

AD-A166 367

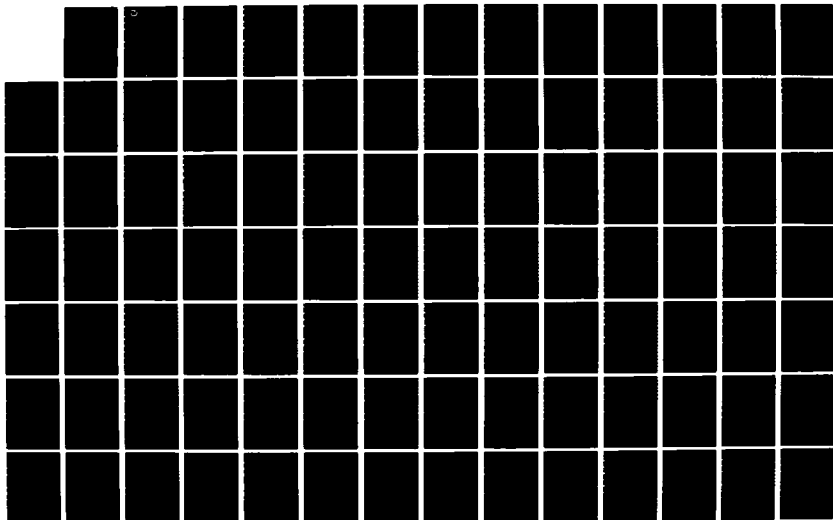
ADA (TRADE NAME) TRAINING CURRICULUM BASIC ADA
PROGRAMMING L202 TEACHER'S GUIDE VOLUME 2(U) SOFTECH
INC WALTHAM MA 1986 DAAB07-83-C-K514

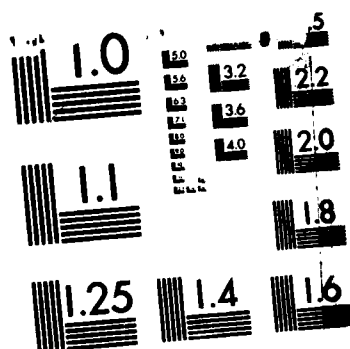
1/8

UNCLASSIFIED

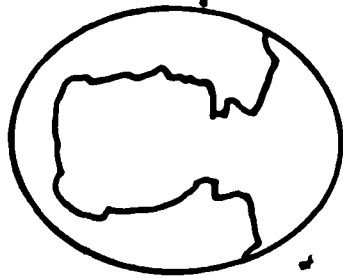
F/G 9/2

NL





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

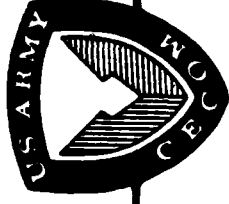


AD-A166 367

①

1986

Ada® Training Curriculum



Basic Ada® Programming L202 Teacher's Guide Volume II

DTIC FILE COPY

DTIC
ELECTE
APR 03 1986
S D E

*Sub A143585
KD*

Prepared By:

U.S. Army Communications-Electronics Command
(CECOM)

SOFTECH, INC.
460 Totten Pond Road
Waltham, MA 02154

Contract DAAB07-83-C-K5D6
14

86 . 4 2 041

PART II
BASIC Ada* PROGRAMMING (L202)
TEACHER'S GUIDE

INSTRUCTOR NOTES

ALLOCATE AT LEAST ONE HOUR AND A HALF OF LECTURE FOR THIS SECTION.

ASSIGN EXERCISES 22 THROUGH 24 OF THE EXERCISE BOOKLET FOR LAB WORK. (PUT HIGHER STRESS ON 24.)

THE OBJECTIVE OF THIS SECTION IS TO INTRODUCE RECORD TYPES AND OBJECTS, RECORD AGGREGATES, OPERATIONS ON RECORD OBJECTS, AND RECORD ATTRIBUTES.

SECTION 9 RECORD TYPES

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By _____	
Distribution/	
Availability Codes	
Avail and/or	
Dist	Special
A-1	



INSTRUCTOR NOTES

POINT OUT THAT ARRAYS ARE COMPOSITE OBJECTS ALSO, YET COMPONENTS OF ARRAYS MUST BE OF THE SAME TYPE. RECORD OBJECTS MAY HAVE COMPONENTS OF VARYING TYPES.

RECORD

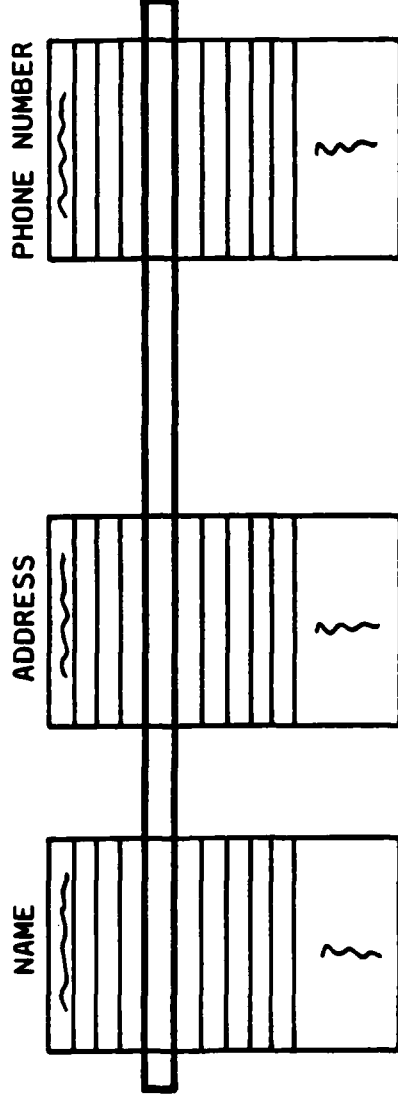
- IS A COMPOSITE OBJECT CONSISTING OF
NAMED COMPONENTS
WHICH MAY BE OF DIFFERENT TYPES
- SPECIFIES A LOGICAL RELATIONSHIP,
NOT STORAGE LAYOUT

INSTRUCTOR NOTES

THE POINT IS THAT RECORDS ALLOW GROUPING TOGETHER OF LOGICALLY RELATED DATA ITEMS.

ARRAY STRUCTURE

- ARRAY STRUCTURE IS LIMITING AS EACH COMPONENT MUST BE OF THE SAME TYPE.
- ARRAY STRUCTURE FOR NAME, ADDRESS, PHONE NUMBER...



- ARRAY VIEW DISALLOWS GROUPING OF ONE INDIVIDUAL'S DATA.
- RECORD VIEW ALLOWS GROUPING OF ONE INDIVIDUAL'S DATA.

INSTRUCTOR NOTES

AN EXAMPLE BEFORE THE RULES ARE SPECIFIED.

POINT OUT THE RESERVED WORDS. EMPHASIZE THAT THE TYPE DECLARATION DEFINES A SET OF VALUES AND A SET OF OPERATIONS.

POINT OUT THE CONSTRAINT SUPPLIED TO THE COMPONENT BELONGING TO THE UNCONSTRAINED ARRAY TYPE STRING.

RECORD TYPE DECLARATIONS

EXAMPLE:

```
type Graphics_Op_Type is (Move, Line, Polygon);

type Instruction_Type is
  record
    Op : Graphics_Op_Type;
    X_Coord : Integer range 0 .. 319;
    Y_Coord : Integer range 0 .. 239;
  end record;

Maximum_Length : constant := 80;

type Buffer_Type is
  record
    Current_Length : Natural;
    Contents : String (1..Maximum_Length);
  end record;
```


RECORD OBJECT DECLARATION

CONTEXT:

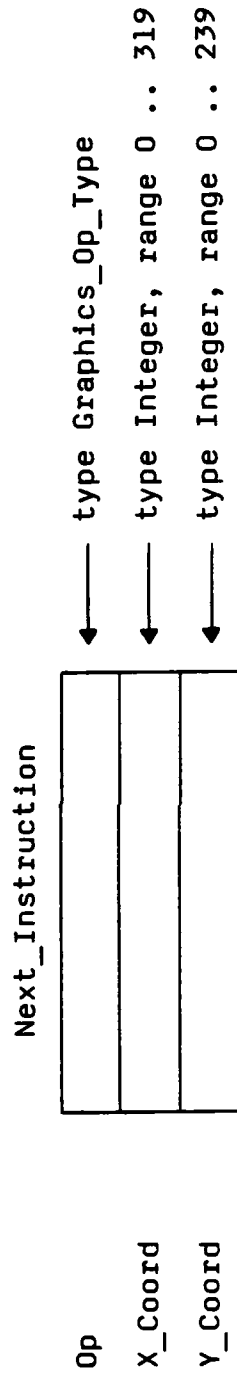
```
type Graphics_Op_Type is (Move, Line, Polygon);

type Instruction_Type is
  record
    Op : Graphics_Op_Type;
    X_Coord : Integer range 0 .. 319;
    Y_Coord : Integer range 0 .. 239;
  end record;
```

EXAMPLE:

```
Next_Instruction : Instruction_Type;
```

CREATES:



INSTRUCTOR NOTES

THIS IS HOW TO GET A COMPONENT (PART) OF THE RECORD.

- POINT OUT THAT YOU CAN ACCESS A COMPONENT OF A COMPONENT.
- POINT OUT INITIALIZATION OF THE OBJECT Line_Buffer.

COMPONENT ACCESS

- USE DOT NOTATION TO ACCESS INDIVIDUAL COMPONENTS.

CONTEXT:

```
Maximum_Length : constant := 30;

type Buffer_Type is
  record
    Current_Length : Natural;
    Contents       : String (1..Maximum_Length) := (others => ' ');
  end record;

Line_Buffer : Buffer_Type := (Contents => "This is a line of text.",
                             Current_Length => 23);
```

EXAMPLE:

```
Line_Buffer.Current_Length -- 23
Line_Buffer.Contents       -- "This is a line of text."
Line_Buffer.Contents (1..4) -- "This"
```

INSTRUCTOR NOTES

- MENTION THAT THE COMPONENT MAY BE ASSIGNED A DEFAULT INITIAL VALUE. DEFAULT INITIAL VALUES FOR COMPONENTS APPLY WHEN AN OBJECT DECLARATION DOES NOT SPECIFY A DEFAULT VALUE FOR THE ENTIRE RECORD OBJECT.
- MENTION THAT THIS IS A SIMPLE SYNTAX OF A RECORD.
- IF THE COMPONENT TYPE IS AN ARRAY TYPE, THE COMPONENT MUST BE CONSTRAINED. EITHER THE ARRAY TYPE OR SUBTYPE MUST BE CONSTRAINED OR THE COMPONENT TYPE NAME MUST BE ACCOMPANIED BY AN INDEX CONSTRAINT. I.E. IF String IS A COMPONENT TYPE IT MUST HAVE A BOUND SPECIFIED.

RECORD TYPE DECLARATION

SYNTAX:

```
type Type_Name is
record
    Component_Name : Component_Type_or_Subtype_Name [:= Default_Initial_Value];
    { Component_Name : Component_Type_or_Subtype_Name [:= Default_Initial_Value];}
end record;
```

INSTRUCTOR NOTES

- COMPONENTS OF TYPE STRING MUST HAVE A RANGE SPECIFIED.
- ARRAY COMPONENTS MUST BE OF AN ARRAY TYPE WHICH HAS BEEN PREVIOUSLY DEFINED.
COMPONENTS OF ANONYMOUS ARRAY TYPES, E.G.

type Rec is

record

Comp: array (1 .. 10) of Boolean;

end record;

ARE ILLEGAL. THEY MAY SEE THIS IN OTHER REFERENCES. IT IS NO LONGER LEGAL. Ada
80 ALLOWED IT.

COMPONENTS

- MUST BE NAMED

- MAY BE OF ANY TYPE

- IF COMPONENT IS AN ARRAY IT MUST BE
FULLY CONSTRAINED.

EITHER (1) THE COMPONENT MUST BELONG TO A
CONSTRAINED ARRAY TYPE, OR

(2) THE COMPONENT MUST HAVE AN INDEX CONSTRAINT.

- ANONYMOUS ARRAY TYPE IS NOT A LEGAL COMPONENT.

INSTRUCTOR NOTES

A RECORD COMPONENT MUST BE CONSTRAINED.

COMPONENT DECLARATIONS

-- LEGAL

type OK_Record_Type is

record

 Name_Component : String (1 .. 20); -- constraint supplied

end record;

-- ** ILLEGAL **

CONTEXT:

type Matrix_Type is array (Integer range <>, Integer range <>) of Integer;

(NON) EXAMPLE:

type Illegal_Record_Type is
record

 Dimension : Positive;

 Matrix : Matrix_Type;

end record;

-- not constrained

INSTRUCTOR NOTES

- RECORD TYPES ARE THE ONLY TYPES WHICH ALLOW INITIAL VALUES FOR OBJECT COMPONENTS TO BE SPECIFIED IN THE TYPE DECLARATION.
- POINT OUT THAT NOT ALL THE COMPONENTS HAVE TO HAVE INITIAL VALUES. IN OTHER WORDS

type Rec is

record

C1: Integer;

C2: Boolean := True;

C3: Character;

end record;

IS LEGAL

- IT IS USUALLY BETTER PROGRAMMING PRACTICE TO REFER TO THE BASE TYPE IN A RANGE CONSTRAINT, THUS THE Minutes_Part AND Seconds_Part COMPONENTS ARE TYPES AS Integer range 0 .. 59 RATHER THAN Natural range 0 .. 59 (WHICH IS PERFECTLY LEGAL) THE REASON IS MAINTAINABILITY AND READABILITY: WHAT IF THE SUBTYPE RANGE CHANGES OR DISAPPEARS ENTIRELY.

DEFAULT VALUES

- DEFAULT VALUES MAY BE GIVEN IN THE TYPE DECLARATION FOR ALL OR SOME OF THE

COMPONENTS

type Latitude_Type is
 record

 Degrees_Part : Integer range -180 .. 180 := 0;
 Minutes_Part, Seconds_Part : Integer range 0 .. 59 := 0;
 end record;

- WHEN A RECORD OBJECT IS DECLARED, A COMPONENT AUTOMATICALLY RECEIVES A VALUE IF THE TYPE DECLARATION ASSIGNS A DEFAULT VALUE TO THAT COMPONENT.

FOR EXAMPLE:

 POSITION : Latitude_Type;

CREATES:

 Position

0
0
0

-- Degrees_Part

-- Minutes_Part

-- Seconds_Part

INSTRUCTOR NOTES

YES, YOU MAY OVERRIDE THE DEFAULT.

POINT OUT THAT THE FOLLOWING REPRESENTATION IS ALSO POSSIBLE

```
type Position_Component_Type is (Degrees, Minutes, Seconds);  
type Latitude_Type is array (Position_Component_Type) of Integer;
```

BUT THAT IT IS LESS APPROPRIATE BECAUSE IT DOESN'T PROVIDE RANGE CHECKING FOR COMPONENT VALUES.

INITIAL VALUE

- AS WITH ANY OBJECT DECLARATION, AN INITIAL VALUE MAY BE SPECIFIED

CONTEXT:

```
type Latitude_Type is
record
  Degrees_Part      : Integer range -180 .. 180 := 0;
  Minutes_Part, Seconds_Part : Integer range 0 .. 59 := 0;
end record;
```

EXAMPLE:

```
Position_1 : Latitude_Type := (90, 5, 1); -- has initial value (90, 5, 1)
```

```
Position_2 : Latitude_Type;           -- has initial value (0, 0, 0) by default
```

- THE INITIALIZATION VALUES IN THE OBJECT DECLARATION OVERRIDE THE DEFAULT VALUES OF THE TYPE DECLARATION.

INSTRUCTOR NOTES

THE FORMS FOR WRITING AN AGGREGATE ARE SPECIFIED ON THIS AND THE NEXT FOIL.

NOTE THAT THERE IS NO CHANCE FOR ERROR WHEN EACH COMPONENT IS NAMED.

RECORD AGGREGATES

- AN AGGREGATE IS USED TO SPECIFY VALUES FOR A WHOLE RECORD.

CONTEXT:

```
type Am_or_Pm_Type is (am, pm);
type Clock_Time_Type is
  record
    Hour      : Integer range 1 .. 12;
    Minute    : Integer range 0 .. 59;
    Second    : Integer range 0 .. 59;
    Am_or_Pm  : Am_or_Pm_Type;
  end record;
```

```
It_is : Clock_Time_Type;
```

- POSITIONAL FORM --- LIST IN ORDER

```
if It_Is = (12, 20, 45, am) then
  Run_Batch;
end if;
```
- NAMED FORM --- MAY BE IN ANY ORDER

```
if It_Is = (Am_or_Pm => am, Hour => 12, Minute => 20, Second => 45) then
  Run_Batch;
end if;
```


INSTRUCTOR NOTES

THE TWO NOTATIONS CAN BE MIXED, WITH CERTAIN RESTRICTIONS (AS NOTED ON THE FOIL).

POINT OUT THAT THIS IS NOT THE CASE WITH ARRAY AGGREGATES, WHICH MUST BE EITHER ALL POSITIONAL OR ALL NAMED.

RECORD AGGREGATES

• MIXED POSITIONAL AND NAMED --- POSITIONAL MUST COME FIRST AND IN ORDER

CONTEXT:

```
type Queue_Type is (Large, User, Graphics, Laser_Printer);
type Process_Descriptor_Type is
record
    Queue           : Queue_Type;
    ID_Number       : Integer range 1 .. 1_000_000;
    Process_Name    : String (1 .. 15);
    Owner_Name      : String (1 .. 12);
end record;
```

Process_A : Process_Descriptor_Type;

EXAMPLE:

```
if Process_A = (Laser_Printer, 987654, Owner_Name => "ASMITH",
                 Process_Name => "SOURCE LISTING") then
    .
    .
    .
end if;
```

INSTRUCTOR NOTES

POINT OUT THAT

(6, others => 2) IS WRONG BECAUSE REMAINING COMPONENTS ARE OF DIFFERENT TYPES.

