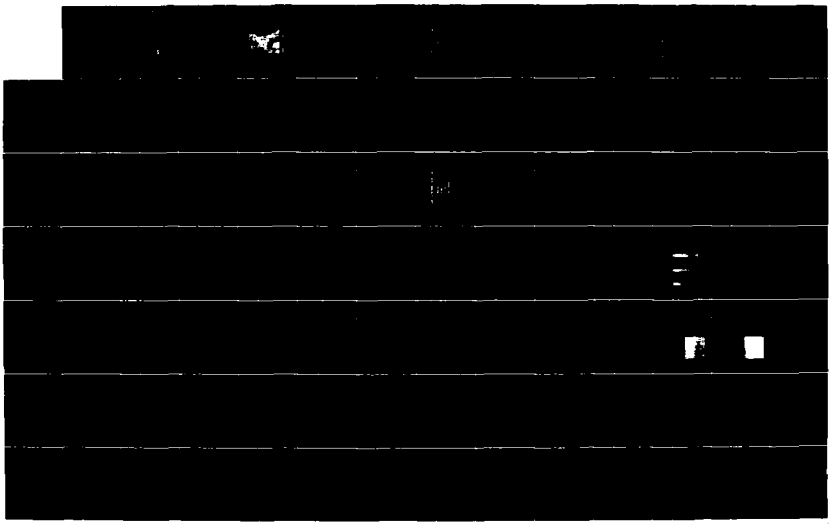
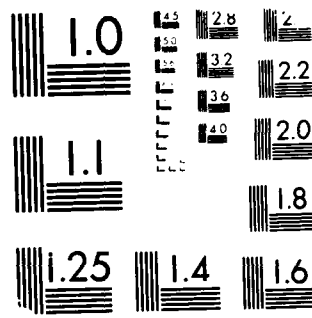


AD-A189 641 ADVANCED ADA WORKSHOP AUGUST 1987(U) ADA JOINT PROGRAM 1/4
OFFICE ARLINGTON VA 21 AUG 87

UNCLASSIFIED

F/G 12/3 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

UNCLASSIFIED

DTIC FILE COPY

E (When Data Entered)

IDENTIFICATION PAGE

AD-A189 641

12. GOVT ACCESSION NO.

READ INSTRUCTIONS
BEFORE COMPLETING FORM

3. RECIPIENT'S CATALOG NUMBER

Advanced Ada Workshop, August, 1987

5. TYPE OF REPORT & PERIOD COVERED
Tutorial, Aug. 17-21, 1987

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)

Ada Software Engineering Education and Training Team (ASEET)

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION AND ADDRESS

Ada Software Education and Training Team
Ada Joint Program Office, 3E114, The
Pentagon, Washington, D.C. 20301-308110. PROGRAM ELEMENT, PROJECT, TASK
AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS

Ada Joint Program Office
3E 114, The Pentagon
Washington, DC 20301-3081

12. REPORT DATE

3 December, 1986

13. NUMBER OF PAGES

380

14. MONITORING AGENCY NAME & ADDRESS (If different from Controlling Office)

Ada Joint Program Office

15. SECURITY CLASS (of this report)

UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING
SCHEDULE

N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20. If different from Report)

UNCLASSIFIED

18. SUPPLEMENTARY NOTES

19. KEYWORDS (Continue on reverse side if necessary and identify by block number)

Ada Programming language, Ada Training, Education, Training, Computer
Programs, Ada Joint Program Office, AJPO, ←

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This document contains prints of slides presented at the Advanced Ada
Workshop, Monday, 17 August to Friday, 21 August, 1987. Keywords: →DTIC
SELECTED
JAN 06 1988
S
C
DDD FORM 1473
1 JAN 73EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ADVANCED Ada WORKSHOP

Applied Ada Software Engineering

Capt Roger D. Beauman

Capt Michael S. Simpson

Ada Software Engineering Education and
Training (ASEET) Team

A black and white woodblock-style illustration of a woman in a kimono, looking down with a sad expression, holding a long object like a fan or a stick. The style is reminiscent of traditional Japanese art, with bold lines and a focus on the figure's posture and expression. The woman is shown from the waist up, wearing a kimono with a patterned collar. Her hair is styled in a traditional manner, and she has a small flower in it. The background is plain white, emphasizing the figure.

				A-1
				A-1

A-1



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**

CHANGING COMPUTER TECHNOLOGY

**REALTIME AND
PARALLEL PROCESSING**

DISTRIBUTED DATABASES

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

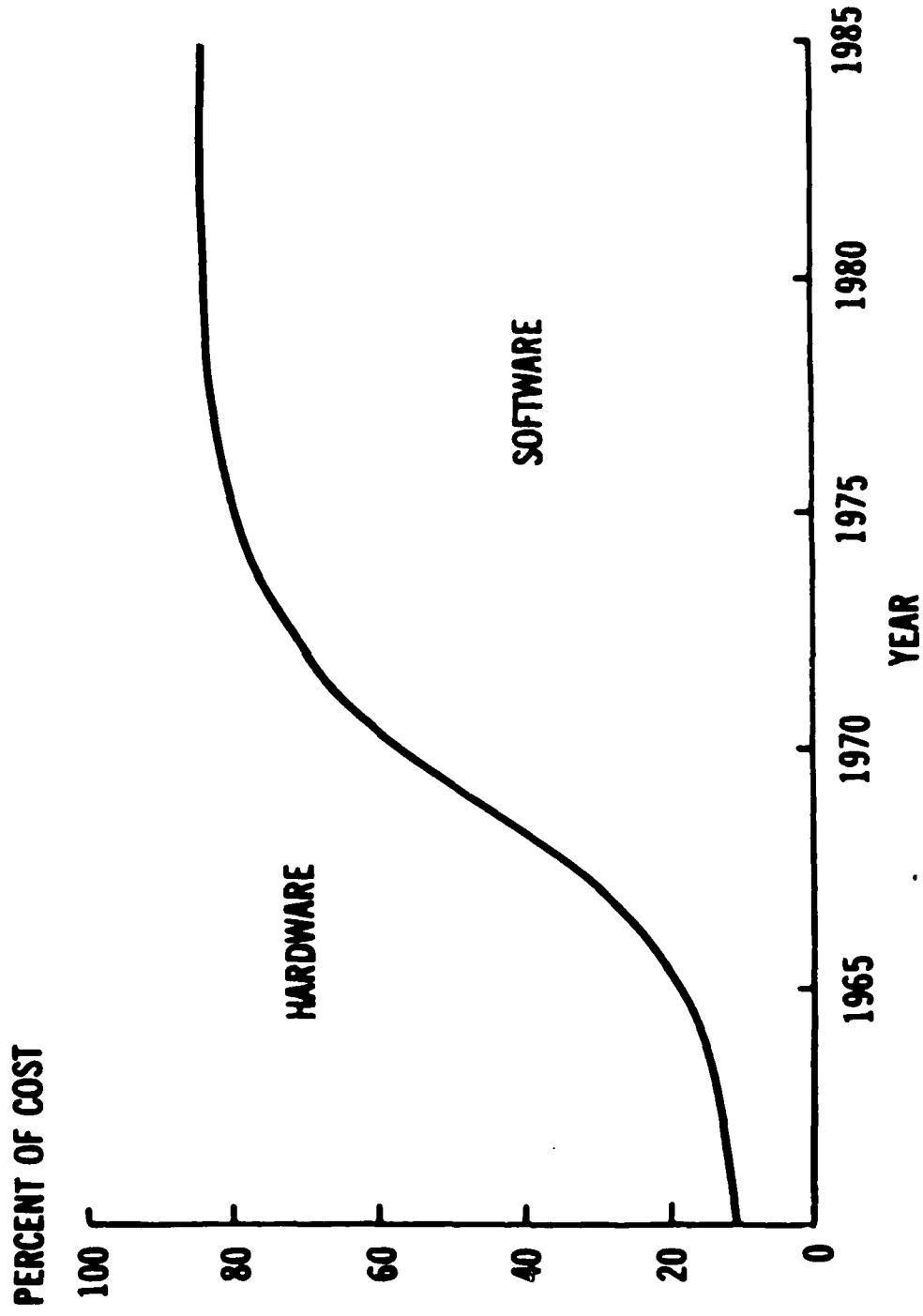
HARDWARE CAPABILITIES

- * 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

HARDWARE/SOFTWARE COST TRENDS



A MACRO SOLUTION

- * Greater Use of Automation**
- * Higher Levels of Abstraction**
- * Reuseability**
 - Isolate Commonality**
 - Create Workable Abstractions**
 - Reuseable 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 $\Rightarrow$

- * Standards

- * Guidelines

- * Practices

SOFTWARE ENGINEERING

TO SUPPORT COMPUTING WHICH IS =>

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

THROUGHOUT THE LIFE CYCLE OF A SYSTEM

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

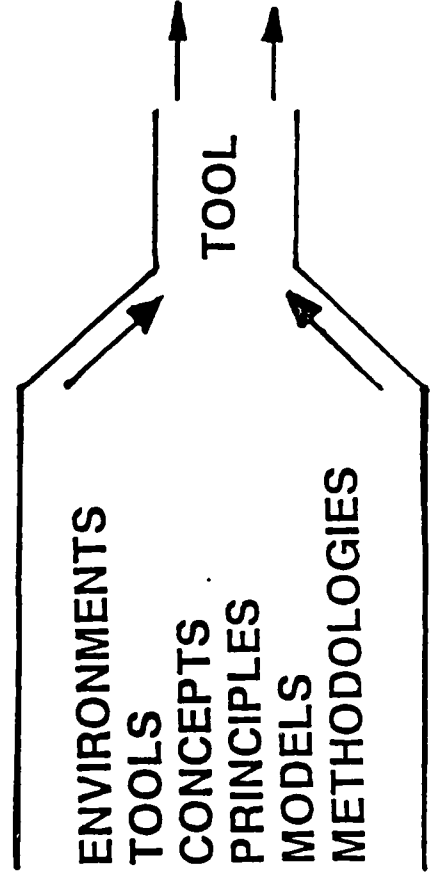
Programming Languages

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

STANDARDS

GUIDELINES

PRACTICES



A PRACTICAL SOLUTION

Ada

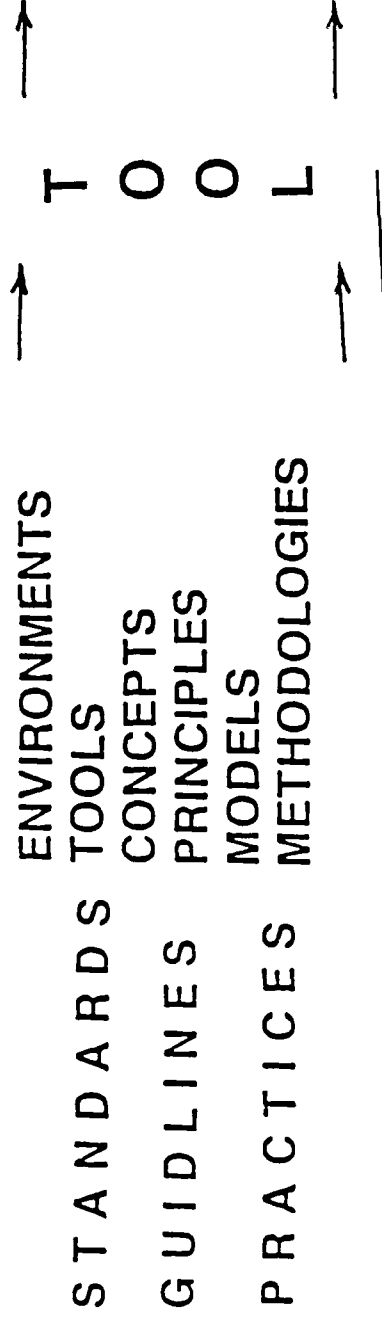
Ada and Software Engineering

- * 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



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

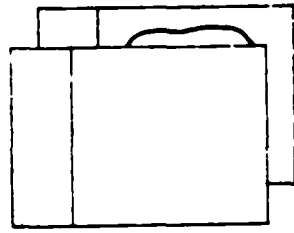
READABILITY

- * 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

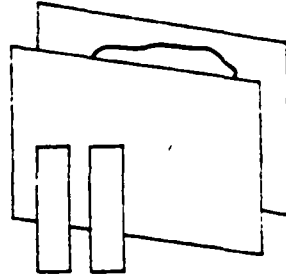
- * Components of Ada which together form a working Ada software system
- * Express the architecture of a system
- * Define and enforce interfaces

PROGRAM UNITS



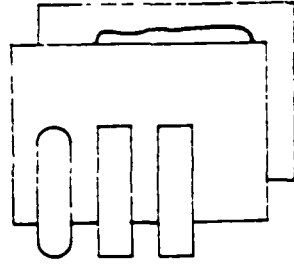
SUBPROGRAMS

Working components that perform some action



TASKS

Performs actions in parallel with other program units



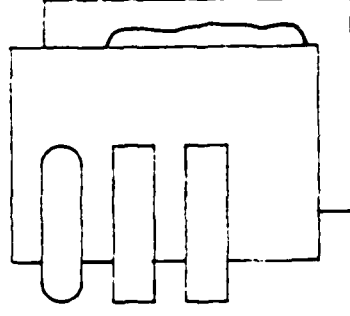
PACKAGES

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)



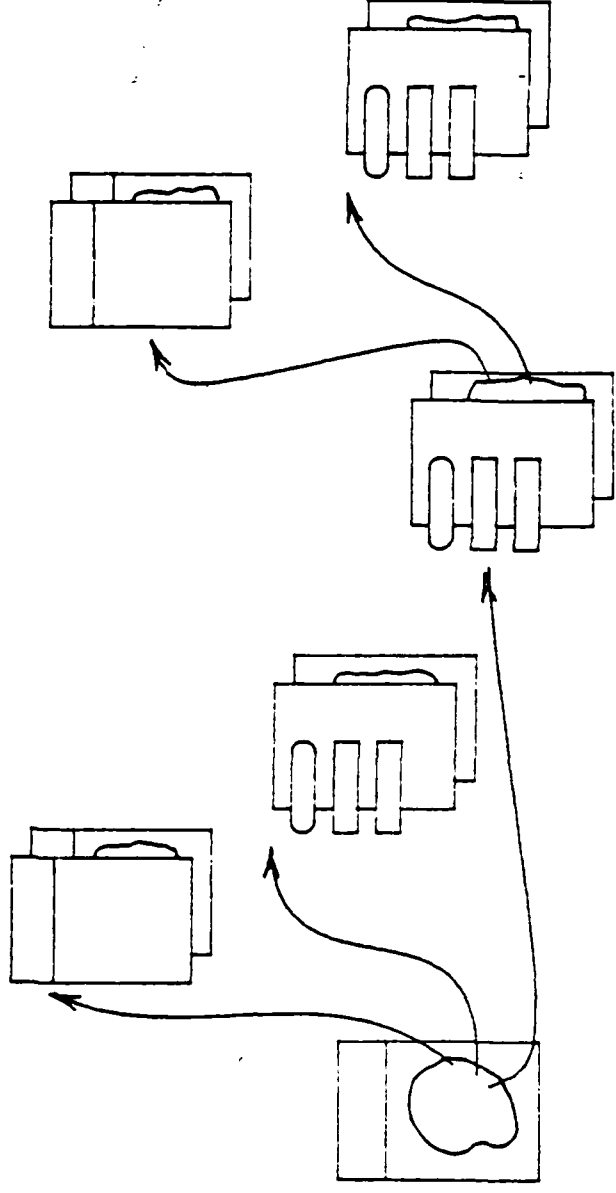
BODY: Defines the implementation of the program unit (the HOW)

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

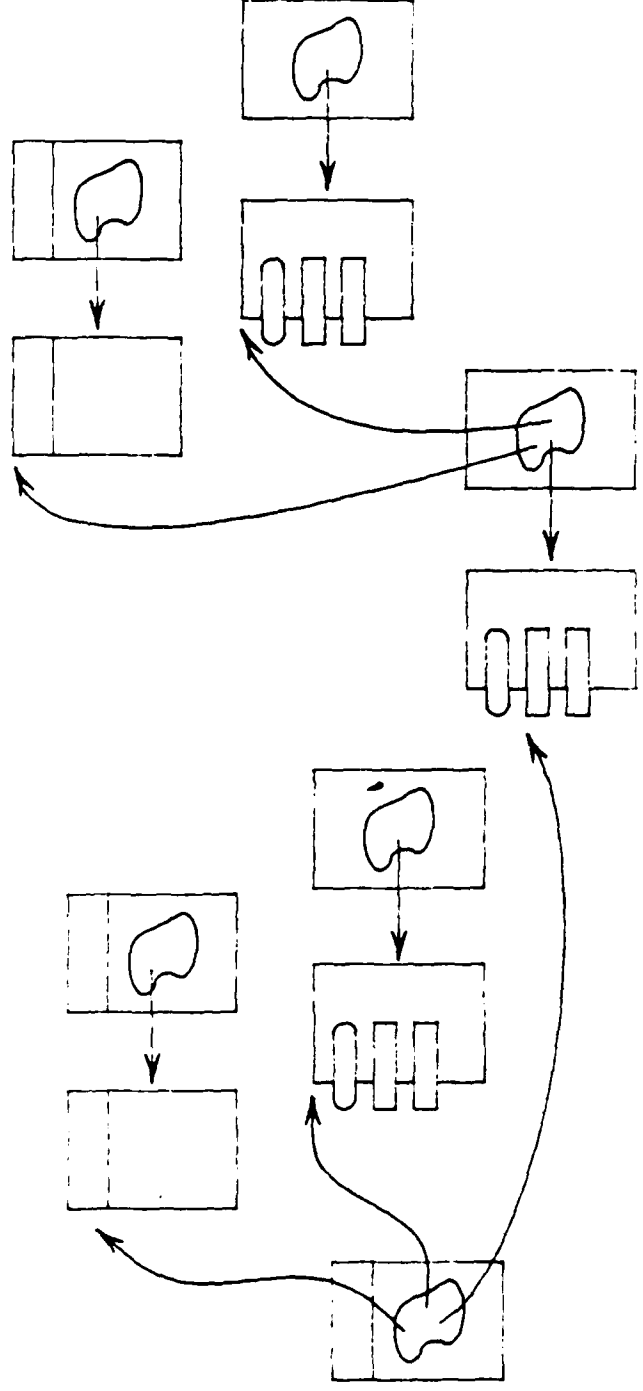
SEPARATE COMPILE

- * 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 COMPIRATION

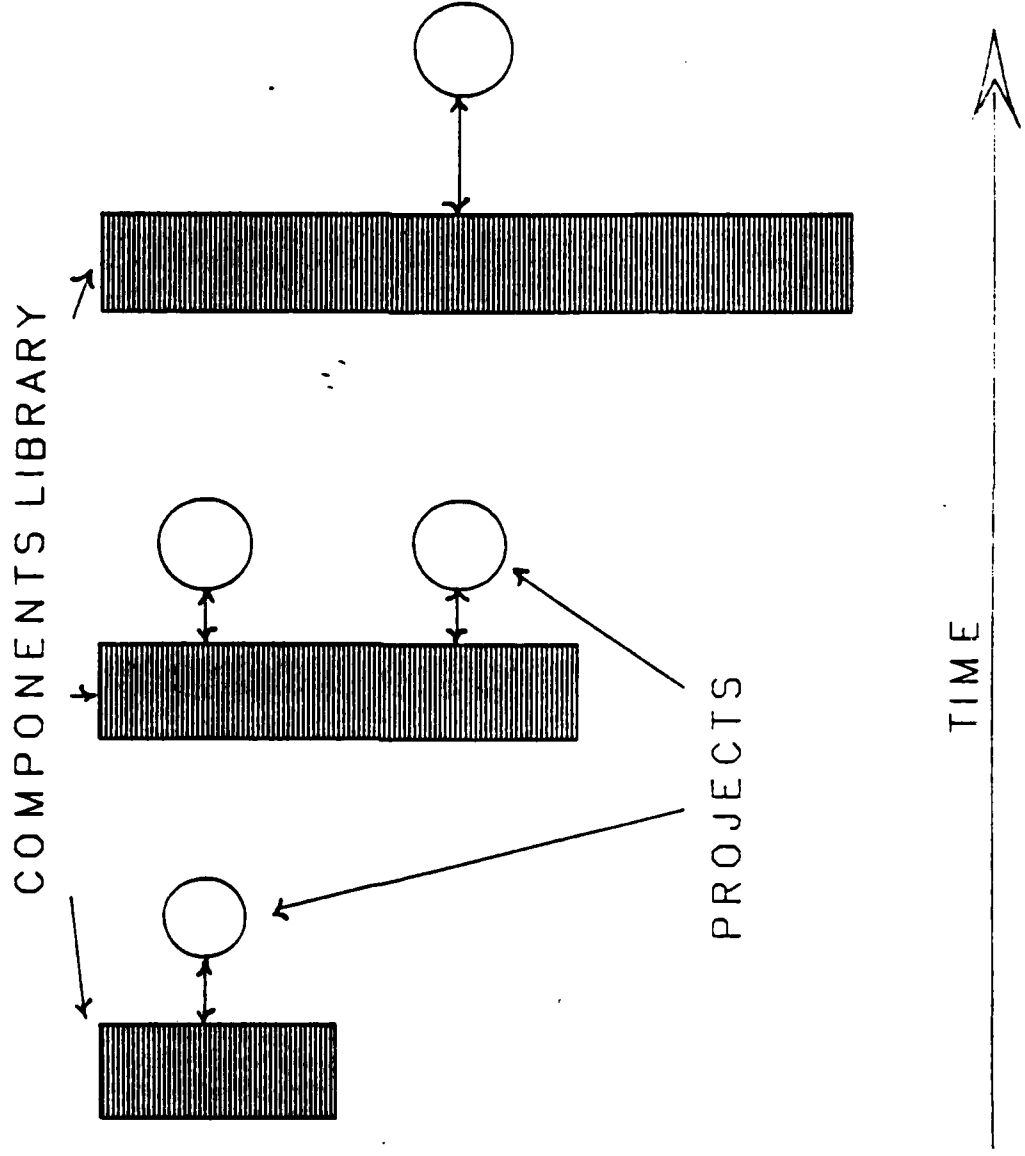
- * A program unit's specification may be compiled separately from its body
- * Realizes not only a logical distinction between architecture and implementation, but also a physical distinction



