-----------
task Monitor is
    entry Do_Something;
end Monitor;
task body Monitor is
...
begin
    accept Do_Something do
      ...
      raise Failure;
      ...
      end Do_Something;
    ...
exception  -- exception handled here
    when Failure =>
      Termination_Message;
end Monitor;
-----------
begin  -- Critical_Code
    ...
    Monitor.Do_Something;
    ...
exception  -- same exception will be handled here
    when Failure =>
      Critical_Problem_Message;
end Critical_Code;
Outline

- Overview
- Naming an exception
- Creating an exception handler
- Raising an exception
- Handling exceptions
- Turning off exception checking
- Tasking exceptions

=> More examples

- Summary
More Examples

- Interactive data input

- Propagating exception out of scope and back in

- Keeping a task alive
with TEXT_IO; use TEXT_IO;
procedure Get_Input (Number : out integer) is

    subtype Input_Type is integer range 0..100;
    package Int_io is new INTEGER_IO (Input_Type);
    In_Number : Input_Type;

    begin -- Get_Input

        loop -- to try again after incorrect input

            begin -- inner block to hold exception handler

                put ("Enter a number 0 to 100");
                Int_io.GET (In_Number);
                Number := In_Number;
                exit; -- to exit loop after correct input

            exception

                when DATA_ERROR =>
                    put ("Try again, fat fingers!");
                    Skip_Line; -- must clear buffer

            end; -- inner block

        end loop;

    end Get_Input;
Propagating Exception Out of Scope and Back In

declare
package Container is
    procedure Has_Handler;
    procedure Raises_Exception;
end Container;

procedure Not_in_Package is
begin
    Container.Raises_Exception;
exception
    when others => raise;
end Not_in_Package;

package body Container is
    Crazy : exception;
    procedure Has_Handler is
begin
    Not_in_Package;
exception
    when Crazy => Tell_Everyone;
end Has_Handler;
    procedure Raises_Exception is
begin
    raise Crazy;
end Raises_Exception;
end Container;
begin
    Container.Has_Handler;
end;
task Monitor is
entry Do_Something;
end Monitor;

task body Monitor is
begin
loop -- for never-ending repetition
    ...
    select
        accept Do_Something do
            begin -- block for exception handler
                ...
                raise Failure;
                ...
                exception
                    when Failure => Recover;
                end; -- block
            end Do_Something; -- exception must be
                -- lowered before exiting
            ...
        end select;
    ...
end loop;

exception
    when others =>
        Termination_Message;
end Monitor;
Outline

• Overview

• Naming an exception

• Creating an exception handler

• Raising an exception

• Handling exceptions

• Turning off exception checking

• Tasking exceptions

• More examples

=> Summary
Summary

• Exception handling principles are consistent

• Suppression of exception checking will usually do more harm than good

• Use of exceptions must become a habit to be useful
Ada* Tasking

Abstraction of Process

Captain David A. Cook
U.S. Air Force Academy
Ada Tasking

- Overview

Define Ada Tasking

Define Synchronization Mechanism

Examples
Task Definition

A Program Unit for Concurrent Execution

Never a Library Unit

Master is a...

Library Package

Subprogram

Block Statement

Other Task
Ada Tasking

Synchronization Mechanisms

- Global Variables
- Rendezvous

Main Program IN A Task

Caller Requests Service

1. Immediate Request
2. Wait for a While
3. Wait Forever
Callee Provides Service

1. Immediate Response
2. Wait for a while
3. Wait forever

Service is requested with an entry call statement

Service is provided with an accept statement
Ada Tasking

Select statements provide ability to program the different request and provide modes.

Guards are "If statements" for providing service [True or False condition]

Termination is an alternative if a service is no longer needed.
TASK MASTERS

Each task must depend on a master

A master can be a task, a currently executing block statement, a currently executing subprogram, or a library package.

Packages declared inside another program unit cannot be masters.

The master of a task is determined by the creation of the task object.

A block, task, or subprogram cannot be left until all of its dependents are terminated.
Actually, the 1815A does not define if tasks that depend on library packages are required to terminate!!
WHEN DOES A TASK START?

Tasks are activated after the elaboration of the declarative part.

Effectively, activation is after the declarative part, and immediately after the 'BEGIN' statement, but before any other statement.

The purpose of this is to allow the exception handler to service task exception.
Task type T1 is ....

Obj : T1;

begin
  declare
    New_Obj : T1;

  begin
    null;
  end;

  ...
  ...
end;
TASKS OBJECTS ACCESSED BY ALLOCATORS DO THINGS A LITTLE BIT DIFFERENTLY

Normally, the scope of a task object determines its master

For an access type, the master is determined by the access type definition

Activation for accessed tasks occurs immediately upon the assignment of a value to the access object
ELABORATION - Declarative part

RUNNING - Task has processor

READY - Task is available for processor, and has all resources to run

BLOCKED - Task is either waiting for a call, or waiting for call to be answered

COMPLETED - At end, or exception

TERMINATED - Completed, and dependent tasks also terminated

ABNORMAL - Task was aborted
task [type] [is
  {entryDeclaration}
  {representationClause}
end [task_simple_name] ]

  task body task_simple_name is
  {declarative_part}
  begin
  {sequence_of_statements}
  {exception
    exception_handler
    {exception_handler}}
end [task_simple_name];
ACCEPT STATEMENT

The Accept statement allows an unknown caller to call an entry.

There can be in and/or out parameters

The construct is 'accept.....do'

During the accept, the calling unit is suspended. Thus, a long accept slows down the system.

A good approach is to use the accept simply to copy in or out data, and allow the caller to continue.
SIMPLEST FORM OF TASK ENTRY

ACCEPT

TASK T1 IS
    ENTRY ENTRY1;
END T1;

TASK BODY T1 IS
    BEGIN
        LOOP
            ACCEPT ENTRY1 DO
                <SOS>
                END ENTRY1;
                <SOS>
            END LOOP;
        END
    END T1;

--WAIT FOREVER FOR CALL TO ENTRY1
TASK T1 IS
  ENTRY ACTION (DATA : SOME_TYPE);
END T1;

TASK BODY T1 IS

BEGIN
  LOOP
    ACCEPT ACTION(DATA:SOME_TYPE) DO
      --SOME LONG PROCESS USING DATA
      -- OCCURS HERE
    END ACTION;
  END LOOP;
END T1;

--NO EXITS OR GOTOS ALLOWED IN ACCEPT,
-- BUT A RETURN IS ALLOWED
TASK T1 IS
ENTRY ACTION (DATA : SOME_TYPE);
END T1;

TASK BODY T1 IS
LOCAL : SOME_TYPE;
BEGIN
LOOP
ACCEPT ACTION (DATA : SOME_TYPE) DO
LOCAL := DATA;
END ACTION;
--PUT PROCESS ON LOCAL HERE
END LOOP;
END T1;
--WHEN THIS CAN BE DONE, IT WILL SPEED
--UP THE SYSTEM.
TASK T1 IS
   ENTRY ACTION(DATA:A_TYPE);
   ENTRY RESULT(DATA :out A_TYPE);
END T1;

TASK BODY T1 IS
   LOCAL : A_TYPE;
BEGIN
   LOOP
      ACCEPT ACTION(DATA:A_TYPE) DO
         LOCAL := DATA;
      END ACTION;
      --PROCESS ON LOCAL HERE
      ACCEPT RESULT(DATA:out A_TYPE) DO
         DATA :
      END RESULT;
   END LOOP;
END T1;
Task T1 is
  entry ENTRY1;
end T1;

Task body T1 is
begin
  loop
    accept ENTRY1; -- 'sync' call only
    <SOS>
    end loop;
  end loop;
end T1;
-- wait forever for call to ENTRY1

-- even if ENTRY1 has parameters associated with
  -- it, the accept block does not have to have a
  -- sequence of statements
SELECT Statement

Used by the task to allow options

Simplest form is the selective wait (wait forever)

Task T1 is
  ENTRY ENTRY1;
  ENTRY ENTRY2;
END T1;

Task body T1 is
BEGIN
  LOOP
    SELECT
      ACCEPT ENTRY1 do
        <SOS>
        END ENTRY1;
        <SOS>
      or
      ACCEPT ENTRY2 do
        <SOS>
        END ENTRY2;
        <SOS>
      --as many 'or' and accept clauses as needed
    END SELECT;
    END LOOP;
  END T1;
--wait for either ENTRY1 or ENTRY2
SELECTIVE WAIT WITH ELSE (DON'T WAIT AT ALL)

TASK T1 IS
  ENTRY ENTRY1;
END T1;

TASK BODY T1 IS
BEGIN
  LOOP
    SELECT
      ACCEPT ENTRY1 DO
        <SOS>
      END ENTRY1;
      <SOS>
    ELSE
      <SOS>
    END SELECT;
  END LOOP;
END T1;

IF THERE IS NOT A CALLER WAITING RIGHT NOW,
DO THE ELSE PART.
SELECTIVE WAIT WITH ELSE, MULTIPLE ACCEPTS

TASK T1 IS
  ENTRY ENTRY1;
  ENTRY ENTRY2;
END T1;

TASK BODY T1 IS
BEGIN
  LOOP
    SELECT
      ACCEPT ENTRY1 DO
        <SOS>
        END ENTRY1;
        <SOS>
      OR
      ACCEPT ENTRY2 DO
        ...-- AS MANY 'OR' AND 'ACCEPT' CLAUSES AS NEEDED
      ELSE
        <SOS>;
      END SELECT;
    END LOOP;
  END LOOP;
END T1;
SELECT WITH DELAY ALTERNATIVE
(WAIT A FINITE TIME)

TASK BODY T1 IS
BEGIN
    LOOP
        SELECT
            ACCEPT ENTRY1 DO....
            [OR
                ACCEPT ENTRY2.....]
            OR
                DELAY 15.0; -- SECONDS
                <SOS>;
        END SELECT;
    END LOOP;
END T1;

IF ENTRY1 CALLED WITHIN 15 SECONDS,
THEN YOU ACCEPT THE CALL. OTHERWISE,
AFTER 15 SECONDS YOU WILL DO SOMETHING.
'DELAY' Rules

You may have several alternatives with a DELAY statement.

Since delays can be static, the shortest delay alternative will be selected.

Zero and negative delays are legal.

You may not have an else part with a DELAY, since the delay would never be accepted.
'DELAY' Rules

You may have several alternatives with a DELAY statement.

Since delays can be static, the shortest delay alternative will be selected.

Zero and negative delays are legal.

You may not have an ELSE part with a DELAY, since the delay would never be accepted.
SELECT WITH DELAY ALTERNATIVE (WAIT A FINITE TIME)

TASK BODY T1 IS
BEGIN
LOOP
SELECT
  ACCEPT ENTRY1 DO....
  [OR
   ACCEPT ENTRY2.....]
OR
  DELAY <EXPRESSION>;
  <SOS>;
OR
  DELAY <EXPRESSION>;
  <SOS>;
  --SHORTEST DELAY WILL GET CHOSEN
END SELECT;
END LOOP;
END T1;
Guards can be used on any accept statement

```plaintext

... ...

WHEN SOME_CONDITION =>
  ACCEPT ENTRY1 ......
```

If there is no guard, the accept statement is said to be open.