(others => 50, Move) IS WRONG BECAUSE others MUST COME LAST.

(Op => Move, others => 50) IS LEGAL BECAUSE TYPES OF REMAINING COMPONENTS ARE ALL
SUBTYPES OF Float

RECORD AGGREGATES

others IS RARELY USEFUL FOR RECORDS AS COMPONENTS USUALLY ARE OF DIFFERENT

TYPES:

CONTEXT:

type Graphics_Op_Type is (Move, Line, Polygon);

type Instruction_Type is
record

Op : Graphics_Op_Type;

X_Coord : Integer range 0 .. 319;

Y_Coord : Integer range 0 .. 239;

end record;

Next_Instruction : Instruction_Type;

EXAMPLE:

(6, others => 2)

-- ** ILLEGAL

(others => 50, Move)

-- ** ILLEGAL

(Op => Move, others => 50)

-- LEGAL

INSTRUCTOR NOTES

THE AGGREGATE MUST CONTAIN A VALUE FOR EACH AND EVERY COMPONENT.

RECORD AGGREGATES

- AN AGGREGATE MUST BE COMPLETE EVEN IF IT SUPPLIES THE SAME VALUES
AS THE DEFAULT VALUES FOR SOME COMPONENTS.

CONTEXT:

```
type Latitude_Type is
  record
    Degrees_Part : Integer range -180 .. 180 := 0;
    Minutes_Part : Integer range 0 .. 59 := 0;
    Seconds_Part : Integer range 0 .. 59 := 0;
  end record;
```

EXAMPLE:

```
Old_Latitude : Latitude_Type := (30, 0, 0);    -- legal
New_Latitude : Latitude_Type := (30, 0, 0);    -- **ILLEGAL
```

INSTRUCTOR NOTES

A SPECIAL CASE.

THE REASON FOR EMBEDDING A COUNTER IN A RECORD IS TO GUARANTEE THAT ANY Counter_Type VARIABLE DECLARED ALWAYS HAS A DEFAULT INITIAL VALUE, IN THIS CASE, 0. THIS IS THE CLOSEST MECHANISM TO AUTOMATIC INITIALIZATION IN Ada.

RECORD AGGREGATE

- A ONE-COMPONENT AGGREGATE MUST BE IN NAMED NOTATION:

CONTEXT:

```
type Counter_Type is
  record
    Value : Integer := 0;
  end record;
```

EXAMPLES:

```
Counter : Counter_Type := (1);
Counter : Counter_Type := (Value => 10);
Counter : Counter_Type ;
-- **ILLEGAL
-- legal
-- legal; Counter.Value has default
-- initial value of 0
```


INSTRUCTOR NOTES

IN CASE SOMEONE WONDERS HOW (3, 2, 1934) COULD BE FEBRUARY 3, TELL HIM/HER THAT'S HOW
MOST OF THE WORLD WOULD READ IT.

NAMED NOTATION SHOULD NORMALLY BE USED

- EASIER TO READ
- IT'S TOO EASY TO WRITE THINGS IN THE WRONG ORDER

CONTEXT:

```
type Date_Type is
  record
    Month_Part : Integer range 1 .. 12;
    Day_Part   : Integer range 1 .. 31;
    Year_Part  : Integer range 1900 .. 2100;
  end record;
Due_Date : Date_Type;
```

EXAMPLE:

```
Due_Date := (3, 2, 1934)      -- FEBRUARY 3, 1934
                        -- OR MARCH 2, 1934 ??
```

INSTRUCTOR NOTES

SOLUTION:

```
type Longitude_Type is
record
  Degrees_Part : Integer range -180 .. 180 := 0;
  Minutes_Part, Seconds_Part : Integer range 0 ..59 := 0;
end record;

type Global_Position_Type is
record
  Latitude_Part : Latitude_Type;
  Longitude_Part : Longitude_Type;
end record

POSITION: Global_Position_Type := ((90, 30, 5), (-10, 5, 5));
      (Other variations are possible)
```

BONUS POINT ANSWER - ??

EXERCISE

- GIVEN THE FOLLOWING TYPE DEFINITION:

```
type Latitude_Type is
record
    Degrees_Part : Integer range -180 .. 180 := 0;
    Minutes_Part, Seconds_Part : Integer range 0 ..59 := 0;
end record;
```

DEFINE A Longitude Type (SIMILAR TO THE ABOVE) AND A Global_Position_Type, WHERE A GLOBAL POSITION CONSISTS OF A LATITUDE AND A LONGITUDE.

- PROVIDE AN AGGREGATE THAT WOULD ASSIGN THE FOLLOWING VALUES TO AN OBJECT OF TYPE Global_Position_Type;

```
Latitude : 90 degrees, 30 minutes, 5 seconds
Longitude : -10 degrees, 5 minutes, 5 seconds
```

(BONUS POINT TO ANYONE WHO CAN GUESS WHERE ON THE GLOBE THAT MIGHT BE.)

INSTRUCTOR NOTES

THESE OPERATORS MAY BE USED ON THE RECORD ITSELF OR ON COMPONENTS OF THE RECORD.

NOTE THAT WITHOUT THE QUADRANT COMPONENT, TYPE `Vertex_Type` MIGHT BE IMPLEMENTED AS AN ARRAY TYPE, SINCE BOTH COMPONENTS WOULD THEN BE OF TYPE `Float`.

RECORD OPERATIONS - EQUALITY/INEQUALITY

EQUALITY/INEQUALITY

- WHOLE RECORDS
- COMPONENTS OF RECORDS

CONTEXT:

```
type Vertex_Type is
record
    X_Part, Y_Part : Float;
    Quadrant : Integer range 1 .. 4;
end record;

Vertex_1, Vertex_2 : Vertex_Type;
```

EXAMPLE:

```
if Vertex_1 = Vertex_2 then
    ...
end if;

or

if Vertex_1.X_Part > 12.0 then
    ...
end if;
```

INSTRUCTOR NOTES

YOU CAN ASSIGN TO WHOLE RECORDS OR INDIVIDUAL RECORD COMPONENTS.

RECORD OPERATIONS - ASSIGNMENT

- WHOLE RECORD ASSIGNMENT
- ASSIGNMENT TO INDIVIDUAL COMPONENTS

CONTEXT:

```
type Vertex_Type is
  record
    X_Part, Y_Part : Float;
    Quadrant : Integer range 1 .. 4;
  end record;
Vertex_1 : Vertex_Type;
```

EXAMPLES:

```
Vertex_1 := (1.0, 3.3, 1);      -- WHOLE RECORD
Vertex_1.Y_Part := 10.5;        -- INDIVIDUAL COMPONENT
```


INSTRUCTOR NOTES

THESE OPERATIONS MAY NOT BE PERFORMED ON AN ENTIRE RECORD.

RECORD OPERATIONS - RELATIONAL, LOGICAL, AND ARITHMETIC

- PERFORMED ON INDIVIDUAL COMPONENTS AND NOT ON THE WHOLE RECORD (E.G. YOU CANNOT ADD Buffer_Type OBJECTS)
- THE OPERATION MUST BE COMPATIBLE WITH THE TYPE OF THE COMPONENT

INSTRUCTOR NOTES

WHAT ABOUT ATTRIBUTES?

ATTRIBUTES

- ATTRIBUTES MAY BE APPLIED TO SELECTED COMPONENTS
- ATTRIBUTES MUST BE COMPATIBLE WITH SELECTED COMPONENTS

INSTRUCTOR NOTES

0 RESTATE THAT AN ARRAY OBJECT WITH A SUBSCRIPT IS ESSENTIALLY THE COMPONENT VARIABLE.

TRACE THROUGH THE STATEMENTS BY FILLING IN THE PICTURE ON THE BOARD.

ARRAYS OF RECORDS

CONTEXT:

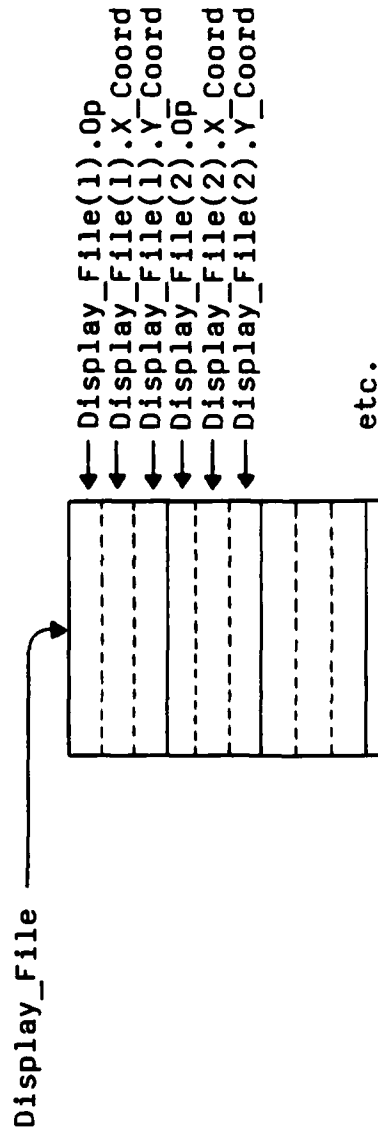
```

type Graphics_Op_Type is (Move, Line, Polygon);

type Instruction_Type is
  record
    Op : Graphics_Op_Type;
    X_Coord : Integer range 0 .. 319;
    Y_Coord : Integer range 0 .. 239;
  end record;

type Display_File_Type is array (1 .. 500) of Instruction_Type;
Display_File : Display_File_Type;
  
```

STRUCTURE:



INSTRUCTOR NOTES

STEP THROUGH EACH EXAMPLE.

ARRAYS OF RECORDS

```
Display_File (1) := (Op      => Line,  
                     X_Coord => 10, Y_Coord => 50);
```

```
Display_File (2) := Display_File(1);
```

```
Display_File (3 .. 4) := Display_File (1 .. 2);
```

```
case Display_File(2).Op is  
  when Move =>  
    ...  
  when Line =>  
    ...  
  when others =>  
    ...  
end case;
```


INSTRUCTOR NOTES

ASK THE CLASS HOW THEY WOULD DO I/O FOR RECORDS.

THEY WOULD HAVE TO DO IT BY COMPONENTS.

THEY WILL SEE IN SECTION 13 THAT THERE IS SOMETHING CALLED Direct_IO AND Sequential_IO.
THESE WOULD BE USED TO DO I/O FOR RECORDS.

ASSIGN EXERCISES 22 THROUGH 24 OF THE EXERCISE BOOKLET FOR LAB WORK. STRESS EXERCISE 24.

ASSIGN CHAPTER 9 OF THE PRIMER.

I/O FOR RECORD TYPES

- NONE IS PREDEFINED
- MUST DO COMPONENT BY COMPONENT
FOR Text_10;

INSTRUCTOR NOTES

VG 728.2

9-251

123 575 584 593 602 611 620 629 638 647 656 665 674 683 692 701 710 719 728 737 746 755 764 773 782 791 800 809 818 827 836 845 854 863 872 881 890 899 908 917 926 935 944 953 962 971 980 989 998

RECORD TYPES WITH DISCRIMINANTS

VG 728.2

9-25

INSTRUCTOR NOTES

- THIS SLIDE AND THE NEXT THREE INTRODUCE THE NEED FOR DISCRIMINANTS
- THIS SLIDE DESCRIBES A VARYING LENGTH STRING TYPE WITH A FIXED MINIMUM LENGTH.
THE NEXT SLIDE INTRODUCES THE NEED FOR THE MAXIMUM LENGTH TO BE A DISCRIMINANT.
- DO NOT GO INTO DETAIL ABOUT VARYING LENGTH STRINGS.
- BULLET #2
 - TELL THE CLASS THAT THIS COULD BE HANDLED BY DEFINING String40_Type, String256_Type, String500_Type, ETC. EACH OF THESE TYPES IS JUST A COPY OF String80 WITH BOTH OCCURRENCES OF 80 REPLACED BY THE REQUIRED MAXIMUM LENGTH.
 - NOTE THAT THIS IS WASTED EFFORT
 - USES SEVERAL TYPE DEFINITIONS WHERE IDEALLY, ONE WOULD BE BETTER
 - USING THE LARGEST OF THE MAXIMUM LENGTHS, SAY 500, WOULD BE WASTEFUL WHEN A MAXIMUM OF LENGTH 16 IS NEEDED
- BULLET #3
 - THE PLOT THICKENS.

WHY DO WE NEED DISCRIMINANTS?

- SUPPOSE WE NEED A VARYING LENGTH STRING RATHER THAN A FIXED STRING
 - MUST BE CAPABLE OF HOLDING ALL STRINGS UP TO SOME MAXIMUM LENGTH
 - SUPPOSE THE MAXIMUM LENGTH IS 80

```
type String80_Type is
record
    Contents      : String (1 .. 80);
    Current_Length : Natural;
end record;
```

- GIVEN THE DECLARATION
String80 : String80_Type;
THE CURRENT STRING VALUE IS
String80.Contents (1 .. String80.Current_Length)
TO ASSIGN THE STRING "Ada" OF LENGTH 3 TO String80, WE NEED THE TWO
ASSIGNMENTS

```
String80.Current_Length := 3;
String80.Contents (1 .. 3) := "Ada";
```

- SUPPOSE WE ALSO NEED VARYING STRINGS WITH MAXIMUM LENGTHS 40, 256, 500, 16,
ETC.
- WHAT IF THE MAXIMUM LENGTH IS NOT KNOWN UNTIL PROGRAM EXECUTION?

INSTRUCTOR NOTES

- DETAILS PROVIDED IN THE REMAINDER OF THIS SECTION.

WHY DO WE NEED DISCRIMINANTS? - Continued

- WE SHOULD LIKE TO DEFINE ONE VARYING STRING TYPE WITH THE MAXIMUM LENGTH LEFT BLANK. THE BLANK WOULD BE FILLED IN WHEN A VARYING STRING OBJECT IS DECLARED.

```
type Varying_String_Type ( MAXIMUM LENGTH ) is
  record
    Contents      : String (1 .. MAXIMUM LENGTH );
    Current_Length : Natural;
  end record;
```

```
String80 : Varying_String_Type (80);
String16 : Varying_String_Type (16);

procedure Example (Name_Length:Natural) is
  Temp : Varying_String_Type (Name_Length);
begin
  ...
end Example;
```

- THIS CAN BE ACCOMPLISHED WITH DISCRIMINANTS.

INSTRUCTOR NOTES

- THIS SLIDE ILLUSTRATES THE NEED FOR VARIANTS
- BULLET #3
 - LET THE CLASS TAKE A CRACK AT IT FIRST
 - PARTIAL LIST OF PROBLEMS
 - NOT CLEAR THAT Customer_Number_Part AND Order_Size_Part ARE VALID ONLY WHEN Variety_Part = Order_Transaction. SIMILARLY FOR Shipment_Size_Part
 - POSSIBLE TO ACCIDENTALLY USE Order_Size_Part WHEN Variety_Part = Items_Received_Transaction
 - POSSIBLE TO LEAVE OBJECT IN AN INCONSISTENT STATE BY CHANGING Variety_Part WITHOUT CHANGING APPROPRIATE COMPONENTS
 - STORAGE WASTED: Shipment_Size_Part NOT NEEDED WHEN Variety_Part = Order_Transaction. (IN CASE ANYONE SUGGESTS COMBINING Order_Size_Part AND Shipment_Size_Part INTO ONE COMPONENT THIS WOULD SAVE SPACE BUT MAKES THE USE OF THE RECORD TYPE MORE DIFFICULT TO UNDERSTAND)
 - VARIANTS TO THE RESCUE ON THE NEXT SLIDE

ANOTHER EXAMPLE

- IN AN INVENTORY SYSTEM WE MIGHT HAVE ORDER TRANSACTIONS AND ITEMS RECEIVED TRANSACTIONS.
 - FOR EACH TRANSACTION WE NEED
 - TIME OF THE TRANSACTION
 - UNIQUE TRANSACTION NUMBER
 - FOR ORDER TRANSACTIONS WE NEED
 - CUSTOMER NUMBER
 - SIZE OF THE ORDER
 - FOR ITEMS RECEIVED TRANSACTIONS WE NEED
 - SIZE OF THE SHIPMENT
- THIS COULD BE MODELLED AS A RECORD TYPE


```

type Transaction_Variety_Type is (Order_Transaction, Items_Received_Transaction);

type Transaction_Type is
  record
    Variety_Part : Transaction_Variety_Type;
    Time_Part   : Calendar.Time;
    Number_Part : Positive
    Customer_Number_Part : Customer_Number_Type;
    Order_Size_Part      : Item_Count_Type;
    Shipment_Size_Part   : Item_Count_Type;
  end record;
      
```

THIS IS CORRECT Ada AND WORKS.
- WHAT'S "WRONG" WITH THIS APPROACH?

INSTRUCTOR NOTES

- DETAILS PROVIDED LATER IN THIS SECTION.
- AVOID THE TERM VARIANTS.