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

SOFTWARE COMPONENTS



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

SUBPROGRAMS

- * A program unit that performs a particular action
 - Procedures
 - Functions
- * Contains an interface (parameter part)
mechanism to pass data to and from the subprogram
- * The basic discrete component which acts like
a black box
- * Gives ability to express abstract actions

MAJOR FEATURES OF Ada

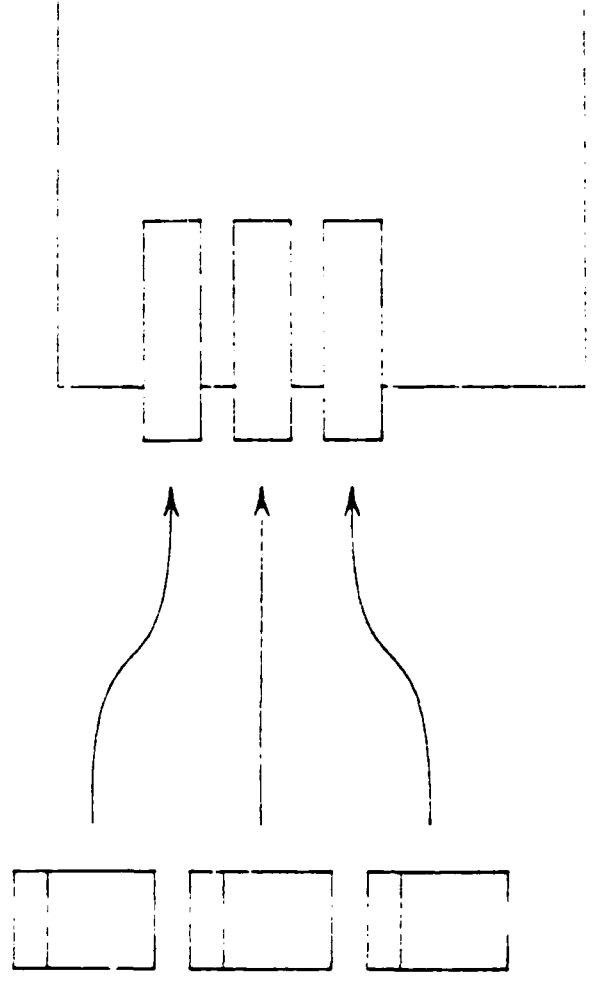
- * Packages
- * Strong Typing
- * Typing Structures
- * Data Abstraction
- * Tasks
- * Exceptions
- * Generics

PACKAGES

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

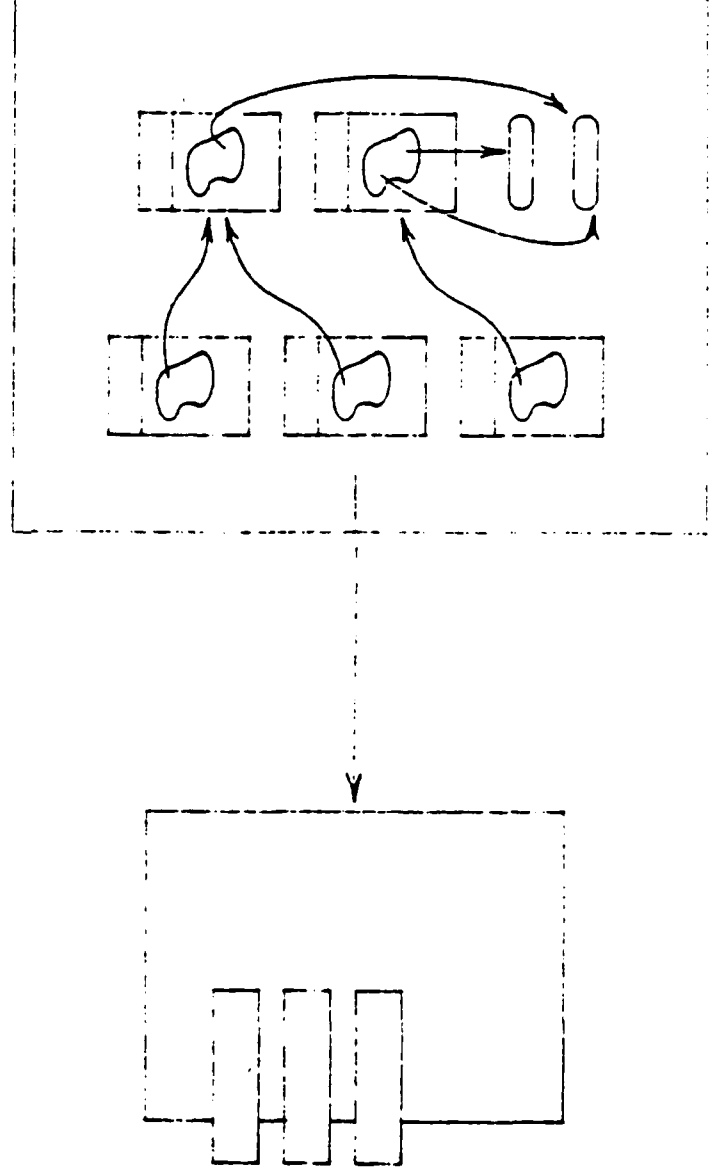
PACKAGES

- * 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:

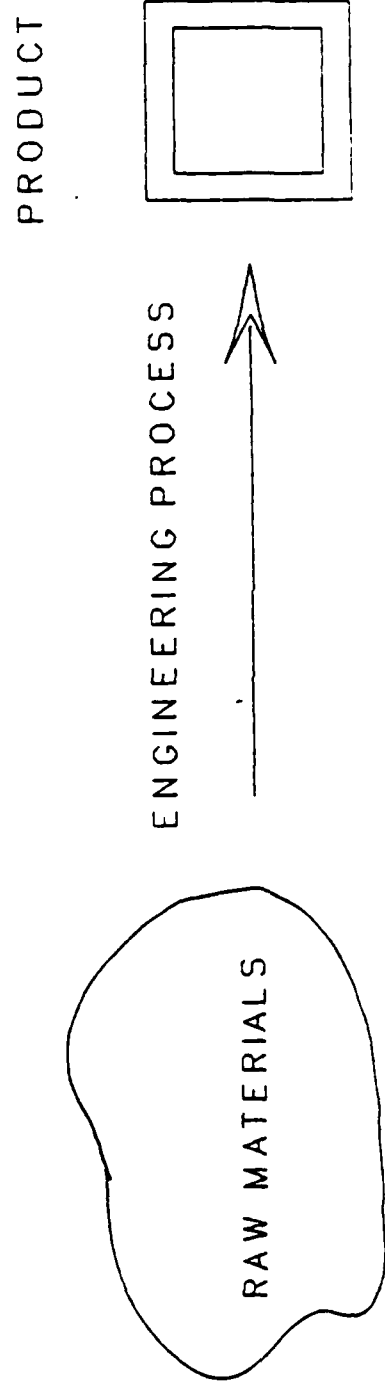
- * Abstraction
 - * Information hiding
 - * Modularity
 - * Localization
-
- * Understandability
 - * Efficiency
 - * Reliability and safety
 - * Modifiability
 - * Correctness

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

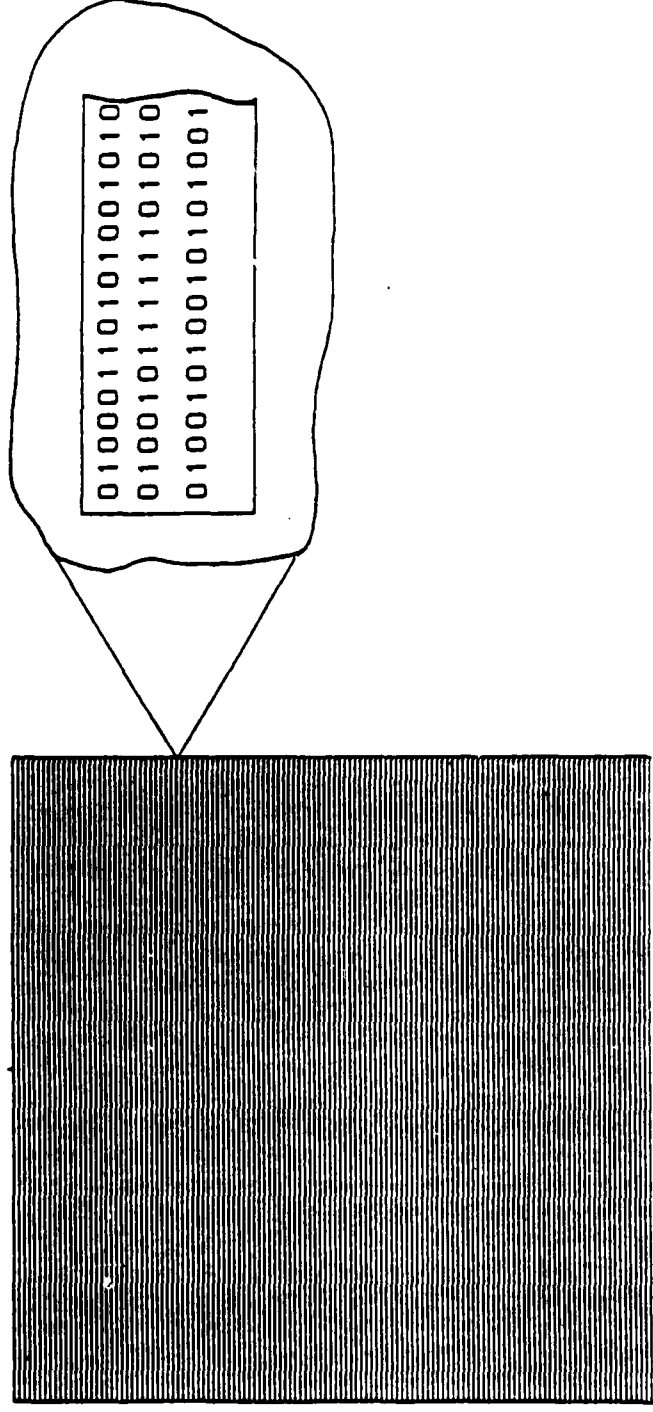


STRUCTURING RAW MATERIALS

- * There is a requirement to structure raw materials
 - To quantify
 - To manage
 - To test
 - To validate
- * Methods of structuring vary across disciplines

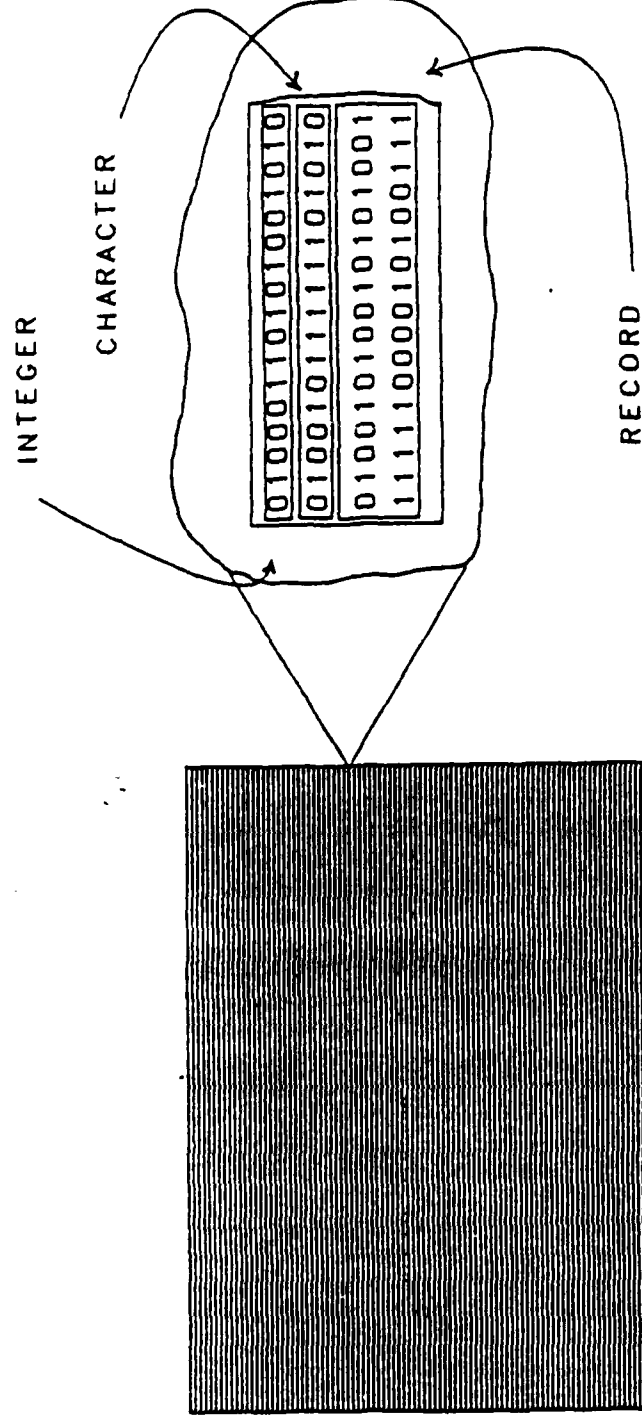
SOME RAW MATERIALS OF SOFTWARE ENGINEERING

- * 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

TYPING STRUCTURES

- * Variety of problems requires a variety of structuring capabilities
- * Ada provides a rich variety of 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

Integers

--Define a set of exact, consecutive values

USER DEFINED

type ALTITUDE is range 0..100_000;

type DEPTH is range 0..20_000;

PLANES_HEIGHT : ALTITUDE;

DIVER_DEPTH : DEPTH;

begin

PLANES_HEIGHT := 10_000;

PLANES_HEIGHT := 200_000; --- error

PLANES_HEIGHT := DIVER_DEPTH; --- error

end;

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

(~7S,~) (p~3 1\ (775 7) (p~c 3\

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

Types

Arrays constrained
 unconstrained

CONSTRAINED

-- Indices are static for all objects of that type

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);

MY_HOURS(SUN)	MY_HOURS(MON)	MY_HOURS(TUE)	MY_HOURS(SAT)
0	8	8	0

Types

Records

undiscriminated
discriminated
variant

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

TUE
26
NOV
1985

DAY_OF_WEEK

DAY_NUMBER

MONTH_NAME

YEAR_NUMBER

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 .. 9;
    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
- * Elimination of data transfer and control

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 TEXTJO; use TEXTJO;
procedure COUNT_NUMBERS is
package INTJO is new INTEGERJO (INTEGER);
use INTJO;
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

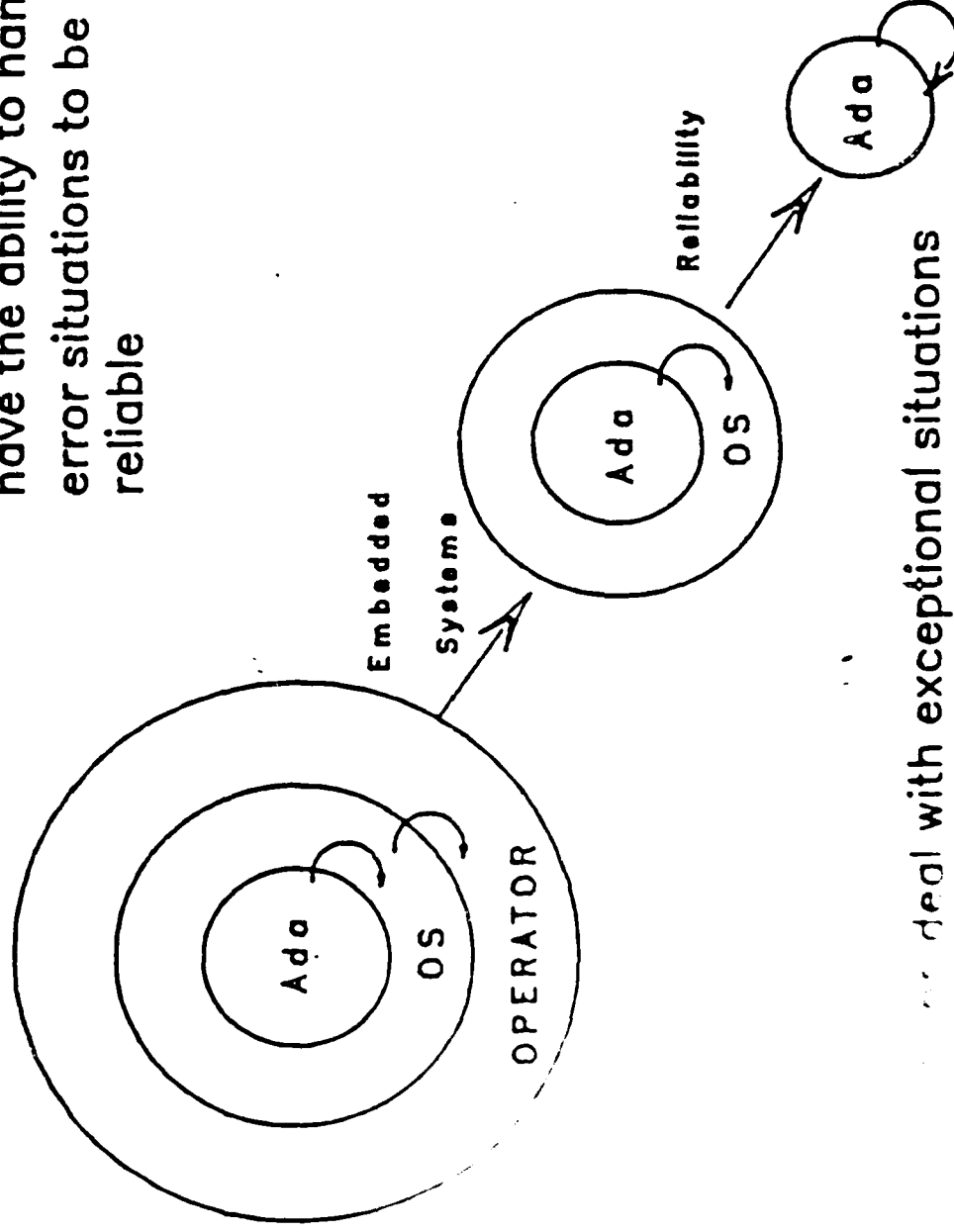
- * Errors will occur
 - Hardware
 - Software
- * Real time systems must be able to operate in a degraded mode
- * Reliability and safety must be engineered into a system
- * Traditional languages lack specific features for dealing with errors and exceptional situations