If there is a guard, and the when condition is true, the accept is also open.

False guard statements are said to be closed.

Open alternatives are considered. If there is more than one, then one is selected arbitrarily.

If there are no open alternatives (and no else part), the exception PROGRAM_ERROR is raised.
TERMINATION

WHEN A TASK HAS COMPLETED ITS SEQUENCE OF STATEMENTS, ITS STATUS IS COMPLETED.

ADDITIONALLY, THERE IS AN OPTION THAT ALLOWS A TASK TO TERMINATE.

```
SELECT
  ACCEPT ENTRY1 DO ..... [OR
  ACCEPT ENTRY2 DO .....]
OR
  TERMINATE;
END SELECT;
```

THIS MAY NOT BE USED WITH EITHER THE DELAY OR AN ELSE CLAUSE.

SINCE THIS IS USED ONLY WITH A 'WAIT FOREVER' ETASK, THIS OPTION ALLOWS A TASK THAT IS WAITING FOREVER TO TERMINATE IF ITS PARENT IS ALSO READY TO QUIT.
REMEMBER....

Tasks are Non-deterministic

select
    accept ENTRY1;

or

    accept ENTRY2;

Might always take ENTRY1!!!!
KILLING A TASK

Often, a 'terminate' alternative is not sufficient.

A parent may kill dependent tasks (or itself) using the abort statement.

This should only be used in very rare circumstances.

A better method is to use an entry to 'accept' a shutdown call.

If you have accepted a 'shutdown' call, then it is OK to abort yourself.
TASK BODY T1 IS
BEGIN
LOOP  -- THE ENDLESS LOOP OF THE
       -- TASK STARTS HERE
       -- EXIT LOOP TO TERMINATE
SELECT
       -- THE REQUIRED ACCEPT
       -- STATEMENTS ARE CODED HERE
OR
       ACCEPT SHUTDOWN;
       -- SPECIAL FINAL ACTIONS HERE
EXIT;  -- EXITS LOOP, ENDS TASK
OR
       TERMINATE;  -- FOR CASES WHERE
                    -- SHUTDOWN NOT CALLED
END SELECT;
END LOOP;
END T1;
PROBLEMS WITH PARALLELISM

MULTIPLE 'THREADS OF CONTROL' CAN CAUSE PROBLEMS IF TWO PROCESSES ARE TRYING TO ACCESS AND UPDATE ONE PIECE OF INFORMATION AT THE SAME TIME.

PRAGMA SHARED
   My-Object : Some-Type;
PRAGMA SHARED (My-Object);

ENFORCES MUTUALLY EXCLUSIVE ACCESS

ONLY WORKS FOR SCALAR AND ACCESS TYPES
Semaphores can also be used to control access to an object —promotes 'polling'

Encapsulating a data item within a task is a better method
TASK SEMAPHORE IS
  ENTRY P; --GET RESOURCE
  ENTRY V; --RELEASE
END SEMAPHORE;

TASK BODY SEMAPHORE IS
  AVAILABLE : BOOLEAN := TRUE;
BEGIN
  LOOP
    SELECT
         WHEN AVAILABLE
         ACCEPT P DO
           AVAILABLE := FALSE;
           END P;
      OR
         WHEN NOT AVAILABLE
         ACCEPT V DO
           AVAILABLE := TRUE;
           END V;
      OR
         TERMINATE;
    END LOOP;
  END LOOP;
END SEMAPHORE;
TASK SPECIAL_OPS IS
   ENTRY ASSIGN ( Object : in SOME_TYPE );
   ENTRY RETRIEVE ( Object : out SOME_TYPE );
END SPECIAL_OPS;

TASK BODY SPECIAL_OPS IS
   THE_OBJECT : SOME_TYPE;
BEGIN
   LOOP
      SELECT
         ACCEPT ASSIGN( Object : in SOME_TYPE ) do
            THE_OBJECT := Object;
         end ASSIGN;
      OR
         ACCEPT RETRIEVE( Object : out SOME_TYPE ) do
            OBJECT := THE_OBJECT;
         end RETRIEVE;
      OR
         TERMINATE;
      end SELECT;
   END LOOP;
END SPECIAL_OPS;
CALLING A TASK ENTRY

When you call a task, you must know the task name.

There are three types

Entry Calls (wait forever)

Timed Entry Calls (wait for specified time)

Conditional Entry Calls (don't wait at all)
CALL AND WAIT FOREVER

To call an entry, specify the task name and then the entry name.

BEGIN

   T1.ENTRY1(DATA);
TIMED ENTRY CALL  
(WAIT FOR A FINITE TIME)

SELECT  
T1.ENTRY1(DATA);  
<SOS>

OR  
DELAY 60;  
<SOS>
END SELECT;

YOU CANNOT USE AN 'OR' TO CALL TWO (OR MORE) TASK ENTRIES!!!

THIS WOULD BE EQUIVALENT TO STANDING IN TWO DIFFERENT LINES AT ONCE.
CONDITIONAL ENTRY CALLS
(DON'T WAIT AT ALL)

SELECT
  TI.ENTRY1(DATA);
  <SOS>
ELSE
  <SOS>
END SELECT;

NOTICE THE 'ORTHOGONALITY' OR THE SELECT STATEMENT. IT IS USED IN EITHER A TASK ENTRY CALL OR AN ACCEPT STATEMENT.

ALSO NOTICE THAT INSTEAD OF 'ACCEPT...BEGIN...END ACCEPT;
IT IS 'ACCEPT...DO....END ENTRY_NAME;
WHY???
TASK ATTRIBUTES

T'CALLABLE - RETURNS BOOLEAN VALUE
TRUE - TASK CALLABLE,
FALSE - TASK COMPLETED,
ABNORMAL OR TERMINATED

T'TERMINATED - BOOLEAN VALUE
TRUE IF TERMINATED

E'COUNT - RETURNS AN UNIVERSAL INTEGER INDICATING THE NUMBER OF ENTRY CALLS QUEUED FOR ENTRY E.

AVAILABLE ONLY WITHIN TASK T ENCLOSING E
TASK PRIORITIES

`pragma PRIORITY (static_expression);`

Used to represent degree of relative urgency.

If two tasks are READY, then the task with the higher priority runs.

Although priorities are static, task rendezvous are dynamic. When tasks are in rendezvous, the priority is the higher of the caller and the callee.
SYNCHRONIZATION OF DATA

TASK SYNC IS
  ENTRY UPDATE (DATA : IN DATA_TYPE);
  ENTRY READ (DATA : OUT DATA_TYPE);
END SYNC;

TASK BODY SYNC IS
  LOCAL : DATA_TYPE;
BEGIN
  LOOP
    SELECT
      ACCEPT UPDATE(DATA : IN DATA_TYPE) DO
        LOCAL := DATA;
      END UPDATE;
    OR
      TERMINATE;
    END SELECT;

    SELECT
      ACCEPT READ (DATA : OUT DATA_TYPE) DO
        DATA := LOCAL;
      END READ;
    OR
      TERMINATE;
    END SELECT;

  END LOOP;
END SYNC;
FAMILIES OF ENTR'ES

TYPE URGENCY IS (LOW, MEDIUM, HIGH);

TASK MESSAGE IS
ENTRY RECEIVE(URGENCY) (DATA : DATA_TYPE);
END MESSAGE;

TASK BODY MESSAGE IS
BEGIN
LOOP
SELECT
  ACCEPT RECEIVE(HIGH) (DATA:DATA_TYPE) DO
  ...
  END RECEIVE;
OR
  WHEN RECEIVE(HIGH)'COUNT = 0 =>
  ACCEPT RECEIVE(MEDIUM) (DATA:DATA_TYPE) DO
  ...
  END RECEIVE;
OR
  WHEN RECEIVE(HIGH)'COUNT+RECEIVE(MEDIUM)'COUNT=0 =>
  ACCEPT RECEIVE(LOW) (DATA:DATA_TYPE) DO
  ...
  END RECEIVE;
OR
  DELAY 1.0; -- SHORT WAIT
END MESSAGE;
SAME THING, WITH NO GUARDS

TYPE URGENCY IS (LOW, MEDIUM, HIGH);

TASK MESSAGE IS
ENTRY RECEIVE(URGENCY) (DATA : DATA_TYPE);
END MESSAGE;

TASK BODY MESSAGE IS
BEGIN
LOOP
SELECT
  ACCEPT RECEIVE(HIGH) (DATA:DATA_TYPE) DO
    ...
  END RECEIVE;
ELSE
  SELECT
    ACCEPT RECEIVE(MEDIUM) (DATA:DATA_TYPE) DO
      ...
    END RECEIVE;
ELSE
  SELECT
    ACCEPT RECEIVE(LOW) (DATA:DATA_TYPE) DO
      ...
    END RECEIVE;
  OR
    DELAY 1.0; -- SHORT WAIT
  END SELECT;
END SELECT;
END SELECT;
END MESSAGE;
REPRESENTATION SPECIFICATIONS

LENGTH CLAUSE

T'STOORAGE_SIZE

TASK TYPE T1 IS
  ENTRY ENTRY_1;
  FOR T1'STOORAGE_SIZE USE 2000*SYSTEM.STORAGE_UNIT);
END T1;

THE PREFIX T DENOTES A TASK TYPE.

THE SIMPLE EXPRESSION MAY BE STATIC, AND IS USED TO SPECIFY THE NUMBER OF STORAGE UNITS TO BE RESERVED OR FOR EACH ACTIVATION (NOT THE CODE) OF THE TASK.
Address Clause

`task type T1 is
    entry ENTRY_1;
    for T1 use at 16#167A#;
end T1;

In this case, the address specifies the actual location in memory where the machine code associated with T1 will be placed.

`task T1 is
    entry ENTRY_1;
    for ENTRY_1 use at 16#40#;
end T1;

If this case, ENTRY_1 will be mapped to hardware interrupt 64.

Only in parameters can be associated with interrupt entries.

An interrupt will act as an entry call issued by the hardware, with a priority higher than any user-defined task.

Depending upon the implementation, there can be many restrictions upon the type of call to the interrupt, and upon the terminate alternatives.

Note: you can directly call an interrupt entry.
TASKS AT DIFFERENT PRIORITIES

Given 5 tasks, 3 of varying priority, 1 to be interrupt driven, and 1 that will be tied to the clock.