EXAMPLE TWO (Continued)

- WE WOULD LIKE A WAY TO INDICATE THAT THE EXISTENCE OF CERTAIN COMPONENTS DEPENDS ON THE TRANSACTION

- FOR Order_Transactions WE WANT Transaction_Type TO BE DEFINED AS

```

type Transaction_Type is
record
    Variety_Part      : Transaction_Variety_Type; -- Always Order_Transaction
    Time_Part         : Calendar.Time;
    Number_Part       : Positive;
    Customer_Number_Part : Customer_Number_Type;
    Order_Size_Part   : Item_Count_Type;
end record;

```

- FOR Items_Received_Transaction WE WANT Transaction_Type TO BE DEFINED AS

```

type Transaction_Type is
record
    Variety_Part      : Transaction_Variety_Type; -- Always Items_Received_Transaction
    Time_Part         : Calendar.Time;
    Number_Part       : Positive;
    Shipment_Size_Part : Item_Count_Type;
end record;

```

- THIS CAN BE ACCOMPLISHED THROUGH THE USE OF DISCRIMINANTS

INSTRUCTOR NOTES

- BULLET #3

- GIVE THE CLASS SOME EXAMPLES THEY WILL RUN INTO LATER IN THIS SECTION
 - AS AN EXAMPLE OF A RECORD TYPE FOR WHICH OBJECTS IN THE TYPE CONTAIN ARRAYS WITH DIFFERENT BOUNDS, WE WILL HAVE AN EXAMPLE OF A VARYING LENGTH STRING, WHERE THE DISCRIMINANT SPECIFIES THE MAXIMUM SIZE STRING THE OBJECT SHOULD HOLD.
 - AS AN EXAMPLE OF A RECORD TYPE FOR WHICH OBJECTS IN THE TYPE MAY HAVE DIFFERENT COMPONENTS, WE HAVE A SENSOR READING TYPE. OBJECTS IN THIS TYPE WILL CONTAIN A FLAG INDICATING WHETHER OR NOT A VALID SENSOR READING WAS OBTAINED. IF THE FLAG INDICATES A VALID READING THEN THE OBJECT WILL ALSO CONTAIN THE READING; OTHERWISE THE OBJECT ONLY CONTAINS THE FLAG.

- BULLET #4

- EMPHASIZE THE DISCRIMINANT AS A RECORD COMPONENT
- MENTION THAT THERE IS A RESTRICTION ON CHANGING THE VALUE OF A DISCRIMINANT, AS WILL BE SEEN LATER.

DISCRIMINANTS

- RECORD TYPES MAY BE DECLARED WITH DISCRIMINANTS.
- DISCRIMINANTS ACT AS "PARAMETERS" TO RECORDS IN THE TYPE.
- DIFFERENT OBJECTS IN THE TYPE MAY HAVE DIFFERENT COMPONENTS, OR MAY CONTAIN ARRAYS WITH DIFFERENT BOUNDS, DEPENDING ON THE "PARAMETERS" SPECIFIED FOR EACH OBJECT.
- BESIDES ACTING AS A "PARAMETER," A DISCRIMINANT IS ALSO A COMPONENT OF THE RECORD.

INSTRUCTOR NOTES

- POINT OUT THE TWO DISTINCT OCCURRENCES OF **DISCRIMINANT SPECIFICATION** IN THE EXAMPLE.
- DO NOT DISCUSS THE USE OF **DEFAULT INITIAL VALUE** NOW.

DECLARING RECORD TYPES WITH DISCRIMINANTS

- TYPE DECLARATION

```

type TYPE NAME ( DISCRIMINANT SPECIFICATION { DISCRIMINANT SPECIFICATION } ) is
    record
        ... -- COMPONENT DECLARATIONS
    end record;

```

- FORM OF A **DISCRIMINANT SPECIFICATION** :

```

IDENTIFIER { , IDENTIFIER } : DISCRETE TYPE NAME [ := DEFAULT INITIAL VALUE ]

```

- EXAMPLE

```

type Message_Type (Maximum_Length, Initial_Length : Integer := 0;
                    End_Character : Character := ',') is
    record
        ...
    end record;

```


INSTRUCTOR NOTES

- BULLET #2

- THE VALUES IN THE CONSTRAINT CAN BE EXPRESSIONS INVOLVING VARIABLES AND FUNCTION CALLS. FOR NOW MOSTLY CONTEND WITH EXPRESSIONS NOT INVOLVING VARIABLES OR FUNCTION CALLS. AT END OF SECTION WE WILL RETURN TO THIS.

DISCRIMINANT CONSTRAINTS

- AN OBJECT DECLARATION FOR A TYPE WITH DISCRIMINANTS MAY CONTAIN A DISCRIMINANT CONSTRAINT.
- THE DISCRIMINANT CONSTRAINT SPECIFIES THE VALUES TO BE USED FOR THAT OBJECT'S "PARAMETERS".
- DISCRIMINANT CONSTRAINTS MAY BE NAMED, POSITIONAL, OR MIXED:

NAMED:
 Message : Message_Type (Maximum_Length => 10, End_Character => '/',
 Initial_Length => 5);

POSITIONAL:
 Message : Message_Type (10, 5, '/');

MIXED:
 Message : Message_Type (10, Initial_Length => 5, End_Character => '/');

INSTRUCTOR NOTES

- SEVERAL EXAMPLES OF SUCH RECORD TYPES WILL APPEAR IN THIS SECTION.
- VARIANTS ALLOW US TO REFLECT THE LOGICAL STRUCTURE OF OUR DATA. THIS IS ILLUSTRATED WITH THE EXAMPLE ON THE NEXT PAGE.

VARIANTS

- A RECORD TYPE WITH DISCRIMINANTS CAN HAVE DIFFERENT FORMS CALLED VARIANTS.
- WITHIN EACH VARIANT THERE ARE DIFFERENT RECORD COMPONENTS WITH DIFFERENT NAMES AND POSSIBLY DIFFERENT TYPES.
- THE VARIANT THAT AN OBJECT BELONGS TO IS DETERMINED BY THE VALUE OF ITS DISCRIMINANTS.

INSTRUCTOR NOTES

- THESE TYPE DEFINITIONS MIGHT APPEAR AS PART OF AN INVENTORY CONTROL SYSTEM.
- Transaction_Type REFLECTS TWO LOGICAL TRANSACTIONS: Order_Transaction AND Items_Received_Transaction. BOTH TRANSACTIONS REQUIRE TIME AND PART NUMBER. AN Order_Transaction MUST RECORD THE CUSTOMER PLACING THE ORDER AND THE SIZE OF THE ORDER, WHILE AN Items_Received_Transaction MUST RECORD THE SIZE OF THE SHIPMENT RECEIVED. USING Transaction_Variety_Type AS A DISCRIMINANT ALLOWS Transaction_Type TO REFLECT BOTH LOGICAL VIEWS OF THE DATA.

DECLARATIONS OF TYPES WITH VARIANTS - EXAMPLE

- DECLARATIONS:

```

type Transaction_Variety_Type is (Order_Transaction, Items_Received_Transaction);

type Transaction_Type (Variety : Transaction_Variety_Type) is
record
    Transaction_Time_Part : Calendar.Time;      -- Calendar is predefined package
    Transaction_Number_Part : Positive;
    case Variety is
        when Order_Transaction =>
            Customer_Number_Part : Customer_Number_Type;
            Order_Size_Part : Item_Count_Type;
        when Items_Received_Transaction =>
            Shipment_Size_Part : Item_Count_Type;
    end case;
end record;

```

- POSSIBLE FORMS OF TRANSACTION TYPE OBJECTS

Variety
Transaction_Time_Part
Transaction_Number_Part
Customer_Number_Part
Order_Size_Part

Variety = Order_Transaction

Variety
Transaction_Time_Part
Transaction_Number_Part
Shipment_Size_Part

Variety = Items_Received_Transaction

INSTRUCTOR NOTES

- TAKE SOME TIME WITH THIS SLIDE. MAKE SURE THE CLASS UNDERSTANDS THE GENERAL FORM.

DECLARATION OF TYPES WITH VARIANTS - GENERAL FORM

type **TYPE NAME** (**DISCRIMINANT SPECIFICATION** { ; **DISCRIMINANT SPECIFICATION** }) is
record

COMPONENT LIST
end record;

- THREE POSSIBLE FORMS OF **COMPONENT LIST** :

ORDINARY COMPONENT DECLARATION	VARIANT PART	null;
{ ORDINARY COMPONENT DECLARATION		
[VARIANT PART]		

- FORM OF A **VARIANT PART** :

case **DISCRIMINANT NAME** is
when **CHOICE LIST** =>
COMPONENT LIST
{ when **CHOICE LIST** =>
COMPONENT LIST }
end case;

- FORM OF A **CHOICE LIST** :
SAME AS FOR A CASE STATEMENT:
CHOICE { | **CHOICE** }
WHERE EACH **CHOICE** IS AN EXPRESSION OR A RANGE

INSTRUCTOR NOTES

- EMPHASIZE HOW THIS TYPE REFLECTS THE LOGICAL STRUCTURE OF THE DATA.

ANOTHER TYPE WITH VARIANTS

- A TYPE TO REPRESENT DATA PROVIDED BY A SENSOR.
- THE SENSOR HAS A BUILT-IN SELF-TEST MECHANISM, AND SETS A "VALIDITY BIT" WHEN THE SELF-TEST SUCCEEDS. IT ALSO PROVIDES DATA OF TYPE Integer THAT IS MEANINGFUL ONLY WHEN THE VALIDITY BIT IS ON.
- AN ITEM OF DATA IS EITHER VALID OR INVALID, AND ONLY VALID ITEMS HAVE INTEGER VALUES ASSOCIATED WITH THEM.

```
type Sensor_Reading_Type (Valid : Boolean := False) is
  record
    case Valid is
      when True =>
        Reading_Part : Integer;
      when False =>
        null;
    end case;
  end record;
```

INSTRUCTOR NOTES

- THE NEXT THREE SLIDES GIVES EXAMPLES OF THIS.

CONTROLLING THE LENGTH OF ARRAYS INSIDE RECORDS

- A RECORD COMPONENT DECLARATION MAY TAKE THE FORM
IDENTIFIER : UNCONSTRAINED ARRAY TYPE NAME INDEX CONSTRAINT ;
- DISCRIMINANTS MAY BE USED WITHIN THE INDEX CONSTRAINT TO SPECIFY BOUNDS OF THE ARRAY.
- DIFFERENT RECORDS IN THE RECORD TYPE MAY HAVE ARRAY-VALUED COMPONENTS OF DIFFERENT SIZES. THE SIZE IS DETERMINED BY THE VALUE OF THE DISCRIMINANTS.

INSTRUCTOR NOTES

- EMPHASIZE THE ABSTRACT USE OF `Varying_String_Type` BEFORE DISCUSSING THE TYPE DECLARATION ITSELF.
 - MENTION THAT `Word` IS INITIALLY SET TO THE `Empty_Varying_String` IN THE OBJECT DECLARATION, WHERE THE MAXIMUM SIZE IS SET TO 80.
 - MENTION THAT EACH CALL TO CATENATE INCREASES THE LENGTH OF THE STRING OBJECT WORD BY ONE.
- NOW DISCUSS THE TYPE DECLARATION
 - CONSIDER OUTLINING THE CATENATE PROCEDURE

```
procedure Catenate (Word : in out Varying_String_Type; C : in Character) is
begin
    if Word.Current_Length < Word.Maximum_Length then
        Word.Current_Length := Word.Current_Length + 1;
        Word.Contents (Word.Current_Length) := C;
    else
        -- error processing
        end if;
    end Catenate;
```
- MENTION THAT OPERATIONS ON `Varying_String_Type` OBJECTS WOULD BE PERFORMED THROUGH SUBPROGRAMS.

EXAMPLE OF A DISCRIMINANT CONTROLLING ARRAY BOUNDS

- A TYPE FOR REPRESENTING CHARACTER STRINGS OF MANY POSSIBLE LENGTHS.
- FROM AN ABSTRACT POINT OF VIEW, THE CURRENT LENGTH OF A Varying_String_Type OBJECT MAY CHANGE DURING PROGRAM EXECUTION, UP TO A SPECIFIED MAXIMUM.

```
Word : Varying_String_Type (Maximum_Length => 80) := Empty_Varying_String;  
C : Character;  
...  
loop  
  Get (C);  
  exit when C = ' ';  
  Catenate (Word, C);  
end loop;
```

- TYPE DECLARATION

```
type Varying_String_Type (Maximum_Length : Positive) is  
  record  
    Current_Length : Natural;  
    Contents       : String (1 .. Maximum_Length);  
  end record;
```


ANOTHER EXAMPLE

- TYPE DECLARATIONS

```
type Matrix_Type is
    array (Positive range<>, Positive range<>) of Float;

type Square_Type (Size : Positive) is
    record
        Square_Part : Matrix_Type (1 .. Size, 1 .. Size);
    end record;
```

- THE Square_Part COMPONENT OF A Square_Type RECORD IS ALWAYS A MATRIX WITH
THE SAME NUMBER OF ROWS AND COLUMNS

- OBJECT DECLARATIONS

```
A : Square_Type (5);  -- A.Square_Part is a 5x5 Matrix,
B : Square_Type (10); -- B.Square_Part is a 10x10 Matrix.
```


INSTRUCTOR NOTES

NESTING RECORD TYPES WITH DISCRIMINANTS

- A RECORD TYPE WITH DISCRIMINANTS MAY BE USED AS A COMPONENT OF SOME LARGER RECORD TYPE.
- A DISCRIMINANT OF THE LARGER RECORD TYPE MAY BE USED IN A DISCRIMINANT CONSTRAINT FOR THE COMPONENT.
- EXAMPLE:

```
type Message_Type (Message_Size_Limit : Positive) is
  record
    Origin, Destination : Transmission_Line_Type;
    Text                 : Varying_String_Type (Maximum_Length => Message_Size_Limit);
  end record;
```

INSTRUCTOR NOTES

- RECALL THAT WE SAW THIS EXAMPLE EARLIER, WITHOUT DISCRIMINANTS; THIS IS A MORE LOGICAL REPRESENTATION.
- Requested_Length AND Maximum_Buffer_Length_Needed CAN BE CONSTANTS, VARIABLES OR FUNCTION CALLS.
- OTHER DISCUSSIONS OF EXPRESSION EVALUATION AND DISCRIMINANTS OCCUR AFTER SUBTYPES OF RECORD TYPES ARE INTRODUCED.

USING DISCRIMINANTS AS DEFAULT INITIAL VALUES OF COMPONENTS

- EXAMPLE

```
type Buffer_Type (Initial_Length, Maximum_Length : Natural) is
  record
    Current_Length : Natural := Initial_Length;
    Contents       : String (1 .. Maximum_Length);
  end record;
```

- EXPRESSIONS IN DISCRIMINANT CONSTRAINTS ARE EVALUATED WHEN THE DECLARATION CONTAINING THE CONSTRAINT IS ELABORATED

- EXAMPLE

```
Buffer : Buffer_Type
  (Initial_Length => Requested_Length,
   Maximum_Length => Maximum_Buffer_Length_Needed);
```

WHEN THIS OBJECT DECLARATION IS ELABORATED, Requested_Length AND Maximum_Buffer_Length_Needed ARE EVALUATED. THESE VALUES ARE USED FOR Initial_Length AND Maximum_Length.

INSTRUCTOR NOTES

- THE FOURTH USE IS DISCUSSED ON THE NEXT SLIDE.
- RULES FOR EVALUATION OF THE DISCRIMINANT VALUES ARE DISCUSSED LATER.



ALLOWABLE USES OF DISCRIMINANTS IN RECORD TYPE DECLARATIONS

- THERE ARE FOUR ALLOWABLE USES:

- FOLLOWING THE WORD case IN A VARIANT PART.
- IN AN INDEX CONSTRAINT OF A RECORD COMPONENT.
- IN A DISCRIMINANT CONSTRAINT OF A RECORD COMPONENT.
- AS THE DEFAULT INITIAL VALUE OF A COMPONENT.

- IN EACH CASE, THE DISCRIMINANT MUST APPEAR AS A FULL EXPRESSION, NOT AS PART OF A LARGER EXPRESSION.

INSTRUCTOR NOTES

- WE WILL SEE EXAMPLES OF SUCH RECORD TYPES IN A FEW SLIDES.

DEFAULT INITIAL VALUES FOR DISCRIMINANTS

- A RECORD TYPE DECLARATION MUST PROVIDE DEFAULT INITIAL VALUES FOR ALL OF ITS DISCRIMINANTS OR FOR NONE OF THEM.
- IF DEFAULT INITIAL VALUES ARE PROVIDED, AN OBJECT IN THE TYPE MAY BE DECLARED WITHOUT A DISCRIMINANT CONSTRAINT. IF DEFAULT INITIAL VALUES ARE NOT PROVIDED, EACH OBJECT IN THE TYPE MUST BE DECLARED WITH A DISCRIMINANT CONSTRAINT.
- AN OBJECT DECLARED WITH A DISCRIMINANT CONSTRAINT IS CONSTRAINED. ITS DISCRIMINANTS MUST KEEP THE SAME VALUES FOR THE LIFE OF THE OBJECT.
- AN OBJECT DECLARED WITHOUT A DISCRIMINANT CONSTRAINT IS UNCONSTRAINED. IT MAY TAKE ON VALUES WITH DIFFERENT DISCRIMINANT COMPONENTS DURING ITS LIFETIME.

INSTRUCTOR NOTES

- WHEN A RECORD AGGREGATE IS USED, THE VALUE FOR THE DISCRIMINANT MAY NOT BE A VARIABLE OR FUNCTION CALL. THIS WILL BE DISCUSSED LATER, SO JUST MENTION IT FOR NOW.