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
 - Propagate exception
- * Increase reliability
- * Reduce complexity

Exceptions

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



deal with exceptional situations

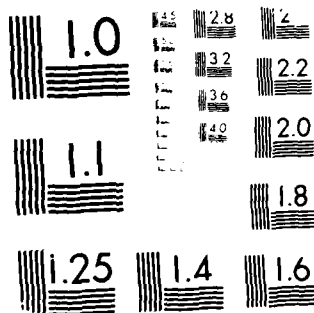
AD-A189 641

ADVANCED ADA WORKSHOP AUGUST 1987(U) ADA JOINT PROGRAM 2/4
OFFICE ARLINGTON VA 21 AUG 87

UNCLASSIFIED

F/G 12/5 NL





MICROCOPY RESOLUTION TEST CHART
 NATIONAL BUREAU OF STANDARDS-1963-A

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

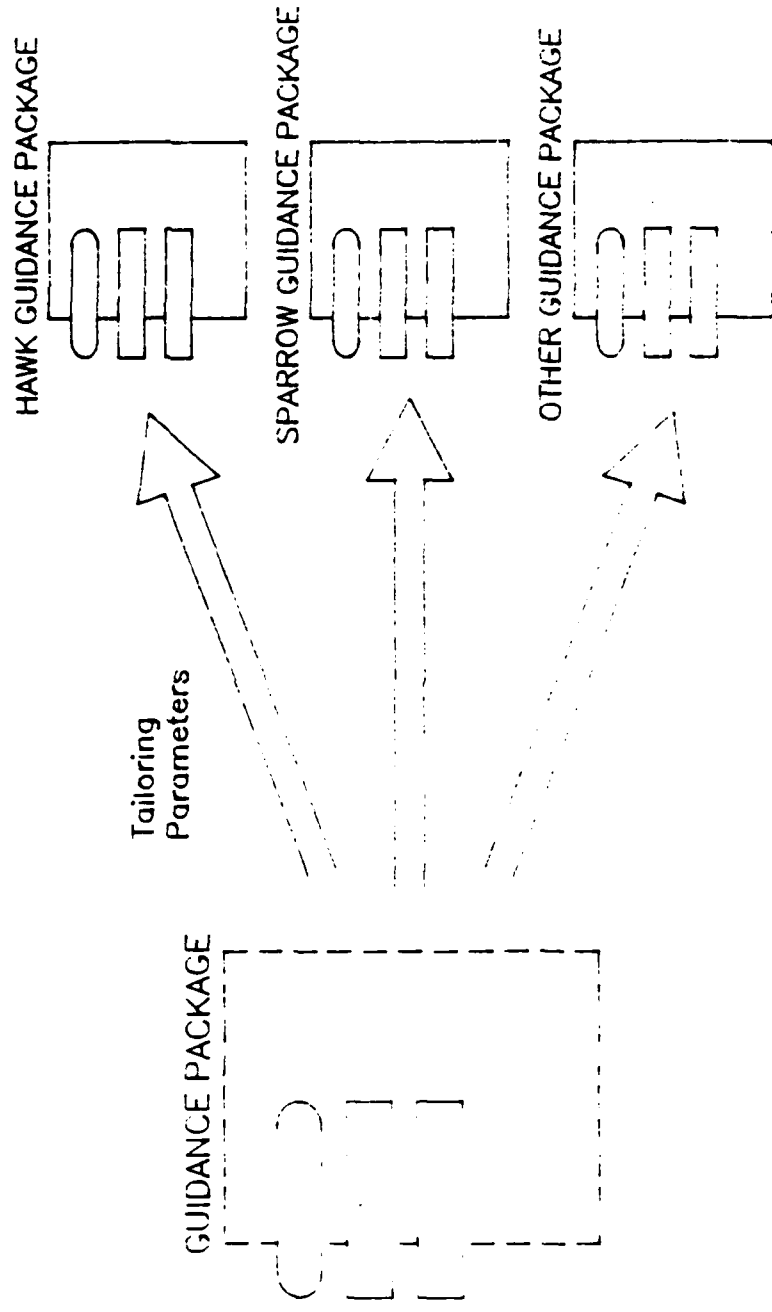
```
begin
loop
begin
  GET ( A_NUMBER );
  NEW_LINE;
  PUT ( "The number is " );
  PUT ( A_NUMBER );
  NEW_LINE;
exception
  when DATA_ERROR => PUT_LINE("Bad number, try again");
end;
end loop;
end GET_NUMBERS;
```

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



GENERIC

- * 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;
```

```
begin
```

```
    TEMP    := FIRST_INTEGER;  
    FIRST_INTEGER := SECOND_INTEGER;  
    SECOND_INTEGER := TEMP;
```

```
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(CHARACTER);

 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
UNIFORMITY
- * 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

Ada Generics

**Part of the
Advanced Ada Workshop**

**Sponsored by the
Ada Software Engineering Education & Training Team
(ASEET)**

**John Bailey
Software Metrics, Inc.**

**703-533-8300
jbailey at Ada20**

18 August 1987

Outline

1. Rationale

**Are generics really necessary in Ada?
What can they do?**

2. Syntax and Semantics

**Generic parameters
Instantiation
Compared with other Ada units**

3. Examples

**See how simple they can be.
See how useful they can be.
See how complicated they can be.**

4. Limitations

**Some generalization is not easy.
Some generalization is not possible.**

5. Advanced Usage

**Retrofitting generics
Generalization judgement calls**

1. Rationale

Are generics really necessary?

Ada language goals:

- Encapsulation of processing desired**
- Encapsulation of resources (objects) desired**
- User-defined types desired**
- Strong, static type checking desired**
- Unnecessary language features not desired**
- Reusability desired**

Other languages answer some of these goals, but not all.

Fortran

- Encapsulation of processing supported**
- Encapsulation of resources (mostly)**
- No user-defined types**
- No static type checking**
- I/O routines part of language definition**
- Some reusability support**

Pascal

- Encapsulation of processing supported**
- No encapsulation of resources**
- User-defined types**
- Strong, static type checking (with loopholes)**
- I/O routines part of language definition**
- No reusability support**

Smalltalk

- Encapsulation of processing supported**
- Encapsulation of resources supported**
- User-defined types supported**
- No static type checking**
- Only primitive I/O language-defined**
- Excellent reusability support**

Generics Are Solution

Some of the above goals appear to be in conflict:

- **Reusability vs. Strong static type checking -- ?**

```
type User_Type is new Integer;  
procedure Increment (X : in out User_Type) is  
begin  
    X := X + 1;  
end Increment;
```

Cannot reuse Increment anywhere else.

- **Strong checking vs. Minimal language -- ?**

```
type User_Type is new Integer;  
X : User_Type;  
I : Integer;  
Put (X);           -- for strong checking, these  
Put (I);           -- must be distinct "Put's"
```

Pascal solves this with "magic" I/O procedures
- extended language

```
Write (X);  
Write (I);  
Write (X, I);  
Write ('The answer is ', X, ' or ', I);
```

Smalltalk provides primitive I/O with each new type

Might not be wanted.

Does not provide static checking

This allows great flexibility but can lead to
runtime errors that could have been
avoided by compilation-time checking

The Case for Generics

Ada generics allow excellent compromise

Minimal additional complexity

**Encapsulation of processing
Generic subprograms**

**Encapsulation of resources (objects)
Generic packages**

**User-defined types desired
Strong, static type checking desired**

**Unnecessary language features avoided
I/O not part of language (rather, part
of the standard environment)**

**Reusability supported
Reasonable flexibility
Capability to separate essential detail
of an algorithm or object from
problem-specific detail**

(Hint: that last point is the essence of generic programming)

What can Generics Do?

Consider the rationale for programming anything:

A programmed solution:

**Generalizes over several occurrences
Appropriate if similar processing,
but with varying values**

Typical programs contain:

- **Conceptual "chunks" related to the general problem class:
Algorithms
Objects or classes of objects**
- **Details about the specific problem
Specific types
Specific routines, such as error recovery
Specific data flow designs**
- **At run time, a specific case is handled
Values specific to a given run are used
Same program is "reused" with next set of values**

Likewise, generic programming uses:

- **Conceptual chunks, object classes**
- **Problem-specifics**
- **Run-time specifics**

Traditional Programming - Diagram

Algorithms, Objects, Resources

- - Intertwined with - -

Problem-specific declarations

package Useful_Object is

type Specific_Type is (Something_or_other);

procedure Do_Something (To : Specific_Type);

**function Status_Of (An_Object_Of : Specific_Type)
return Predefined_Type_Perhaps;**

end Useful_Object;

Generic Programming - Diagram

application-domain
algorithms

application-domain
object classes

problem-specific declarations

problem-specific processing

instantiations of the application-domain
chunks (above the dashed line)

generic

type Formal_Type is private;

package General_Object is

procedure Do_Something (To : Formal_Type);

function Status_Of (An_Object_Of : Formal_Type)
return Predefined_Type_Perhaps;

end General_Object;

type Specific_Type is (Something_or_other);

package Useful_Object is new
General_Object (Specific_Type);

Another Quick Example

Frequently, the following occurs in Ada programs:

```
package Global_Types is  
    type Useful is (This, That, The_Other);  
end Global_Types;
```

```
with Global_Types;  
package Service is  
    procedure Operation  
        (On : Global_Types.Useful);  
end Service;
```

```
with Global_Types;  
with Service;  
procedure User is  
    My_Object : Global_Types.Useful;  
begin  
    Service.Operation (My_Object);  
end User;
```

**Cannot separately use Service without also using
Global_Types due to visibility requirements.**

Alternative, Equivalent Program

```
package Global_Types is  
    type Useful is (Whatever);  
end Global_Types;
```

```
    -- no context clause  
generic  
    type Formal is private;  
package General_Service is  
    procedure Operation (On : Formal);  
end General_Service;
```

```
with Service;  
with Global_Types;  
procedure User is  
    My_Object : Global_Types.Useful;  
    package Service is new  
        General_Service (Global_Types.Useful);  
begin  
    Service.Operation (My_Object);  
end User;
```

**Now, General_Service can be reused without
Global_Types.**

**Just as a program becomes a specific case for a
given run, a generic program is instantiated
into a specific case for a given program.**

Another Reason Generics are Unavoidable

With only predefined types, reusability is simplified:

```
function Math_Operation (X : Real) return Real;
```

**If users only have type Real for floating point,
the above function is always usable**

However:

Ada allows user-defined types, such as:

```
type Low_Precision is digits 3;  
type High_Precision is digits 7;
```

**And, Ada requires strong static type checking, so
if Math_Operation were needed for both of
the above types, two functions would be needed:**

```
function Math_Operation (X : Low_Precision)  
  return Low_Precision;
```

```
function Math_Operation (X : High_Precision)  
  return High_Precision;
```

Generics solve this problem:

```
generic  
  type Real is digits <>;  
function Math_Operation (X : Real) return Real;
```

Analogy with Programming

Typical programming:

A programmed solution can be written once and "used" several times. The following motivate the creation of a programmed solution:

Reusability:

Similar processing will be required repeatedly.

Reliability:

Testing and verification can be performed to help ensure all runs will be correct.

Readability:

By allowing variables to stand for specific values during run time, the program can be understood in the abstract.

Maintainability:

Making a change in the program will apply to all usages.

Generic Programming:

All the same arguments can be applied to generic programming.

Reusability:

Similar program components needed repeatedly, but different enough to preclude run time parameterization.

Reliability:

A properly tested generic need not be retested each time it is used.

Readability:

By supressing problem-specific detail the higher-level concept can be understood.

Maintainability:

By helping to avoid repetition, the (hopefully single) location where a change is required is easier to determine.

Simplified View

Just as types are templates for describing objects,

Generics are merely templates for other program units.

Generic packages

Generic subprograms

Generic procedures

Generic functions

There are no generic tasks

- a task is already an object of a type.**
- a generic package can contain a task**

**A generic (template) is instantiated by a
declaration just as an object is an "instance"
of a type (template).**

**This instantiation is accomplished by the compiler,
and not (necessarily) at run time.**

**The effect of Ada generics can be obtained through
editor-like substitution (but this is not the
smartest implementation of them)**

Sometimes likened to assembly language Macros.

Simple Examples

If you can write an Ada package, procedure, or function
(you can, can't you?)
then you can write an Ada generic:

```
procedure Easy is
begin
    Text_io.Put_Line ("pie");
end Easy;
```

```
-- a call to Easy:
Easy;
```

Generic version:

```
generic
procedure Easy is
begin
    Text_io.Put_Line ("pie");
end Easy;
```

```
-- first an instantiation:
procedure Easy_Instance is new Easy;
```

```
-- then a call:
Easy_Instance;
```

Note the need to instantiate first, then treat as a
regular procedure.

The above generic is a trivial case with no parameters.

It's possible (but not too likely) that such a simple
generic might be useful.

**The preceeding example must have been in the scope
of package Text_lo.**

**Or, it could have been a library unit with its own
context clause:**

```
with Text_lo;  
generic  
procedure Easy is  
begin  
    Text_lo.Put_Line ("ple");  
end Easy;
```

Generics are one of the possible library units.

Recalling the Ada grammar:

Library_Unit ::=

subprogram_declaration

package_declaration

generic_declaration

generic_instantiation

subprogram_body

Note that a generic instantiation can also be a library unit.

Given the foregoing, the following two lines also form a library unit:

```
with Easy;  
procedure Easy_Instance is new Easy;
```

And a user could be:

```
with Easy_Instance;  
procedure User is  
begin  
    Easy_Instance;  
end User;
```

Resulting in "ple" being printed.

These trivial cases should illustrate the hidden simplicity in the declaration and use of Ada generics.

More Useful Example

Normally, one or more parameters would be used to allow a variety of instances to be declared.

```
generic
    Prompt : String := "A>";
procedure Issue_Prompt;

with Text_io;
procedure Issue_Prompt is
begin
    Text_io.Put (Prompt);
end Issue_Prompt;
```

Note, the context clause could have preceeded the spec but it isn't needed until the body.

Note that the separate spec and body is Required with generic subprograms, unlike regular subprograms.

The following is Not allowed:

```
with Text_io;
generic
    Prompt : String := "A>";
procedure Issue_Prompt is
begin
    Text_io.Put (Prompt);
end Issue_Prompt;
```

Instantiations

The above could be instantiated with the following:

```
with Issue_Prompt;  
with Text_io;  
function User_Reply return String is  
    procedure Prompt is new  
        Issue_Prompt ("What is your wish? ");  
    Buffer : String (1..80);  
    Length: Natural;  
begin  
    Prompt;  
    Text_io.Get_Line (Buffer, Length);  
    return Buffer (1..Length);  
end User_Reply;
```

```
with Issue_Prompt;  
with Text_io;  
function User_Reply (To_Question: String := "")  
    return String is  
    procedure Prompt is new  
        Issue_Prompt (To_Question);  
    Buffer : String (1..80);  
    Length: Natural;  
begin  
    Prompt;  
    Text_io.Get_Line (Buffer, Length);  
    return Buffer (1..Length);  
end User_Reply;
```

Continued...

Resp1 : constant String := User_Reply; --what's wrong?

Resp2 : constant String := User_Reply ("Why? ");

Resp3 : constant String := User_Reply;

What Is your wish? __

Why? __

A> __

Another Example

Generic formal type parameter (any floating point type)
followed by an object parameter of that type:

```
generic
    type Real is digits <>;
    Max : Real := Real'Last;
    procedure Useless;

    procedure Useless is
        Local : Real := Max / 2.0;
    begin
        if Real'Last > Max then
            Local := Real'Last / 2.0;
        end if;
    end Useless;

    with Useless;
    procedure Useless_User is
        type Coarse is digits 3;
        procedure Zip is new Useless
            (Real => Coarse,
             Max => 10.0);
        procedure Zap is new Useless (Coarse);
        procedure Zot is new Useless (Float, -1.0);
    begin
        Zip;
    end Useless_User;
```


Object Parameters - Detail

As with subprogram parameters, generic formal parameters can have modes:

Mode in is the default.

```
generic
    Obj1 : Integer;
    Obj2 : in Integer;
procedure Testing;

procedure Testing is
begin
    Obj1 := Obj2; -- no!
end Testing;
```

In Parameters

Mode in parameters act as constants.

Mode in parameters are passed values (as with in parameters of subprograms).

```
with Testing;  
procedure Test is  
    X, Y : Integer := 29;  
    procedure Fee is new Testing (3 + X, 7);  
    procedure Fie is new Testing (X, Y);  
begin  
    Fee;  
end Test;
```

Mode in parameters are given values as in assignment
(unlike subprogram in parameters - see later).

```
with Testing;  
procedure Test is  
    X : Integer; -- uninitialized  
    procedure Bomb is new Testing (X, X);  
begin  
    Bomb; -- erroneous  
end Test;
```

Default Evaluation

Defaults are evaluated at time of instantiation
(Not at time of elaboration of generic)

package Declaration_Shell is

function Num return Integer;

generic

Val : Integer := Num;

procedure Demo;

end Declaration_Shell;

package body Declaration_Shell is

Local : Integer := 0;

function Num return Integer is

begin

Local := Local + 1;

return Local;

end Num;

procedure Demo is separate;

end Declaration_Shell;

with Text_io;

separate (Declaration_Shell)

procedure Demo is

begin

Text_io.Put_Line (Integer'Image (Val));

end Demo;

So, . . .

```
with Declaration_Shell;  
procedure Show is
```

```
    procedure Demo is  
        new Declaration_Shell.Demo;
```

```
    procedure Demo1 is  
        new Declaration_Shell.Demo;
```

```
begin  
    Demo;  
    Demo1;  
end Show;
```

output:

1
2

Not:

1
1

Mode in out

Mode in out parameters must be passed variables, not values (same as in out subprogram parameters)

Formal objects of mode in out are aliases for their actual counterparts.

Can be confusing - and is usually *not* recommended.

The evaluation of the variable represented by a name supplied as the actual parameter to an in out generic formal object parameter occurs *once*

Therefore, if some expression in the name evaluation changes after the instantiation, no change is made to the object represented by the formal name.

Test your knowledge!

declare

Y : array (1..5) of Character := "kitty";

Index : Integer := 1;

generic

X : In out Character;

procedure Gen;

procedure Gen is

begin

Index := 5;

X := 'w';

Put (String (Y));

end;

procedure P is new Gen (Y (Index));

begin

P;

end;

**What would happen if the object passed depended
on the value of a discriminant?**

Is this Reasonable?

```
declare
  type Furniture is (Bed, Couch, Table);
  type Style (F : Furniture := Bed) is record
    case F is
      when Bed =>
        Four_Poster : Boolean;
      when Couch =>
        Convertible : Boolean;
      when Table =>
        Legs : Integer;
    end case;
  end record;
  S : Style;
  generic
    X : in out Boolean;
  procedure Gen;

  procedure Gen is
  begin
    S := (Table, 4);
    -- what is the value of X ??
  end;

  procedure P is new Gen (S.Four_Poster);

begin
  P;
end;
```

Would the above be allowed?

Other points

Mode out is not available.

What would that mean, anyway?

Formal objects are not static, so they can't be used
in the generic in case alternatives, type ranges,
floating point precisions, etc.

```
declare
  generic
    X : Integer;
  procedure Gen (Val : Integer);

  procedure Gen (Val : Integer) is
  begin
    case Val is
      when X =>
        ...
      when others =>
        ...
    end case;
  end Gen;

  procedure P is new Gen (X => 5);
begin
  P (Val => 8);
end;
```

You guessed it, this is illegal, too!

Parametric Confusion

When the subprograms have parameters, the syntax can be confusing.