PROCEDURE HEAVY_STUFF IS

    TASK HIGH_PRIORITY IS
       Pragma Priority(50); --or as high as system allows
        Entry Point;
    END HIGH_PRIORITY;

    TASK MEDIUM_PRIORITY IS
       Pragma Priority(25);
        Entry Point;
    END MEDIUM_PRIORITY;

    TASK LOW_PRIORITY IS
       Pragma Priority(1);
        Entry Point;
    END LOW_PRIORITY;

    TASK INTERRUPT_DRIVEN IS
        ENTRY Point;
        For Point use at 16#61#; --interrupt 97
    END INTERRUPT_DRIVEN;

    TASK CLOCK_DRIVEN IS
        --there are two ways to do this
        --first way is to have another task monitor
        --the clock, and call CLOCK_DRIVEN.CALL
        --every time unit.
        ENTRY CALL;

        --second way is to actually tie CALL to an
        --clock interrupt, and let CALL determine when
        --he wishes to perform an action
        For CALL use at 16#32#; --assume interrupt 50
        --is a clock interrupt
    END CLOCK_DRIVEN;

END HEAVY_STUFF;
TASK QUEUE IS
   ENTRY INSERT(DATA : IN DATA_TYPE);
   ENTRY REMOVE(DATA : OUT DATA_TYPE);
END QUEUE;

TASK BODY QUEUE IS
   HEAD, TAIL : INTEGER := 0;
   Q : ARRAY (1..100) OF DATA_TYPE;
BEGIN
   LOOP
      SELECT
         WHEN TAIL - HEAD + 1 /= 0 AND THEN TAIL - HEAD + 1 /= 100 =>
            ACCEPT INSERT(DATA : IN DATA_TYPE) DO
               IF HEAD = 0 THEN HEAD := 1; END IF;
               IF TAIL = 100 THEN TAIL := 0; END IF;
               TAIL := TAIL + 1;
               Q(TAIL) := DATA;
            END INSERT;
         OR
         WHEN HEAD /= 0 =>
            ACCEPT REMOVE(DATA : OUT DATA_TYPE) DO
               DATA := Q(HEAD);
               IF HEAD = TAIL THEN
                  HEAD := 0;
                  TAIL := 0;
               ELSE
                  HEAD := HEAD + 1;
                  IF HEAD > 100 THEN HEAD := 1; END IF;
               END IF;
            END REMOVE;
         OR
         TERMINATE;
      END SELECT;
   END LOOP;
END QUEUE;
TASK TYPE QUEUE IS
  ENTRY INSERT(DATA : IN DATA_TYPE);
  ENTRY REMOVE(DATA : OUT DATA_TYPE);
END QUEUE;

TASK BODY QUEUE IS
  HEAD, TAIL : INTEGER := 0;
  Q : ARRAY (1..100) OF DATA_TYPE;
BEGIN
  LOOP
    SELECT WHEN TAIL - HEAD + 1 /= 0 AND THEN
      TAIL - HEAD + 1 /= 100 =>
      ACCEPT INSERT(DATA : IN DATA_TYPE) DO
        IF HEAD = 0 THEN HEAD := 1; END IF;
        IF TAIL = 100 THEN TAIL := 0; END IF;
        TAIL := TAIL + 1;
        Q(TAIL) := DATA;
      END INSERT;
    OR WHEN HEAD /= 0 =>
      ACCEPT REMOVE(DATA : OUT DATA_TYPE) DO
        DATA := Q(HEAD);
        IF HEAD = TAIL THEN
          HEAD := 0;
          TAIL := 0;
        ELSE
          HEAD := HEAD + 1;
          IF HEAD > 100 THEN HEAD := 1; END IF;
        END IF;
      END REMOVE;
    OR
      TERMINATE;
    END SELECT;
  END LOOP;
END QUEUE;

MY_QUEUE, YOUR_QUEUE : QUEUE; -- TWO TASKS
GEcERIC
DATA_TYPE : PRIVATE;
QUEUE_SIZE : POSITIVE := 100;

PACKAGE QUEUE_PACK IS

TASK QUEUE IS
    ENTRY INSERT (DATA : IN DATA_TYPE);
    ENTRY REMOVE (DATA : OUT DATA_TYPE);
END QUEUE;

PACKAGE BODY QUEUE_PACK IS
    TASK BODY QUEUE IS
        HEAD, TAIL : INTEGER := 0;
        Q : ARRAY (1..QUEUE_SIZE) OF DATA_TYPE;
        BEGIN
            LOOP
                SELECT
                    WHEN TAIL - HEAD + 1 /= 0 AND THEN
                    TAIL - HEAD + 1 /= QUEUE_SIZE =>
                        ACCEPT INSERT (DATA : IN DATA_TYPE) DO
                            IF HEAD = 0 THEN HEAD := 1; END IF;
                            IF TAIL = QUEUE_SIZE THEN TAIL := 0; END IF;
                            TAIL := TAIL + 1;
                            Q(TAIL) := DATA;
                        END INSERT;
                    OR
                    WHEN HEAD /= 0 =>
                        ACCEPT REMOVE (DATA : OUT DATA_TYPE) DO
                            DATA := Q(HEAD);
                            IF HEAD = TAIL THEN
                                HEAD := 0;
                                TAIL := 0;
                            ELSE
                                HEAD := HEAD + 1;
                                IF HEAD > QUEUE_SIZE THEN HEAD := 1; END IF;
                            END IF;
                        END REMOVE;
                    OR
                    TERMINATE;
                END SELECT;
            END LOOP;
        END QUEUE;

PACKAGE NEW_QUEUE IS NEW QUEUE_PACK(MY_RECORD, 250);
PACKAGE OLD_QUEUE IS NEW QUEUE_PACK(INTEGER);
PROCEDURE INSERT_INTEGER (DATA : in INTEGER) renames
OLD_QUEUE.INSERT;

PROCEDURE REMOVE_INTEGER (DATA : out INTEGER) renames
OLD_QUEUE.REMOVE;
PROCEDURE SPIN (R : RESOURCE) IS
BEGIN
  LOOP
    SELECT
      R.SEIZE;
    RETURN;
    ELSE
      NULL; --BUSY WAITING
    END SELECT;
  END LOOP;
END;

--OR--

PROCEDURE SPIN (R : RESOURCE) IS
BEGIN
  R.SEIZE;
  RETURN;
END;
Ada Tasking

Scenario I

"The Golden Arches"

McD Tasks:
Service Provided: Food
Service Requested: None

Gonzo Tasks:
Service Provided: None
Service Requested: Food
Task McD is
   entry SERVE(TRAY_OF : out FOOD_TYPE);
end McD;

Task GONZO;

Task Body McD is
   NEW_TRAY : FOOD_TYPE;
   function COOK return FOOD_TYPE is ......
   begin
      loop
         accept SERVE(TRAY_OF : out FOOD_TYPE) do
            TRAY_OF := COOK;
         end;
      end loop;
end end McD;
Task Body GONZO is
  MY_TRAY : FOOD_TYPE;

procedure CONSUME(MY_TRAY:in FOOD_TYPE) is ...

begin
  loop
    McD.SERVE (MY_TRAY);
    CONSUME(MY_TRAY);
  end loop;
end GONZO;
Task Body McD is

NEW_TRAY : FOOD_TYPE;

function COOK return FOOD_TYPE is
...
end COOK;

begin
  loop
    NEW_TRAY := COOK;
    accept SERVE(TRAY_OF:out FOOD_TYPE) do
      TRAY_OF := NEW_TRAY;
    end SERVE;
  end loop;
end GONZO;
loop
    NEW_TRAY := COOK;
    select
        accept SERVE(TRAY_OF : out FOOD_TYPE) do
            TRAY_OF := NEW_TRAY;
            end SERVE;
    else
        null;
    end select;
end loop;
loop
  NEW_TRUNC := COOK;
  select
    accept SERVE(TRAY_OF : out FOOD_TYPE) do
      TRAY_OF := NEW_TRUNC;
    end SERVE;
    else
      terminate;
    end select;
  end loop;
loop
  NEW_TRAY := COOK;
  select
    accept SERVE(TRAY_OF : out FOOD_TYPE) do
      TRAY_OF := NEW_TRAY;
      end SERVE;
  or
    delay 15.0 * MINUTES;
  end select;
end loop;
loop

select
    McD.SERVE(MY_ORDER); CONSUME(MY_ORDER);
else
    select
        BK.SERVE(MY_ORDER); CONSUME (MY_ORDER);
    else
        exit;
    end select;
end select;

end loop;
loop

select
  McD.SERVE(MY_ORDER); CONSUME(MY_ORDER);
or
  delay 5.0 * MINUTES;
select
  BK.SERVE(MY_ORDER); CONSUME(MY_ORDER);
or
  delay 5.0 * MINUTES;
exit;
end select;
end select;

end loop;
loop

    select
        McD.SERVE (MY_ORDER);
    or
        BK.SERVE (MY_ORDER);
    end select;

    CONSUME (MY_ORDER);

end loop;
loop

select
    McD.SERVE (MY_ORDER);
or
    BK.SERVE(MY_ORDER);
else
    delay 10.0 * MINUTES;
    exit;
end select;

CONSUME(MY_ORDER);

end loop;
Ada Tasking

Scenario II

"No Free Lunch"

McD Task
Service Provided: Food
Service Requested: Money

Gonzo Task
Service Provided: Money
Service Requested: Food
Task McD is
    entry SERVE(ORDER: out FOOD_TYPE;
       COST: in MONEY_TYPE);
end McD;

Task GONZO;

--OR

Task McD is
    entry SERVE(ORDER: out FOOD_TYPE);
end McD;

Task GONZO is
    entry PAY(COST: in MONEY_TYPE;
       PAYMENT: out MONEY_TYPE);
end GONZO;
Task Body McD is
CASH_DRAWER, AMOUNT_PAID: MONEY_TYPE;
NEW_ORDER : FOOD_TYPE;
function COOK .......... 
function CALC_COST(ORDER: in FOOD_TYPE)
    return MONEY_TYPE ..........

begin
    loop
        NEW_ORDER := COOK;
        select
            accept SERVE(ORDER:out FOOD_TYPE) do
                ORDER := NEW_ORDER;
                COST := CALC_COST(NEW_ORDER);
                GONZO.PAY(COST, AMOUNT_PAID); --***
                CASH_DRAWER :=
                    CASH_DRAWER + AMOUNT_PAID;
            end SERVE;
            or
                delay 15.0 * MINUTES;
        end select;
    end loop;
end McD;
Task Body GONZO IS
ACCOUNT_BALANCE : MONEY_TYPE;
MY_ORDER : FOOD_TYPE;
function GO_TO_WORK return MONEY_TYPE ......

begin
    ACCOUNT_BALANCE :=
    ACCOUNT_BALANCE + GO_TO_WORK;
end loop
    McD.SERVE(MY_ORDER);
    accept PAY (COST : in MONEY_TYPE;
                PAYMENT : out MONEY_TYPE) do
        ACCOUNT_BALANCE :=
        ACCOUNT_BALANCE - COST;
        PAYMENT := COST;
    end PAY;
end loop;
end GONZO;
Scenario II a

"No Wait for the Waiters"

McD Task
Service Provided: Food
Service Requested: Money

Gonzo Task
Service Provided: Money
Service Requested: Food

Manager Task
Service Provided: Make new waiter
Service Requested: None
Task type McD is
  entry SERVE.....
end McD;

Task GONZO is
  entry PAY.....
end GONZO;

Task MANAGER;

Type CASHIER_POINTER is access McD;

Type REGISTER_TYPE is array (1..NO_REGISTERS)
  of CASHIER_POINTER;

THE_REGISTERS :& REGISTER_TYPE
  := (others => new McD);
Task Body McD is

... ... ... begin
  loop
    NEW_ORDER := COOK;
    select
      accept SERVE.....
    ... end SERVE;
    or
      delay 2, 0 * MINUTES;
      exit;
    end select;
  end loop;
Task Body GONZO is

...  
...  
begin

...  
...  
--- Now, GONZO has to search for the open
--- registers, and select the one with
--- the shortest line

...  
...  
THE_REGISTERS(MY_REGISTER).SERVE...