CHANGING THE DISCRIMINANTS OF UNCONSTRAINED OBJECTS

- IT IS ILLEGAL TO CHANGE THE DISCRIMINANT COMPONENT OF A RECORD BY ITSELF.
REASON: THIS DISCRIMINANT DETERMINES WHAT OTHER DATA THE RECORD MAY CONTAIN. CHANGING ONLY THE DISCRIMINANT COULD LEAD TO AN INCONSISTENT RECORD.
- IT IS POSSIBLE TO ASSIGN AN ENTIRE RECORD VALUE WITH DIFFERENT DISCRIMINANT COMPONENTS TO AN UNCONSTRAINED RECORD OBJECT.
- SPECIFICALLY, DISCRIMINANT COMPONENTS MAY NEVER APPEAR
 - ON THE LEFTHAND SIDE OF :=
 - AS AN ACTUAL PARAMETER OF MODE out OR in out.

INSTRUCTOR NOTES

- SINCE Sensor_Reading IS UNCONSTRAINED, EITHER CONSTRAINED OBJECT MAY BE ASSIGNED TO IT.
- THE FIRST ILLEGAL ASSIGNMENT ATTEMPTS TO CHANGE THE DISCRIMINANT VALUE OF A CONSTRAINED OBJECT.
- THE SECOND ILLEGAL ASSIGNMENT ATTEMPTS TO DIRECTLY CHANGE A DISCRIMINANT VALUE.

EXAMPLE - CHANGING DISCRIMINANTS OF UNCONSTRAINED OBJECTS

```
Sensor_Reading : Sensor_Reading_Type;           -- Unconstrained
Valid_Reading  : Sensor_Reading_Type (Valid => True);  -- Constrained
Invalid_Reading : Sensor_Reading_Type (Valid => False); -- Constrained
...
Sensor_Reading := Valid_Reading;
Sensor_Reading := Invalid_Reading;
...
Valid_Reading := Invalid_Reading;           -- Illegal
Sensor_Reading.Valid := True;               -- Illegal
```

INSTRUCTOR NOTES

- THIS EXAMPLE SHOWS THE USE OF RECORD TYPES WITH DISCRIMINANTS AS PROGRAM PARAMETERS.
- THIS SUBPROGRAM COPIES ONE `Varying_String_Type` OBJECT TO ANOTHER
 - IF THE `Maximum_Length` OF THE `To String` IS LESS THAN THE `Current_Length` OF THE `From String`, THEN THE `From String` IS TRUNCATED TO THE `Maximum_Length` OF THE `To String` AND ASSIGNED TO THE `To String`
 - WE NEEDED THE CAPABILITY OF EXAMINING THE DISCRIMINANT VALUE `Maximum_Length` OF THE `out` PARAMETER
 - A DISCRIMINANT OF A RECORD OBJECT ALWAYS HAS A VALUE. THIS IS ONE OF THE CONSISTENCY RULES WE WILL SEE LATER

EXAMPLE OF SUBPROGRAM PARAMETERS WITH DISCRIMINANTS

```
procedure Copy_String
  (From : in Varying_String_Type;
   To   : out Varying_String_Type) is
begin
  if From.Current_Length <= To.Maximum_Length then
    To.Current_Length := From.Current_Length;
    To.Contents (1 .. To.Current_Length) := From.Contents (1 .. From.Current_Length);
  else
    To.Current_Length := To.Maximum_Length;
    To.Contents (1 .. To.Maximum_Length) := From.Contents (1 .. To.Maximum_Length);
  end if;
end Copy_String;

● To.Maximum_Length IS EXAMINED EVEN THOUGH TO IS OF MODE out.
- THIS IS LEGAL SINCE Maximum_Length IS A DISCRIMINANT
- ILLEGAL TO EXAMINE To.Contents

● STILL CANNOT ASSIGN TO DISCRIMINANT DIRECTLY.

● CANNOT CHANGE THE VALUE OF THE DISCRIMINANT THROUGH A RECORD ASSIGNMENT UNLESS THE
  PARAMETER IS NOT CONSTRAINED.
```

AD-A166 367

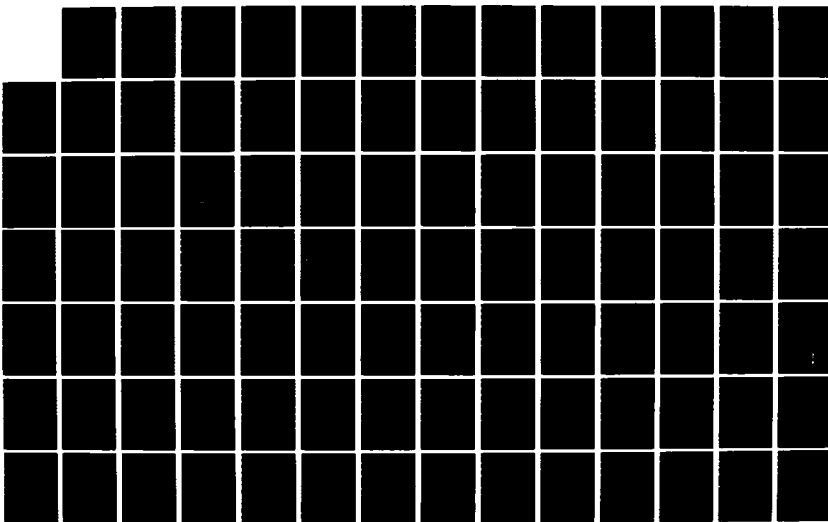
ADA (TRADE NAME) TRAINING CURRICULUM BASIC ADA
PROGRAMMING L202 TEACHER'S GUIDE VOLUME 2(U) SOFTECH
INC WALTHAM MA 1986 DAB07-83-C-K514

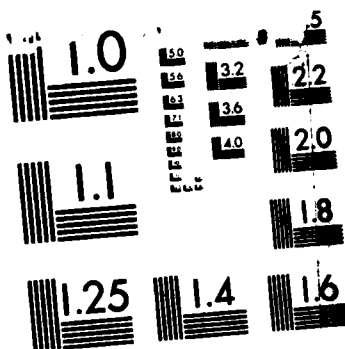
2/8

UNCLASSIFIED

F/G 9/2

NL





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

INSTRUCTOR NOTES

- THIS VERSION OF Copy_String IS SIMILAR TO THE PREVIOUS VERSION, EXCEPT WHEN THE Current_Length OF THE To STRING EXCEEDS THE Maximum_Length OF THE To STRING,
 1. TRUNCATION OCCURS ONLY IF To IS A CONSTRAINED OBJECT
 2. IF To IS UNCONSTRAINED, THEN IT IS ASSIGNED A RECORD AGGREGATE THAT CHANGES THE Maximum_Length DISCRIMINANT VALUE TO BE THE VALUE OF THE Current_Length OF THE From STRING
- TO DETERMINE WHETHER THE OBJECT IS CONSTRAINED, THE CONSTRAINED ATTRIBUTE IS USED.
- ANOTHER POINT OF VIEW MIGHT BE TO INDICATE AN ERROR IF To IS CONSTRAINED. THIS SHOULD BE MENTIONED.

USING THE Constrained ATTRIBUTE

```
procedure Copy_String
  (From : in Varying_String_Type;
   To   : out Varying_String_Type) is
begin
  if From.Current_Length = To.Maximum_Length then
    To.Current_Length := From.Current_Length;
    To.Contents(1 .. From.Current_Length) := From.Contents(1 .. From.Current_Length);
  elsif To'Constrained then
    To.Current_Length := To.Maximum_Length;
    To.Contents(1 .. To.Maximum_Length) := From.Contents(1 .. To.Maximum_Length);
  else
    To := (Maximum_Length => From.Current_Length,
           Current_Length => From.Current_Length,
           Contents      => From.Contents(1 .. From.Current_Length));
  end if;
end Copy_String;
```

- IF X IS A VALUE IN A RECORD TYPE WITH DISCRIMINANTS, THE X'Constrained HAS THE VALUE True; OTHERWISE X'Constrained HAS THE VALUE False.

INSTRUCTOR NOTES

- THIS EXAMPLE ILLUSTRATES THE USE OF A RECORD TYPE WITH DISCRIMINANTS AS THE COMPONENT TYPE OF AN UNCONSTRAINED ARRAY AND AS THE RESULT TYPE OF A FUNCTION.
- THE FUNCTION IS PASSED A LIST OF SENSOR READINGS. IF MORE THAN HALF THE SENSOR READINGS ARE VALID, THEN AN AVERAGED SENSOR READING IS RETURNED; OTHERWISE AN INVALID READING IS RETURNED.

ANOTHER SUBPROGRAM EXAMPLE

```
type Sensor_Reading_List_Type is array (Positive range <>) of Sensor_Reading_Type;

function Average_Reading (Sensor_Reading_List : Sensor_Reading_List_Type)
    return Sensor_Reading_Type is
    Number_Valid : Natural := 0;
    Valid_Sum    : Float   := 0.0;
begin
    for I in Sensor_Reading_List'Range loop
        if Sensor_Reading_List (I).Valid then
            Number_Valid := Number_Valid + 1;
            Valid_Sum    := Valid_Sum + Sensor_Reading_List (I).Reading_Part;
        end if;
    end loop;

    if Number_Valid > Sensor_Reading_List'Length/2 then
        return (True, Valid_Sum/Float(Number_Valid));
    else
        return (Valid => False);
    end if;
end Average_Reading;
```

INSTRUCTOR NOTES

- ITEM 1

- Ada DOES NOT ALLOW A RECORD OBJECT IN A RECORD TYPE WITH DISCRIMINANTS TO BE CREATED UNLESS A VALUE CAN BE DETERMINED FOR EACH DISCRIMINANT.

- ITEM 2

- IT IS NOT LEGAL, FOR EXAMPLE, TO REFER TO THE Reading_Part COMPONENT OF A Sensor_Reading_Type OBJECT UNLESS ITS VALID COMPONENT HAS THE VALUE True.

- ITEM 3

- THE RECORD AGGREGATE (False, 10.3) IS INVALID FOR A Sensor_Reading_Type OBJECT SINCE A Sensor_Reading_Type OBJECT WITH False FOR ITS DISCRIMINANT, HAS NO Reading_Part COMPONENT.

CONSISTENCY RULES

- DISCRIMINANTS ALWAYS HAVE VALUES.
- REFERENCES TO COMPONENTS MUST BE CONSISTENT WITH DISCRIMINANTS.
- AGGREGATES MUST BE CONSISTENT.
- DISCRIMINANTS CANNOT BE CHANGED BY THEMSELVES
 - NOT THROUGH ASSIGNMENT (:=)
 - NOT THROUGH SUBPROGRAM CALLS

INSTRUCTOR NOTES

- REMIND THE CLASS THAT THEY HAVE SEEN SUBTYPES BEFORE.
- IF THE CLASS HAS PROBLEMS WITH RECORD SUBTYPES, CONSIDER USING THE Aircraft_Type AS ANOTHER SOURCE OF EXAMPLES.

SUBTYPES OF RECORD TYPES

• TWO FORMS FOR DECLARATIONS

```

subtype SUBTYPE NAME is RECORD TYPE NAME
subtype SUBTYPE NAME is RECORD TYPE NAME DISCRIMINANT CONSTRAINT;

```

- WITH THE FIRST FORM, THE SUBTYPE CONSISTS OF THE ENTIRE RECORD TYPE
EXAMPLE:

```

subtype Line_Type is Varying_String_Type;

```

- WITH THE SECOND FORM, THE SUBTYPE CONSISTS OF THOSE RECORD VALUES THAT
SATISFY DISCRIMINANT CONSTRAINT
EXAMPLES:

```

subtype Valid_Sensor_Reading_Subtype is Sensor_Reading_Type (Valid => True);
-- THIS SUBTYPE CONSISTS OF ALL VALUES IN Sensor_Reading_Type
-- EXCEPT Sensor_Reading_Type (Valid => False)
subtype Line_of_80 is Varying_String_Type (80);
-- THIS SUBTYPE CONSISTS OF ALL VALUES IN Varying_String_Type
-- WHOSE Maximum_Length COMPONENT HAS THE VALUE OF 80.

```


INSTRUCTOR NOTES

- THIS SLIDE SHOWS THE CLASS THAT MEMBERSHIP TESTS ALSO EXIST FOR RECORD SUBTYPES.



MEMBERSHIP TESTS

- TO DETERMINE IF A RECORD VALUE BELONGS TO A PARTICULAR SUBTYPE, DISCRIMINANTS CAN BE TESTED
 - if Sensor_Reading.Valid then
...
end if;
- MEMBERSHIP TESTS CAN ALSO BE USED
 - if Sensor_Reading in Valid_Sensor_Reading_Subtype then
...
end if;
ESPECIALLY USEFUL IF THE RECORD TYPE HAS MORE THAN ONE DISCRIMINANT.

INSTRUCTOR NOTES

VG 728.2

9-52i

EXPRESSIONS INVOLVING DISCRIMINANTS

- FOR A RECORD TYPE HAVING A VARIANT, AN AGGREGATE FOR THE RECORD TYPE MAY NOT HAVE VARIABLES OR FUNCTION CALLS IN ANY EXPRESSION FOR DISCRIMINANTS SELECTING THE CURRENT VARIANT.

EXAMPLE: ASSUMING Aircraft_Id : Aircraft_Identification_Type;

```
(Aircraft_Identification => Aircraft_Id,  
Tracking_Data_Part      => Tracking_Data,  
Threat_Level_Part       => Low_Threat)
```

IS AN INVALID AGGREGATE SINCE Aircraft_Identification SELECTS THE VARIANT

EXAMPLE: ASSUMING Number_Of_Blanks : Natural;

```
(Maximum_Length => Number_Of_Blanks,  
Current_Length  => Number_Of_Blanks,  
Content         => (1 .. Number_Of_Blanks => ' '))
```

- THE DEFAULT INITIAL VALUE EXPRESSION IN A DISCRIMINANT SPECIFICATION IS EVALUATED WHEN A RECORD OBJECT IS CREATED, AND ONLY IF VALUES ARE NOT GIVEN IN A DISCRIMINANT CONSTRAINT OR IN AN INITIAL VALUE.
- EXPRESSIONS IN A DISCRIMINANT CONSTRAINT ARE EVALUATED WHEN THE DECLARATION CONTAINING THE CONSTRAINT IS ELABORATED.

Buffer_Size : Positive;

```
...  
subtype Buffer_Subtype is Varying_String_Type (Maximum_Length => Buffer_Size);
```

WHEN THIS SUBTYPE IS ELABORATED, THE CURRENT VALUE OF Buffer_Size IS USED TO DETERMINE THE VALUE OF Maximum_Length. THIS STAYS FIXED, EVEN IF Buffer_Size CHANGES.

INSTRUCTOR NOTES

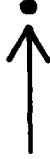
SECTION 10

ACCESS TYPES

INSTRUCTOR NOTES

THIS SLIDE IS JUST TO LET THE STUDENTS "TAKE A STEP BACK" AND PUT INTO PERSPECTIVE ALL OF THE CLASSES OF TYPES.

CLASSES OF TYPES

- SCALAR TYPES
 - INTEGER
 - REAL
 - ENUMERATION
- COMPOSITE TYPES
 - ARRAY
 - RECORD
-  ACCESS TYPES
- PRIVATE TYPES

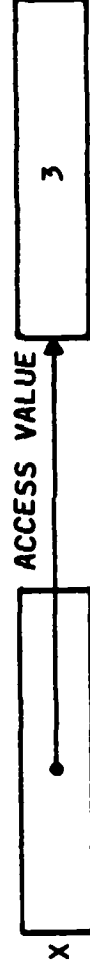
INSTRUCTOR NOTES

A NULL ACCESS VALUE IS DENOTED (IN PROGRAMS AND IN OUR DIAGRAMS) BY THE RESERVED WORD
"null."

ACCESS VALUES

AN ACCESS VALUE IS AN INDIRECT REFERENCE TO AN OBJECT CONTAINING SOME OTHER VALUE.

ACCESS VALUES ARE ALSO KNOWN AS POINTER VALUES. WE DRAW AN INDIRECT REFERENCE TO AN OBJECT AS AN ARROW POINTING TO THAT OBJECT.



X { CONTAINS AN INDIRECT REFERENCE TO
POINTS TO
DESIGNATES } AN OBJECT CONTAINING 3

THE SPECIAL ACCESS VALUE NULL DOES NOT POINT TO ANY OBJECT.

INSTRUCTOR NOTES

THE BOXES ON THE RIGHT REPRESENT TWO DIFFERENT OBJECTS, BOTH CONTAINING THE NUMBER 3.

THE BOXES ON THE LEFT REPRESENT THREE VARIABLES NAMED A, B, AND C, EACH CONTAINING ACCESS VALUES.

A AND B POINT TO THE SAME OBJECT, SO THEY HOLD THE SAME ACCESS VALUE.

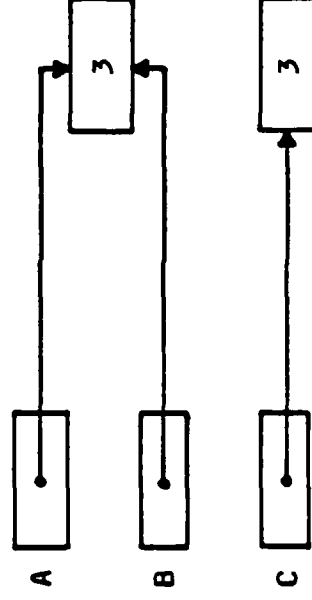
B AND C POINT TO DIFFERENT OBJECTS, SO THEY HOLD DIFFERENT ACCESS VALUES. IT IS IRRELEVANT THAT THESE TWO OBJECTS HAPPEN TO HAVE THE SAME CONTENTS.

ACCESS VALUES (Continued)

TWO ACCESS VALUES ARE EQUAL IF THEY DESIGNATE THE SAME OBJECT.

(TWO ARROWS DEPICT THE SAME ACCESS VALUE IF THEY POINT TO THE SAME PLACE.)

ACCESS VALUES DESIGNATING DIFFERENT OBJECTS WITH IDENTICAL CONTENTS ARE DIFFERENT ACCESS VALUES.