Generic formals precede specification but parameter list follows instantiation.

Subsequent parameters to instance also follow, but name is instance name now not generic name.

So, the text never quite matches up, as with subprograms!

```
declare
  generic
    Gen_Formal : Integer;
  procedure G (Proc_Formal : Boolean);

  procedure G (Proc_Formal : Boolean) is
  begin
    Put (Gen_Formal);
    Put (Proc_Formal);
  end G;

  procedure P is new G (Gen_Formal => 3);
begin
  P (Proc_Formal => False);
end;
```

Enough on objects... Type Parameters!

The real power of generics is revealed through the use of type (and subprogram - later) parameters

Type Parameters

type T is digits <>; -- any floating point type

type T is delta <>; -- any fixed point type

type T is range <>; -- any integer type

type T is (<>); -- any discrete type, which
-- includes integer types

Note allowable operations on above, such as 'First, 'Last, 'Succ, 'Pred, "", "+", etc. , are available only as appropriate (minimum assumptions)

generic

type Counter is (<>);

function Gen_F (X : Counter) return Counter;

function Gen_F (X : Counter) return Counter is
begin

return X + 1; -- oops

end Gen_F;

Private Formal Parameters

type T is private;

Good news: any type except a limited type will match this formal.

Bad news: you can only declare objects, assign values to objects, and test for equality (just those operations you would expect to be able to perform on a private type).

type T is limited private;

Good news: any type including a limited type will match this formal.

Bad news: you can only declare objects and nothing else (just what you would expect to be able to do with a limited private type).

Remember, object parameters are given values by assignment, so can't have limited private object parameters (rats!).

generic

File : Text_io.File_Type;

procedure Oops;

Static Uses Not Allowed

declare

generic

X : Integer;

package Static_Uses_Illegal is

type Length is range 1 .. X;

type Precision is digits X;

N : constant := X;

end Static_Uses_Illegal;

package S is new Static_Uses_Illegal (3);

begin

null;

end;

Surprise

```
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);
begin
  P;
end;
```

-- will raise Constraint_Error at time of instantiation

```
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);
begin
  P;
end;
```

-- will execute OK - in spite of apparent error

Different Integer Types

Ada allows user defined Integer types:

```
type Dimension is range 0 .. 100;
```

This is really a derived type, and therefore a distinct type from type Standard.Integer.

Therefore, a utility that worked with Integers would not work with type Dimension:

```
function Is_Prime (P : Integer) return Boolean;  
D : Dimension := 27;
```

```
if Is_Prime ( D ) then ... -- would not compile
```

One solution:

```
if Is_Prime ( Integer ( D ) ) then ... -- OK
```

Generic solution:

```
generic  
  type Int_Like is range <>;  
function Is_Prime ( P : Int_Like) return Boolean;  
  
function Prime_Dimension is new  
  Is_Prime ( Int_Like => Dimension);  
  
if Prime_Dimension ( P => D) then ... --OK
```

Other Numeric Generic Parameters

Similarly, a generic could manipulate floating or fixed point numeric objects of user-defined types.

```
function Sqrt (X : Float) return Float;
```

Would only work with values of type Float:

```
type Precise is digits 9;
Measurement : Precise := 2.33442_545;
begin
  Ans := Sqrt (Measurement); -- would not compile
  Ans := Sqrt (Float (Measurement)); -- OK, but ugh
```

Better to make Sqrt generic:

```
generic
  type Precision is digits <>;
function Gen_Sqrt (X : Precision) return Precision;
```

```
function Sqrt is new Gen_Sqrt (Precise);
```

```
M := Sqrt (Measurement);
```

Note that you Don't say:

```
function Sqrt is new Gen_Sqrt (9);
```

Fixed Point Does Not Give an Alternative

Recall that there is no general purpose fixed point type so all fixed point types (except Duration) are user-defined.

```
function Sqrt (X : No_Global_Fixed_Type)
```

```
...
```

So, fixed point routines must be generic if not in the scope of the fixed point declarations they will operate on:

```
generic
  type Fixed is delta <>;
function Gen_Fixed_Sqrt (F : Fixed) return Fixed;
```

And, instantiations would look as you would expect:

```
type Fix is delta 0.01;
function Fix_Sqrt is new Gen_Fixed_Sqrt (Fix);
```

Note, again, that the syntax of the instantiation is not:

```
function Fix_Sqrt is new Gen_Fixed_Sqrt (0.01);
```


Enumeration Types Can Also Be Passed

If a generic needs to know about an enumeration type, there is a generic formal parameter for any discrete type.

Note that integer types are also considered discrete types, so an instantiation can pass either an enumeration type or an integer type.

```
generic
  type Things is (<>);
function Number_Of_Things return Integer;

function Number_Of_Things return Integer is
begin
  return
    Things'Pos (Things'Last) -
    Things'Pos (Things'First) + 1;
end Number_Of_Things;

function Two is new
  Number_Of_Things (Boolean);

function Look_Out is new
  Number_Of_Things (Integer);

function Barely_Make_It is new
  Number_Of_Things (Positive);
```

Review of Simple Parameters

To pass

Use the form

Integer

type Int is range <>;

Floating

type Flt is digits <>;

Fixed

type Fix is delta <>;

Discrete

type Enum is (<>);

(Integers are also discrete)

Access Type Parameters

```
generic
  type Int_Ptr is access Integer;
package Probably_Not_Too_Useful is ...
```

When one parameter depends on another:

```
package Access_Example is

  type Candy is (MM, Mars, Hershey);
  type Pointer is access Candy;

  generic
    type Blind is limited private;
    type Ptr is access Blind;
  package Lists is

    type List is limited private;
    -- must be limited    Why?

    procedure Make (L : in out List);
    -- etc. ...

  private
    type List is
      record
        Data : Blind; -- because of this
        Link : Ptr;
      end record;
  end Lists;

  package Candy_Chain is new Lists
    (Blind => Candy,
     Ptr => Pointer
    );

end Access_Example;
```

Structured Types

How do you pass an array to a generic?

```
generic
  type Arr is private;
  Obj : Arr;
procedure Try;
```

```
S : String (1 .. 5) := "kitty";
```

```
procedure Nice_Try is new Try (String, S);
```

```
procedure Try is
begin
  Obj ( 1 ) := 'w';
end Try;
```

Sorry - no dice

To be treated as an array, the structure of an object must be "known" to the generic:

How To Teach Your Generic About Arrays

```
generic
  type Int_Array is array
    (Integer range <>) of Integer;
  procedure Sort_Array (Arr : in out Int_Array);

  procedure Sort_Array (Arr : in out Int_Array) is
    Temp : Integer;
  begin
    for I in Arr'First + 1 .. Arr'Last loop
      for J in Arr'First .. I - 1 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;

  type List is array (Integer range <>) of Integer;

  procedure S is new Sort_Array (List);

  L : List (1..5) := (3, 2, 4, 7, -3);

begin
  S (L);
```

Here, the component type had to be Integer, and the index range had to be an unconstrained range of Integer.

More Flexibility Possible

Above, the index type was Integer and the component type was type Integer.

Here, the index type is a range of any integer type and the component type is also any integer type.

```
generic
    type Index is range <>;
    type Component is range <>;
    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 Arr'First + 1 .. Arr'Last loop
        for J in Arr'First .. I - 1 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;
type Short is range 1 .. 5;
type Dimension is new Integer range 0 .. 100;
type List is array (Short) of Dimension;
procedure S is new Sort_Array
    (Short, Dimension, List);
    L : List := (2, 5, 4, 6, -3);
begin
    S ( L );
```

Even More Flexibility

An array must be indexed by a discrete type, but not necessarily an integer type - an enumerated type is also OK.

Also, the component type can be anything (but if assignment is needed in the generic, then it cannot be limited).

```
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;
type List is array (Boolean) of Float;
procedure S is new Sort_Array
    (Boolean, Float, List);
L : List (4.5, 2.6945);
begin
    S ( L );
```

Constrained vs. Unconstrained Arrays

Note that the first example used an unconstrained type for the formal array parameter and was instantiated with an unconstrained actual array type.

The next two examples showed a constrained array type parameter and actual type.

There is NO array parameter to generics that allows either an unconstrained or constrained array type to be passed to it.

Typically, if you would like to allow either, make the generic handle unconstrained arrays, and make the user declare constrained array types based on named unconstrained types, which are the ones used for the instantiation.

```
type Short is range 1 .. 5;  
type List is array (Short) of Things;
```

The above is really shorthand for

```
type Anon is new Integer; -- or other parent type  
subtype Short is Anon range 1 .. 5;
```

```
type Anon_List is array (Anon range <>) of Things;  
subtype List is Anon_List (Short);
```

So, just don't take shortcuts in declaring the user array types.

No Generic Record Types

Would be nice, but the syntax and rules would have to be quite complex.

For example,

```
generic
  type First_Component is private;
  type Second_Component is private;
  type Rec is record
    Name1 : First_Component;
    Name2 : Second_Component;
  end record;
  procedure Nice_Try;
```

How would you handle different sized record structures?

How would you handle initialized components?

What would you do with such general records once they were passed to the generic?

Food for thought . . .

Exceptions Raised by the Instance

Exceptions can be raised and propagated by an instance during processing.

These must be handled by the user of the instance:

```
generic
procedure Action;

procedure Action is
    Error : exception;
begin
    ...
end Action;

procedure Act1 is new Action;
procedure Act2 is new Action;
procedure Act3 is new Action;

begin

    Act1;
    Act2;
    Act3;

exception
    when Act1.Error => ...;
    when Act2.Error => ...;
    when Act3.Error => ...;
```

No Exception Parameters To a Generic

Can't pass an exception to a generic to be raised:

```
generic
    When_Trouble : exception; -- nope
    ...
procedure Sort ...
    ...
exception
    when others =>
        raise When_Trouble;
end Sort;

My_Exception : exception;
procedure S is new Sort (My_Exception);
...
begin
    S;
exception
    when My_Exception => -- this isn't possible
        whatever...
end;
```

But, There Is Something Just As Useful

Passing a subprogram to a generic is possible:

```
generic
    with procedure Call_Me_Sometime;
procedure General_Stuff;

procedure General_Stuff is
begin
    ...
    Call_Me_Sometime;
    ...
end General_Stuff;

procedure Wait_For_Call is
begin
    Put ("I've been called!");
end Wait_For_Call;

procedure My_Instance is new
    General_Stuff (Wait_For_Call);

begin

    My_Instance;
        -- Wait_For_Call could now be called
        -- by the instance
```

Subprogram Parameters

Replaces the dynamic passing of subprograms such as in standard Pascal.

Enables more complete type checking, i.e., types of parameters to passed subprogram can be checked against calls to it.

Pascal problem:

```
program P;  
  type  
    Color = (Red, Green, Blue);  
  var  
    Bucket : Color;  
  
  procedure Print (C : Color);  
  begin  
    case C of  
      Red : write ('Red');  
      Green : write ('Green');  
      Blue : write ('Blue');  
    end;  
  end;  
  
  procedure Proc (P : procedure);  
  begin  
    P (Bucket);      (* OK *)  
    P (5);           (* runtime error *)  
  end;  
  
begin  
  Proc (Print);
```

Ada Solution

declare

type Color is (Red, Green, Blue);
Bucket : Color := Green;

procedure Print (C : Color) is
begin
 Text_io.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;

procedure Proc is new Gen_Proc (Print);

begin
 Proc;
end;

Sending the Parameter Types, Too

One of the most common uses of subprogram parameters to generics is to provide the generic with operations on user-supplied types.

Some operations are implied by the generic formal parameter:

type T is private;

allows values of this type to be assigned to variables and compared for equality

type T is (<>);

**matches any discrete type
allows use of 'First, 'Last, 'Succ, 'Pred, 'Image, 'Value, 'Pos, 'Val, "<", ">", as well as above.**

type T is range <>;

**matches any integer type
since integer types are also discrete types,
allows all of the above plus the integer operations
such as "+", "-", etc.**

type Ar is array (Index) of Component;

**allows indexing, slicing, assigning, equating
(which are special for arrays), 'Length, 'First, 'Last, etc.**

When Additional Operations are Needed

For example, with array, private, and limited private types, the generic cannot perform text output of values without help from the user:

```
type Handle is access Integer;
Ptr : Handle := new Integer'(57);

function Handle_Image (L : Handle) return String is
begin
    return Integer'Image (L.all);
end Handle_Image;

generic
    type Any is limited private;
    with function String_Of (X : Any) return String;
    procedure Gen_Proc (Obj : Any);

procedure Gen_Proc (Obj : Any) is
begin
    Put ("This is Gen_Proc processing . . . ");
    Put (String_Of ( Obj ) );
end Gen_Proc;

procedure Example is new Gen_Proc
    (Handle, Handle_Image);

procedure Interesting is new Gen_Proc
    (Integer, Integer'Image);

begin
    Example (Ptr);
    Interesting (75);
end;
```


Default Subprogram "Values"

As with generic object and value parameters (and unlike generic type parameters - why?) generic subprogram parameters can be supplied by defaults.

For example, consider the following:

```
with Text_io;
package Shell is

    generic

        type Any is limited private;
        with procedure Print (Val : Any) ;
    package Any_Lists is

        type Any_List is . . .

        procedure Put (L : Any_List);
        -- the body would call the generic
        -- parameter Print procedure
    end Any_Lists;

    package Char_Lists is new Any_Lists
        (Character, Text_io.Put);

end Shell;
```

With judicious naming of the generic subprogram parameter, a default might be possible:

```
with Text_io;
package Shell is

    generic
        type Any is limited private;
        with procedure Put (Val : Any) is <>;
    package Any_Lists is

        type Any_List is ...

        procedure Put (L : Any_List);
        -- the body would call the generic
        -- parameter Put procedure
    end Any_Lists;

    package Char_Lists is new Any_Lists
        (Character);

    -- what's missing?

end Shell;
```

Remember - the resolution of the default takes place at the point of instantiation (not at the generic declaration). Otherwise, it would be trivial.

Speaking of Point of Declaration vs. Instantiation...

Also remember that global references from within a generic refer to those at the point of declaration not those at the point of instantiation. But, default references refer to matching names from the point of instantiation. Confused?

```
with Text_lo;  
use Text_lo;  
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);  
    end Demo;
```

```
end Shell;
```

```
with Shell;  
package Inner is  
    Global : Integer := 39;  
  
    procedure Put (I : Integer);  
  
    procedure User is new 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;
```

What gets printed?

So, generic instantiation is not simple text substitution.

Exercise

Modify the Sorting example from before so that the user can optionally change whether the sort is ascending or descending.

As before:

```
generic
  type Int_Array is array
    (Integer range <>) of Integer;
  procedure Sort_Array (Arr : in out Int_Array);

  procedure Sort_Array (Arr : in out Int_Array) is
    Temp : Integer;
  begin
    for I in Arr'First + 1 .. Arr'Last loop
      for J in Arr'First .. I - 1 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;
```

Make minimal changes to satisfy requirement.

Solution

```
generic
  type Int_Array is array
    (Integer range <>) of Integer;
    with function "<" (Left, Right : Integer)
      return Boolean is <>;
  procedure Sort_Array (Arr : in out Int_Array);

  procedure Sort_Array (Arr : in out Int_Array) is
    Temp : Integer;
  begin
    for I in Arr'First + 1 .. Arr'Last loop
      for J in Arr'First .. I - 1 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;
```

Instances could be declared:

```
type List is array (Integer range <>) of Integer;

procedure Ascending1 is new Sort_Array
  (List, "<");

procedure Ascending2 is new Sort_Array (List);

procedure Descending is new Sort_Array
  (List, ">");
```

Nesting of Generics

It can be useful to export a generic utility from a generic package. Don't clutch:

```
generic
  type Element is private;
  with function "<" (Left, Right : Element)
    return Boolean is <>;
package Sorted_Binary_Trees is
  type Tree is private;

  procedure Insert
    (X : Element; Into : in out Tree);

  generic
    with procedure Operate (X : Element);
  procedure Search_And_Operate (T : Tree);

end Sorted_Binary_Trees;
```

(The body is, of course, "trivially obvious to even the most casual reader". . .)

Exercise: Give an example instantiation of the above generics. (Not the body.)

A Simple Solution

```
package Int_Trees is
    new Sorted_Binary_Trees (Integer);
```

```
procedure Put_Int (X : Integer) is
begin
    Text_IO.Put ( Integer'Image (X) );
end Put_Int;
```

```
procedure Print_In_Order is new
    Int_Trees.Search_And_Operate (Put_Int);
```


Other Type Parameters

As with arrays, unconstrained types are allowed for private and limited types using the customary syntax:

```
generic
    type Furniture (Upholstered : Boolean)
                                is private;
package P is
    Table : Furniture (False);
    Couch : Furniture (True);
end P;
```

Note that a default for the discriminant is Not allowed.

This means that an unconstrained object cannot be declared (unlike typical discriminated records)

Incidentally, a type mark must be used in a generic part, and not a subtype:

```
generic
    subtype Short is Integer range <>;
procedure Not_A_Chance;
```

More on Unconstrained Types

Peculiar situations can arise when matching a private or limited type with an actual type which is unconstrained.

If the actual type is unconstrained:

- an unconstrained array**
- an unconstrained record without a default
discriminant**

then you can't declare an object of that type inside the generic (which includes allocators without initial values)

unless the object is a constant, meaning the initial value must have been supplied by the instantiation.

```

generic
  type T is private;
package Experiment is
  Val : T;
  type Ptr is access T;
  P : Ptr := new T;
end Experiment;

```

```

type Boo is array (Boolean range <>) of Integer;
package P is new Experiment (Boo); -- will fail

```

```

type Rec (B : Boolean) is record
  null;
end record;
package Q is new Experiment (Rec); -- will fail

```

```

generic
  type T is private;
  Init : T;
package Experiment is
  Val : T := Init; -- still a problem
  type Ptr is access T;
  P : Ptr := new T'(Init); -- this helps
end Experiment;

```

```

type Boo is array (Boolean range <>) of Integer;
I : Boo (False .. True) := (6, 12);
package P is new Experiment (Boo, I); -- will fail

```

```

type Rec (B : Boolean) is record
  null;
end record;
R : Rec (False) := Rec'(B => False);
package Q is new Experiment (Rec, R); -- will fail

```

```

generic
  type T is private;
  Init : T;
package Experiment is
  Val : constant T := Init;           -- OK
  type Ptr is access T;
  P : Ptr := new T'(Init);
end Experiment;

```

```

type Boo is array (Boolean range <=>) of Integer;
I : Boo (False .. True) := (6, 12);
package P is new Experiment (Boo, I); -- OK

```

```

type Rec (B : Boolean) is record
  null;
end record;
R : Rec (False) := Rec'(B => False);
package Q is new Experiment (Rec, R); -- OK

```

Retrofitting Generics

Look familiar?

```
package Counter is
  type Count is limited private;
  procedure Increment (C : in out Count);
  procedure Reset (C : out Count);
private ...
end Counter;

package Trees is
  type Tree is limited private;
  procedure Get (T : out Tree);
  function Is_Leaf (T : Tree) return Boolean;
  procedure Split (T : in out Tree;
                  Left, Right : out Tree);
  procedure Return (T : in out Tree);
private ...
end Trees;

with Trees;
package Piles is
  type Pile is limited private;
  function Empty (P : Pile) return Boolean;
  procedure Put (T : in out Trees.Tree;
                On : in out Pile);
  procedure Initialize (P : in out Pile);
  procedure Get (T : out Trees.Tree;
                From : in out Pile);
private ...
end Piles;

with Trees, Piles, Counter;
procedure Count_Leaves is ...
```

Generalization

```
generic
  type Object is limited private;
package General_Piles is
  type Pile is limited private;
  function Empty (P : Pile) return Boolean;
  procedure Put (T : in out Object;
                On : in out Pile);
  procedure Initialize (P : in out Pile);
  procedure Get (T : out Object;
                From : in out Pile);
private ...
end Piles;
-- package is the same except for substitution
-- of Object for Trees.Tree

with Trees, General_Piles, Counter;
procedure Count_Leaves is ...
  package Tree_Piles is
    new General_Piles (Trees.Tree);
    ...
  -- can be the same otherwise
```

Note that this generalization was made easy due to the lack of assumptions that the original components made about each other's types.