...  

end GONZO;
Task Body MANAGER is

... ...
begin
  loop
    -- The Manager will look at the queue lengths of
    -- the open registers, and, when necessary,
    -- will open registers that are currently
    -- closed
    ...
    ...
    if .............then
        THE_REGISTERS(CLOSED_REGISTER):= new McD;
    end if;
  end loop;
end MANAGER;
Scenario III

"A Sugar Cone, Please:

BR Task
Service Provided: Ice Cream
Service Requested: An Order

Servomatic Task
Service Provided: A Number

Customers Task
Service Provided: An Order
Service Requested: Ice Cream
Task BR is
    entry SERVE(ICE_CREAM: out DESSERT_TYPE);
end BR;

Task SERVOMATIC is
    entry TAKE(A_NUMBER: out SERVOMATIC_NUMBERS);
end SERVOMATIC;

Task type CUSTOMER_TASK is
    entry REQUEST(ORDER: out ORDER_TYPE);
    enter CUSTOMER_TASK;

Type CUSTOMER is access CUSTOMER_TASK;

CUSTOMERS : array (SERVOMATIC_NUMBERS) of CUSTOMER;
Task Body BR is

NEXT_CUSTOMER : SERVOMATIC_NUMBERS :=
    SERVOMATIC_NUMBERS'last;
CURRENT_ORDER : ORDER_TYPE;
ICE_CREAM : DESSERT_TYPE;

function MAKE(ORDER : in ORDER_TYPE) return
    DESSERT_TYPE is ......

begin
    loop
        begin
            NEXT_CUSTOMER := (NEXT_CUSTOMER + 1)
                mod SERVOMATIC_NUMBERS'last;
            CUSTOMERS(NEXT_CUSTOMER).REQUEST
                (CURRENT_ORDER);
            ICE_CREAM := MAKE (CURRENT_ORDER);
            accept SERVE(ICE_CREAM:out DESSERT_TYPE) do
                ICE_CREAM := BR.ICE_CREAM;
            end SERVE;
            exception
                when TASKING_ERROR => null; -- customer not here
            end exception;
        end loop
    end;
end;
Task Body SERVOMATIC is

    NEXT_NUMBER : SERVOMATIC_NUMBERS :=
        SERVOMATIC_NUMBERS'first;

begin
    loop
        accept TAKE(A_NUMBER:out SERVOMATIC_NUMBERS) do
            A_NUMBER := NEXT_NUMBER;
        end TAKE;
        NEXT_NUMBER := (NEXT_NUMBER + 1) mod
            SERVOMATIC_NUMBERS'last;
    end loop;

end SERVOMATIC;
Task Body CUSTOMER_TASK is

   MY_ORDER : ORDER_TYPE := ... -- some value
   MY_DESSERT : DESSERT_TYPE;

begin
   accept REQUEST(ORDER:out ORDER_TYPE) do
      ORDER := MY_ORDER;
   end REQUEST;
   BR.SERVE(MY_DESSERT);
   -- eat the dessert, or do whatever
end;
Ada Tasking

Scenario IV

"Let's Hide the Spooler Task"

printer_package
Action—"Hides" the print spooler
by renaming task entry

spooler task
Service Provided: virtual print
Service Requested: physical print

printer task
Service Provided: physical print
Service Requested: file name
Package PRINTER_PACKAGE is

    ...
    ...

    task SPOOLER is
        entry PRINT_FILE(NAME : in STRING;
                           PRIORITY : in NATURAL);
        entry PRINTER_READY;
    end SPOOLER;
    ...
    ...

    procedure PRINT (NAME : in STRING;
                     PRIORITY : in NATURAL := 10)
        renames SPOOLER.PRINT_FILE;
    end PRINTER_PACKAGE;

Package Body PRINTER_PACKAGE is

    ...

    task PRINTER is
        entry PRINT_FILE(NAME : 'in STRING);
    end PRINTER;
    ...

    end PRINTER_PACKAGE;
Task Body SPOOLER is
    begin
      loop
        select
          accept PRINTER_READY do
            PRINTER.PRINT_FILE(REMOVE(QUEUE));
            --Remove would determine the next job
            -- and send it to the actual printer
          end PRINTER_READY;
        else
          null;
        end select;
      end loop;
    end SPOOLER:
Task Body PRINTER is
begin
    loop
        SPOOLER.PRINTER_READY;
        accept PRINT_FILE (NAME : in STRING) do

            if NAME'length /= 0 then ......

                -- print the file

            else
                delay 10.0 * SECONDS;
            end if;

        end if;

    end PRINT_FILE;
end loop;

end PRINTER;
with PRINTER_PACKAGE;

procedure MAIN is

    ...
    ...
    ...
    ...
    loop
        -- process several files
        PRINTER_PACKAGE.PRINT (A_FILE, A_PRIORITY);
        ...
    end loop;
end MAIN;
TASKING MINDSET

Simple problem - write a task spec to let Task A send an integer to Task B.

Solution 1 - A calls an entry in B
Solution 2 - B calls for an entry in A
Solution 3 - Write a 'buffer' task to call entry in A, get integer, and then call entry in B to send integer
Solution 4 - Write buffer task C to accept integers from A, and also accept requests from B
IN-CLASS EXERCISE

Let us design the task specifications for the following scenario.

Three tasks have access to a type known as MESSAGE_TYPE.

Task_1 produces messages. Task_2 can receive messages, hold them in a buffer (if necessary), and sends them to Task_3 when the date/time field (part of MESSAGE_TYPE) says to.

Task TASK_1 is

END TASK_1;

Task TASK_2 is

END TASK_2;

Task TASK_3 is

END TASK_3;
Tasking Exercise

Write a main program and two tasks to simulate a house alarm system. The main program is an input simulator to the tasks. One task keeps track of the status of the house. Another is the actual alarm system.

Task 1: The House Status (Task name: HOUSE)
Three entries => OK, NOT_OK, WRITE

The entries OK and NOT_OK set or reset a flag that determines the status of the house. NOT_OK will also set a variable to tell you which alarm is currently going off. Both OK and NOT_OK should print out a message verifying that they were called. The WRITE entry will print the status of the house. If there is an alarm currently going off, WRITE will tell you the alarm number.

Task 2: The Alarm System (Task name: ALARM)
Three entries => FIRE, INTRUDER, SHUTOFF

The alarm system will accept any of the three entry calls from the input simulator. If there are no entry calls within 5 seconds, it will call HOUSE.WRITE to display the status. FIRE and INTRUDER each have a parameter indicating the alarm location. FIRE locations are '1' thru '9'. INTRUDER locations are 'A' thru 'Z'. FIRE and INTRUDER should call HOUSE.NOT_OK (and tell the house where the alarm is sounding), and then print out a message.

Main Program

The main program will read in characters from the keyboard. If the character is a '1' thru '9', call the fire alarm. If the character is a 'A' thru 'Z', then it calls the intruder alarm. If the character is a '0' (zero), the house is reset to OK. If the character is a '!', then the alarm is shutdown, and the program ends. All other characters do nothing.

The house status should be OK to start.
The house is ok

&
Invalid character. Try again

The house is ok

G
House alarm set to not OK at location G
Intruder in room G

The house is not ok ...alarm is off at location G

The house is not ok ...alarm is off at location G

4
House alarm set to not OK at location 4
Fire Alarm # 4 has been set off.

The house is not ok ...alarm is off at location 4

G
House alarm reset to OK.

The house is ok

The house is ok

The alarm has been turned off

7)
WITH TEXT_IO;
USE TEXT_IO;

PROCEDURE COOKIE IS
CHAR : CHARACTER;

TASK HOUSE IS
ENTRY OK;
ENTRY NOT_OK (WHERE: CHARACTER);
ENTRY WRITE;
END HOUSE ;

TASK ALARM IS
ENTRY FIRE (LOCATION: CHARACTER);
ENTRY INTRUDER (LOCATION: CHARACTER);
ENTRY SHUTOFF;
END ALARM ;
TASK BODY HOUSE IS
   TYPE CONDITION IS (OK, NOT_OK);
   ALARM_STATUS : CONDITION := OK;
   ALARM_LOCATION : CHARACTER;
BEGIN
   LOOP
      SELECT
         ACCEPT OK DO
            ALARM_STATUS := OK;
            PUT_LINE("HOUSE ALARM RESET TO OK.");
            END OK;
         OR
         ACCEPT NOT_OK (WHERE:CHARACTER) DO
            ALARM_STATUS := NOT_OK;
            ALARM_LOCATION := WHERE;
            PUT_LINE("HOUSE ALARM SET TO NOT OK AT"&
                  "LOCATION " & ALARM_LOCATION);
            END NOT_OK;
         OR
         ACCEPT WRITE DO
            NEW_LINE;
            CASE ALARM_STATUS IS
               WHEN OK => PUT_LINE("THE HOUSE IS OK");
               WHEN NOT_OK => PUT_LINE
                  ("THE HOUSE IS NOT OK"&
                   "...ALARM IS OFF AT LOCATION " &
                   ALARM_LOCATION);
               END CASE;
            NEW_LINE;
            END WRITE;
         OR
         TERMINATE;
      END SELECT;
   END LOOP;
END HOUSE;
TASK BODY ALARM IS
BEGIN
LOOP
SELECT
  ACCEPT FIRE (LOCATION:CHARACTER) DO
    HOUSE.NOT_OK(LOCATION);
    PUT ("FIRE ALARM # ");
    PUT (LOCATION);
    PUT_LINE (" HAS BEEN SET OFF.");
  END FIRE;
OR
  ACCEPT INTRUDER (LOCATION:CHARACTER) DO
    HOUSE.NOT_OK(LOCATION);
    PUT ("INTRUDER IN ROOM ");
    PUT (LOCATION);
    NEW_LINE;
  END INTRUDER;
OR
  ACCEPT SHUTOFF;
  PUT_LINE ("THE ALARM HAS BEEN TURNED OFF");
  EXIT;
OR
  DELAY 5.0;
  HOUSE.WRITE;
END SELECT;
END LOOP;
END ALARM;
BEGIN
-- MAIN
LOOP
GET (CHAR);
SKIP_LINE;
CASE CHAR IS
  WHEN 'I' .. '9' => ALARM.FIRE (CHAR);
  WHEN 'A' .. 'Z' => ALARM.INTRUDER (CHAR);
  WHEN 'A' .. 'Z' => ALARM.INTRUDER (CHAR);
  WHEN 'O' => HOUSE.OK;
  WHEN 'I' => ALARM.SHUTOFF;
  WHEN OTHERS => PUT_LINE
    ("INVALID CHARACTER. TRY AGAIN");
END CASE;
EXIT WHEN CHAR = 'I';
END LOOP;
END COOKIE;
Towers of Hanoi
An example of recursion

recursion: n., see recursion.