A = B, BUT B $\neq$ C

INSTRUCTOR NOTES

THE NOTION OF ACCESS VALUE MAY BE ALIEN TO FORTRAN PROGRAMMERS, BUT ASSEMBLY LANGUAGE PROGRAMMERS ARE FAMILIAR WITH SOMETHING QUITE SIMILAR -- ADDRESSES USED AS VALUES.

ACCESS VALUES MAY BE USED IN ASSIGNMENT STATEMENTS AND COMPARISONS FOR EQUALITY OR INEQUALITY. THEY MAY ALSO BE USED TO PROVIDE ACCESS TO THE OBJECTS THEY DESIGNATE.

INTERNALLY AN ACCESS VALUE DESIGNATING AN OBJECT MAY OR MAY NOT BE REPRESENTED AS THE ADDRESS OF THE OBJECT. THIS DEPENDS ON THE COMPILER AND IS OF NO CONCERN TO THE PROGRAMMER.

ACCESS VALUES ARE SIMILAR TO ADDRESSES ...

- THEY PROVIDE ACCESS TO ANOTHER OBJECT.
- THEY MAY BE USED AS VALUES THEMSELVES.
- TWO VARIABLES CONTAINING THE SAME { ADDRESS
ACCESS VALUE } ARE INDIRECT REFERENCES TO
THE SAME OBJECT.

... BUT THERE ARE IMPORTANT DIFFERENCES:

- AN ACCESS VALUE CANNOT POINT TO AN ARBITRARY PLACE IN MEMORY, BUT ONLY TO AN OBJECT OF A PARTICULAR TYPE.
- ACCESS VALUES ARE NOT NUMBERS. THEY CANNOT BE ADDED, SUBTRACTED, MULTIPLIED, OR DIVIDED. ONE ACCESS VALUE IS NOT GREATER THAN OR LESS THAN ANOTHER ACCESS VALUE, I.E., THE RELATIONAL OPERATORS DO NOT APPLY.

INSTRUCTOR NOTES

DYNAMIC ALLOCATION IS AN ACTION PERFORMED BY A PROGRAM.

IT USUALLY RESULTS FROM EXECUTION OF A STATEMENT THAT CALLS FOR AN OBJECT TO BE ALLOCATED. THE PART OF THE STATEMENT THAT CALLS FOR THE ALLOCATION SPECIFIES THE TYPE OF THE OBJECT TO BE ALLOCATED.

THE OBJECT ALLOCATED IS NEVER A CONSTANT.

[NOTE TO INSTRUCTOR: THE FACT THAT ALLOCATORS ARE EXPRESSIONS WILL BE EXPLAINED IN DETAIL LATER. THE REFERENCE TO STATEMENTS ABOVE IS INTENDED ONLY TO CONVEY THE FACT THAT DYNAMIC ALLOCATION IS CONTROLLED ALGORITHMICALLY.]

DYNAMIC ALLOCATION

DYNAMIC ALLOCATION IS THE PROCESS OF CREATING AN ALLOCATED VARIABLE DURING PROGRAM EXECUTION:

1. STORAGE FOR THE OBJECT IS SET ASIDE.
2. AN ACCESS VALUE POINTING TO THE OBJECT IS MADE AVAILABLE TO THE PROGRAM.

A DYNAMICALLY ALLOCATED OBJECT IS ALWAYS A VARIABLE OF A SPECIFIED TYPE.

THE PROGRAM MAY ONLY REFER TO THE VARIABLE IN TERMS OF THE ACCESS VALUE POINTING TO IT.

INSTRUCTOR NOTES

THE DECLARED VARIABLES AND ALLOCATED VARIABLES DO NOT OVERLAP.

ALLOCATED VARIABLES CAN ONLY BE REFERRED TO USING ACCESS VALUES AND ACCESS VALUES CAN ONLY POINT TO ALLOCATED VARIABLES.

[NOTE TO INSTRUCTOR: A DECLARED VARIABLE IS AN OBJECT RESULTING FROM A VARIABLE DECLARATION. A RENAMING DECLARATION IS NOT A VARIABLE DECLARATION, AND A RENAMING DECLARATION LIKE

S : Some_Type renames Y.all;

ESTABLISHES AN IDENTIFIER NAMING AN ALLOCATED VARIABLE. DO NOT ADDRESS THIS ISSUE UNLESS IT IS BROUGHT UP BY A STUDENT!]

EVERYTHING WE'VE SEEN SO FAR HAS BEEN A DECLARED VARIABLE, NOW WE'RE LOOKING AT ALLOCATED VARIABLES ...

THE TWO CATEGORIES OF Ada VARIABLES

DECLARED VARIABLES

- ARISE FROM VARIABLE DECLARATIONS
- REFERRED TO BY THEIR DECLARED NAMES

→ ALLOCATED VARIABLES

- ARISE FROM DYNAMIC ALLOCATION
- REFERRED TO BY THE ACCESS VALUES MADE AVAILABLE DURING DYNAMIC ALLOCATION

ACCESS VALUES NEVER POINT TO DECLARED VARIABLES.

INSTRUCTOR NOTES

PAUSE AT THIS POINT FOR QUESTIONS.

TELL CLASS YOU'VE INTRODUCED CONCEPTS AND NOW YOU'LL GO INTO GREATER DETAIL ON ACCESS VALUES.

RECAP

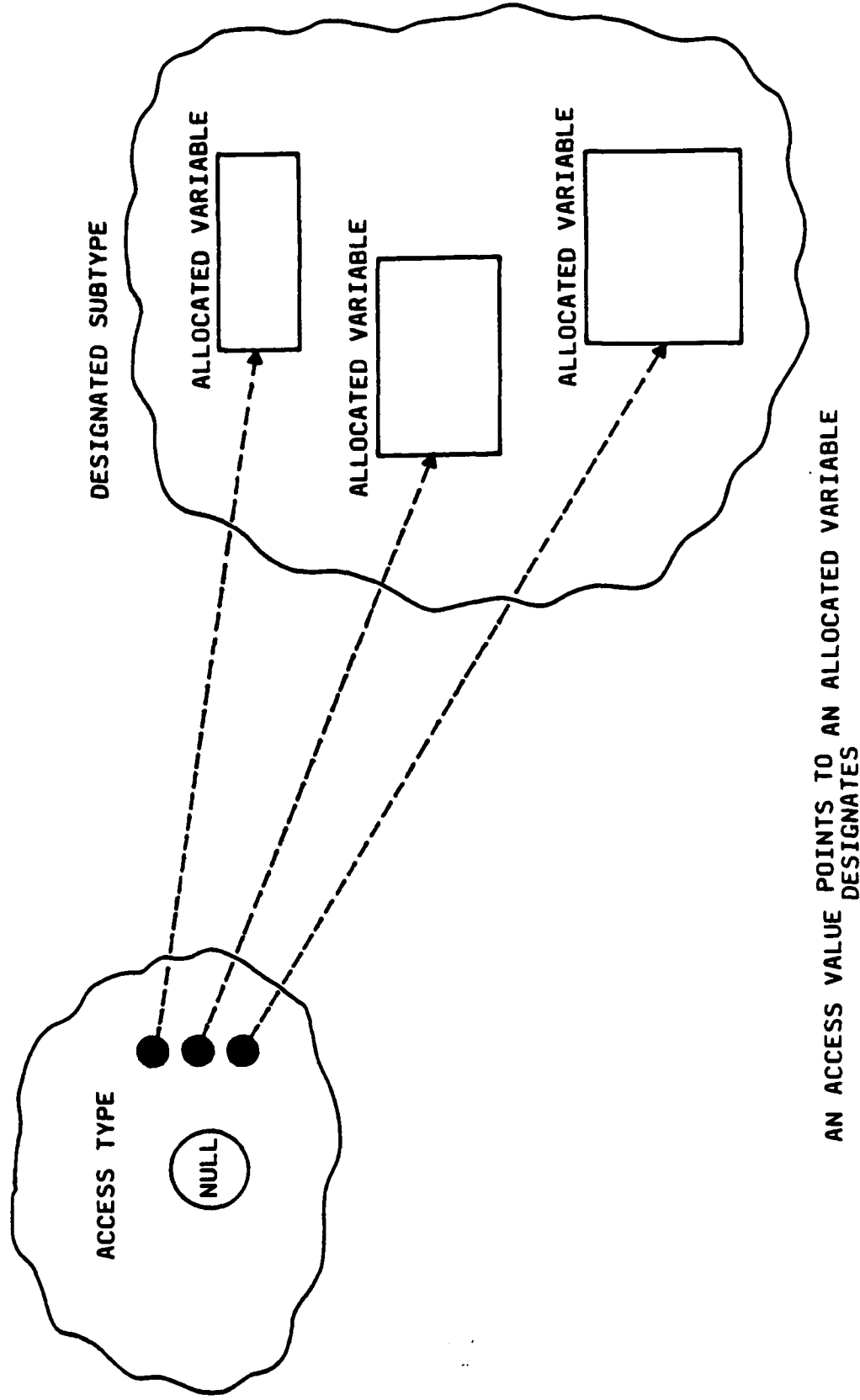
1. ACCESS VALUES ARE INDIRECT REFERENCES (THEY POINT) TO OTHER VARIABLES.
2. VARIABLES MAY BE DECLARED OR ALLOCATED.
3. ALLOCATED VARIABLES ARE ALWAYS POINTED TO BY ACCESS VALUES.
DECLARED VARIABLES ARE NEVER POINTED TO BY ACCESS VALUES.
4. ALLOCATED VARIABLES MAY ONLY BE REFERRED TO IN TERMS OF THE ACCESS VALUES POINTING TO THEM.

INSTRUCTOR NOTES

THE FOLLOWING SLIDE ANSWERS THE QUESTION, "WHERE DO ACCESS VALUES FIT INTO THE LANGUAGE?"

POINT OUT THE NULL POINTER.

ACCESS TYPES



INSTRUCTOR NOTES

AN ACCESS TYPE DECLARATION SPECIFIES THE SUBTYPE OF THE VARIABLES TO BE POINTED TO BY VALUES IN THE ACCESS TYPE. EACH SUCH DECLARATION ESTABLISHES A SEPARATE COLLECTION OF ALLOCATED VARIABLES. TWO ACCESS TYPE DECLARATIONS CAN NAME THE SAME TYPE OR SUBTYPE TO BE DESIGNATED, BUT OBJECTS IN THE TWO ACCESS TYPES CANNOT BOTH DESIGNATE THE SAME ALLOCATED VARIABLE. THIS IS BECAUSE EACH ALLOCATED VARIABLE BELONGS TO EXACTLY ONE COLLECTION.

IP, NP, AND DP ARE VARIABLES CONTAINING ACCESS VALUES.

ACCESS TYPES

AN ACCESS TYPE IS A TYPE CONSISTING OF ACCESS VALUES.

ALL VALUES IN A GIVEN ACCESS TYPE DESIGNATE ALLOCATED VARIABLES OF A SINGLE SUBTYPE.

ACCESS TYPE DECLARATION:

type access type name is access designated type or subtype name [Constraint];

EXAMPLES:

type Integer_Pointer_Type is access Integer;
type Natural_Pointer_Type is access Natural;
type Digit_Pointer_Type is access Integer range 0 .. 9;

IP : Integer_Pointer_Type;
NP : Natural_Pointer_Type;
DP : Digit_Pointer_Type;

OBJECTS IN ACCESS TYPE (SUCH AS IP, NP, AND DP) ARE AUTOMATICALLY INITIALIZED TO NULL UNLESS ANOTHER VALUE IS GIVEN IN THE OBJECT DECLARATION.

INSTRUCTOR NOTES

LIKE A FUNCTION CALL, AN ALLOCATOR REQUIRES CERTAIN ACTIONS TO BE PERFORMED IN ORDER TO PRODUCE A VALUE.

REMINDER: DYNAMIC OBJECTS ARE CREATED DYNAMICALLY AND EXPLICITLY DURING PROGRAM EXECUTION.

ALLOCATORS

ALLOCATED VARIABLES ARE CREATED BY EVALUATION OF AN ALLOCATOR.

AN ALLOCATOR IS AN EXPRESSION.

EVALUATION OF THIS EXPRESSION CONSISTS OF TWO STEPS:

1. A VARIABLE OF A SPECIFIED SUBTYPE IS DYNAMICALLY ALLOCATED.
2. AN ACCESS VALUE POINTING TO THE NEW ALLOCATED VARIABLE BECOMES THE VALUE OF THE EXPRESSION.

INSTRUCTOR NOTES

THIS SLIDE PROVIDES THE STUDENTS WITH A BRIEF LOOK AT ALLOCATORS. AVOID GETTING CAUGHT IN SYNTAX RULES, THE FORMS OF AN ALLOCATOR ARE PRESENTED ON THE FOLLOWING SLIDE.

EXAMPLES OF ALLOCATORS

```
CONTEXT:  type Integer_Pointer_Type is access Integer;
          type Cell_Type is
            record
              Value : String (1 .. 8);
              Priority : Positive;
            end record;
          type Link_Type is access Cell_Type;
          type Bit_String_Type is array (Integer range <>) of Boolean;
          type Bit_String_Pointer_Type is access Bit_String_Type;
```

EXAMPLES:

```
A : Integer_Pointer_Type := new Integer;
Halfword : Bit_String_Pointer_Type := new Bit_String_Type (1 .. 4);

Cell_1, Cell_2 : Link_Type;
Cell_1 := new   Cell_Type ' (*****M100", 1);
Cell_2 := new   Cell_Type ' (Value => "*****M200", Priority => 2);
```

INSTRUCTOR NOTES

THE THIRD FORM DEFINES THE CONTENTS OF THE ALLOCATED OBJECT BEING CREATED. THE OTHER FORMS DO NOT.

THE PARENTHESES FORM PART OF THE INDEX CONSTRAINT OR DISCRIMINANT CONSTRAINT.

FORMS OF AN ALLOCATOR

- new type or subtype name
- new unconstrained array subtype name index constraint
- new type or subtype name' (initial value)

-- IF THE INITIAL VALUE IS AN AGGREGATE, THE RESULTING DOUBLE SET OF PARENTHESES MAY BE ABBREVIATED BY A SINGLE SET, E.G.

new Vector'(1,2,3,4)

INSTEAD OF

new Vector'((1,2,3,4))

- new unconstrained record subtype name discriminant constraint

INSTRUCTOR NOTES

THE DECLARATION OF A AND B IS EQUIVALENT IN EVERY RESPECT TO SEPARATE AND IDENTICAL DECLARATIONS FOR EACH VARIABLE. IN PARTICULAR, THE ALLOCATOR SPECIFYING THE INITIAL VALUE IS EVALUATED TWICE, CREATING TWO ALLOCATED VARIABLES DESIGNATED BY TWO DIFFERENT ACCESS VALUES. EACH ALLOCATED VARIABLE IS INITIALIZED TO 3.

BECAUSE NO INITIAL VALUE IS SPECIFIED FOR C, C IS INITIALIZED TO null.

THE ASSIGNMENT C := B MAKES C DESIGNATE THE SAME OBJECT AS B.

THE ASSIGNMENT B := new Integer (4) CAUSES THE CREATION OF A NEW ALLOCATED VARIABLE CONTAINING THE VALUE 4, AND PLACES THE ACCESS VALUE DESIGNATING THAT VARIABLE IN B.

C CONTINUES TO DESIGNATE THE VARIABLE ORIGINALLY DESIGNATED BY B.

POINT OUT THAT THE ALLOCATOR HAS BEEN EVALUATED TWICE, ONCE FOR A AND ONCE FOR B. IF IT HAD ONLY BEEN ALLOCATED ONCE, THEN A AND B WOULD BOTH BE POINTING TO THE SAME ALLOCATED VARIABLE.

USE OF ALLOCATORS

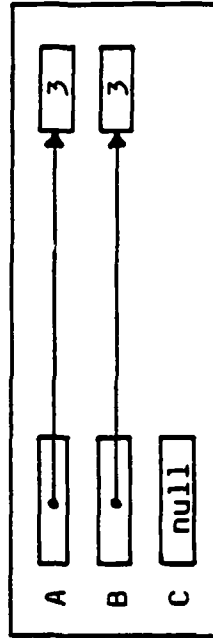
procedure Allocator_Example is

type Integer_Pointer_Type is access Integer;

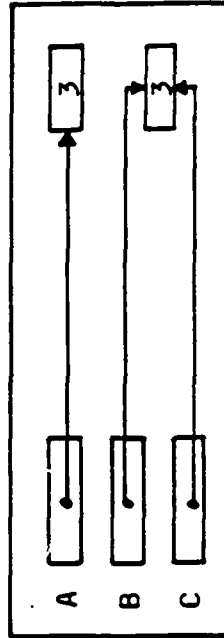
A, B : Integer_Pointer_Type := new Integer'(3); -- ALLOCATOR EVALUATED TWO TIMES

C : Integer_Pointer_Type; -- DEFAULT INITIAL VALUE IS NULL

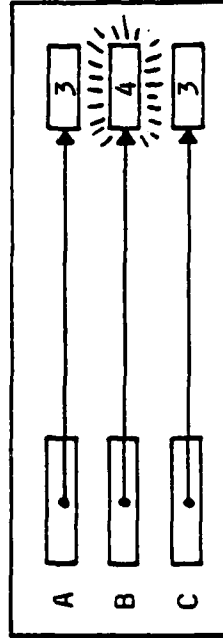
begin -- Allocator_Example



C := B;



B := new Integer'(4);



end Allocator_Example;

INSTRUCTOR NOTES

THIS SLIDE EMPHASIZES A RULE GIVEN TWO SLIDES EARLIER: WHEN ALLOCATING A VARIABLE IN AN UNCONSTRAINED ARRAY SUBTYPE, YOU MUST PROVIDE EITHER AN INDEX CONSTRAINT (AS WITH A AND B) OR AN INITIAL VALUE (AS WITH C AND D; C IS INITIALIZED WITH A NAMED AGGREGATE, D WITH A POSITIONAL ONE).