Clue: private and limited private types assist in future generalizations of your program components.

What if the types aren't private?

Generalizations are still possible, but require more complicated generic parts.

```
package Export_Array is
    type Global_Array is array (1..10) of Integer;
    ...

with Export_Array;
package Needs_To_See_Global_Array is
    procedure Sort
        (Ar : in out Export_Array.Global_Array);
    ...
```

To generalize the above:

```
generic
    type GA is array (1..10) of Integer;
package Needs_To_See_Global_Array is
    procedure Sort (Ar : in out GA);
    ...
```

Or better would be:

```
generic
    type Index_Range is range <>;
    type GA is array (Index_Range) of Integer;
package Needs_To_See ...
```

But would need to ensure that the body of the package did not depend on literals unique to the original array type (i.e., that the range went from 1 to 10).

Simple Generic

Given a single state-machine implementation of stacks, the customary way to introduce many stacks is to turn it into an abstract data type:

Simple stack package:

```
package Stack is
  procedure Push (I : Integer);
  procedure Pop (I : out Integer);
  function Empty return Boolean;
  function Full return Boolean;
end Stack;
```

Conventional way to convert to many stacks:

```
package Stacks is
  type Stack is private;
  procedure Push (I : Integer; On : in out Stack);
  procedure Pop
    (I : out Integer; From : in out Stack);
  function Empty (S : Stack) return Boolean;
  function Full (S : Stack) return Boolean;
private ...
end Stacks;
```

User code:

```
S : Stacks.Stack;
I : Integer;
begin
  Push (25, S);
  Pop (I, S);
```


Many Stacks, "Generically"

Review the original single stack:

```
package Stack Is
  procedure Push (I : Integer);
  procedure Pop (I : out Integer);
  function Empty return Boolean;
  function Full return Boolean;
end Stack;
```

Illustration of generic method (not necessarily recommended, just for example):

```
generic
package Stack Is
  procedure Push (I : Integer);
  procedure Pop (I : out Integer);
  function Empty return Boolean;
  function Full return Boolean;
end Stack;
```

No change to the body in this conversion! (Unlike the body of the private type example.)

User code:

```
package S1 Is new Stack;
package S2 Is new Stack;
I : Integer;
begin
  S1.Push (25);
  S1.Pop (I);
```

Generic Implementations

What happens when a generic is elaborated?

What happens when an instantiation is elaborated?

It depends . . .

Suppose an initialization block appears at the end of a generic package body:

```
with Text_io;  
package body Stack is  
    . . .  
begin  
    Text_io.Put ("Stack instance elaborated");  
end Stack;
```

Then, each of the following will cause the above Put statement to execute:

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

In the case of packages (not generics) there would only be one logical package and only one elaboration initialization would occur.

This emphasizes the fact that there are as many logical packages (or subprograms) as there are instantiations of a generic package (or subprogram).

Logical or Physical?

Debate over whether an implementation should actually create physical copies of a generic for each instance or somehow "code share" among all the instances.

Code sharing is more the way a procedure or function behaves (re-entrant).

Physical expansion (code copying) is more the way a macro expansion behaves.

There are advantages and disadvantages of each:

Physical Expansion, Pro:

Object code executes faster because all the work is done at compile time and no context switching is needed among the instances of the same generic code.

Implementation is simpler.

Physical Expansion, Con:

Space requirements for object code can become a real problem if there are a lot of instances.

Recompilation of a generic spec is always required even if just the body changes. (Why?)

Code Sharing, Pro:

Significant savings in object code size.

Generic bodies can be changed without recompiling the specs (and therefore the instantiations).

Code Sharing, Con:

Execution can be slower due to run time instantiations.

Implementation is more difficult.

So, which is it? There is a "best" answer:

For a development compiler, on a host machine, where maintenance, modification, and recompilation is likely and turnaround time is more important than saving a few milliseconds at run time, you want a code sharing compiler.

For a target compiler, in a time-critical application, where execution efficiency is more important than recompilation efficiency, you want a code copying compiler.

It's 11:00 . . . Do you know what your compiler is doing?

(Do you know what the different vendors offer?)

More Retrofitting - Make the following general:

```
package Project is
  subtype Line is String (1..80);
  subtype Word is String (1..80);
  type Break_Chars is (' ', '-', ':', ',');
end Project;

with Project; use Project;
procedure Next_Word (From : in out Line; Into : out Word) is
begin
  Into := (others => ' ');
  for C in 1 .. 80 loop
    -- find first non-Break_Chars letter in From
  end loop;
  for W in C .. 80 loop
    -- copy contiguous non-Break_Chars to Into
  end loop;
  for R in W+1 .. 80 loop
    -- left justify remaining letters in From
    -- and set rest to a Blank
  end loop;
end Next_Word;

with Next_Word;
with Project; use Project;
with Text_io;
procedure Main is
  Input_Line : Line;
  Length : Natural;
  A_Word : Word;
begin
  ... -- initializations, file openings
  while not Text_io.End_Of_File (In_File) loop
    Input_Line := (others => ' ');
    Get_Line (In_File, Input_Line, Length);
    loop
      Next_Word (From => Input_Line, Into => A_Word);
      exit when A_Word = (others => ' ');
      Put_Line (A_Word);
    end loop;
  end loop;
end Main;
```

```

package Project is
    subtype Line is String (1..80);
    subtype Word is String (1..80);
    type Break_Chars is (' ', '-', ':', ',');
end Project;

generic
    type Line is array (Positive range <>) of Character;
    type Word is array (Positive range <>) of Character;
    Break_Chars : String;
package Next_Word_Package is
    procedure Next_Word
        (From : in out Line; Into : out Word);
end Next_Word_Package;

package body Next_Word_Package is
    procedure Next_Word
        (From : in out Line; Into : out Word) is
    begin
        Into := (others => Break_Chars (Break_Chars'First));

        ...

with Next_Word;
with Project; use Project;
with Text_IO;
procedure Main is
    Input_Line : Line;
    Length : Natural;
    A_Word : Word;
    package Words is new
        ...
begin
    ... -- initializations, file openings
    while not Text_IO.End_Of_File (In_File) loop
        Input_Line := (others => ' ');
        Get_Line (In_File, Input_Line, Leng'h);
        loop
            Next_Word (From => Input_Line, Into => A_Word);
            exit when A_Word = (others => ' ');
            Put_Line (A_Word);
        end loop;
    end loop;
end Main;

```

Unwrapping Ada[®] Packages

LCDR Melinda Moran
Computer Science Department
U. S. Naval Academy
Annapolis, MD 21401
(301) 267-2797
AV 281-2797
LINDY@USNA

Ada Program Structure

☐ 4 Basic Program Units:

☒ Packages

☐ Subprograms

☐ Tasks

☐ Generic Units

Packages, a Powerful Tool for Software Engineering:

☒ Abstraction

- ☐ both data and process abstraction supported
- ☐ layers of abstraction supported
- ☐ facilitate managing complexity by allowing division of problem into layers of abstraction with 7 ± 2 components each

☒ Information Hiding

- ☐ irrelevant information hidden
- ☐ only information relevant to each level of abstraction is visible and accessible at that level

☒ Modularity

- ☐ facilitate creation of libraries of software modules which can be reused to implement many systems
- ☐ facilitate easily modified systems where packages can easily be added and subtracted

☒ Localization

- ☐ facilitate creation of very cohesive packages which can be less tightly coupled

☐ Uniformity

☐ Completeness

☒ Confirmability

- ☐ communication only through visible interface minimizes debugging problems by minimizing "ripple" effects
- ☐ clear definition of interfaces simplifies testing problems

PACKAGES

Definition: collection of computational resources, which may *encapsulate* data types, data objects, subprograms, tasks and even other packages.

Purpose: to express and *enforce* user's logical abstractions within the language.

Applications/Common Usages:

- ☐ Encapsulate related data types, constants or objects.
- ☐ Encapsulate related program units.
- ☐ Embody an Abstract Data Type.
- ☐ Embody an Abstract-state Machine.
- ☐ Encapsulate tasks.*

Structure of a Package

☐ Composed of 2 parts

☐ Specification

- ☐ defines the packages interface with client modules-- the types and services it offers
- ☐ defines the portion of the package "visible" to other modules
- ☐ must be compiled before program units which use it
- ☐ may contain a "private" portion

☐ Body

- ☐ contains implementation details of "how" the package fulfills its contract with the client modules
- ☐ facilitates "information hiding" -- hides irrelevant information from client modules
- ☐ protects data structures and operations it encapsulates from inadvertent or malicious tampering by client module
- ☐ may be separately compiled at a later time than specification
- ☐ may be replaced with different implementation without recompilation "ripples" to other modules as long as specification is unchanged
- ☐ may not be present if specification only contains types or object declarations
- ☐ may contain a section of initialization statements and an exception handler

```

package TERMINAL_PMS is
  procedure ClearScreen;
  procedure SetCursorAt(Column, Row: in integer);
  procedure CursorUp;
  procedure CursorDown;
  procedure CursorRight;
  procedure CursorLeft;
  procedure Run;
end TERMINAL_PMS;

```

```

with TEXT_IO; use TEXT_IO;
package body TERMINAL_PMS is
  ESC : constant character := character'val(27);

  procedure Run is
  begin
    PUT(ESC);
    PUT("W");
  end;

  --
  procedure ClearScreen is
  begin
    Run;
    PUT(ESC);
    PUT("J");
  end ClearScreen;

  --
  procedure SetCursorAt(column, row: in integer) is
  begin
    PUT(ESC);
    PUT("Y");
    PUT(character'val(row+32));
    PUT(character'val(column+32));
  end SetCursorAt;

  --
  procedure CursorUp is
  begin
    PUT(ESC);
    PUT("A");
  end;

  --
  procedure CursorDown is
  begin
    PUT(ESC);
    PUT("B");
  end;

  --
  procedure CursorRight is
  begin
    PUT(ESC);
    PUT("C");
  end;

  --
  procedure CursorLeft is
  begin
    PUT(ESC);
    PUT("D");
  end;

  --
end TERMINAL_PMS;

```

AD-A189 641

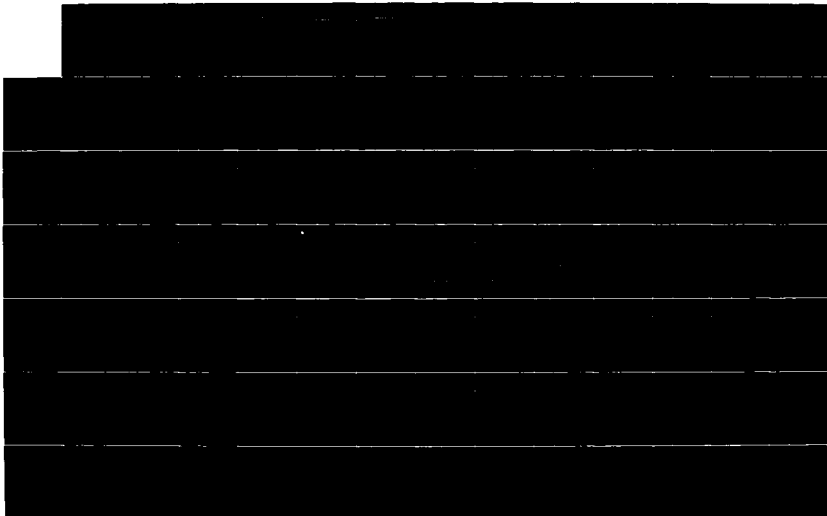
ADVANCED ADA WORKSHOP AUGUST 1987(U) ADA JOINT PROGRAM
OFFICE ARLINGTON VA 21 AUG 87

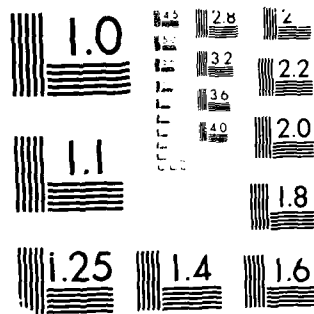
3/4

UNCLASSIFIED

F/G 12/3

NL





MICROCOPY RESOLUTION TEST CHART
 NATIONAL BUREAU OF STANDARDS-1963-A

```

-- Unit Name: Basic_IO
-- Author: LCDR MORAN
-- Date: 4 AUG 1987
-- Function: Provides user ability to do input and output of
--           character, string, integer, and floating point
--           data types.

```

```

package Basic_IO is

```

```

    procedure New_Line;
    -- advances cursor to next line on the screen

    procedure Get(Item : out character);
    -- reads a single character from the keyboard

    procedure Get_Line(Item : out character);
    -- reads a single character and carriage return from the keyboard

    procedure Put(Item : in character);
    -- outputs a single character to the screen

    procedure Put_Line(Item : in character);
    -- outputs a single character to the screen and advances the cursor
    -- to the next line on the screen

    procedure Get(Item : out string);
    -- reads a string of characters from the keyboard

    procedure Get_Line(Item : out string);
    -- reads a string of characters and a carriage return from the keyboard

    procedure Put(Item : in string);
    -- outputs a string of characters to the screen

    procedure Put_Line(Item : in string);
    -- outputs a string of characters to the screen and advances the cursor
    -- to the next line on the screen

    procedure Get(Item : out integer);
    -- reads an integer from the keyboard

    procedure Get_Line(Item : out integer);
    -- reads an integer and a carriage return from the keyboard

    procedure Put(Item : in integer);
    -- outputs an integer to the screen

    procedure Put_Line(Item : in integer);
    -- outputs an integer to the screen and advances the cursor to the
    -- next line on the screen

    procedure Get(Item : out float);
    -- reads a floating point value from the keyboard

    procedure Get_Line(Item : out float);
    -- reads a floating point value and a carriage return from the keyboard

    procedure Put(Item : in float);
    -- outputs a floating point value to the screen

    procedure Put_Line(Item : in float);
    -- outputs a floating point value to the screen and advances the cursor
    -- to the next line on the screen

```

```

end Basic_IO;

```

```

with Text_IO;
package body Basic_IO is

  package Int_IO is new Text_IO.Integer_IO(Integer);
  package Float_Point_IO is new Text_IO.Float_IO(Float);

  procedure New_Line is
  begin
    Text_IO.New_Line;
  end;

  procedure Get_Line is
  Info : string(1..80);
  Count : natural;
  begin
    Text_IO.Get_Line(Info, Count);
  end;

  procedure Get(Item : out character) is
  begin
    Text_IO.Get(Item);
  end;

  procedure Get_Line(Item : out character) is
  begin
    Text_IO.Get(Item);
    Get_Line;
  end;

  procedure Put(Item : in character) is
  begin
    Text_IO.Put(Item);
  end;

  procedure Put_Line(Item : in character) is
  begin
    Text_IO.Put_Line;
  end;

  procedure Get(Item : out string) is
  begin
    Text_IO.Get(Item);
  end;

  procedure Get_Line(Item : out string) is
  begin
    Text_IO.Get_Line;
  end;

  procedure Put(Item : in string) is
  begin
    Text_IO.Put(Item);
  end;

  procedure Put_Line(Item : in string) is
  begin
    Text_IO.Put_Line;
  end;

  procedure Put_Line(Item : in string) is
  begin
    Text_IO.Put_Line;
    New_Line;
  end;

  procedure Get(Item : out Integer) is
  begin
    Int_IO.Get(Item);
  end;

  procedure Get_Line(Item : out Integer) is
  begin
    Int_IO.Get(Item);
    Get_Line;
  end;

  procedure Put(Item : in Integer) is
  begin
    Int_IO.Put(Item);
  end;

  procedure Put_Line(Item : in Integer) is
  begin
    Int_IO.Put(Item);
    New_Line;
  end;

  procedure Get(Item : out Float) is
  begin
    Float_Point_IO.Get(Item);
    Get_Line;
  end;

  procedure Put(Item : in Float) is
  begin
    Float_Point_IO.Put(Item);
  end;

  procedure Put_Line(Item : in Float) is
  begin
    Float_Point_IO.Put(Item);
    New_Line;
  end;

end Basic_IO;

```


Overloading

```
package COMPLEX is
  type COMPLEX_NUMBER is record
    REAL_PART      : FLOAT;
    IMAGINARY_PART : FLOAT;
  end record;
  function "+" (A,B : in COMPLEX_NUMBER) return COMPLEX_NUMBER;
  function "-" (A,B : in COMPLEX_NUMBER) return COMPLEX_NUMBER;
end COMPLEX;

package body COMPLEX is
  function "+" (A,B : in COMPLEX_NUMBER) return COMPLEX_NUMBER is
    RESULT : COMPLEX_NUMBER;
  begin
    RESULT.REAL_PART := A.REAL_PART + B.REAL_PART;
    RESULT.IMAGINARY_PART := A.IMAGINARY_PART + B.IMAGINARY_PART;
    return RESULT;
  end;

  function "-" (A,B : in COMPLEX_NUMBER) return COMPLEX_NUMBER is
    RESULT : COMPLEX_NUMBER;
  begin
    RESULT.REAL_PART := A.REAL_PART - B.REAL_PART;
    RESULT.IMAGINARY_PART := A.IMAGINARY_PART - B.IMAGINARY_PART;
    return RESULT;
  end;
end COMPLEX;
```

Overloading continued

```
package FRACTIONS is
  type FRACTION is record
    NUMERATOR      : NATURAL;
    DENOMINATOR    : NATURAL;
  end record;
  function "+" (A,B : in FRACTION) return FRACTION;
  function "-" (A,B : in FRACTION) return FRACTION;
end FRACTIONS;
```

```
package body FRACTIONS is
  function "+" (A,B : in FRACTION) return FRACTION is
    RESULT : FRACTION;
  begin
    RESULT.NUMERATOR := (A.NUMERATOR * B.DENOMINATOR) +
      (B.NUMERATOR * A.DENOMINATOR);
    RESULT.DENOMINATOR := A.DENOMINATOR * B.DENOMINATOR;
    return RESULT;
  end;

  function "-" (A,B : in FRACTION) return FRACTION is
    RESULT : FRACTION;
  begin
    RESULT.NUMERATOR := (A.NUMERATOR * B.DENOMINATOR) -
      (B.NUMERATOR * A.DENOMINATOR);
    RESULT.DENOMINATOR := A.DENOMINATOR * B.DENOMINATOR;
    return RESULT;
  end;
end FRACTIONS;
```

Overloading continued

```
with COMPLEX, FRACTIONS;  
use COMPLEX, FRACTIONS;  
procedure ARITHMETIC is  
  X : FRACTION := (3,4);  
  Y : FRACTION := (5,6);  
  FRACTION_RESULT : FRACTION := (1,1);  
  
  A : COMPLEX_NUMBER := (5,7);  
  B : COMPLEX_NUMBER := (8,9);  
  COMPLEX_RESULT : COMPLEX_NUMBER := (0,0);  
  
begin  
  FRACTION_RESULT := X + Y;  
  COMPLEX_RESULT := A + B;  
  
  FRACTION_RESULT := X - Y;  
  COMPLEX_RESULT := A - B;  
end;
```

Packages with No Body

package CALENDAR is

type DAY is (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
SATURDAY, SUNDAY);

type MONTH is (JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE,
JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER,
DECEMBER);

type YEAR is range 0 .. INTEGER'LAST;
end CALENDAR;

package METRIC_EARTH_CONSTANTS is

EQUATORIAL_RADIUS : constant := 6378.145; --km
GRAVITATION_CONSTANT : constant := 3.986012e5; --km**3/sec**2
SPEED_UNIT : constant := 7.90536828; --km/sec
TIME_UNIT : constant := 806.8118744; -- sec

end METRIC_EARTH_CONSTANTS;

[METRIC_EARTH_CONSTANTS package taken from Software Engineering
with Ada by Grady Booch]

```

with TYPES;
use TYPES;
package GIGI_GRAPHICS_PKG is