Problem: Move disks from one tower to another tower.

Constraints:
Move only 1 disk at a time.
Place no disk on a smaller disk.

Top down design approach:
Assume a procedure to move N disks:

```pascal
procedure Move (N : in positive; From, To, Other : in Towers);
```

Use the procedure and solve the problem:

```pascal
Move (N=>3, From => Middle, To => Left, Other => Right);
```
Using this approach, we can now create a complete Ada program:

```ada
procedure Towers_of_Hanoi is
  type Towers is (Middle, Left, Right);
  procedure Move (N : in positive;
                  From, To, Other : in Towers)
    is separate;
  begin
    Move(3, From => Middle,
         To   => Left,
         Other => Right);
  end Towers_of_Hanoi;
```
Implement the procedure in pseudocode:

```pseudo
separate (Towers_of_Hanoi)

procedure Move (N : in positive;
    From,
    To,
    Other : in Towers)

is
    begin
        null;
        -- if more than one disk to move,
        -- Move(N-1, from => ______,
        --     to   => ______,
        --     other => ______);

        -- move the only disk left on 'from' to 'to'
        -- if more than one disk to move,
        -- Move(N-1, from => ______,
        --     to   => ______,
        --     other => ______);
    end Move;
```
Now rewrite the procedure in Ada:

```ada
with Text_I0;
separate (Towers_of_Hanoi)
procedure Move (N : in positive;
    From,
    To,
    Other : in Towers)
is
begin
    if N > 1 then
        Move(N-1, From => From,
            To => Other,
            Other => To);
    end if;

    Text_I0.put_line("Move disk from "
        & Towers'Image(From)
        & " tower to "
        & Towers'Image(To)
        & " tower.");

    if N > 1 then
        Move(N-1, From => Other,
            To => To,
            Other => From);
    end if;

end Move;
```
Sieve of Eratosthenes

Eratosthenes, of Alexandria, was a Greek mathematician. He developed an elegant algorithm for generating prime numbers.

1. 2 is the first prime number.
2. For each positive number, N, greater than 2, if it is not divisible by any prime less than N, it is prime.

This algorithm has a natural implementation in Ada.

Imagine that a separate process is available for each prime number, that can check the "relative" primeness of a number.

We can now "pipeline" these processes with all the positive numbers, any number that makes it through the "pipe" is prime!

Create a task which feeds numbers into the pipe:

    task Feeder;

Create a task template which accepts a value and checks it for primeness:

    task type Checker is
        entry Check_It (In_Value : Positive);
    end Checker;
But, this checker task needs to know what prime number it uses. Often we find the case in Ada tasks where the task must be initialized with information:

```ada
task type Checker is
  entry Who_Am_I (In_Value : Positive);
  entry Check_It  (In_Value : Positive);
end Checker;
```

Finally, we need to create new tasks when we find that a number is prime:

```ada
procedure Make_New_Checker
  (A_Prime_Number : in  Positive;
   New_Checker     : out Checker_Ptr);
```

We can create an operation to construct a task only by using a pointer to the new task.
There are many ways to link the checker tasks together into the "pipe". This linking determines and is determined by the manner used to pass the numbers being checked from task to task.

I chose to have each task contain the name of the next task in the pipe.
procedure Primes is

    task Feeder;

    type Checker,
    type Checker_ptr is access Checker;
    task type Checker is
        entry Who_Am_I (In_Value : Positive);
        entry Check_It   (In_Value : Positive);
    end Checker;

procedure Make_New_Checker (A_Prime_Number : in Positive;
    New_Checker    : out Checker_Ptr);

    Front : Checker_ptr; -- This is the front of the "pipe".

    task body Feeder is separate;

    task body Checker is separate;

procedure Make_New_Checker (A_Prime_Number : in Positive;
    New_Checker    : out Checker_Ptr)
    is separate;

begin
    null;
end Primes,
with Text_10, Integer_10,
separate (Primes)
procedure Make_New_Checker (A_Prime_Number : in Positive;
                           New_Checker : out Checker_Ptr) is

  Result : Checker_Ptr;

begin
  -- We have been given a prime number, display it:
  Integer_10.Put (A_Prime_Number);

  -- Make a new prime task for it:
  Result := new Checker;
  Result.Who_Am_I (A_Prime_Number);  -- Tell the task it's prime.

  -- Allow the task to be used in the "pipe".
  New_Checker := Result;

exception
  when Storage_Error =>
    Text_10.Put_Line (" Not enough room to make new tasks.");

end Make_New_Checker;

with Text_10, Integer_10;
separate (Primes)
task body Feeder is
  Upper Limit : Positive;

begin
  Text_10.Put ("Upper limit for primes? ");
  Integer_10.Get (Upper Limit);

  -- Generate the first prime:
  Make_New_Checker (2, Front);

  -- Feed the "pipe":
  for Counter in 3 .. Upper_Limit loop
    Front.Check_It (Counter);  
  end loop;

end Feeder;
separate (Primes)
task body Checker is

   My_Value, 
   Value_to_Check : Positive;
   Next_Checker   : Checker_Ptr;
   Prime          : Boolean;

begin

   accept Who_Am_I (In_Value : Positive) do
      My_Value := In_Value;
   end Who_Am_I;

   loop
      select
         accept Check_It (In_Value : Positive) do
            Value_to_Check := In_Value;
         end Check_It;
         or
         terminate;
      end select;

      Prime := (Value_to_Check / My_Value) * My_Value /= Value_to_Check;

      if Prime then
         if Next_Checker /= null then
            -- It's not divisible by my number, pass the value on
            Next_Checker.Check_It (Value_to_Check);
         else
            -- It really is prime.
            Make_New_Checker (Value_to_Check, Next_Checker);
         end if;
      end if;

   end loop;

end Checker;

The Dining Philosophers

The scenario: Five philosophers sit at a table. A lazy Susan containing dishes of Chinese food is in the center of the table. Each philosopher has a plate, but there are only five single chopsticks, one between each philosopher.

The problem: Develop a program that allows each philosopher to alternately eat and think forever. Of course, no philosopher should preclude any other from eating for an indefinite amount of time.

The constraints: Each philosopher must have control of two chopsticks to eat. But, he can only use those that were originally on either side of his plate.

\[\text{Aristotle} \quad \text{Socrates} \quad \text{Einstein} \quad \text{Plato} \quad \text{Buddha}\]

\[\text{1 This problem was first stated by Edsger Dijkstra as a challenge to the multitasking community}\]
Approach:
Each philosopher can wait in a queue for his chopsticks. If at least one philosopher is eating, we will not have deadlock. If philosophers are not blocked from entering a queue, then we will not have indefinite postponement.

Model the chopsticks as counting semaphores.

Do not make all the philosophers right-handed or left-handed.

Object Oriented Design:
Object – Chopstick
Operations – Pick_Up, Put_Down

Object – Philosopher
Operations – Give_Names

Object – Console
Operations – Display
with Wrap_Around,
procedure Diners is
   type Names is (Socrates, Plato, Buddha, Einstein, Aristotle),
   task Chopstick is
      entry Pick_Up;
      entry Put_Down;
   end Chopstick;
   task type Philosopher is
      entry Give_Names (My_Name, First_Strip, Second_Strip : in Names);
   end Philosopher;
   Chopsticks : array (Names) of Chopstick;
   Philosophers : array (Names) of Philosopher;
   task Console is
      entry Display (Message : in String);
   end Console;
   task body Console is separate;
   task body Chopstick is separate;
   task body Philosopher is separate;
   function Wrap is new Wrap_Around (Names);
begin
   -- Tell each philosopher his name.
   Philosophers (Names'First).Give_Names
      (My_Name => Names'First,
       First_Strip => Wrap(Names'First),
       Second_Strip => Names'First),
   for Loop_Index in Wrap (Names'First) .. Names'Last
   loop
      Philosophers (Loop_Index).Give_Names
         (My_Name => Loop_Index,
          First_Strip => Loop_Index,
          Second_Strip => Wrap(Loop_Index));
   end loop;
end Diners,
with Text_IO,
separate (Diners)
task body Chopstick is
begin
loop
select
accept Pick_Up, -- Callers will be queued here.
accept Put_Down, -- Resource is released here.
or
terminate; -- Server task offers to quit.
end select;
end loop;

exception
when others =>
   Text_IO.Put_Line ("Chopstick task died.");
end Chopstick;
with Text_IO;
separate (Diners)
task body Philosopher is
  My_Name,
  First_Stick,
  Second_Stick : Names;
begin
  accept Give_Names (My_Name,
       First_Stick,
       Second_Stick : in Names) do
    Philosopher.My_Name := My_Name;
    Philosopher.First_Stick := First_Stick;
    Philosopher.Second_Stick := Second_Stick;
  end Give_Names;

  declare
    Eating_Message : constant String :=
      Names.Image(My_Name) & " eating";
    Thinking_Message : constant String :=
      Names.Image(My_Name) & " thinking";
  begin
    loop
      Chopsticks (First_Stick).Pick_Up;
      Chopsticks (Second_Stick).Pick_Up;
      Console.Display (Eating_Message);
      Chopsticks (First_Stick).Put_Down;
      Chopsticks (Second_Stick).Put_Down;
      Console.Display (Thinking_Message);
    end loop;
  exception
    when others =>
      Text_IO.Put_Line ("Philosopher task died"),
  end Philosopher,
with Text_IO,
separate (Diners)
task body Console is
begin
loop
select
accept Display (Message : in String) do
  Text_IO.Put_Line (Message);
end Display;

or
  terminate; -- Server task offers to quit.
end select;
end loop;

exception
  when others =>
    Text_IO.Put_Line ("Console task died.");

end Console;
Ada Generics

by

Lindy Moran
USNA

Maj Chuck Engle
SEI

Part of the
Advanced Ada Workshop
sponsored by the
Ada Software Engineering Education
& Training Team (ASEET)
Acknowledgements:

Much of this material was taken from earlier Aseet Advanced Generics Tutorials - in particular the tutorial of 18 Aug '87 presented by Mr. John Bailey of Software Metrics, Inc.
GENERICS

- Why program at all?
- Why program generically?
- What does generics provide?
- How do you write a generic unit?
  - Parameterless Generics
  - Parameterized Generics
    - Value and Object Parameters
    - Type Parameters
    - Subprogram Parameters
- What are the Cons of generics?
- What are the Pros of generics?
- What are the unresolved issues?
- How do you teach generics?
Why program at all?

- **Reusability** - a programmed solution can be used over and over

- **Reliability** - program can be tested and verified to ensure correct results for subsequent runs

- **Readability** - program formalizes human solution and represents it in more abstract readable form

- **Maintainability** - making a change to a program ensures that the change is consistently applied to all problem solutions
Why program generically?

- Reusability - similar program units needed but different enough to preclude simply entering differing values at run time.

- Reliability - generic unit once tested and verified does not need to be retested for each new use or "instantiation".

- Readability - using generic unit allows extraction of the "essence" of the unit eliminating application specific details and produces a very uncluttered readable unit.