AN APOSTROPHE IS USED WITH AN INITIAL VALUE, BUT NOT WITH AN INDEX CONSTRAINT.

WHEN AN INITIAL VALUE IS GIVEN, IT MUST BE POSSIBLE TO DEDUCE THE INDEX BOUNDS FROM THE INITIAL VALUE. THEREFORE, WHEN THE ARRAY SUBTYPE IS NOT CONSTRAINED, AN AGGREGATE GIVING THE INITIAL VALUE MAY NOT CONTAIN others. (FOR A POSITIONAL AGGREGATE, THE FIRST VALUE OF THE INDEX SUBTYPE -- Natural'First, OR 0, IN THIS CASE -- IS USED AS THE LOWER BOUND OF THE ARRAY.)

IF AN ARRAY SUBTYPE IS CONSTRAINED, YOU DON'T HAVE TO SPECIFY AN INITIAL VALUE OR AN INDEX CONSTRAINT.

RATIONALE IS THAT THE ALLOCATOR NEEDS ENOUGH INFORMATION TO KNOW HOW BIG OF AN ARRAY TO ALLOCATE.

ALLOCATION OF ARRAYS

- WHEN ALLOCATING A VARIABLE IN AN UNCONSTRAINED ARRAY TYPE, YOU MUST SPECIFY EITHER AN INDEX CONSTRAINT OR AN INITIAL VALUE. (INITIAL VALUES ARE PRECEDED BY APOSTROPHES; INDEX CONSTRAINTS ARE NOT.)

- EXAMPLE

procedure Array_Allocation is

type Vector_Type is array (Natural range <>) of Float;

type Vector_Pointer_Type is access Vector_Type;

A,B,C,D : Vector_Pointer_Type;

begin

... A := new Vector_Type (1 .. 10);

... B := new Vector_Type (20 .. 100);

...

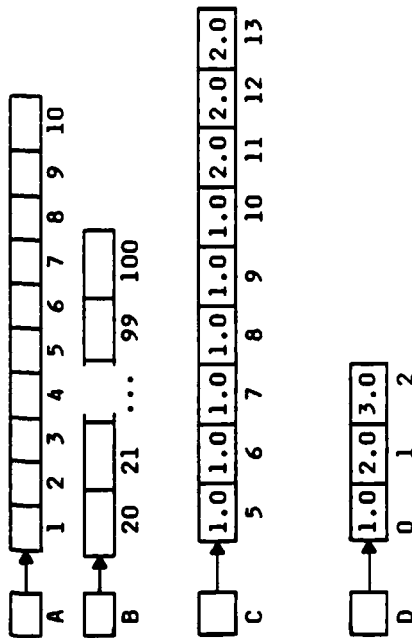
C := new Vector_Type '(5 .. 10 => 1.0, 11 .. 13 => 2.0);

...

D := new Vector_Type '(1.0, 2.0, 3.0);

...

end Array_Allocation;



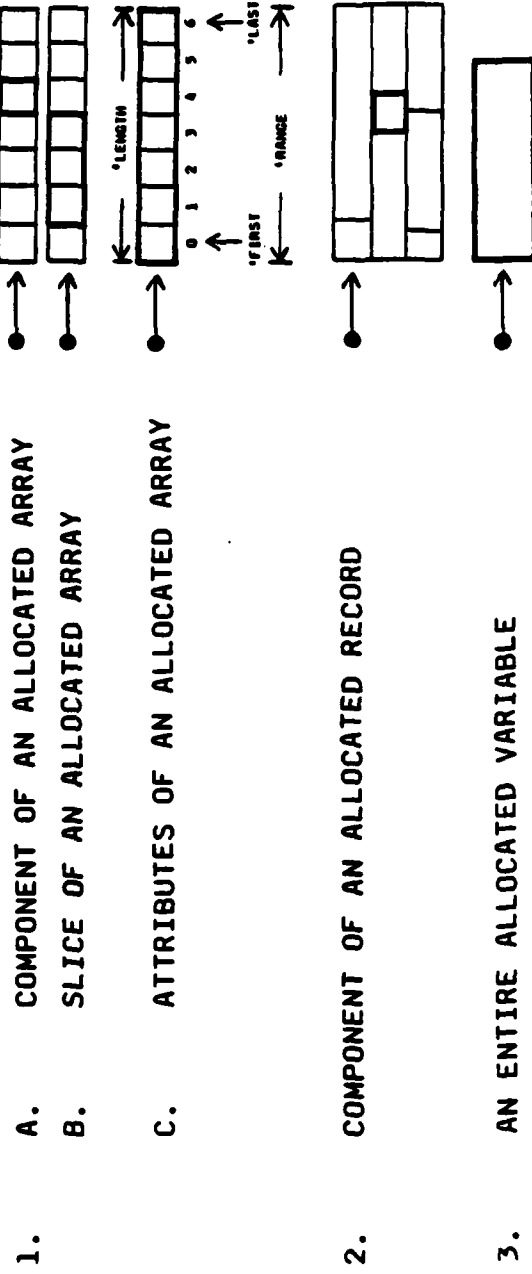
INSTRUCTOR NOTES

THE PROCESS OF GETTING YOUR HANDS ON AN ALLOCATED VARIABLE, GIVEN AN ACCESS VALUE DESIGNATING THAT VARIABLE, IS SOMETIMES CALLED DEREFERENCING.

THE ARROWS POINT TO THE VALUE AS A WHOLE, NOT INDIVIDUAL COMPONENTS.

REFERRING TO ALLOCATED VARIABLES

- ALLOCATED VARIABLES AND COMPONENTS OF THOSE VARIABLES MAY BE REFERRED TO IN TERMS OF ACCESS VALUES POINTING TO THEM.
- THEY MAY BE REFERRED TO ON EITHER SIDE OF AN ASSIGNMENT STATEMENT.
- SPECIAL NOTATION FOR DIFFERENT CASES:



EXAMPLES OF EACH TO FOLLOW...

INSTRUCTOR NOTES

IF SP IS AN ACCESS VALUE DESIGNATING AN N-DIMENSIONAL ARRAY, THEN SP FOLLOWED BY A PARENTHEZIZED LIST OF N SUBSCRIPTS NAMES A COMPONENT OF THE DESIGNATED ARRAY.

ON THE SLIDE, S CONTAINS A STRING OF LENGTH 5, THAT IS, AN ARRAY CONTAINING FIVE COMPONENTS OF TYPE Character. SP CONTAINS AN ACCESS VALUE DESIGNATING A STRING OF LENGTH 12.

IN THE FIRST ASSIGNMENT STATEMENT, S(1) REFERS TO THE FIRST COMPONENT OF S ITSELF, WHILE SP (12) REFERS TO THE TWELFTH COMPONENT OF THE ARRAY DESIGNATED BY SP. THE EFFECT OF THIS ASSIGNMENT IS TO REPLACE THE PERIOD FOLLOWING "Hello there" WITH AN EXCLAMATION POINT.

(AN EXPRESSION LIKE SP(12) CAN ALSO APPEAR IN EXPRESSIONS, NAMING THE VALUE OF THE TWELFTH COMPONENT OF THE ARRAY DESIGNATED BY SP. THE SLIDE DOES NOT ILLUSTRATE THIS POSSIBILITY.)

SINCE THE ARRAY DESIGNATED BY SP IS ONE-DIMENSIONAL, IT IS POSSIBLE TO TAKE SLICES OF THE ARRAY. SP FOLLOWED BY A PARENTHEZIZED DISCRETE RANGE NAMES A SLICE OF THE ARRAY DESIGNATED BY SP.

IN THE SECOND ASSIGNMENT STATEMENT, SP (1 .. 5), NAMES SUCH A RANGE. THIS ASSIGNMENT REPLACES THE CONTENTS OF S WITH THE STRING "Hello".

IN THE THIRD ASSIGNMENT, SP (7 .. 11) NAMES SUCH A RANGE. THIS ASSIGNMENT REPLACES THE SEVENTH THROUGH ELEVENTH COMPONENTS OF THE ARRAY DESIGNATED BY SP WITH THE CHARACTERS IN S, NAMELY "Hello". THIS LEAVES "Hello Hello!" IN THE ARRAY DESIGNATED BY SP.

AS SHOWN ON THE BOTTOM OF THE SLIDE, AN ATTRIBUTE DESIGNATOR NORMALLY MEANINGFUL ONLY FOR ARRAY OBJECTS CAN BE APPLIED TO SP TO NAME AN ATTRIBUTE OF THE ARRAY DESIGNATED BY SP.

HAVE STUDENTS FILL IN.

SP'LENGTH = 12, SP'FIRST = 1, SP'LAST = 12

INSTRUCTOR NOTES

IF LP IS AN ACCESS VALUE DESIGNATING A RECORD WITH COMPONENTS LATITUDE AND LONGITUDE, THEN LP.Latitude AND LP.Longitude NAME THE COMPONENTS OF THE DESIGNATED RECORD.

ON THE SLIDE, L CONTAINS A RECORD OF TYPE Location_Type AND LP CONTAINS AN ACCESS VALUE POINTING TO SUCH A RECORD.

THE FIRST ASSIGNMENT STATEMENT ASSIGNS THE LATITUDE COMPONENT OF THE RECORD DESIGNATED BY LP TO THE LATITUDE COMPONENT OF L.

ANSWERS:	LP : ☐	55.75	L :	55.75
		37.70		-77.01

THE SECOND ASSIGNMENT STATEMENT ASSIGNS THE LONGITUDE COMPONENT OF L TO THE LONGITUDE COMPONENT OF THE RECORD DESIGNATED BY LP.

ANSWERS:	LP : ☐	55.75	L :	38.90
		-77.01		-77.01

THE NOTION LP.all MEANS THAT ALL THE COMPONENTS OF THE RECORD DESIGNATED BY LP, IN OTHER WORDS, THE ENTIRE RECORD (THE WORD all IS RESERVED, SO IT CANNOT BE THE NAME OF AN ORDINARY RECORD COMPONENT).

THE THIRD ASSIGNMENT STATEMENT ASSIGNS THE ENTIRE RECORD DESIGNATED BY LP TO L.

ANSWERS:	LP : ☐	55.75	L :	55.75
		37.70		37.70

THE FOURTH ASSIGNMENT STATEMENT PLACES A NEW Location_Type VALUE IN THE VARIABLE DESIGNATED BY LP, REPLACING THE ENTIRE RECORD.

ANSWERS:	LP : ☐	51.50	L :	38.90
		0.00		-77.01

HAVE STUDENTS FILL IN.

NOTE: HAVE STUDENT REFER TO THE ORIGINAL DEFINITION FOR EACH EXAMPLE.

2. COMPONENTS OF ALLOCATED RECORDS

type Location_Type is
record

Latitude : Float range -90.0 .. 90.0;
Longitude : Float range -180.0 .. 180.0;
end record;

type Location_Pointer_Type is access Location_Type;

LP : Location_Pointer_Type := new Location_Type'(55.75, 37.70);
L : Location_Type := (38.90, -77.01);

LP :   Latitude
Longitude

L :  Latitude
Longitude

L.Latitude := LP.Latitude;

-- READ THE LATITUDE COMPONENT OF THE RECORD
-- POINTED TO BY LP

LP :  

L : 

LP.Longitude := L.Longitude;

-- CHANGE THE LONGITUDE COMPONENT OF THE
-- RECORD POINTED TO BY LP

LP :  

L : 

L := LP.all;

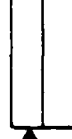
-- READ THE ENTIRE RECORD POINTED TO BY LP

LP :  

L : 

LP.all := (51.50, 0.00);

-- REPLACE THE ENTIRE RECORD POINTED TO BY LP

LP :  

L : 

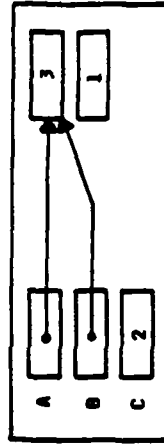
INSTRUCTOR NOTES

IN FACT, IF X IS ANY NON-NULL ACCESS VALUE, X.all NAMES THE ENTIRE ALLOCATED VARIABLE POINTED TO BY X. THE ALLOCATED VARIABLE DOES NOT HAVE TO BELONG TO A RECORD TYPE.

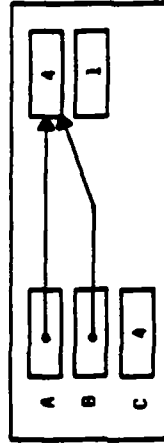
THIS SLIDE SHOWS HOW TO NAME THE INTEGER VARIABLES DESIGNATED BY VALUES OF THE Integer Pointer Type. A.all NAMES THE BOX THAT A POINTS TO, AND B.all NAMES THE BOX THAT B POINTS TO.

THE ALLOCATOR IN THE DECLARATION OF A AND B IS EVALUATED ONCE FOR A AND ONCE FOR B, SO A AND B INITIALLY POINT TO TWO SEPARATE ALLOCATED VARIABLES, EACH CONTAINING THE VALUE 1.

THE FIRST ASSIGNMENT STATEMENT PLACES THE VALUE 3 IN THE BOX POINTED TO BY A. IT IS AN ASSIGNMENT OF INTEGER VALUES. THE SECOND ASSIGNMENT MAKES B POINT TO THAT SAME BOX. IT IS AN ASSIGNMENT OF Integer Pointer Type VALUES.



THE THIRD ASSIGNMENT INCREMENTS THE VARIABLE POINTED TO BY B. SINCE A AND B POINT TO THE SAME BOX AT THIS POINT, A.all AND B.all ARE CURRENTLY TWO DIFFERENT NAMES FOR THE SAME ALLOCATED VARIABLE. INCREMENTING B.all CAUSES A.all TO INCREASE. THUS THE FOURTH ASSIGNMENT TO C.



THE CLASS SHOULD BE GIVEN A MOMENT TO STUDY THE FIFTH ASSIGNMENT AND CONCLUDE THAT IT INCREMENTS B.all (TO 5) AND DOES NOT AFFECT C.

REMEMBER TO FILL IN ARROWS AS WELL AS BOXES.

USE .all TO NAME ENTIRE ALLOCATED VARIABLES OF ANY TYPE

procedure All_Example is

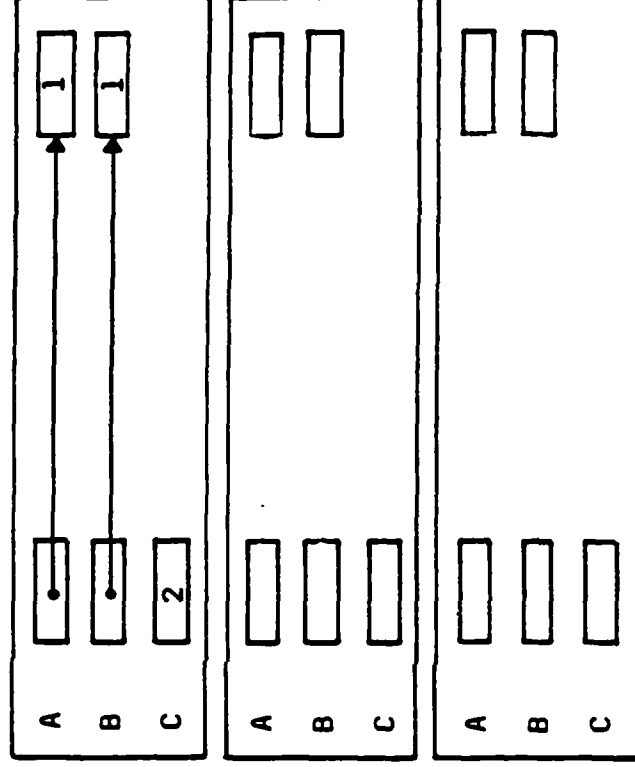
```
type Integer_Pointer_Type is access Integer;
A, B : Integer_Pointer_Type := new Integer'(1);
C : Integer := 2;
```

begin -- All_Example

```
-----
A.all := 3;
B := A;
```

```
-----
B.all := B.all + 1;
C := A.all;
```

```
-----
A.all := A.all + 1;
-- HOW DOES THIS AFFECT THE VALUE OF B.all? OF C?
end All_Example;
```