```

```

-----
  procedure SETSCREENSCALE(left,right,top,bottom: in integer);
  procedure CLEARSCREEN;
  procedure SETSCREENCOLOR(color : in colortype);
  procedure SETREVERSEVIDEO;
  procedure SETNORMALVIDEO;
  procedure FREEZESCREEN(ticks : in integer);
  procedure MOVECURSORABS(xcoord,ycoord : in integer);
  procedure SETCURSORAT(xcoord,ycoord : in integer);
  procedure MOVECURSORREL(xincr,yincr : in integer);
  procedure QUERYCURSORPOS(xcoord,ycoord : out integer);
  procedure SVELASTMOVELOC;
  procedure RESTORELASTMOVELOC;
  procedure PUTDOT;
  procedure DRAWLINETO(xcoord,ycoord : in integer);
  procedure DRAWPOLYLINE(coordarray : in coordarraytype;
                           num_of_coord_pairs : in integer);
  procedure DRAWCIRCLE(xcoord,ycoord,radius : in integer);
  procedure SETSHADINGREFLINE(y_value: in integer);
  procedure SETSHADINGCHAR(shading_char : in character);
  procedure SETSHADINGON;
  procedure SETSHADINGOFF;
  procedure SVELASTDRAWLOC;
  procedure RESTORELASTDRAWLOC;
  procedure SETWRITINGCOLOR(color : in colortype);
  procedure SETWRITINGMODE(mode : in modetype);
  procedure SETTEXTSIZE(size : in integer);
  procedure SETTEXTPATH(angle : in integer);
  procedure SETTEXTHEIGHT(height : in integer);
  procedure SETITALICSANGLE(angle : in integer);
  procedure SETBLINKOFF;
  procedure SETBLINKON;
  procedure SAVETEXTATTRIBUTES;
  procedure RESTORETEXTATTRIBUTES;
  procedure OUTPUT(integer_val : in integer);
  procedure OUTPUT(text_val : in string);
  procedure OUTPUT(char_val : in character);
end GIGI_GRAPHICS_PKG;
-----

```

```

with TEXT_HANDLER;
use TEXT_HANDLER;
package body GIGI_GRAPHICS_PKG is

```

```

    -- current screen scale bounds
    CLEFT : integer := 0;           -- current left x-axis value
    CRIGHT : integer := 767;        -- current right x-axis value
    CTOP : integer := 0;            -- current top y-axis value
    CBOTTOM : integer := 479;       -- current bottom y-axis value

    --current cursor position
    CP_X : integer := 0;            -- current position, x-value
    CP_Y : integer := 0;            -- current position, y-value

```

```

-----
task SEMAPHORE is

```

```

    entry SEIZE;
    entry RELEASE;

```

```

end SEMAPHORE;

```

```

task body SEMAPHORE is

```

```

    IN_USE : boolean := false;

```

```

begin

```

```

    loop

```

```

        select

```

```

            when not IN_USE =>

```

```

                accept SEIZE do

```

```

                    IN_USE := true;

```

```

                end SEIZE;

```

```

            or

```

```

            when IN_USE =>

```

```

                accept RELEASE do

```

```

                    IN_USE := false;

```

```

                end RELEASE;

```

```

            end select;

```

```

        end loop;

```

```

end SEMAPHORE;
-----

```

```

function INBOUNDS(xcoord,ycoord : in integer) return boolean is
    xvalid, yvalid : boolean;
begin
    -- check if xcoord between left and right screen scale bounds
    if ((xcoord >= cleft) and (xcoord <= cright)) or
        ((xcoord <= cleft) and (xcoord >= cright)) then
        xvalid := true;
    else
        xvalid := false;
    end if;

    -- check if ycoord between top and bottom screen scale bounds
    if ((ycoord >= ctop) and (ycoord <= cbottom)) or
        ((ycoord <= ctop) and (ycoord >= cbottom)) then
        yvalid := true;
    else
        yvalid := false;
    end if;
    if (xvalid) and (yvalid) then
        return true;
    else
        return false;
    end if;
end;
-----
procedure CLEARSCREEN is
    T : TEXT;
begin
    T := TO_TEXT(esc) & "Pp" & "S(E)" & esc & "\";
    PUT(T);
end;
-----
procedure SETSCREENCOLOR(color : in colortype) is
    x : integer;
    T : text;
begin
    x := colortype'pos(color); -- x=ordinal position of color chosen
                                -- see colortype in types package
    T := TO_TEXT(esc) & "Pp" & "S(I" & integer'image(x) & ")"
        & esc & "\";
    PUT(T);
end;
-----
procedure SETREVERSEVIDEO is
    T : TEXT;
begin
    T := TO_TEXT(esc) & "Pp" & "S(Nl)" & esc & "\";
    PUT(T);
end;
-----

```

Specification Structure

- ☐ 2 Parts

- ☐ Visible portion

- ☐ extends from beginning of package specification up to word "private"
- ☐ Client modules can "see" and use the types, objects, subprograms in this section
- ☐ Resources in this part of the spec are said to be "exported"

- ☐ Private portion
 - ☐ Part of the specification that follows the word "private"
 - ☐ Types declared in this portion of the spec can be seen "textually" but their structural components are not accessible to/corruptible by the client
 - ☐ Declaration of types as private facilitates use of abstraction where client operates with logical properties of the type and cannot access the details of its physical implementation
 - ☐ Private types can only be declared in the visible part of a package; their implementation is in the private part of the package
 - ☐ 2 categories of private type
 - ☐ private types - can use operations declared in visible part of pkg spec, assignment operator, eq. and inequality op.
 - ☐ limited private types- same as for private types except assignment, eq and inequality operations unavailable

```

package Fractions is
    type Fraction is
        record
            Numerator: Integer := 0;
            Denominator: positive := 1;
        end record;

    function MakeFraction(N, D: Integer) return Fraction;
        -- "create"

    function Numer(x: Fraction) return Integer;
    function Denom(x: Fraction) return Integer;
        -- decomposition

    function "+"(x, y: Fraction) return Fraction;
    function "-"(x, y: Fraction) return Fraction;
    function "*" (x, y: Fraction) return Fraction;
    function "/"(x, y: Fraction) return Fraction;
        -- arithmetic

    function Equal(x, y: Fraction) return boolean;
    function "<"(x, y: Fraction) return boolean;
    function ">"(x, y: Fraction) return boolean;
        -- comparison

    function Reduce(x: Fraction) return Fraction;
    function FractionToInt(x: Fraction) return Integer;
    function IntToFraction(i: Integer) return Fraction;
        -- miscellaneous

end Fractions;

```

Figure 1-7 Package specification for Fractions

```

package Fractions is
    type Fraction is private;

    function MakeFraction(N, D: Integer) return Fraction;
        -- "create"

    function Numer(x: Fraction) return Integer;
    function Denom(x: Fraction) return Integer;
        -- decomposition

    function "+"(x, y: Fraction) return Fraction;
    function "-"(x, y: Fraction) return Fraction;
    function "*" (x, y: Fraction) return Fraction;
    function "/"(x, y: Fraction) return Fraction;
        -- arithmetic

    function Equal(x, y: Fraction) return boolean;
    function "<"(x, y: Fraction) return boolean;
    function ">"(x, y: Fraction) return boolean;
        -- comparison

    function Reduce(x: Fraction) return Fraction;
    function FractionToInt(x: Fraction) return Integer;
    function IntToFraction(i: Integer) return Fraction;
        -- miscellaneous

private
    type Fraction is
        record
            Numerator: Integer := 0;
            Denominator: positive := 1;
        end record;

end Fractions;

```

Figure 1-9 Package specification using private type

```

package body Fractions is
    -- code for function body MakeFraction

    function Numer(x: Fraction) return Integer is
    begin
        return x.Numerator;
    end Numer;

    -- code for function body Denom
    -- code for function body "+"

    function "-"(x, y: Fraction) return Fraction is
    N: Integer;
    D: positive;
    begin
        N := Numer(x)*Denom(y) - Numer(y)*Denom(x);
        D := Denom(x)*Denom(y);
        return Reduce(MakeFraction(N,D));
    end "-";

    -- code for function body "*"
    -- code for function body "/"
    -- code for function body Reduce

    function Equal(x, y: Fraction) return boolean is
    begin
        return Numer(x)*Denom(y) = Numer(y)*Denom(x);
    end Equal;

    -- code for function body "<"
    -- code for function body ">"

    function FractionToInt(x: Fraction) return Integer is
    begin
        return Numer(x)/Denom(x);
    end FractionToInt;

    function IntToFraction(i: Integer) return Fraction is
    begin
        return i;
    end IntToFraction;

end Fractions;

```

Figure 1-8 Partial package body for Fractions

[Taken from Data Structures with Ada by Michael B. Feldman]

```

package KEY_MANAGER is
  type KEY is private;
  procedure GET_KEY(K : out KEY);
  function "<" (X,Y : KEY) return BOOLEAN;
private
  type KEY is new INTEGER range 0 .. INTEGER'LAST;
end KEY_MANAGER;

package body KEY_MANAGER is
  NEXT_KEY : KEY := 1; -- own variable - exists between
                        -- procedure calls
  procedure GET_KEY(K : out KEY) is
  begin
    K := NEXT_KEY;
    NEXT_KEY := NEXT_KEY + 1;
  end GET_KEY;

  function "<"(X,Y : KEY) return BOOLEAN is
  begin
    return INTEGER(X) < INTEGER(Y);
  end "<";
end KEY_MANAGER;

```

[Taken from Programming with Ada by Peter Wegner]

```

package B_R is
    type NUMBERS is range 0 .. 99;

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

end B_R;

package body B_R is
    SERV_A_MATIC : NUMBERS := 1;

    procedure TAKE (A_NUMBER : out NUMBERS) is
    begin
        A_NUMBER := SERV_A_MATIC;
        SERV_A_MATIC := SERV_A_MATIC + 1;
    end TAKE;

    procedure SERVE (NUMBER : in NUMBERS) is separate;

    function NOW_SERVING return NUMBERS is separate;

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;

```

[Taken from Keesler AFB, ATC Course Student Handout]

```

package B_R is
    type NUMBERS is private;

    procedure TAKE (A_NUMBER : out NUMBERS);
    procedure SERVE (NUMBER : in NUMBERS);
    function NOW_SERVING return 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);
    procedure SERVE (NUMBER : in NUMBERS);
    function NOW_SERVING return NUMBERS;
    function "=" (LEFT, RIGHT : in NUMBERS)
        return BOOLEAN;
    function CLOSE_ENOUGH (A_NUMBER : 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);
    if NOW_SERVING = YOUR_NUMBER then
        SERVE (YOUR_NUMBER);
    elsif CLOSE_ENOUGH (YOUR_NUMBER) then
        while NOW_SERVING /= YOUR_NUMBER loop
            null; -- wait your turn
        end loop;
        SERVE (YOUR_NUMBER);
    else
        GO_TO_DQ;
    end if;

end ICE_CREAM;

```

Implications of Using the Specification's Private Section

- ☐ Forces recompilation of specification and any client modules if the data structure used to implement a type changes.
- ☐ Can be avoided if data structure needed for a package can be internal to the body of the package but this disables client's ability to declare multiple objects of a particular type. Haberman and Perry refer to "open type" and "unique object" solutions.
- ☐ Can be avoided by making private type an "access" (pointer) type

Ada Open Type Solution

package QueueManager is

```

    qsize : constant INTEGER := 10;
    subtype qindex is INTEGER range 0 .. qsize - 1;

    type bodies is array(qindex of FLOAT);
    type queue is record
        head : qindex := 0;
        length : INTEGER range 0 .. qsize := 0;
        qbody : array(qindex) of FLOAT;
    end record;

    function FULL (q in queue) return BOOLEAN;
    function EMPTY (q in queue) return BOOLEAN;
    procedure ENQ (q in out queue, r in FLOAT);
    procedure DEQ (q in out queue, r out FLOAT);

    overflow, underflow : exception;
end QueueManager;
```

package body QueueManager is

```

    function FULL (q in queue) return BOOLEAN is
    begin
        return q.length = qsize;
    end FULL;

    function EMPTY (q in queue) return BOOLEAN is
    begin
        return q.length = 0;
    end EMPTY;

    procedure ENQ (q in out queue, r in FLOAT) is
    begin
        if FULL then raise overflow; end if;
        q.qbody(q.head + q.length) mod qsize := r;
        q.length := q.length + 1;
    end ENQ;

    procedure DEQ (q in out queue, r out FLOAT) is
    begin
        if EMPTY then raise underflow; end if;
        r := q.qbody(q.head);
        q.head := (q.head + 1) mod qsize;
        q.length := q.length - 1;
    end DEQ;
end QueueManager;
```

Ada Program for the Queue as a Unique Object

package RealQueue is

```

    function FULL return BOOLEAN;
    function EMPTY return BOOLEAN;
    procedure ENQ (r in FLOAT);
    function DEQ return FLOAT;

    overflow, underflow : exception;
end RealQueue;

package body RealQueue is

    qsize : constant INTEGER := 10;
    subtype qindex is INTEGER range 0 .. qsize - 1;

    head : qindex;
    length : INTEGER range 0 .. qsize;
    qbody : array(qindex) of FLOAT;

    function FULL return BOOLEAN is
    begin
        return length = qsize;
    end FULL;

    function EMPTY return BOOLEAN is
    begin
        return length = 0;
    end EMPTY;

    procedure ENQ (r in FLOAT) is
    begin
        if FULL then raise overflow; end if;
        qbody (head + length) mod qsize := r;
        length := length + 1;
    end ENQ;

    function DEQ return FLOAT is
    begin
        if EMPTY then raise underflow; end if;
        head := (head + 1) mod qsize;
        length := length - 1;
        return r;
    end DEQ;

begin
    head := 0;
    length := 0;
end RealQueue;
```

from Ada for Experienced Programmers by N Nico Habermann and Dewayne E Perry

```

package Fractions is
-- modified version of Fractions Package taken from Data
-- Structures with Ada by M. B. Feldman

type Fraction is private;

function MakeFraction(N, D : integer) return Fraction;
procedure Put(X : in Fraction);

function Numer(X : Fraction) return integer;
function Denom(X : Fraction) return integer;

function "+"(X, Y : Fraction) return Fraction;
function "-"(X, Y : Fraction) return Fraction;
function "*" (X, Y : Fraction) return Fraction;
function "/"(X, Y : Fraction) return Fraction;
function Reciprocal(X : Fraction) return Fraction;

function Equal(X, Y : Fraction) return boolean;
function "<"(X, Y : Fraction) return boolean;
function ">="(X, Y : Fraction) return boolean;

function Reduce(X : Fraction) return Fraction;
function ToDecimal(X : Fraction) return float;
function IntegerToFraction(I : integer) return Fraction;
function FractionToInteger(X : Fraction) return integer;

ZeroDenominatorError : exception;

private
type Fraction_Node;
type Fraction is access Fraction_Node;

end Fractions;

```



```
with Text_IO;
package body Fractions is
```

```
type fraction_Node is array(1..2) of integer;
```

```
package Int_IO is new Text_IO.Integer_IO(integer);
```

```
function MakeFraction(N, D : integer) return Fraction is
```

```
  F : Fraction;
```

```
begin
```

```
  if D /= 0 then
```

```
    F := new Fraction_Node;
```

```
    -- for fractions < 0, the numerator carries the negative sign
```

```
    if D < 0 then
```

```
      F(1) := -N;
```

```
      F(2) := -D;
```

```
    else
```

```
      F(1) := N;
```

```
      F(2) := D;
```

```
    end if;
```

```
    return F;
```

```
  else
```

```
    raise ZeroDenominatorError;
```

```
  end if;
```

```
end MakeFraction;
```

```
function Numer(X : Fraction)
```

```
return integer is
```

```
begin
```

```
  return X(1);
```

```
end Numer;
```

```
function Denom(X : Fraction)
```

```
return integer is
```

```
begin
```

```
  return X(2);
```

```
end Denom;
```

```
procedure Put(X : in Fraction) is
```

```
begin
```

```
  Int_IO.Put(Numer(X),1);
```

```
  Text_IO.Put("/");
```

```
  Int_IO.Put(Denom(X),1);
```

```
  Text_IO.Put("hi there");
```

```
end Put;
```

```
function GCD(X, Y : integer)
```

```
return integer is
```

```
begin
```

```
  if Y = 0 then
```

```
    return X;
```

```
  else
```

```
    return GCD(Y,X mod Y);
```

```
  end if;
```

```
end GCD;
```

```
procedure Swap(X, Y : in out integer);
```

```
procedure Swap(X, Y : in out integer) is
```

```
  Temp : integer;
```

```
begin
```

package body Fractions is

type Fraction_Type is

record
 Numerator : integer := 0;
 Denominator : integer := 1;
end record;

package Int_IO is new Text_IO.Integer_IO(integer);

function MakeFraction(N, D : integer) return Fraction is
 F : Fraction;

begin
 if D /= 0 then
 F := new Fraction_Type;
 -- for fractions < 0, the numerator carries the negative sign
 if D < 0 then
 F.Numerator := -N;
 F.Denominator := -D;
 else
 F.Numerator := N;
 F.Denominator := D;
 end if;
 return F;
 else
 raise ZeroDenominatorError;
 end if;
end MakeFraction;

function Numer(X : Fraction) return integer is
begin
 return X.Numerator;
end Numer;

function Denom(X : Fraction) return integer is
begin
 return X.Denominator;
end Denom;

procedure Put(X : in Fraction) is
begin
 Int_IO.Put(Numer(X),1);
 Text_IO.Put("/");
 Int_IO.Put(Denom(X),1);
end Put;

function GCD(X, Y : integer) return integer is
begin
 if Y = 0 then
 return X;
 else
 return GCD(Y, X mod Y);
 end if;
end GCD;

procedure Swap(X, Y : in out integer);

procedure Swap(X, Y : in out integer) is

Temp : integer;
begin
 Temp := X;
 X := Y;
 Y := Temp;
end Swap;

function Reduce(X : Fraction) return Fraction is
 N, D : integer;
begin
 N := Abs(Numer(X));
 D := Abs(Denom(X));
 if N > D then
 Swap(N,D);
 end if;
 return MakeFraction(Numer(X)/GCD(N,D), Denom(X)/GCD(N,D));
end Reduce;

function Reciprocal(X : Fraction) return Fraction is
begin

```

function "+"(X, Y : Fraction)      return Fraction is
begin
    return Reduce( MakeFraction(Numer(X)*Denom(Y)+Numer(Y)*Denom(X),
                                Denom(X)*Denom(Y)) );
end "+";

function "-"(X, Y : Fraction)      return Fraction is
begin
    return Reduce( MakeFraction(Numer(X)*Denom(Y)-Numer(Y)*Denom(X),
                                Denom(X)*Denom(Y)) );
end "-";

function "**"(X, Y : Fraction)      return Fraction is
begin
    return Reduce( MakeFraction(Numer(X)*Numer(Y),Denom(X)*Denom(Y)) );
end "**";

function "/"(X, Y : Fraction)      return Fraction is
begin
    return Reduce( (X*Reciprocal(Y)) );
end "/";

function Equal(X, Y : Fraction)    return boolean is
begin
    return Numer(X)*Denom(Y) = Numer(Y)*Denom(X);
end Equal;

function "<"(X, Y : Fraction)      return boolean is
begin
    return Numer(X)*Denom(Y) < Numer(Y)*Denom(X);
end "<";

function ">="(X, Y : Fraction)     return boolean is
begin
    return Numer(X)*Denom(Y) >= Numer(Y)*Denom(X);
end ">=";

function ToDecimal(X : Fraction)   return float is
begin
    return float(Numer(X)) / float(Denom(X));
end ToDecimal;

function IntegerToFraction(I : integer) return Fraction is
begin
    return MakeFraction(I,1);
end IntegerToFraction;

function FractionToInteger(X : Fraction) return integer is
begin
    return Numer(X) / Denom(X);
end FractionToInteger;

end Fractions;

```

```
package MANAGER is
  type PASSWORD is private;
  NULL_PASSWORD : constant PASSWORD;
  function GET return PASSWORD;
  function IS_VALID(P : in PASSWORD) return BOOLEAN;
private
  type NODE;
  type PASSWORD is access NODE;
end MANAGER;

package body MANAGER is
  type NODE is range 0 .. 7000;
  .
  .
  .
end MANAGER;
```

[Taken from Software Engineering with Ada by Grady [Book]]

- Reason private types must be included in private part of the specification rather than put in the body of a package is that compiler needs to be able to determine how much storage to allocate for instances of private types declared by client modules.
- "Inclusion of private data types in the package specification implies that a change in representation of the private data type will require recompilation of the program unit which contains the package specification, thus violating the principle that package specifications are recompiled only when there are specification changes which affect the user. This appears to be a very heavy price to pay for the simplification in compiling and loading that is achieved by including private declarations in the specification part." [Taken from Programming with Ada by Peter Wegner]

Using a Package

- via textual inclusion in the client module
 - package has NOT been separately compiled
 - package is visible in client from the point it is first declared
 - spec and body do not have to be textually contiguous in client but spec must come first

☐ via context specification using the "with" clause

☐ package has been separately compiled and is made visible via the "with" clause

☐ this method supports ideas of modularity and localization better than textual inclusion method

☐ must prefix references to items with the package's name and a period

☐ necessitates prefix form of notation for use of an overloaded operator such as "+" or "-", etc.