- Maintainability - a change made to the unit applies to all uses of the unit.
Traditional Programming

Algorithms, Objects, Resources

-- intermixed with --

Problem specifics
The Price of Strong Typing
An Example

procedure Swap(X,Y : in out INTEGER) is
  Temp : INTEGER;
begin
  Temp := X;
  X := Y;
  Y := Temp;
end Swap;

procedure Swap(X,Y : in out CHARACTER) is
  Temp : CHARACTER;
begin
  Temp := X;
  X := Y;
  Y := Temp;
end Swap;

procedure Swap(X,Y : in out FLOAT) is
  Temp : FLOAT;
begin
  Temp := X;
  X := Y;
  Y := Temp;
end Swap;

type GRADE is range 0..100;
procedure Swap(X,Y : in out GRADE) is
  Temp : GRADE;
begin
  Temp := X;
  X := Y;
  Y := Temp;
end Swap;
Generic Programming

Algorithms, Objects, Resources

---

separated from

---

Problem specifics
A "generic" Swap Procedure

generic
    type ELEMENT is private;
procedure Swap(X,Y : in out ELEMENT);

procedure Swap(X,Y : in out ELEMENT) is
    Temp : ELEMENT;
begin
    Temp := X;
    X := Y;
    Y := Temp;
end Swap;

procedure SwapThings is
    X : integer := 5;
    Y : integer := 10;
    Letter1 : character := 'A';
    Letter2 : character := 'Z';

    procedure IntSwap is new Swap(integer);
    procedure CharacterSwap is
        new Swap(ELEMENT->character);

begin
    IntSwap(X,Y);
    CharacterSwap(Letter1,Letter2);
end SwapThings;
Syntax and Semantics

generic
... formal parameters go here ...
subprogram or package specification

subprogram or package body
... body goes here ...

instantiation to create a usable unit
What does generics provide?

- Generics serve as "templates" for creating or "instantiating" similar conceptual "chunks" of code (packages, functions, or procedures).

- Generics allow removing the problem specifics from a program unit adding greater clarity to its understandability.

- Generics allows the programmer to introduce a level of abstraction to increase program understandability.

- Generics reduce user's source code size thereby making it more readable and maintainable.

- Generics enhance REUSE of software components, facilitating modular system development and easier verifiability.

- Generics provide an elegant solution to the restrictions imposed by strong typing.

- Generics provides a mechanism for passing subprograms as parameters.

- Generics provides a mechanism for doing IO (if needed) for all predefined and user-defined types.
Parameterless Generics
"Cloning" Units

A nongeneric "unique object" Stack package:

package Stack is
  procedure Pop(I : out integer);
  procedure Push(I : in integer);
  function Empty return boolean;
  function Full return boolean;
end Stack;

A non-generic "many objects" solution:

package Stacks is
  type Stack is . . . ;
  procedure Pop(S : in out Stack; I : out integer);
  procedure Push(S : in out Stack; I : in integer);
  function Empty(S : Stack) return boolean;
  function Full(S : Stack) return boolean;
end Stacks;

-- changes must be made to body of package also

A sample user program:
procedure StackUp is
  S1, S2 : Stack;  Item : integer;
begin
  Push(S1,10); Push(S2,5); Pop(S1,Item);
end;
A generic "many objects" solution:

generic
package Stack is
  procedure Pop(I : out integer);
  procedure Push(I : in integer);
  function Empty return boolean;
  function Full return boolean;
end Stack;

-- generic body is identical to non-generic one
-- no changes have to be made to get many stacks

A sample user program:

with Stack;
procedure StackUp is
  Item : integer;
  package S1 is new Stack;
  package S2 is new Stack;
begin
  S1.Push(10); S2.Push(5);
  S1.Pop(Item); S2.Pop(Item);
end StackUp;
Parameterless Generics cont.

Stack implementations compared

- Non-generic package - only one elaboration and initialization occur

- Generic package - multiple elaborations and initializations occur - once for each package

Example: with Text_IO;
package body Stack is
...
begin
  Text_IO.Put("New stack created.");
end Stack;

package S1 is new Stack; -- message prints
package S2 is new Stack; -- message prints again
package S3 is new Stack; -- message prints again
...
Parameterless Generics
"Cloning" Things

"Making The Mold"

package VDU is
    subtype Y.Range is integer range 1 .. 24;
    subtype X.Range is integer range 1 .. 80;
    procedure Write(S : in string);
        -- writes S to screen at current cursor loc
    procedure Move(Y : in Y.Range; X : X.Range);
        -- changes cursor position to (X,Y)

end VDU;

generic package VDU is
    subtype Y.Range is integer range 1 .. 24;
    subtype X.Range is integer range 1 .. 80;
    procedure Write(S : in string);
        -- writes S to screen at current cursor loc
    procedure Move(Y : in Y.Range; X : X.Range);
        -- changes cursor position to (X,Y)

end VDU;
Generic Instantiation
"Cloning" Things Continued... "Making The Copies"

package VDU1 is new VDU;
package VDU2 is new VDU;

VDU1.Write("VDU 1");  VDU2.Write("VDU 2");

**What if we included "Use VDU1, VDU2;"? Would we still need to be explicit and use the package name and dot prefix notation?

[*VDU example taken from ADA Language and Methodology by Watt, Wichmann, and Findlay]
Creating Library Units of Generic Instantiations

-- compile following separately into the library

with Stack;
package S1 is new Stack;

-- S1 is now a usable library unit

with S1; use S1;
procedure StackUp is
  Item : integer;
begi
  Push(10);
  Push(20);
  Pop(Item);
end StackUp;
Parameterized Generics

- Generic Parameters
  - Value and Object Parameters
  - Type Parameters
  - Subprogram Parameters
Value and Object Parameters

- Value Parameters
  - Are of mode IN
  - Serve as local constants in generic units

- Object Parameters
  - Are of mode IN OUT
  - Serve as global objects in generic units
Value Parameters

generic
    Max : in integer;
    Min : integer; -- default mode is IN
procedure BigNSmall(X : integer);

procedure BigNSmall(X : in integer) is
begin
    if X > Max then
        Max := X; -- not with mode IN
    end if;
    if X < Min then
        Min := X; -- not with mode IN
    end if;
end BigNSmall;
Value Parameters and Initialization Before Instantiation

☐ Actual parameters which are to match with formal generic value parameters must have been initialized before the instantiation occurs.

Example:

```pascal
generic
  Max : in integer;
  Min : integer;  -- default mode is IN
procedure BigNSmall1(X : integer);

procedure UseBigNSmall1 is
  LocalMin : integer;  -- no initial value
  LocalMax : integer;  -- no initial value
  X : integer := -100;

  procedure Extremes is new
    BigNSmall1(Max -> LocalMax, Min -> LocalMin);
  -- error occurs due to lack of initialization

begin
  Extremes(X);
end UseBigNSmall1;
```
Value Parameters and Levels of Abstraction

generic
  Lower, Upper : in character;
function In_Range(S : in string) return boolean;

function In_Range(S : in string) return boolean is
begin
  for I in S'Range loop
    if S(I) not in Lower .. Upper then
      return FALSE;
    end if;
  end loop;
  return TRUE;
end In_Range;

A non-generic version of In_Range:

function In_Range(S : in string; Upper,Lower : character) return boolean is
begin
  for I in S'Range loop
    if S(I) not in Lower .. Upper then
      return FALSE;
    end if;
  end loop;
  return TRUE;
end In_Range;
Value Parameters and Levels of Abstraction cont.

Compare clarity in user's programs using generics to add another level of abstraction in "customized" names for `InRange` function

```ada
with InRange;
procedure InBounds is
    Name: string(1..4) := "JACK";
    Phone: string(1..7) := "6725643";
begin
    if InRange(Name,'A','Z') then ... 
    if InRange(Phone,'0','9') then ...
end InBounds;
```

```ada
with InRange;
procedure InBounds is
    Name: string(1..4) := "JACK";
    Phone: string(1..7) := "6725643";

    function Is_All_Upper_Case is new InRange('A','Z');
    function Is_All_Lower_Case is new InRange('a','z');
    function Is_All_Decimal is new InRange('0','9');

begin
    if Is_All_Upper_Case(Name) then ... 
    if Is_All_Decimal(Phone) then ...
end InBounds;
```

[In.Range taken from Ada Language and Methodology]
Object Parameters

Our Stack Example Revisited

generic
  Size : in natural;
package Stacks is
  type Stack is limited private;
  procedure Push(S : in out Stack; I : in integer);
  procedure Pop(S : in out Stack; I : out integer);
private
  subtype NumberOfElements is integer
    range 0..Size;
  type ElementArray is
    array(NumberOfElements) of integer;
  type Stack is record
    Elements : ElementArray;
    Top :NumberOfElements := 0;
  end record;
end Stacks;

with Stacks;
procedure StackUp is
  package SmallStack is new Stacks(5);
  package BigStack is new Stack(5000);
begi
    BigStack is new Stack(5000);
    begin
      ... StackUp;
end StackUp;
Object Parameters and Default Values

generic
  Rows : in positive := 24;
  Columns : in positive := 80;
package Terminal is
  ...
end Terminal;

-- some possible instantiations

package MicroTerminal is new Terminal(24,40);
-- using positional notation

package WordProcessor is new Terminal(Columns=>85,Rows=>66);
-- using named notation

package DefaultTerminal is new Terminal;
-- using the default values of 24 and 80
Object Parameters
and
The Subtleties of Default Values

What are the outputs of the following?

```pascal
package CountingPackage is
  function NextNum return integer;
  generic
    Val : integer := NextNum;
  procedure Count;
end CountingPackage;

with TextIO;
package body CountingPackage is
 CurrentValue : integer := 0;
  function Num return integer is
  begin
    CurrentValue := CurrentValue + 1;
    return CurrentValue;
  end Num;

  procedure Count is
  begin
    TextIO.Put_Line(integer'image(Val));
  end Count;
end CountingPackage;

with CountingPackage;
procedure StartCounting is
  procedure FirstCount is new CountingPackage.Count;
  procedure CountAgain is new CountingPackage.Count;
  begin
    FirstCount;
    CountAgain;
  end StartCounting;
```

```
Object Parameters

A More Useful Example

generic
   Control_Block : in out DeviceData;
   Kind : in VDU.Kind := Basic.Kind;
package VDU is

   ...
end VDU;

with VDU;
procedure ManyVDUs is
   DeviceTable : array(1..N) of DeviceData;

   package VDU1 is new
      VDU(DeviceTable(1),Kind_A);
   package VDU2 is new
      VDU(DeviceTable(2),Kind_B);

begin
   ...
end ManyVDUs;

[Taken from Ada Language and Methodology]
Object Parameters
and
Subtleties

- Object parameters passed by reference
  not by copy-restore method

- Object parameters are "aliases" for their
  actual parameter counterparts

Example:

```pascal
with Text_IO; use Text_IO;

procedure X is
  Global : integer := 99;
  procedure Z(Param : in out integer) is
    begin
      Param := Param + 1;
      Put_Line(integer'image(Param));
      Put_Line(integer'image(Global));
    end Z;
  begin
    Z(Global);
  end X;