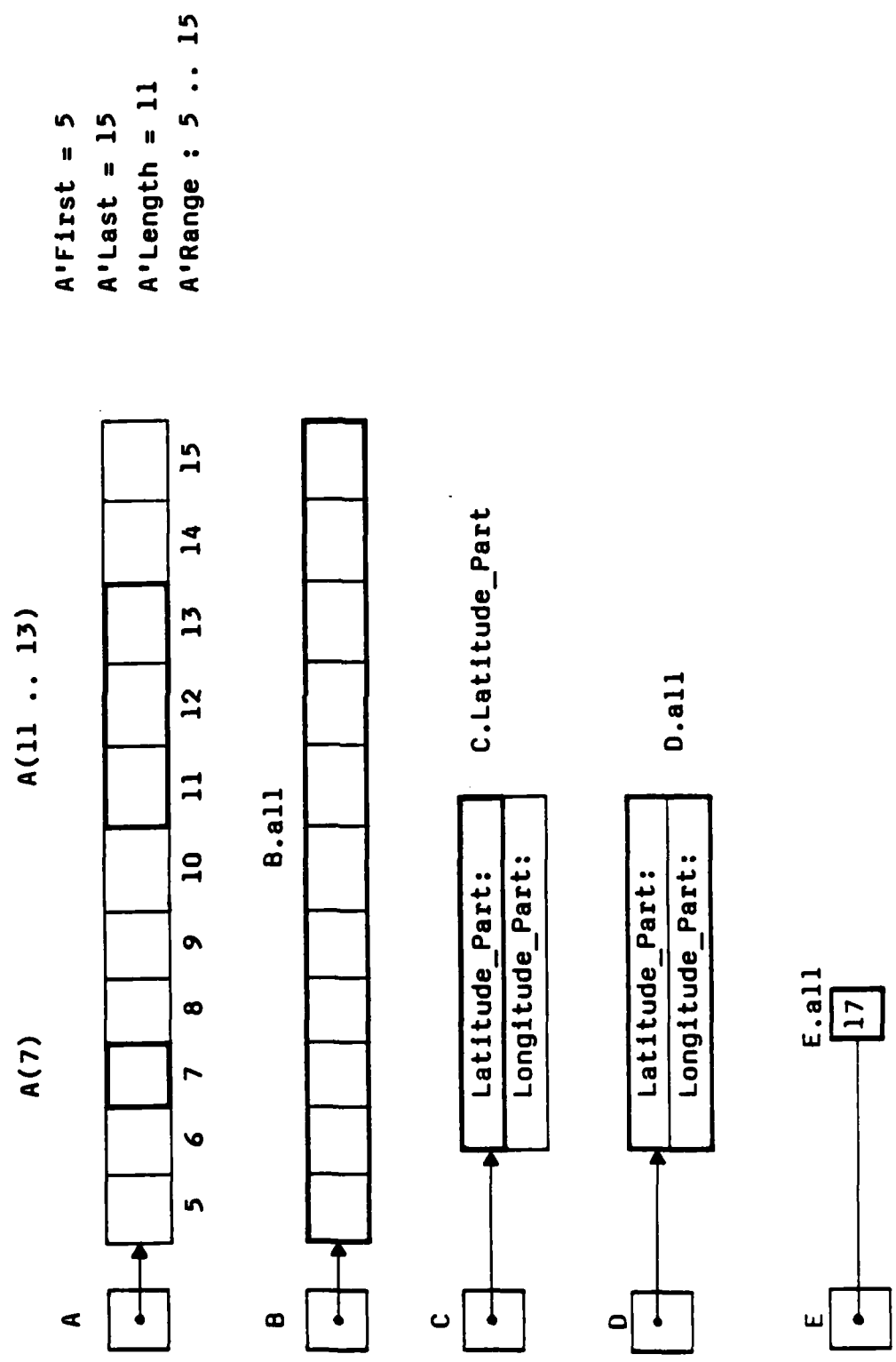


INSTRUCTOR NOTES

REVIEW.



NAMING ALLOCATED VARIABLES -- SUMMARY



INSTRUCTOR NOTES

THE FIRST FOUR TOPICS WILL BE COVERED HERE AND THE FIFTH WILL BE COVERED IN L305.

AS WILL BE EXPLAINED LATER, A RAGGED ARRAY IS AN ARRAY WHOSE COMPONENTS THEMSELVES ARE ARRAYS OF DIFFERENT LENGTHS.

USES OF ACCESS VALUES

1. TO PROVIDE ACCESS TO SHARED DATA.
2. TO BUILD RAGGED ARRAYS.
3. TO AVOID MOVING LARGE AMOUNTS OF DATA.
4. TO CHANGE, AT RUNTIME, THE VARIABLE ASSOCIATED WITH THE NAME.
5. TO DEFINE A DATA TYPE WHOSE OBJECT REFERS TO OTHER OBJECTS OF THE SAME TYPE.

INSTRUCTOR NOTES

SINCE MOST PEOPLE HAVE MAILING ADDRESSES IDENTICAL TO THEIR RESIDENTIAL ADDRESSES, MOST Person_Type RECORDS, AS DEFINED ON THE TOP PART OF THE SLIDE, CONTAIN A LARGE AMOUNT OF DUPLICATE INFORMATION.

AN ALTERNATIVE, SHOWN AT THE BOTTOM OF THE SLIDE, IS TO HAVE COMPONENTS OF THE Person_Type RECORD POINT TO Address_Type RECORDS. COMPONENTS REPRESENTING THE SAME ADDRESS CAN POINT TO THE SAME Address_Type RECORD, SO THAT THE ADDRESS NEED ONLY BE STORED IN ONE PLACE.

THE FOLLOWING SLIDE DEPICTS THESE ALTERNATIVES GRAPHICALLY.

THE LAST STATEMENT IS A PUN. IN THE FIRST CASE, P.Residence_Part MEANS THE STATE PART OF THE COMPONENT OF THE ADDRESS TYPE RECORD P.Residence_Part. IN THE SECOND CASE, P.Residence_Part MEANS THE STATE PART COMPONENT OF THE ADDRESS TYPE RECORD POINTED TO BY P.Residence_Part.

1. USE OF ACCESS VALUES TO PROVIDE ACCESS TO SHARED DATA

EXAMPLE 1: SAVE SPACE BY AVOIDING DUPLICATION OF INFORMATION

```
type Address_Type is
record
    Street_Part : String (1 .. 25);
    City_Part   : String (1 .. 15);
    State_Part  : State_Type;
    Zip_Code_Part : Integer range 0 .. 99999;
end record;
```

```
type Person_Type is
record
    Name_Part      : String (1 .. 30);
    Residence_Part : Address_Type;
    Mailing_Address_Part : Address_Type;
    Phone_Number_Part : String (1 .. 10);
end record;
```

ALTERNATIVELY:

```
type Address_Pointer_Type is access Address_Type;
```

```
type Person_Type is
record
    Name_Part      : String (1 .. 30);
    Residence_Part : Address_Pointer_Type;
    Mailing_Address_Part : Address_Pointer_Type;
    Phone_Number_Part : String (1 .. 10);
end record;
```

EITHER WAY, PERSON P'S STATE OF RESIDENCE IS P.Residence_Part.State_Part

INSTRUCTOR NOTES

THIS SLIDE ILLUSTRATES HOW THE USE OF ACCESS VALUES SAVES SPACE.

WHEN SOMEONE HAS DIFFERENT RESIDENCE AND MAILING ADDRESSES, COMPONENTS OF HIS Person_Type RECORD WILL POINT TO TWO SEPARATE Address_Type RECORDS; BUT IN THE USUAL CASE (EXEMPLIFIED BY JANE SMITH), TWO COMPONENTS OF THE Person_Type RECORD POINT TO THE SAME Address_Type RECORD, AND ADDRESS INFORMATION IS ONLY REPRESENTED ONCE.

1. USE OF ACCESS VALUES TO PROVIDE ACCESS TO SHARED DATA (Continued)

ILLUSTRATION OF SPACE SAVINGS

ORIGINAL METHOD	
"John Doe"	"
"123 Main Street"	"
"Waltham"	"
MA	
02154	
"P.O. Box 999"	"
"Cambridge"	"
MA	
02138	
"6172345678"	
"Jane Smith"	"
"100 3rd Street"	"
"Falls Church"	"
VA	
22041	
"100 3rd Street"	"
"Falls Church"	"
VA	
22401	
"7033456789"	

METHOD USING ACCESS TYPES	
"Jane Smith"	"
"100 3rd Street"	"
"Falls Church"	"
VA	
22041	
"7033456789"	
"John Doe"	"
"123 Main Street"	"
"Waltham"	"
MA	
02154	
"P.O. Box 999"	"
"Cambridge"	"
MA	
02138	
"6172345678"	

INSTRUCTOR NOTES

IN ESSENCE, Ada REQUIRES ALL COMPONENTS OF AN ARRAY TO HAVE THE SAME LENGTH.

2. USE OF ACCESS VALUES TO BUILD RAGGED ARRAYS

A "RAGGED ARRAY" IS ONE WHICH ITSELF CONTAINS ARRAYS OF DIFFERENT LENGTHS.

THE FOLLOWING IS ILLEGAL:

```
type Name_List_Type is array (Positive range <>) of String;  
Beatles : Name_List_Type (1..4) :=  
    ("John", "Paul", "George", "Ringo");
```

-- THIS IS NOT AN EMBEDDED APPLICATION

'J'	'O'	'h'	'n'
'P'	'a'	'u'	'l'
'G'	'e'	'o'	'r'
'R'	'i'	'n'	'g'
		'o'	'e'

REASON:

STRING IS AN UNCONSTRAINED ARRAY TYPE. THE DECLARATION OF

Name_List_Type MUST THEREFORE SPECIFY AN INDEX CONSTRAINT FOR ITS COMPONENTS.

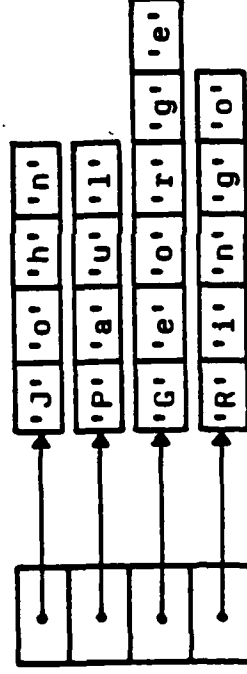
INSTRUCTOR NOTES

NOW THE ARRAY COMPONENTS ARE ACCESS VALUES. EVEN THOUGH THEY POINT TO STRINGS OF VARIOUS LENGTHS, THE COMPONENTS THEMSELVES ARE ALL THE SAME SIZE.

2. USE OF ACCESS VALUES TO BUILD RAGGED ARRAYS (Continued)

THE FOLLOWING IS LEGAL:

```
type String_Pointer_Type is access String;  
type Name_List_Type is  
  array (Positive range <>) of String_Pointer_Type;  
  
Beatles : Name_List_Type (1 .. 4) :=  
  (new String_T("John"),  
   new String_T("Paul"),  
   new String_T("George"),  
   new String_T("Ringo"));
```



INSTRUCTOR NOTES

THE MOST EFFICIENT WAY TO DO THIS IS TO PLACE THE TANK STATUS RECORDS IN AN ARRAY AND SORT THE ARRAY ACCORDING TO THE DISTANCE OF EACH TANK FROM THE SPECIFIED POINT.

THIS SLIDE SHOWS THE SPECIFICATION OF A PROCEDURE TO DO THIS. Sort_By_Distance REARRANGES THE COMPONENTS OF THE ARRAY Tanks IN ORDER OF DISTANCE FROM THE POINT Reference_Point.

3. USE OF ACCESS VALUES TO AVOID MOVING LARGE AMOUNTS OF DATA

EXAMPLE: A SET OF LARGE RECORDS, EACH CONTAINING STATUS INFORMATION ABOUT A TANK. THIS INFORMATION INCLUDES THE CURRENT LOCATION OF THE TANK. THE RECORDS ARE TO BE PROCESSED IN ORDER OF THE TANKS' PROXIMITY TO A GIVEN LOCATION (CLOSEST TANK FIRST).

APPROACH 1:

```
type Tank_Status_Type is
  record
    ...
    Location_Part : Location_Type;
    ...
  end record;

type Tank_Status_List_Type is
  array (Positive range <>) of Tank_Status_Type;

procedure Sort_By_Distance (Tanks      : in out Tank_Status_List_Type;
                           Reference_Point : in Location_Type);
```


INSTRUCTOR NOTES

THE ALTERNATIVE IS TO DEFINE Tank_Status_List_Type VALUES TO BE ARRAYS OF POINTERS TO RECORDS, RATHER THAN ARRAYS OF RECORDS.

THEN Tanks (1).Location_Part REFERS TO A COMPONENT OF THE RECORD POINTED TO BY Tanks (1) RATHER THAN A COMPONENT OF Tanks (1).

IN MOST IMPLEMENTATIONS, ACCESS VALUES ARE SMALL AND ACCESS VALUE ASSIGNMENTS ARE QUITE FAST.

3. USE OF ACCESS VALUES TO AVOID MOVING LARGE AMOUNTS OF DATA (Continued)

APPROACH 2:

```
type Tank_Status_Pointer_Type is
  access_Tank_Status_Type;

type Tank_Status_List_Type is
  array (Positive range <>) of Tank_Status_Pointer_Type;

procedure Sort_By_Distance (Tanks      : in out Tank_Status_List_Type;
                             Reference_Point : in Location_Type);

-- ACCESS VALUES ARE SORTED IN ORDER OF DISTANCE FROM THE REFERENCE POINT
-- OF THE TANKS WHOSE RECORDS THEY POINT TO.
```

ADVANTAGE: WHEN Sort_By_Distance EXCHANGES TWO COMPONENTS OF A
 Tank_Status_List_Type ARRAY, IT MOVES ACCESS VALUES INSTEAD OF LARGE
 RECORD VALUES.

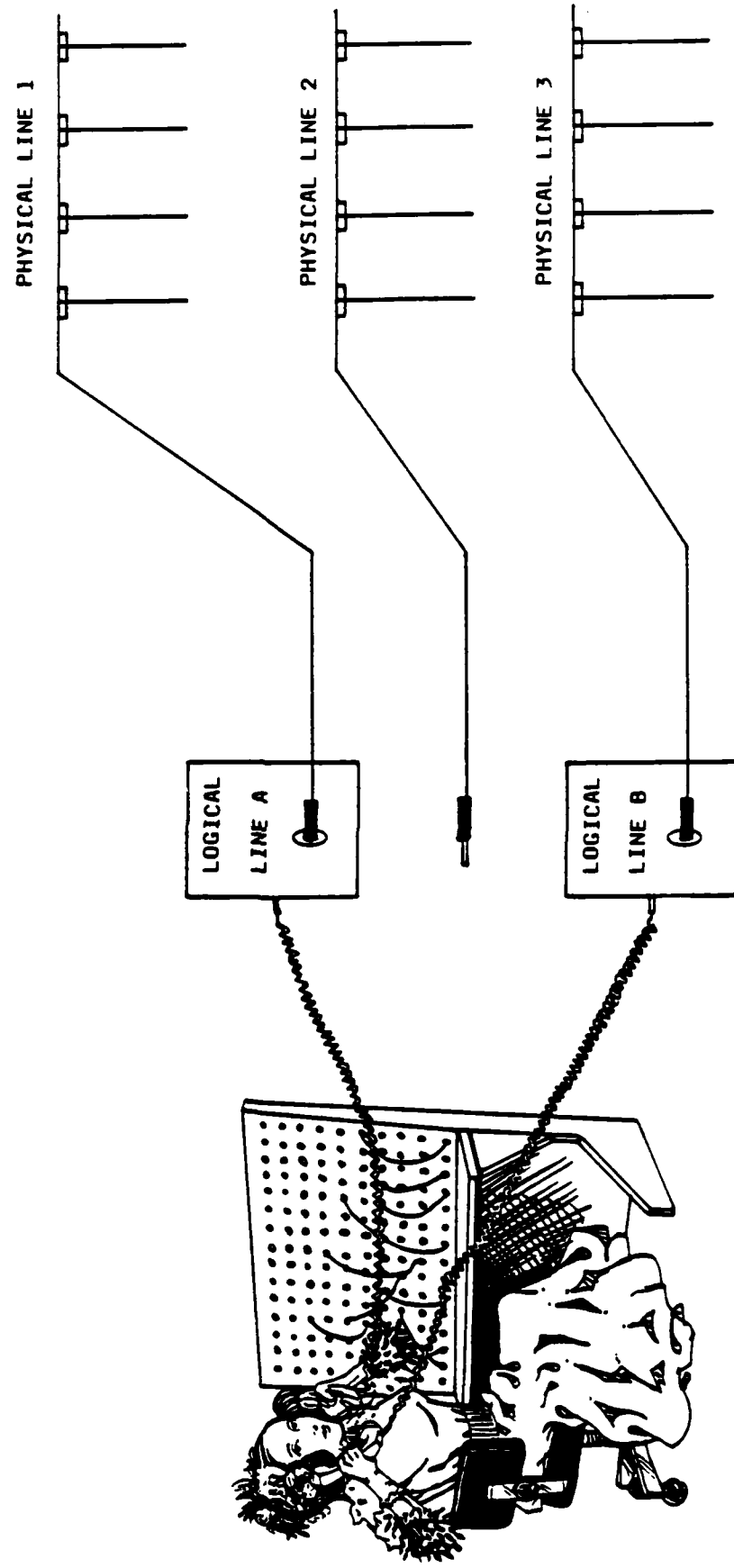
INSTRUCTOR NOTES

A LOGICAL LINE IS ANALOGOUS TO A PHONE NUMBER, A PHYSICAL LINE TO THE WIRE CONNECTED TO THAT PHONE NUMBER.

A MESSAGE SWITCH DETERMINES DYNAMICALLY WHEN A PHYSICAL LINE IS TO PLAY THE ROLE OF TRANSMITTING MESSAGES ADDRESSED TO A GIVEN LOGICAL LINE.

USE THE FOLLOWING SLIDE FOR A GRAPHIC REPRESENTATION.

4. USE OF ACCESS VALUES TO CHANGE, AT RUNTIME, THE VARIABLES ASSOCIATED WITH THE NAME



4. USE OF ACCESS VALUES TO CHANGE, AT RUNTIME, THE VARIABLES ASSOCIATED WITH THE NAME (Continued)

EXAMPLE: A MESSAGE SWITCH

ABSTRACT VIEW: A MESSAGE IS SENT OVER ONE OF A FIXED SET OF "LOGICAL LINES."

IMPLEMENTATION: MESSAGES ADDRESSED TO A GIVEN LOGICAL LINE MAY BE SENT OVER ANY OF SEVERAL "PHYSICAL LINES" AT DIFFERENT TIMES. AT ANY ONE TIME, THERE IS ONE PHYSICAL LINE PLAYING THE ROLE OF A GIVEN LOGICAL LINE.