☐ can use "renames" clause to allow use of operators in infix form

☐ can use "USE" clause to gain direct visibility to items in the package

☐ use of "USE" clause can create ambiguity and pollute the name space

☐ Both methods import ALL elements of the visible part of a package -- no selective importation such as in Modula 2 -- argues for carefully segmenting packages to achieve same effect

```

procedure COMPLEX_COMPUTATIONS is
  procedure ONE,
  procedure TWO;
  package COMPLEX is ...
  procedure ONE is ...
  procedure TWO is ...
  package body COMPLEX is ...

```

```

  X, Y, Z : COMPLEX.COMPLEX_NUMBER;
begin
  Z := COMPLEX."+"(X,Y);
end;

```

```

with COMPLEX;
procedure COMPLEX_COMPUTATIONS is
  procedure ONE;
  procedure TWO;
  procedure ONE is ...
  procedure TWO is ...

```

```

  X, Y, Z : COMPLEX.COMPLEX_NUMBER;
begin
  Z := COMPLEX."+"(X,Y);
end;

```

```

with COMPLEX; use COMPLEX;
procedure COMPLEX_COMPUTATIONS is
  procedure ONE,
  procedure TWO;
  procedure ONE is ...
  procedure TWO is ...

```

```

  X, Y, Z : COMPLEX_NUMBER;
begin
  Z := X + Y;
end;

```


Example of a PACKAGE

package GOLF_INFO is

```
type GOLF_CLUB    is (DRIVER, IRON, PUTTER, WEDGE, MASHIE);
type GOLF_SCORE   is range 1 .. 200;
type HOLE_NUMBER  is range 1 .. 18;
type HANDICAP     is range 0 .. 36;
type SCORE_DATA   is array (HOLE_NUMBER) of GOLF_SCORE;
PAR_FOR_COURSE : constant GOLF_SCORE := 72;
PAR_VALUES : constant SCORE_DATA :=
    (1 => 5, 2 => 3, 3 => 4, 4 => 4, 5 => 3, 6 => 4,
     7 => 5, 8 => 4, 9 => 4, 10=> 3, 11=> 4, 12=> 4,
     13=> 4, 14=> 5, 15=> 3, 16=> 4, 17=> 4, 18=>5);
```

```
procedure COMPUTE_TOTAL_SCORE(SCORES: in SCORE_DATA;
                              TOTAL: out GOLF_SCORE);
```

end GOLF_INFO;

package body GOLF_INFO is

```
procedure COMPUTE_TOTAL_SCORE(SCORES: in SCORE_DATA;
                              TOTAL: out GOLF_SCORE) is
```

```
begin
    TOTAL := 0;
    for HOLE in HOLE_NUMBER loop
        TOTAL := TOTAL + SCORES(HOLE);
    end loop;
end;
```

end GOLF_INFO;

[Taken from *Ada: An Introduction* by Henry Ledgard]

Using the PACKAGE

```
with GOLF_INFO, TEXT_IO;  
use GOLF_INFO, TEXT_IO;  
procedure KEEP_SCORE is
```

```
    MY_SCORES    : SCORE_DATA;  
    TOTAL_SCORE  : GOLF_SCORE;
```

```
begin
```

```
    PUT("Let's have the scores for each hole.");  
    for HOLE in HOLE_NUMBER loop  
        NEW_LINE;  
        PUT(HOLE); PUT(" ");  
        GET(MY_SCORES(HOLE));  
    end loop;
```

```
    COMPUTE_TOTAL_SCORE(MY_SCORES, TOTAL_SCORE);  
    NEW_LINE;  
    PUT("Your total is "); PUT(TOTAL_SCORE);
```

```
    NEW_LINE;  
    if TOTAL_SCORE < PAR_FOR_COURSE then  
        PUT(PAR_FOR_COURSE - TOTAL_SCORE); PUT(" Under Par");  
    elsif TOTAL_SCORE = PAR_FOR_COURSE then  
        PUT("An Even Par");  
    else  
        PUT(TOTAL_SCORE - PAR_FOR_COURSE); PUT(" Over Par");  
    end if;
```

```
end KEEP_SCORE;
```

[Taken from *Ada: An Introduction* by Henry Ledgard]

```

procedure OUTER is
  package HEIGHT is
    ID : INTEGER;
    VALUE : FLOAT;
  end HEIGHT;

  package WEIGHT is
    ID : INTEGER;
    VALUE : FLOAT;
  end WEIGHT;

  procedure INNER is
    use HEIGHT, WEIGHT;
  begin
    HEIGHT.VALUE := 65.3;
    WEIGHT.VALUE := HEIGHT.VALUE;
    VALUE := 63.5;  -- unqualified name causes ambiguity
  end INNER;

begin
  VALUE := 63.5;  -- illegal because outside scope of USE
                  -- statement so name must be qualified
end;

```

[Example taken from Programming with Ada by Peter Wegner]

PROGRAM 3.8 Modula-2 Version of Program 3.7

(• This definition module is a separate compilant and is compiled before the program ReverseName is compiled. •)

DEFINITION MODULE Stacks;

(• \$SEG:=8; •)

(• This is a compiler directive required by the particular operating system being used. It assigns a segment number of a definition module. In general, this is not required. •)

EXPORT QUALIFIED

(• type •) Stack,
(• proc •) Empty,
(• proc •) Pop,
(• proc •) Push,
(• proc •) Initialize,
(• proc •) Remove;

TYPE Stack; (• Representational details hidden •)

PROCEDURE Empty(S: Stack) : BOOLEAN;
(• Returns true if stack is empty. •)

PROCEDURE Pop(VAR S: Stack) : CHAR;
(• Strips top element off stack. •)

PROCEDURE Push(VAR S: Stack; X: CHAR);
(• Adds element to top of stack. •)

PROCEDURE Initialize(VAR S: Stack);
(• Sets stack to empty. •)

PROCEDURE Remove(VAR S: Stack);
(• Removes stack from memory. •)

END Stacks.

[Taken from ...]

PROGRAM 3.8 Modula-2 Version of Program 3.7

(• This definition module is a separate compilant and is compiled before the program ReverseName is compiled. •)

DEFINITION MODULE Stacks;

(• \$SEG := 8; •)

(• This is a compiler directive required by the particular operating system being used. It assigns a segment number of a definition module. In general, this is not required. •)

EXPORT QUALIFIED

- (• type •) Stack;
- (• proc •) Empty;
- (• proc •) Pop;
- (• proc •) Push;
- (• proc •) Initialize;
- (• proc •) Remove;

TYPE Stack; (• Representational details hidden •)

PROCEDURE Empty(S: Stack) : BOOLEAN;

(• Returns true if stack is empty. •)

PROCEDURE Pop(VAR S: Stack) : CHAR;

(• Strips top element off stack. •)

PROCEDURE Push(VAR S: Stack; X: CHAR);

(• Adds element to top of stack. •)

PROCEDURE Initialize(VAR S: Stack);

(• Sets stack to empty. •)

PROCEDURE Remove(VAR S: Stack);

(• Removes stack from memory. •)

END Stacks.

Implementation Details of Module Stacks

IMPLEMENTATION MODULE Stacks;

FROM Terminal IMPORT WriteString, WriteLn;

FROM Storage IMPORT ALLOCATE, DEALLOCATE;

CONST STACKSIZE = 80;

TYPE Objects = ARRAY [1..STACKSIZE] OF CHAR;

Stack = POINTER TO RECORD

ITEM: Objects;

TOP : CARDINAL

END;

PROCEDURE Empty(S: Stack) : BOOLEAN;
BEGIN

IF S.TOP = 0 THEN

RETURN TRUE

ELSE

RETURN FALSE

END

END Empty;

PROCEDURE Pop(VAR S: Stack) : CHAR;

(• Strips top element off stack. •)

BEGIN

IF Empty(S) THEN

WriteString("Stack underflow.");

WriteLn;

HALT;

ELSE

DEC(S.TOP);

RETURN S.ITEM[S.TOP+1];

END

END Pop;

[Taken from Software Engineering with Modula-2 and Ada by F. Wiener and R. Sincoe]

MODULE ReverseName; (* This is the program. *)

(* In Modula-2, all reserved words must be given in uppercase. *)

FROM Stacks IMPORT Empty, Pop, Initialize, Remove, Stack;

FROM InOut IMPORT WriteString, WriteLn, Write, Read, EOL;

PROCEDURE Find Middle (VAR S : Stack) : CHAR;

(* We use an algorithm different from that in Program 3.7. *)

VAR LocalStack: Stack;
 ch : CHAR;

BEGIN

 Initialize(LocalStack);

 REPEAT

 ch := Pop(S);

 Push(LocalStack, ch);

 UNTIL ch = '.';

 ch := Pop(S);

 Push(LocalStack, ch);

 (* Now we restore the stack S to its original form. *)

 WHILE NOT Empty(LocalStack) DO

 Push(S, Pop(LocalStack));

 END;

 RETURN ch

END FindMiddle;

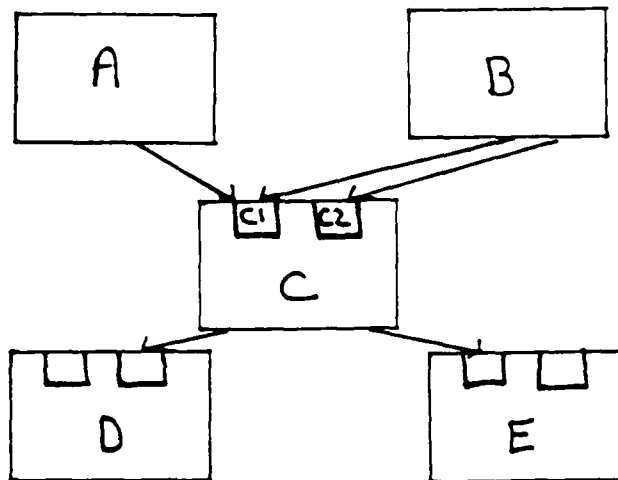
VAR Name : ARRAY[1..40] OF CHAR;
 A : Stack;

•
•
•

[Taken from Software Engineering with Modula-2 and Ada
by R. Wiener and R. Sincovec]

Limitations on Structure Specifications in Ada

- ☐ No support for declaration of structure graph interconnections at specification level
- ☐ Can include this information with comments as an alternative



```
package C is
  required(D,E);
  is required by (A,B);
  procedure C1 (...);
  procedure C2 (...);
end package C;
```

[Taken from System Design with Ada by R.J.A. Buhr]

Package Applications

- ☐ **Encapsulate related data types, constants or objects.**
 - ☐ only objects and types exported, no pgm units
 - ☐ package body may or may not be present
 - ☐ facilitates consistency by factoring out common elements used by many modules
- ☐ **Encapsulate related program units.**
 - ☐ only program units exported, no objects and types
- ☐ **Embody an Abstract Data Type (ADT)**
 - ☐ objects, types, and program units exported
 - ☐ state information not maintained in body of the pkg
 - ☐ supports abstraction and information hiding
- ☐ **Embody an Abstract-state machine.**
 - ☐ objects, types, and program units exported
 - ☐ state information maintained in body of pkg
- ☐ **Encapsulate tasks.***
 - ☐ often used purely to enclose task(s) because they cannot be separately compiled

[Info from Software Engineering with rules by or for people with experience in it]


```
package TRIG_PKG is
```

```
function FACTORIAL(n: in integer) return float;  
-- returns n! as a floating point value
```

```
function SIMPLIFY(angle_in_degrees : in integer) return integer;  
-- returns an equivalent angle (in degrees) for ANGLE_IN_DEGREES  
-- which is positive and between 0 and 360
```

```
function RADIANS(degrees : in integer) return float;  
-- returns radian value for DEGREES
```

```
function SIN(angle_in_degrees : in integer) return float;  
-- returns the sine of ANGLE_IN_DEGREES
```

```
function COS(angle_in_degrees : in integer) return float;  
-- returns the cosine of ANGLE_IN_DEGREES
```

```
end TRIG_PKG;
```

```
package body TRIG_PKG is
```

```
    FACT : array(0..15) of float :=  
        (1.0,  
         1.0,  
         2.0,  
         6.0,  
         24.0,  
         120.0,  
         720.0,  
         5040.0,  
         40320.0,  
         362880.0,  
         3628800.0,  
         39916800.0,  
         479001600.0,  
         6227020800.0,  
         87178291200.0,  
         1307674368000.0);
```

```
function FACTORIAL(n: in integer) return float is  
    x : float := 1.0;  
begin  
    if n < 0 then  
        -- if n is NEGATIVE return ZERO to indicate ERROR  
        return 0.0;  
    elsif ((n) = 0) and (n <= 15) then  
        return FACT(n);  
    elsif n > 15 then  
        for i in 1..n loop  
            x := x * float(i);  
        end loop;  
        return x;  
    end if;  
end;
```

```

function SIMPLIFY(angle_in_degrees : in integer) return integer is
    angle : integer;
begin
    angle := angle_in_degrees rem 360;
    if angle < 0 then
        angle := 360 + angle;
    end if;
    return angle;
end;

function RADIANS(degrees : in integer) return float is
    pi : float := 3.14159265;
begin
    return ((pi/float(180)) * float(degrees));
end;

function SIN(angle_in_degrees : in integer) return float is
    sum : float := 0.0;
    angle : integer;
    temp_angle : integer;
begin
    temp_angle := SIMPLIFY(angle_in_degrees);
    if temp_angle >= 180 then
        -- First and second quadrant angles used for computation
        -- because computation is more accurate
        angle := temp_angle - 180;
    else
        angle := temp_angle;
    end if;
    for i in 0..7 loop
        sum := sum + ( ( float((-1)**i) * (RADIANS(angle**2**i)) )
                        / ( FACT(2*i) ) );
    end loop;

    -- Account for use of only first and second quadrant angles
    -- in computation
    if temp_angle > 180 then
        return (-sum);
    else
        return sum;
    end if;
end;

function COS(angle_in_degrees : in integer) return float is
    sum : float := 0.0;
    angle : integer;
    temp_angle : integer;
begin
    temp_angle := SIMPLIFY(angle_in_degrees);
    if temp_angle >= 180 then
        -- First and second quadrant angles used for computation
        -- because computation is more accurate
        angle := temp_angle - 180;
    else
        angle := temp_angle;
    end if;
    for i in 0..7 loop
        sum := sum + ( ( float((-1)**i) * (RADIANS(angle**2**i)) )
                        / ( FACT(2*i) ) );
    end loop;

    -- Account for use of only first and second quadrant angles
    -- in computation
    if temp_angle >= 180 then
        return (-sum);
    else
        return sum;
    end if;
end;

end TRIG_PKG;

```

```

with TRIG_PKG, GIGI_GRAPHICS_PKG, TYPES:
use TRIG_PKG, GIGI_GRAPHICS_PKG, TYPES:
package TURTLES is

```

```

    subtype ANGLE is integer range 0..360;
    type TURTLE is private;

```

```

    -- TURTLE CONTROLS --

```

```

-- PEN CONTROLS

```

```

    procedure UP(myturtle : in out TURTLE);
    procedure DOWN(myturtle : in out TURTLE);
    function PEN_IS_DOWN(myturtle : in turtle) return boolean;
    procedure NEW_PEN(myturtle : in out TURTLE; colour : in COLORTYPE);

```

```

-- MOVEMENT CONTROLS

```

```

    procedure NORTH(myturtle : in out TURTLE);
    procedure SOUTH(myturtle : in out TURTLE);
    procedure EAST(myturtle : in out TURTLE);
    procedure WEST(myturtle : in out TURTLE);
    procedure MOVE(myturtle : in out TURTLE; n : in integer);
    procedure TURN(myturtle : in out TURTLE; a : in ANGLE);
    procedure TURN_TO(myturtle : in out TURTLE; a : in ANGLE);

```

```

private

```

```

    type POSITION is (pen_up, pen_down);

```

```

    type PEN is record

```

```

        pen_position : POSITION;
        pen_colour : COLORTYPE;
    end record;

```

```

    subtype XCOORD is integer range 0..768;

```

```

    subtype YCOORD is integer range 0..479;

```

```

    type POINT is record

```

```

        x : XCOORD;
        y : YCOORD;
    end record;

```

```

    type TURTLE is record

```

```

        pen_status : PEN := (pen_down, white);
        heading : ANGLE := 0;
        location : POINT := (350, 250);
    end record;

```

```

; TURTLES;

```

```
package body TURTLES is
```

```
-- TURTLE PEN CONTROLS --
```

```
procedure SET_PEN_COLOUR(myturtle : in out TURTLE; colour : in COLORTYPE);  
procedure SET_PEN_COLOUR(myturtle : in out TURTLE; colour : in COLORTYPE) is  
begin  
    myturtle.pen_status.pen_colour := colour;  
end;
```

```
procedure NEW_PEN(myturtle : in out TURTLE; colour : in COLORTYPE) is  
begin  
    SET_PEN_COLOUR(myturtle, colour);  
end;
```

```
function PEN_IS_DOWN(myturtle : in TURTLE) return boolean is  
begin  
    if myturtle.pen_status.pen_position = pen_down then  
        return true;  
    else  
        return false;  
    end if;  
end;
```

```
procedure UP(myturtle : in out TURTLE) is  
begin  
    myturtle.pen_status.pen_position := pen_up;  
end;
```

```
procedure DOWN(myturtle : in out TURTLE) is  
begin  
    myturtle.pen_status.pen_position := pen_down;  
end;
```

-- DRAWING and UN-DRAWING the TURTLE --

```

procedure TRANSFORM(myturtle : in TURTLE; x : in integer; y : in integer;
    new_x : out XCOORD; new_y : out YCOORD;
    theta : in ANGLE);
procedure TRANSFORM(myturtle : in TURTLE; x : in integer; y : in integer;
    new_x : out XCOORD; new_y : out YCOORD;
    theta : in ANGLE) is

```

```

begin
    new_x := myturtle.location.x +
        integer(float(x)*COS(theta) - float(y)*SIN(theta));
    new_y := myturtle.location.y +
        integer(float(x)*SIN(theta) + float(y)*COS(theta));
end;

```

```

procedure DRAW_TURTLE(myturtle : in TURTLE);
procedure DRAW_TURTLE(myturtle : in TURTLE) is
    x,y : integer;
begin
    SETWRITINGMODE(replace);
    SETWRITINGCOLOR(myturtle.pen_status.pen_colour);
    SETCURSORAT(myturtle.location.x, myturtle.location.y);
    TRANSFORM(myturtle, 0, -18, x, y, myturtle.heading);
    DRAWLINETO(x, y);
    TRANSFORM(myturtle, 24, 0, x, y,
        myturtle.heading);
    DRAWLINETO(x, y);
    TRANSFORM(myturtle, 0, 18, x, y,
        myturtle.heading);
    DRAWLINETO(x, y);
    TRANSFORM(myturtle, 0, 0, x, y,
        myturtle.heading);
    DRAWLINETO(x, y);
    SETCURSORAT(myturtle.location.x, myturtle.location.y);
end;