  -- output is 100, 99 for copy-restore method
  -- output is 100,100 for pass by reference
```
Object Parameters
and
Subtleties cont.

☐ Object parameters passed by reference
not by name -- not like Algol's "copy
rule"

☐ Address of actual parameter corresponding
to formal generic object parameter is
evaluated ONCE and does not change

☐ Using generic object parameter NOT like
doing textual substitution of actual
parameter's name
declare
Y : array(1..5) of character := "kitty";
Index : integer := 1;

generic
  X : in out character;
procedure Replace;

procedure Replace is
begin
  Index := 5;
  X := 'w';
  -- X -> Y(1), NOT Y(5)
  Put(String(Y));
end Replace;

procedure Update is new Replace(Y(Index));
-- Index = 1 when this instantiation occurs
begin
  Update;
end;
Object Parameters and Subtleties cont.

☐ ADDRESS of actual parameter corresponding to a generic formal object parameter is evaluated at time of instantiation -- VALUE in that address not evaluated or copied into the formal parameter
declare
    subtype Small is Integer range 1 .. 10;
    X : integer := 27;
generic
    S : in Small;
procedure Gen;
procedure Gen is
begin
    Put("All OK");
end Gen;
procedure P is new Gen(X);
-- Constraint_Error raised at time of instant.
begin
    P;
end;

declare
    subtype Small is Integer range 1..10;
    X : integer := 27;
    generic
        S : in out Small;
    procedure Gen;
    procedure Gen is
    begin
        Put("All OK");
    end Gen;
    procedure P is new Gen(X);
    -- executes OK -- would NOT if value of
    -- S was used inside Gen
    begin
        P;
    end;
Object Parameters

☐ Use not recommended because suffer from all same fallacies as global objects

☐ Generic object parameters usually SHOULD have been regular formal parameters in the subprogram
Object Parameters cont.

generic
  Variable : in out integer;
  Limit, ResetValue : in integer;
procedure ResetIntegerTemplate;

procedure ResetIntegerTemplate is
begin
  if Variable > Limit then
    Variable := ResetValue;
  end if;
end ResetIntegerTemplate;

Better written as . . .

generic
  Limit, ResetValue : in integer;
procedure ResetIntegerTemplate(Variable : in out integer);

procedure ResetIntegerTemplate(Variable : in out integer) is
begin
  if Variable > Limit then
    Variable := ResetValue;
  end if;
end ResetIntegerTemplate;
Type Parameters

- type identifier is range <>;
- type identifier is digits <>;
- type identifier is delta <>;
- type identifier is (<>);
- type identifier is array(typemark range <>, ..., typemark range <>) of typemark;
- type identifier is array(typemark, ..., typemark) of typemark;
- type identifier is access typemark;
- type identifier is private;
- type identifier is limited private;
Integer Type Parameters

- type *identifier* is range ↩;

- matches an integer type, predefined or user-defined

- operations defined are those defined for integers such as +, -, /, *, **, rem, mod, negation, abs, >, <, =, /=, <=, >=

- attributes defined are those defined for integers such as 'first, 'last, 'succ, ...
Integer Type Parameters
An Example

generic
   type IntType is range <>;
function Increment(X : IntType) return IntType;

function Increment(X: IntType) return IntType is
begin
   return X+1;
end Increment;

with Increment;
procedure IncrementThings is
   type Age is range 0 .. 130;
   type Temp is range -100 .. 100;

   MyAge : Age 30;
   CurrentTemp : Temp := 80;

   function YearOlder is new Increment(Age);
   function TempUp is new Increment(IntType->Temp);

begin
   MyAge := YearOlder(MyAge);
   CurrentTemp := TempUp(CurrentTemp);
end IncrementThings;
Float Type Parameters

- type `identifier` is `digits <>`;

- matches any floating point type, predefined or user-defined

- operations defined are those available for floating point types such as `+`, `-`, `/`, `*`, `**`, negation, `abs`, `>`, `<`, `=-`, `<=-`, `>=`

- attributes defined are those available for floating point types such as `'small`, `'large`, `'digits`, `'mantisa`, `'epsilon`, ...

- useful in providing mathematical routines where user can control the precision used
Float Type Parameters
An Example

generic
type FloatType is digits <>;
function Sqrt(X : FloatType) return FloatType;

function Sqrt(X : FloatType) return FloatType is
begin
...
end Sqrt;

with Sqrt;
procedure Rooting is
type VeryPrecise is digits 7;
type Imprecise is digits 3;

X : VeryPrecise := 0.1234;
Y : Imprecise := 0.12;

function ExactRoot is new Sqrt(VeryPrecise);
function RoundRoot is new Sqrt(Imprecise);

begin
X := ExactRoot(X);
Y := RoundRoot(Y);
end Rooting;
Discrete Type Parameters

- type *identifier* is (<>);

- matches any discrete type -- includes integer types and enumeration types (boolean also)

- attributes defined are those available for any discrete/scalar type such as 'first, 'last, 'succ, 'pred, 'image, 'value, 'pos, 'val

- operations defined are those defined for discrete/scalar types such as >, <, -, /, -, <=
Discrete Type Parameters
An Example

generic
type Element is (<>);
package Sets is
type Set is private;
function Intersection(S1, S2 : Set) return Set;
function Union(S1, S2 : Set) return Set;
function IsIn(Item : Element; S : Set) return boolean;
function IsNull(S : Set) return boolean;
private
type Set is array(Element) of boolean;
end Sets;

-- some possible instantiations

package CharacterSet is new Sets(character);

package IntegerSet is new Sets(integer);

type Student is (John, Joan, Ann, Sue, ..., Zip);
package StudentSet is new Sets(Student);
Discrete Type Parameters cont.

- Minimal assumptions about the type must be made - operations must apply to ALL discrete types.

Example:

generic
  type Element is (<>);
function Next(X : Element) return Element;

function Next(X : Element) return Element is begin
  X := X + 1;  -- not defined for ALL
          -- discrete types
end Next;

Use attributes:

function Next(X : Element) return Element is begin
  if X = Element'Last then
    return Element'First;
  else
    return Element'Succ(X);
  end if;
end Next;
Constrained Array Type Parameters

- type identifier is array \((\text{typemark}, \ldots, \text{typemark})\) of \text{typemark};

- matches any constrained array type where:
  1) number of dimensions match,
  2) index subtypes of corresponding dimensions match,
  3) bounds in corresponding dimensions are identical,
  4) component types match

- attributes defined are those available for constrained arrays such as \'first(n), \'last(n), \'range(n), \'length(n)

- operations defined include those available for constrained arrays such as -, :, using slice notation (for one dimensional arrays)
Constrained Array Type Parameters
An Example

generic
    type Component is <>;
    type AnArray is array(1..10) of Component;
procedure Sort(A : in out AnArray);
procedure Sort(A : in out AnArray) is
    Temp : Component;
begin
    for I in 2..10 loop
        for J in 1..I-1 loop
            if A(I) < A(J) then
                Temp := A(J);
                A(J) := A(I);
                A(I) := Temp;
            end if;
        end loop;
    end loop;
end loop;
end Sort;

-- in user program
type Age is integer range 0..130;
type AgeArray is array(1..10) of Age;
X : AgeArray := (8,0,9,4,50,35,87,97,1,124);

procedure AgeSort is new
    Sort(Component,AgeArray);

... AgeSort(X); ...
Unconstrained Array Type Parameters

- type *identifier* is array(*typemark* range <>), . . ., *typemark* range <> of *typemark*;

- matches any unconstrained array where:
  1) number of dimensions the same
  2) subtype of index for corresponding dimensions is the same
  3) component types match

- attributes defined are those available for unconstrained arrays such as 'first(n), 'last(n), 'range(n), 'length(n)

- operations defined include those available for unconstrained arrays such as -, :=, using slice notation (for one dimensional typearrays)
Unconstrained Array Type
Parameters
An Example

generic
  type Index is range <>;
  type Component is range <>;
  type AnArray is array(Index) of Component;
procedure Sort(A : in out AnArray);
procedure Sort(A : in out AnArray) is
  Temp : Component;
begin
  for I in A'First+1 .. A'Last loop
    for J in A'First .. I-1 loop
      if A(I) < A(J) then
        Temp := A(J);
        A(J) := A(I);
        A(I) := Temp;
      end if;
    end loop;
  end loop;
end loop;
end Sort;

-- in user's program

type Age is range 0..130;
type EmployeeNumber is range 1..100;
type EmpList is array(EmployeeNumber) of Age;
procedure EmployeeAgeSort is new
  Sort(EmployeeNumber,Age,EmpList);
Employees : EmpList := (....);
...
EmployeeAgeSort(Employees); ...
Private Type Parameters

- type identifier is private;
- matches any type except a limited type
- operations available are only declaring objects of the type, testing for equality and inequality, and assigning values to objects of the type
Private Type Parameters
An Example

generic
  type Index is (<>);
  type Component is private;
  type AnArray is array(Index) of Component;
function Found(A : AnArray; T : Component)
  return boolean;
function Found(A : AnArray; T : Component)
  return boolean is
begin
  for I in A'First..A'Last loop
    if A(I) = T then
      return TRUE;
    end if;
  end loop;
  return FALSE;
end Found;

-- in user's program
type Student is (Joan, John, Sue, ..., Debbie);
type Grade is range 0..100;
type GradeArray is array(Student) of Grade;
function GradeMade is new
  Found(Student, Grade, GradeArray);
Grades : GradeArray := (....);

... if GradeMade(Grades, 100) then ...
Private Type Parameters cont. and Restrictions Imposed

What's wrong here?

generic
  type Index is (<>);
  type Component is private;
  type Int_Array is array(Index) of Component;
procedure Sort_Array(Arr : in out Int_Array);

procedure Sort_Array(Arr : in out Int_Array) is
  Temp : Component;
begin
  for I in Index'Succ(Arr'First)..Arr'Last loop
    for J in Arr'First..Index'Pred(I) loop
      if Arr(I) < Arr(J) then
        Temp := Arr(J);
        Arr(J) := Arr(I);
        Arr(I) := Temp;
      end if;
    end loop;
  end loop;
end Sort_Array;
Private Type Parameters
Another Caution

What's wrong here?

generic
type Element is private;
procedure Swap(X,Y : in out Element);

procedure Swap(X,Y : in out Element) is
    Temp : Element;
begin
    Temp := X;
    X := Y;
    Y := Temp;
end Swap;

-- in user's program
HerName : string(1..5) := "Lindy";
HisName : string(1..5) := "Chuck";

procedure NameSwap is new Swap(string);
Limited Private Type Parameters

- matches any type including a limited type
- only declaration of objects of the type permitted and NOTHING else

Example:

generic
  MyFile : Text_IO.File_Type; -- illegal
procedure Oops;
Access Type Parameters

☐ matches any access type

☐ operations defined for access types available such as setting object to null, use of NEW allocator, use of .ALL notation

-- Example follows introduction of subprogram parameters
## Generic Formal Type Parameters
### A Synopsis

<table>
<thead>
<tr>
<th>Generic formal parameter</th>
<th>Actual parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>type T is limited private;</td>
<td>any type</td>
</tr>
<tr>
<td>type T is private;</td>
<td>any non-limited type</td>
</tr>
<tr>
<td>type T is (&lt;&gt;);</td>
<td>any discrete type</td>
</tr>
<tr>
<td>type T is range&lt;&gt;;</td>
<td>any integer type</td>
</tr>
<tr>
<td>type T is digits &lt;&gt;;</td>
<td>any float type</td>
</tr>
<tr>
<td>type T is delta &lt;&gt;;</td>
<td>any fixed point type</td>
</tr>
</tbody>
</table>

[*Taken from Ada Language and Methodology*]
Type Parameters

and

The Standard Generic IO Packages

package Text_IO is
    ... non-generic part of Text_IO
generic
    type NUM is range <>;
package Integer_IO is
    ...
end Integer_IO;

generic
    type NUM is digits <>;
package Float_IO is
    ...
end Float_IO;

generic
    type NUM is delta <>;
package Fixed_IO is
    ...
end Fixed_IO;

generic
    type ENUM is (<>);
package Enumeration_IO is
    ...
end Enumeration_IO;
end Text_IO;
Generic Formal Type Parameters
How To Choose?

☐ What operations are performed on the type in the generic body?

☐ How restrictive on the type that the user can choose do you want to be?
Subprogram Parameters

- allow definition and "pass in" of additional operations for generic formal type parameters - especially private and limited private types

- can pass functions or procedures

- formal parameters of generic formal subprogram parameter are checked to ensure match with actual parameters in a call to that subprogram at compile time
program P;
  type Color = (Red, Green, Blue);
  var Bucket : Color;

  procedure Print(C : Color);
  begin
    case C of
      Red : writeln('Red');
      Green : writeln('Green');
      Blue : writeln('Blue');
    end case;
  end;

  procedure Proc(P : procedure);
  begin
    P(Bucket);         (* OK *)
    P(5);              (* RUNtime error *)
  end;

begin
  Proc(Print);
end.
declare
    type Color is (Red, Green, Blue);
    Bucket : Color := Green;

procedure Print(C : in Color) is
begin
    TextIO.Put(Color' Image(C));
end Print;

generic
    with procedure P(Val : Color);
procedure Gen_Proc;

procedure Gen_Proc is
begin
    P(Bucket);  -- OK
    P(5);       -- COMPILe time error
end Gen_Proc;

begin
    Proc;
end;
Subprogram Parameters
An Example

generic
  type Index is (<>);
  type Component is private;
  type Int_Array is array(Index) of Component;
  with function "<"(X,Y:Component)
    return boolean;
procedure Sort_Array(Arr : in out Int_Array);

procedure Sort_Array(Arr : in out Int_Array) is
  Temp : Component;
begin
  for I in Index'Succ(Arr'First)..Arr'Last loop
    for J in Arr'First..Index'Pred(I) loop
      if Arr(I) < Ar(J) then
        Temp := Arr(J);
        Arr(J) := Arr(I);
        Arr(I) := Temp;
      end if;
    end loop;
  end loop;
end Sort_Array;
Subprogram Parameters
An Example cont.

type Day is range 1..31;
type WeatherRec is record
   RainFall : natural;
   AvgTemp : float;
end record;
type WeatherArray is array(Day) of WeatherRec;

function LT(X,Y: WeatherRec) return boolean is
begin
   return X.Rainfall > Y.Rainfall;
end LT;

function "<"(X,Y: WeatherRec) return boolean is
begin
   return X.AvgTemp > Y.AvgTemp;
end "<";

procedure RainSort is new Sort_Array(Day,
   WeatherRec, WeatherArray, LT);

procedure TempSort is new Sort_Array
   (Index->Day, Component->WeatherRec,
    Int_Array->WeatherArray,"<"->"<");

WeatherData : WeatherArray := ( . . . );
begin
   RainSort(WeatherData);
   TempSort(WeatherData); . . . end;
Subprogram Parameters 
and 
Default Values

generic
  type Index is (<>);
  type Component is private;
  type IntLArray is array(Index) of Component;
with function 
"<"(X,Y:Component)
  return boolean is <>;
procedure Sort_Array(Arr : in out IntLArray);

-- in user's program

function 
"<"(X,Y : WeatherRec) return boolean is
begin
  return X.AvgTemp > Y.AvgTemp;
end ";

procedure DefaultSort is new Sort_Array
(Index->Day, Component->WeatherRec,
 Int_Array->WeatherArray);

... DefaultSort(WeatherData); -- will sort on
    -- temp values
Another example:

```plaintext
type SmallRange is range 1..10;
type Values is array(SmallRange) of integer;

procedure IntegerSort is new Sort_Array
    (Index->SmallRange, Component->integer,
     Int_Array->Values);
-- default > for

    V : Values := (....);
begin
    IntegerSort(V); -- default "<" for integers used
end;

-- using Put for subprogram parameter name
-- results in default to generic Put routines
-- in the IO packages
```
Global references inside a generic are resolved to those at point of DECLARATION.

For subprogram parameters, default references resolve to matching names from point of INSTANTIATION.
with Text_IO; use Text_IO;

package Shell is
    Global : integer := 17;
    generic
        with procedure Put(Val : integer) is <>;
    procedure Demo;
end Shell;

package body Shell is
    procedure Demo is
        begin
            Put(Global);
            Put(Global);
        end Demo;
end Shell;

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

with Shell;

package Inner is
    Global : integer := 39;
    procedure Put(I : integer);

    procedure User is new Shell.Demo;
end Inner;

with Text_IO;

package body Inner is
    procedure Put(I : integer) is
        begin
            Text_IO.Put(" Surprise" & integer'image(I));
        end Put;
end Inner;

... Inner.User; ...
Subprogram Parameters and Access Type Parameters

An Example

generic
  type KeyType is private;
  type ElementType is private;
  with function "<"(Left,Right : KeyType)
    return boolean is <>;
package BinaryTreeMaker is
  type Kind is private;
  function Make return Kind;
  function IsEmpty(T : Kind) return boolean;
  procedure Insert(T : in out Kind;
    K : KeyType;
    E : ElementType);
  function Retrieve(T : Kind; K : KeyType)
    return ElementType;
  KeyNotFound : exception;

  generic
    with procedure Operation(K : KeyType;
    E : ElementType);
  procedure InorderTraverse(TheTree: in Kind);
private
  type InternalRecord;
  type Kind is access InternalRecord;
end BinaryTreeMaker;
with BinaryTreeMaker;

package EmployeeDataBase is
  NameLength : constant := 40;
  subtype NameType is string(1..NameLength);
  type Dollar is delta 0.01 range 0.0..1.0e8;
  type AgeType is range 0 .. 150;
  type YearType is range 1900..2100;
  type EmployeeInfo is record
    Salary : Dollar;
    Age : AgeType;
    Hired : YearType;
  end record;

package EmployeeTree is new
  BinaryTreeMaker(KeyType=>NameType,
                  ElementType=>EmployeeInfo);

  RootNode : EmployeeTree.Kind;
end EmployeeDataBase;
with EmployeeDataBase; use EmployeeDataBase;
with Text_IO; use Text_IO;

procedure PrintReports is
    package SalaryIO is new Fixed_IO(Dollar);
    package AgeIO is new Integer_IO(AgeType);
    use SalaryIO, AgeIO;

    procedure PrintSalary(Key : NameType;
        Info : EmployeeInfo) is
    begin
        ... Put(Info.Salary);
    end;

    procedure PrintAge(Key : NameType;
        Info : EmployeeInfo) is
    begin
        ... Put(Info.Age);
    end;

    procedure ReportSalaries is new
        EmployeeTree.InorderTraverse
            (Operation => PrintSalary);

    procedure ReportAge is new
        EmployeeTree.InorderTraverse
            (Operation => PrintAge);
    begin
        ReportSalaries(RootNode);
        New_Line;
        ReportAges(RootNode);
    end PrintReports;

[*From Understanding Ada *]
Subprogram Parameters
and
Handling Exceptions

generic
package Stack is
    ... same as before

    Overflow, Underflow : exception;
end Stack;

-- in user's program

    package S1 is new Stack;
    package S2 is new Stack;

begin
    S1.Push(5);
    S2.Pop(Item);
exception
    when S1.Underflow => ...;
    when S1.Overflow => ...;
    when S2.Underflow => ...;
    when S2.Overflow => ...;
end;
Subprogram Parameters and Handling Exceptions cont.

Cannot pass exceptions as generic parameter

```pascal
generic
  When_Error : exception;  -- NOT allowed
...
procedure X ...
...
exception
  when others -> raise When_Error;
end X;

My_Exception : exception;
procedure S is new X(My_Exception);
...
begin
  S;
exception
  when My_Exception -> ...; -- NOT allowed
end;
```
Subprogram Parameters
and
Handling Exceptions cont.

generic
  with procedure OverflowHandler;
package Stack is
  ... same as before;
end Stack;

package body Stack is

  ... in Push procedure ...
  when Constraint_Error -> OverflowHandler;

end Stack;

-- in user program
with Stack;
...
procedure OverflowHandler is
begin
  Text_IO.Put_Line("Overflow has occurred");
end OverflowHandler;

package S1 is new Stack(OverflowHandler);

begin
  ... 
  S1.Push(5); -- if overflow occurs msg prints
end;
Generic Can'ts

☐ No generic SUBtype parameters, only TYPES

☐ No generic record types

☐ No generic tasks

☐ Wrap a package around it
What are the Cons of Generics?

- Takes longer/is harder to write generic code
- Usually some efficiency sacrificed for the generality -- use of application specifics could lead to increased efficiency
- Difficult to make component robust/reliable enough to survive all uses
What are the Pros of generics?

- Reusability - no reinventing the wheel for each specific application

- Levels of abstraction added - separation of abstraction and implementation

- Source code size of user programs reduced
  - Maintainability, readability, and understandability increased
  - Verification more manageable

- When used in conjunction with user-defined types increases portability across machines

- Provides necessary answer to strong typing without sacrificing increased reliability of compile time checks

- Provides flexible IO packages which can be used (if needed) for predefined AND user-defined types
Unresolved Issues in Generics

□ Compiler Issues

■ Use “code sharing” or “code copying” to implement generics

□ Management Issues

■ How to facilitate creation of generic units
  □ In retrospect, after recognizing similarity in produced units
  □ Beforehand using "domain analysis"

■ How to manage storage and retrieval of units in a library of generic units

■ How to "publicize" availability of units in generic library and provide criterion for selecting proper unit

■ How to manage updating of used generic units as "bugs" are uncovered

□ Legal Issues

■ Who owns the generic module

■ Who is liable for the generic module's performance

[*See Software Components with Ada]
How do you TEACH generics?

☐ Necessary as IO is an issue arising early and should not be kept a "magic" process

☐ One key is to use concrete examples

☐ Driver's licence form is a generic template -- individual's license is a usable instantiation

☐ One key is to tie to previous learning

☐ Use old/familiar packages, procedures, and functions - Stacks, Swap, etc.
END
DATE
FILMED
6-88
DTIC