```

```

procedure UNDRAW_TURTLE(myturtle : in TURTLE);
procedure UNDRAW_TURTLES(myturtle : in TURTLE) is
    x,y : integer;
begin
    SETWRITINGMODE(erase);
    SETWRITINGCOLOR(dark);
    SETCURSORAT(myturtle.location.x, myturtle.location.y);
    TRANSFORM(myturtle, 0, -18, x, y, myturtle.heading);
    DRAWLINETO(x, y);
    TRANSFORM(myturtle, 24, 0, x, y,
        myturtle.heading);
    DRAWLINETO(x, y);
    TRANSFORM(myturtle, 0, 18, x, y,
        myturtle.heading);
    DRAWLINETO(x, y);
    TRANSFORM(myturtle, 0, 0, x, y,
        myturtle.heading);
    DRAWLINETO(x, y);
    SETCURSORAT(myturtle.location.x, myturtle.location.y);
end;

```

-- TURTLE MOVEMENT CONTROLS --

```
procedure MOVE(myturtle : in out TURTLE; n : in integer) is
begin
  UNDRAW_TURTLE(myturtle);
  if PEN_IS_DOWN(myturtle) then
    SETWRITINGMODE(replace);
    SETWRITINGCOLOR(myturtle.pen_status.pen_colour);
    DRAWLINETO(integer( float(n)*COS(myturtle.heading)+
                        float(myturtle.location.x) ),
               integer( float(n)*SIN(myturtle.heading)+
                        float(myturtle.location.y) ));
  else
    MOVECOURSEARBS(integer( float(n)*COS(myturtle.heading)+
                        float(myturtle.location.x) ),
                   integer( float(n)*SIN(myturtle.heading)+
                        float(myturtle.location.y) ));
  end if;
  QUERYCURSORPOS(myturtle.location.x,myturtle.location.y);
  DRAW_TURTLE(myturtle);
end;
```

```
procedure TURN(myturtle : in out TURTLE; a : in ANGLE) is
begin
  UNDRAW_TURTLE(myturtle);
  myturtle.heading := SIMPLIFY(myturtle.heading + a);
  DRAW_TURTLE(myturtle);
end;
```

```
procedure TURN_TO(myturtle : in out TURTLE; a : in ANGLE) is
begin
  UNDRAW_TURTLE(myturtle);
  myturtle.heading := a;
  DRAW_TURTLE(myturtle);
end;
```

```
procedure NORTH(myturtle : in out TURTLE) is
begin
  TURN_TO(myturtle,90);
  MOVE(myturtle,1);
end;
```

```
procedure SOUTH(myturtle : in out TURTLE) is
begin
  TURN_TO(myturtle,270);
  MOVE(myturtle,1);
end;
```

```
procedure EAST(myturtle : in out TURTLE) is
begin
  TURN_TO(myturtle,0);
  MOVE(myturtle,1);
end;
```

```
procedure WEST(myturtle : in out TURTLE) is
begin
  TURN_TO(myturtle,180);
  MOVE(myturtle,1);
end;
```

```
begin
  SETSCREENSCALE(0.760,470,0);
  CLEARSCREEN;
  SETCURSORAT(0,0);
end TURTLES;
```

```

package body LEXICAL_ANALYZER is

  type STATE is (START, BUILD_IDENTIFIER, BUILD_NUMBER, STOP);
  PRESENT_STATE : STATE := START;

  subtype ALPHA is CHARACTER range 'A' .. 'Z';
  subtype DIGIT is CHARACTER range '0' .. '9';
  procedure SET_START_STATE is
  begin
    PRESENT_STATE := START;
  end SET_START_STATE;

  procedure RECEIVE_SYMBOL(C : in CHARACTER) is
  begin
    case PRESENT_STATE is
      when START
        => if (C in ALPHA) then
            PRESENT_STATE := BUILD_IDENTIFIER;
          elsif (C in DIGIT) then
            PRESENT_STATE := BUILD_NUMBER;
          else
            raise INVALID_CHARACTER;
          end if;
        when BUILD_IDENTIFIER => if (C in ALPHA) or (C in DIGIT) then
            null;
          else
            PRESENT_STATE := STOP;
            raise IDENTIFIER_ACCEPTED;
          end if;
        when BUILD_NUMBER
        => if (C not in DIGIT) then
            PRESENT_STATE := STOP;
            raise NUMBER_ACCEPTED;
          end if;
        => raise MACHINE_HALTED;
      when STOP
        => raise RECEIVE_SYMBOL;
      end case;
    end RECEIVE_SYMBOL;
  end LEXICAL_ANALYZER;

```

[Taken from Software Engineering with Ada by
Grady Booch]

```
package Terminal_Driver_Package is
```

```
    task Terminal_Driver is
        entry Read_Character(C : out Character);
        entry Write_Character(C : in Character);
        entry Reset;
        entry ShutDown;
    end Terminal_Driver;
```

```
end Terminal_Driver_Package;
```

```
with Queue_Package, Low_Level_IO, System;
use Low_Level_IO;
```

```
package body Terminal_Driver_Package is
```

```
    task body Terminal_Driver is
```

```
        -- Group all of the machine dependent constants together
```

```
        Console_Input_Vector : constant System.Address := 8#60#;
        Console_Output_Vector : constant System.Address := 8#64#;
        Enable_Interrupts : Integer := 8#100#;
        Write_Time_Out : constant Duration := 0.5;
        Number_Of_Lines : constant := 2;
        LineLength : constant := 132;
```

```
        task type Device_Reader is
            entry Interrupt;
            entry StartUpDone;
            for Interrupt use at Console_Input_Vector;
        end Device_Reader;
```

```
        task type Device_Writer is
            entry Interrupt;
            entry StartUpDone;
            for Interrupt use at Console_Output_Vector;
        end Device_Writer;
```

```
        package Char_Queue_Package is new Queue_Package(Character);
        use Char_Queue_Package;
```

```
        type DriverStateBlock is
            record
                InputCharBuffer, OutputCharBuffer :
                    Blocking_Queue(Number_Of_Lines*LineLength);
                CurReader : Device_Reader;
                CurWriter : Device_Writer;
            end record;
```

```
        type RefToBlock is access DriverStateBlock;
        CurState : RefToBlock;
```

```
        task body Device_Reader is
            TempInput : Character;
        begin
            accept StartUpDone;
            Send_Control(Console_Keyboard_Control, Enable_Interrupts);
            loop
                accept Interrupt do
                    Receive_Control(Console_Keyboard_Data, TempInput);
                end Interrupt;
                Append(CurState.InputCharBuffer, TempInput);
            end loop;
        end Device_Reader;
```



```

task body Device_Writer is
    TempOutput : Character;
begin
    accept StartUpDone;
    Send_Control(Console_Printer_Control, Enable_Interrupts);
    accept Interrupt; -- spurious interrupt caused by Send_Control
    loop
        Remove(CurState.OutputCharBuffer, TempOutput);
        Send_Control(Console_Printer_Data, TempOutput);
        select
            accept Interrupt;
        or
            delay Write_Time_Out;
        end select;
    end loop;
end Device_Writer;

procedure ShutDownOld is
begin
    abort CurState.CurReader;
    abort CurState.CurWriter;
    Destroy_Queue(CurState.InputCharBuffer);
    Destroy_Queue(CurState.OutputCharBuffer);
end ShutDownOld;

procedure StartUp is
begin
    CurState := new DriverStateBlock;
    Init_Queue(CurState.InputCharBuffer);
    Init_Queue(CurState.OutputCharBuffer);
    CurState.CurReader.StartUpDone;
    CurState.CurWriter.StartUpDone;
end StartUp;

begin
    StartUp;

Console_Operations:
    loop
        select
            accept Read_Character(C : out Character) do
                Remove(CurState.InputCharBuffer, C);
            end Read_Character;
        or
            accept Write_Character(C : in Character) do
                Append(CurState.OutputCharBuffer, C);
            end Write_Character;
        or
            accept Reset do
                ShutDownOld;
                StartUp;
            end Reset;
        or
            accept ShutDown;
            ShutDownOld;
            exit Console_Operations;
        or
            terminate;
        end select;
    end loop Console_Operations;
exception
    when others =>
        ShutDownOld;

end Terminal_Driver;

```

Teaching Packages

- ☐ Start at the PACKAGE level and then introduce the syntax by example -- NOT at the syntax level and then intro packages -- good way to produce Fortran programs written in Ada
- ☐ Give students preconstructed packages and have them build defined products with them
- ☐ Have students play "client" package and package developer
- ☐ Have students develop a larger scale system with packages that must interface
- ☐ Have students build a system which has a heirarchical structure of packages
- ☐ Have students do a package and replace its implementation and observe results

- ☐ Give students package with "open" data structure in the spec and have them corrupt its integrity and then challenge them to do the same thing with a package with the data structure in the private part of the spec
- ☐ Data structures course very easy vehicle for introducing packages under topic of abstraction
- ☐ Digital design course offers opportunity to intro packages to encapsulate abstract-state machines and tasks



David A. Cook

INSTRUCTOR, COMBAT AIR SCHOOL
U.S. AIR FORCE ACADEMY (C-110041)

472 3590
AV 259 3590
DFCS

HOME 472 6935
QTHS 4206F
USAF A (O 80841)

Ada* Tasking Abstraction of Process

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

** Ada is a registered trademark of the U.S.
Government, Ada Joint Program Office

ADA TASKING

- OVERVIEW

DEFINE ADA TASKING

DEFINE SYNCHRONIZATION
MECHANISM

EXAMPLES

ADA TASKING

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 REQUESTS
AND PROVIDE MODES

GUARDS ARE "IF STATEMENTS" FOR
PROVIDING SERVICE

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.

FOR THE MAIN PROGRAM, TERMINATION DOES
NOT DEPEND ON TASK WHOSE MASTER IS A
LIBRARY PACKAGE.

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

Task Type T1 is...
Obj : T1;
Type T1_Ptr is access T1;
Ptr_Obj : T1_Ptr := new T1;

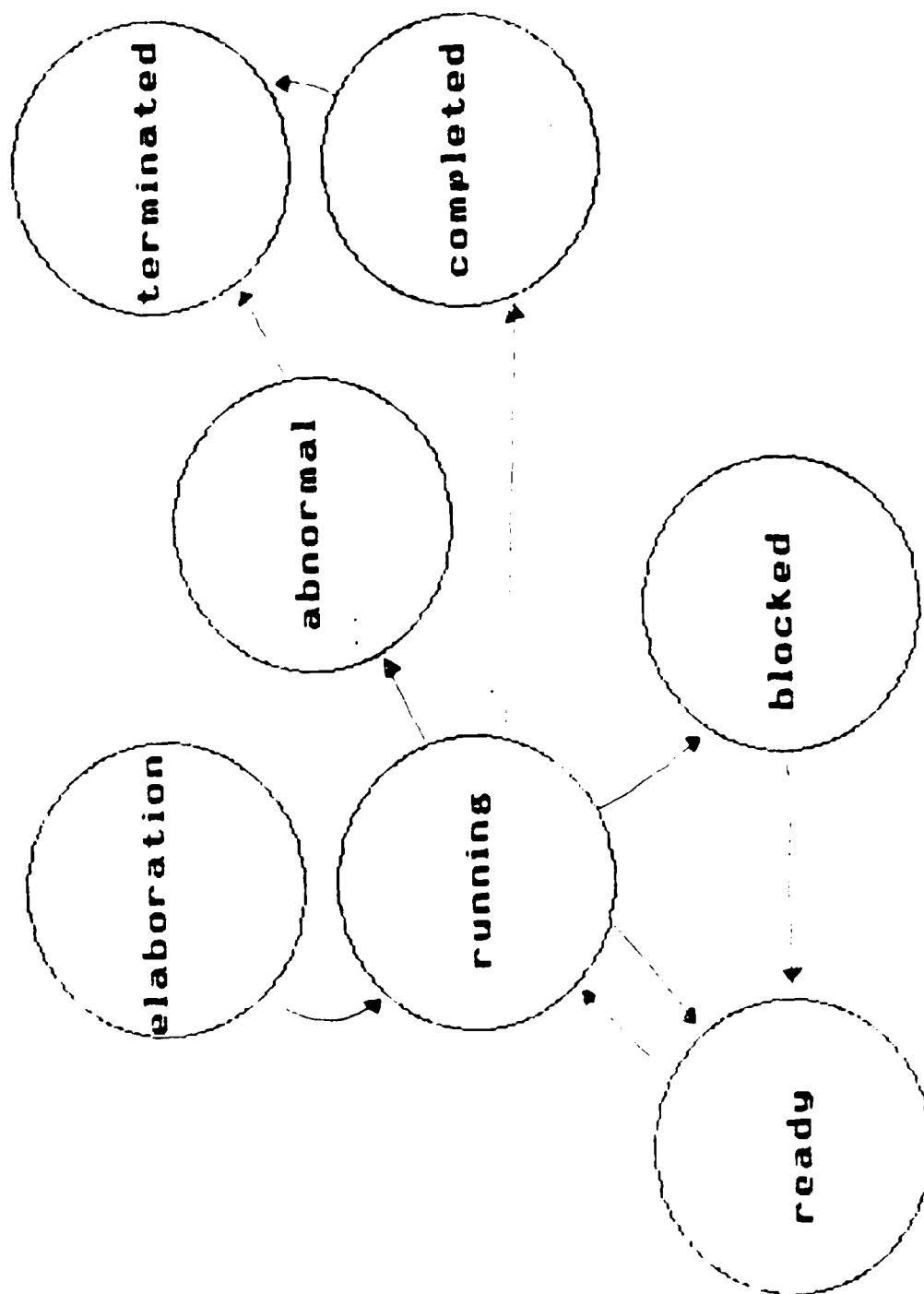
begin
declare

New_Ptr_Obj:T1_Ptr:=new T1;

begin
null;
end;

...
...
end;

+



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  
    {entry_declaration}  
    {representation_clause}  
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 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 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 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

```
...  
...  
...  
    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
THE DELAY OR AN ELSE CLAUSE.

SINCE THIS IS USED ONLY WITH A 'WAIT FOREVER'
TASK, 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 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  
  T1.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 ENTRIES

```
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 MESSAGE;

```

REPRESENTATION SPECIFICATIONS

LENGTH CLAUSE

T'SORAGE_SIZE

```
TASK TYPE T1 IS  
  ENTRY ENTRY_1;  
  FOR T1'SORAGE_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

AD-A189 641

ADVANCED ADA WORKSHOP AUGUST 1987(U) ADA JOINT PROGRAM
OFFICE ARLINGTON VA 21 AUG 87

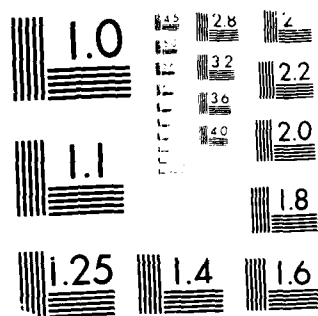
4/4

UNCLASSIFIED

F/G 12/3

NL

END
DATE
INDEXED
4 88
FILE



MICROCOPY RESOLUTION TEST CHART
 NATIONAL BUREAU OF STANDARDS-1963-A


```

GENERIC
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 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_TRAY := COOK;
  select
    accept SERVE<TRAY_OF : out FOOD_TYPE> do
      TRAY_OF := NEW_TRAY;
    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;

```

-- 某某

```

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;

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;

ADA TASKING

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;
```

ADA TAS ING

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;

CUREENT_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;
end loop
end;
```

```

Task Body SERVOMATIC is
  NEXT_NUMBER : SERVOMATIC_NUMBERS :=
    SERVOMATIC_NUMBERS'first;

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

```

loop

end

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

"LETS 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;

select

accept PRINT_FILE<NAME : in STRING;

PRIORITY : NATURAL > do

INSERT <NAME, PRIORITY>;

--put name on queue or queues

-- according to priority

end PRINT_FILE;

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 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
SENARIO.

THREE TASKS HAVE ACCES 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 INDICATION 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.

run cookie

The house is ok

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

0
House alarm reset to OK.

The house is ok

The house is ok

!
The alarm has been turned off

*)

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 '1' .. '9' => ALARM.FIRE (CHAR);
      WHEN 'A' .. 'Z' => ALARM.INTRUDER (CHAR);
      WHEN 'a' .. 'z' => ALARM.INTRUDER (CHAR);
      WHEN '0'      => HOUSE.OK;
      WHEN '!'      => ALARM.SHUTOFF;
      WHEN OTHERS   => PUT_LINE
        ("INVALID CHARACTER. TRY AGAIN");
    END CASE;
    EXIT WHEN CHAR = '!';
  END LOOP;

END COOKIE;

```

Tutorial on Ada* Exceptions

by

Major Patricia K. Lawlis

lawlis%asu@csnet-relay

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

21 August 1987

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

References

- Student Handout, "Ada Applications Programmer - Advanced Ada Software Engineering", USAF Technical Training School, Keesler Air Force Base, July 1986.
- J. G. P. Barnes, Programming in Ada, 2nd edition, Addison-Wesley, 1984.
- Grady Booch, Software Engineering with Ada, Benjamin/Cummings, 1983.
- Theodore F. Elbert, Embedded Programming in Ada, Van Nostrand Reinhold, 1986.
- Putnam P. Texel, Introductory Ada: Packages for Programming, Wadsworth, 1986.
- ANSI/MIL-STD-1815A, "Military Standard - Ada Programming Language" (LRM), U. S. Department of Defense, 22 January 1983.

Outline

=> Overview

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

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

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

The American Way

```
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;
```

Using Exceptions

```
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

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;
USE_ERROR      : exception;      --attempt to use
                                   --invalid operation

STATUS_ERROR   : exception;
MODE_ERROR     : exception;
DEVICE_ERROR   : exception;
END_ERROR      : exception;      --attempt to read
                                   --beyond end of file

DATA_ERROR     : exception;      --attempt to input
                                   --wrong type

LAYOUT_ERROR   : exception;      --for text processing
```

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

Creating an Exception Handler

- Defining an exception handler
- Restrictions
- Handler example

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
```

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

Handler Example

```
procedure Whatever is
    Problem_Condition : exception;
begin
    ...
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

Raising an Exception

- How exceptions are raised
- Effects of raising an exception
- Raising example

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

- Control transfers to exception handler at end of frame (if one exists)
- Exception is lowered
- Sequence of statements in exception handler is executed
- Control passes to end of frame
- If frame does not contain an appropriate exception handler, the exception is propagated

Raising Example

```
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

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

- If exception is raised during normal execution, system looks for an exception handler at the end of the frame in which the exception occurred
- If exception is raised during elaboration of the declarative part of a frame
 - elaboration is abandoned and control goes to the end of the frame with the exception still raised
 - exception part of the frame is not searched for an appropriate handler
 - effectively, the calling unit will be searched for an appropriate handler
 - 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

Sequence of Statements in Exception Handler

- Handler completes the execution of the frame
 - handler for a function should usually contain a return statement
- Statements can be of arbitrary complexity
 - can use most any language construct that makes sense in that context
 - cannot use goto statement to transfer into a handler
 - if handler is in a block inside a loop, could use exit statement
- Handler at end of package body applies only to package initialization

Propagation

- Occurs if no handler exists in frame where 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
- Propagation continues until
 - an appropriate handler is found
 - exception propagates to main program (still with no handler) and program execution is abandoned

Propagation Example

```
procedure Do_Nothing is
    -----
    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;
    -----
begin -- Do_Nothing
    ...
    Calls_It;
    ...
exception
    when others => Fix_Everything;
end Do_Nothing;
```

Outline

- Overview
- Naming an exception
- Creating an exception handler
- Raising an exception
- Handling exceptions

=> Turning off exception checking

- Tasking exceptions
- More examples

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 and foremost, make program work
 - 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

Moral

Improving the algorithm is far better - and easier in
the long run - than suppressing checks

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

Tasking Exceptions

- Exception handling is trickier for tasks
- Exceptions during task rendezvous
- 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
 - 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 Rendezvous

- If the called task terminates abnormally

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)

- 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

Tasking Example

```
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

More Examples

- Interactive data input
- Propagating exception out of scope and back in
- Keeping a task alive

Interactive Data Input

```
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 | CONSTRAINT_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;
```

Keeping a Task Alive

```
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;
```

SUMMARY

- cover the topic well
- use good visual aids
- make organization obvious
- don't leave this as an isolated topic
continue to use exceptions from after

DAT
FILM
4