THE MESSAGE SWITCH MUST MAINTAIN STATUS INFORMATION ABOUT EACH LOGICAL LINE (E.G., THE "NAME" OF THE LINE AND THE MESSAGE IT IS CURRENTLY TRANSMITTING) AND EACH PHYSICAL LINE (E.G., WHETHER THE LINE IS WORKING, HOW NOISY THE LINE IS, AND THE DEVICE ADDRESS OF THE LINE).

4. USE OF ACCESS VALUES TO CHANGE, AT RUNTIME, THE VARIABLES ASSOCIATED WITH THE NAME (Continued)

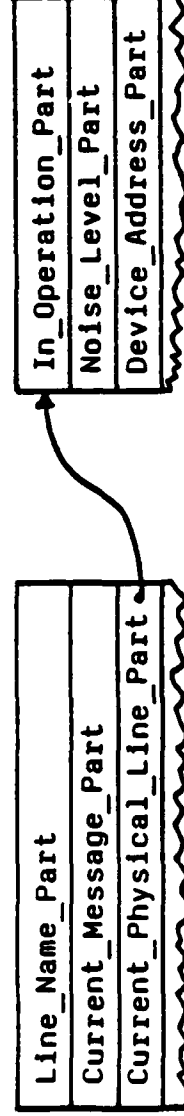
```

type Physical_Line_Status_Type is
record
    In_Operation_Part    : Boolean;
    Noise_Level_Part     : Noise_Level_Type;
    Device_Address_Part  : Device_Address_Type;
    ...
end record;

type Physical_Line_Pointer_Type is access Physical_Line_Status_Type;

type Logical_Line_Status_Type is
record
    Line_Name_Part      : Logical_Line_Name_Type;
    Current_Message_Part : Message_Type;
    Current_Physical_Line_Part : Physical_Line_Pointer_Type;
    ...
end record;

```



MESSAGES ADDRESSED TO LOGICAL LINE X ARE TRANSMITTED USING THE DEVICE ADDRESS X.Current_Physical_Line_Part.Device_Address_Part.

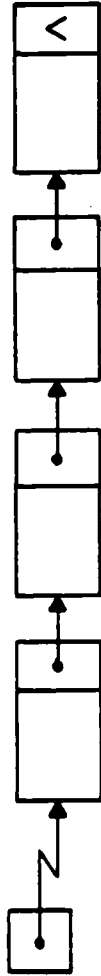
REASSIGNING PHYSICAL LINES REQUIRES ONLY ACCESS VALUE ASSIGNMENTS.

INSTRUCTOR NOTES

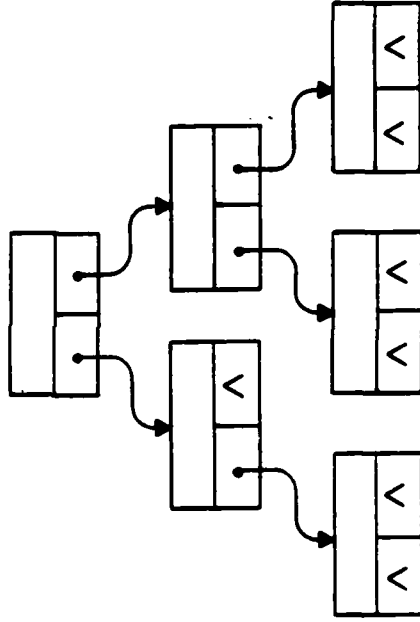
USE THIS SLIDE TO PRESENT THE IDEA OF RECURSIVE DATA TYPES. THE TOPIC WILL BE COVERED
IN L305.

5. USE OF ACCESS VALUES TO DEFINE A DATA TYPE WHOSE OBJECT REFERS TO OTHER OBJECTS OF THE SAME TYPE

LINKED LIST:



BINARY TREE:



INSTRUCTOR NOTES

THIS IS THE FIRST OF THREE FORMS FOR DYNAMICALLY ALLOCATING RECORD OBJECTS WITH DISCRIMINANTS.

WHILE THE ALLOCATED OBJECT IS CONSTRAINED THE ACCESS OBJECT MAY POINT TO OTHER ALLOCATED OBJECTS HAVING DIFFERENT DISCRIMINANT VALUES. THIS WILL BE SHOWN WHEN DISCUSSING THE SECOND FORM.

DYNAMIC ALLOCATION OF RECORDS WITH DISCRIMINANTS - FIRST FORM

NEW

RECORD TYPE NAME

- ONLY ALLOWED IF DISCRIMINANTS HAVE DEFAULT INITIAL VALUES
- ALLOCATES A RECORD OBJECT
 - DISCRIMINANTS ARE GIVEN THEIR DEFAULT INITIAL VALUES
 - NO INITIAL VALUES SPECIFIED FOR OTHER COMPONENTS
- THE ALLOCATED RECORD OBJECT IS CONSTRAINED

INSTRUCTOR NOTES

THE DISCRIMINANT HAS A DEFAULT INITIAL VALUE, SO THE ALLOCATION IS VALID.

EXAMPLE

```
type Aircraft_Identification_Type is
  (Civilian_Aircraft, Military_Aircraft, Enemy_Aircraft, Unknown_Aircraft);
type Threat_Level_Type is (Low_Threat, Medium_Threat, High_Threat);
type Aircraft_Type (Aircraft_Identification: Aircraft_Identification_Type :=
  Unknown_Aircraft) is
  record
    Tracking_Data_Part: Tracking_Data_Type;
    case Aircraft_Identification is
      when Civilian_Aircraft =>
        null;
      when Military_Aircraft =>
        Description_Part : Description_Type;
        when Enemy_Aircraft | Unknown_Aircraft =>
          Threat_Level_Part : Threat_Level_Type;
        end case;
    end record;

  type Aircraft_Pointer_Type is access Aircraft_Type;
  Aircraft_Pointer : Aircraft_Pointer_Type;

  ...

  Aircraft_Pointer := new Aircraft_Type;
  -- Aircraft_Pointer.all is a constrained record object
```

INSTRUCTOR NOTES

THE DISCRIMINANT DOES NOT HAVE A DEFAULT INITIAL VALUE, SO THE ALLOCATION IS ILLEGAL,
SINCE THERE IS NO WAY TO DETERMINE A VALUE FOR THE DISCRIMINANT.

EXAMPLE

```
type Varying_String_Type (Maximum_Length: Positive) is
  record
    Current_Length : Natural;
    Contents       : String (1 .. Maximum_Length);
  end record;

type Varying_String_Pointer_Type is access Varying_String_Type;

Line: Varying_String_Pointer_Type;

...

Line:= new Varying_String_Type;
      -- Illegal since discriminant does not have default initial value
```


INSTRUCTOR NOTES

THIS SLIDE DISCUSSES THE SECOND OF THE TWO FORMS FOR DYNAMICALLY ALLOCATING RECORD OBJECTS WITH DISCRIMINANTS.

NOTE THAT THE AGGREGATE SIMPLIFICATION EXAMPLE ALSO SHOWS HOW TO ALLOCATE A RECORD OBJECT IN A TYPE WHOSE DISCRIMINANTS DO NOT HAVE DEFAULT INITIAL VALUES.

DYNAMIC ALLOCATION OF RECORDS WITH DISCRIMINANTS - SECOND FORM

new RECORD TYPE NAME (EXPRESSION)

- EXPRESSION MUST BELONG TO TYPE SPECIFIED BY RECORD TYPE NAME

- ALLOCATES A RECORD OBJECT

- VALUE OF RECORD OBJECT IS SPECIFIED BY EXPRESSION

- DISCRIMINANT VALUES OBTAINED FROM EXPRESSION

- THE ALLOCATED OBJECT IS CONSTRAINED

- EXPRESSION IS AN AGGREGATE SO COULD WRITE

Line := new Varying_String_Type' ((Maximum_Length => 64));

BUT Ada ALLOWS THIS TO BE WRITTEN SIMPLY AS

Line := new Varying_String_Type'(Maximum_Length => 64);

INSTRUCTOR NOTES

THIS EXAMPLE SHOWS THAT A RECORD TYPE HAVING DEFAULT INITIAL VALUE FOR ITS DISCRIMINANTS MAY BE ALLOCATED USING EITHER FORM.

NOTE THAT ONLY THE ALLOCATED OBJECTS ARE CONSTRAINED. THE ACCESS VARIABLE `Second_Sensor_Reading` MAY POINT TO AN OBJECT WHOSE DISCRIMINANT VALUE IS TRUE OR FALSE.

EXAMPLE

```
type Sensor_Reading_Type (Valid : Boolean := False) is
record
  case Valid is
    when True =>
      Reading_Part : Integer;
    when False =>
      null;
  end case;
end record;

type Sensor_Reading_Pointer_Type is access Sensor_Reading_Type;

Invalid_Reading : Sensor_Reading_Pointer_Type := new Sensor_Reading_Type;

First_Sensor_Reading, Second_Sensor_Reading : Sensor_Reading_Pointer_Type;

...

Second_Sensor_Reading := Sensor_Reading_Type'(Valid => True, Reading_Part => 0);
-- Second_Sensor_Reading.all is constrained

Second_Sensor_Reading := First_Sensor_Reading;

First_Sensor_Reading := Invalid_Reading;
```

INSTRUCTOR NOTES

THIS SLIDE DISCUSSES THE THIRD FORM FOR DYNAMICALLY ALLOCATING RECORD OBJECTS IN RECORD
TYPE WITH DISCRIMINANTS.

AT THE END OF THIS SECTION RESTRICTIONS ON DISCRIMINANT CONSTRAINTS WILL BE GIVEN.

DYNAMIC ALLOCATION OF RECORDS WITH DISCRIMINANTS - THIRD FORM

NEW

RECORD TYPE NAME

DISCRIMINANT CONSTRAINT

RECORD TYPE NAME MUST BE AN UNCONSTRAINED RECORD TYPE

ALLOCATES A RECORD OBJECT

DISCRIMINANTS ARE GIVEN THE INITIAL VALUES SPECIFIED IN

DISCRIMINANT CONSTRAINT

NO INITIAL VALUES ARE SPECIFIED FOR OTHER COMPONENTS

THE ALLOCATED OBJECT IS CONSTRAINED

EXAMPLE

```
type Varying_String_Type (Maximum_Length : Positive) is
  record
    Current_Length : Natural;
    Contents       : String (1 .. Maximum_Length);
  end record;

type Varying_String_Pointer_Type is access Varying_String_Type;

Line : Varying_String_Pointer_Type;

Line_40 : Varying_String_Type (40);

...

Line := new Varying_String_Type (40);

Line.all := Line_40;
```


INSTRUCTOR NOTES

ALLOCATE 3 HOURS OF LECTURE FOR THIS SECTION.

ASSIGN EXERCISES 25 AND 26 OF THE EXERCISE BOOKLET FOR LAB ASSIGNMENTS.

THE OBJECTIVE OF THIS SECTION IS TO REVIEW AND FORMALIZE PROGRAM STRUCTURE, INTRODUCE FORMAL AND ACTUAL PARAMETERS, AND INTRODUCE THE CONCEPT OF SEPARATE COMPILATION.

VG 728.2

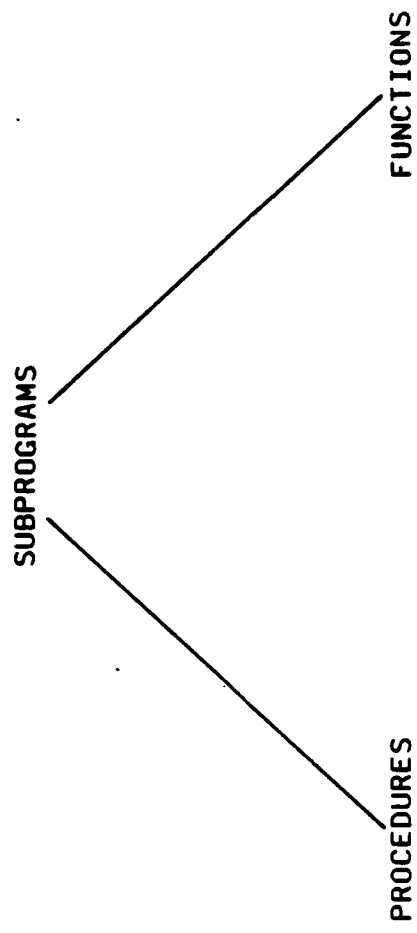
11-1

SECTION 11

PROGRAM STRUCTURE AND SEPARATE COMPILATION

INSTRUCTOR NOTES

"PROCEDURES ARE INVOKED BY PROCEDURE CALL STATEMENTS. FUNCTIONS ARE INVOKED WHEN AN EXPRESSION CONTAINING A FUNCTION CALL IS EVALUATED. FUNCTIONS RETURN VALUES, BUT PROCEDURES DON'T."



INSTRUCTOR NOTES

"MORE RIGOROUS FOR TYPE CHECKING."

SUBPROGRAMS

- ENABLE PROGRAMMERS TO MODULARIZE PROGRAMS
- ENHANCE SOFTWARE READABILITY
- ENHANCE SOFTWARE MAINTAINABILITY
- ANALOGOUS TO THE FORTRAN SUBROUTINES AND FUNCTIONS, HOWEVER, MORE RIGOROUS

INSTRUCTOR NOTES

POINT OUT THE DISTINCTION BETWEEN WHAT SOMETHING DOES AND HOW IT IS DONE.

SUBPROGRAMS

- PROVIDE AN ABSTRACT NAME FOR SOME ALGORITHM OR COMPUTATION
- SEPARATE THE IMPLEMENTATION OF THE ALGORITHM FROM THE SPECIFICATION OF ITS INTERFACE
- AVOID DUPLICATION OF CODE EXECUTED IN MORE THAN ONE PLACE

AD-A166 367

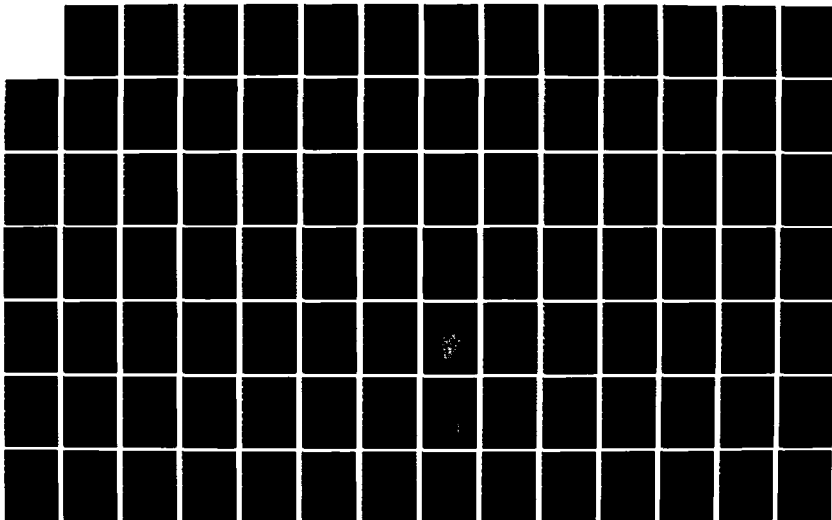
ADA (TRADE NAME) TRAINING CURRICULUM BASIC ADA
PROGRAMMING L202 TEACHER'S GUIDE VOLUME 2(U) SOFTECH
INC WALTHAM MA 1986 DAAB07-83-C-K514

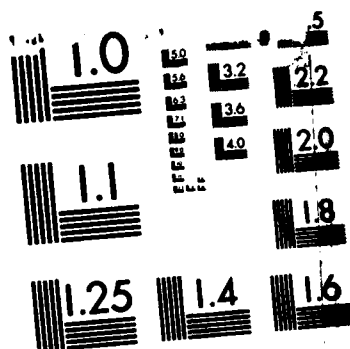
3/8

UNCLASSIFIED

F/G 9/2

NL





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

INSTRUCTOR NOTES

REMIND THE STUDENTS THAT A PROCEDURE IS A SUBPROGRAM.

QUESTION: CAN THE PARAMETER LIST BE LEFT OUT?

ANSWER: YES IF THE FORMAL PARAMETERS HAVE DEFAULT VALUES. (THESE WILL BE COVERED LATER) OR IF THE SUBPROGRAM DOES NOT NEED PARAMETERS.

PROCEDURE

- DEFINES A SEQUENCE OF ACTIONS
- IS INVOKED WITH A PROCEDURE CALL
- A PROCEDURE CALL IS A STATEMENT. THE STATEMENT LOOKS LIKE

Procedure_Name [(Parameter_List)];

INSTRUCTOR NOTES

EXPLAIN THAT A PROCEDURE CALL CAN OCCUR ANYWHERE A STATEMENT CAN (E.G. AFTER THE THEN IN AN IF STATEMENT).

PROCEDURE CALL EXAMPLE

```
with Text_IO; use Text_IO;
with Vector_Services; use Vector_Services;
procedure Compute_Tracking_Data Is
```

```
    Last_Point, Current_Point : Point_Type;
```

```
    Time                      : Time_Type;
```

```
    Velocity                  : Float;
```

```
    ...
```

```
begin -- Compute_Tracking_Data
```

```
    ...
```

```
    Calculate_Velocity (Last_Point, Current_Point, Time, Velocity);
```

```
    ...
```

```
end Compute_Tracking_Data;
```

procedure

call



INSTRUCTOR NOTES

POINT OUT OR HIGHLIGHT NON OPTIONAL PARTS OF A PROCEDURE BODY. I.E. THE begin IS NOT
OPTIONAL.

PROCEDURES

GENERAL FORM OF A PROCEDURE BODY

```
procedure Procedure_Name [(Parameter_List : [mode] Parameter_Type
                           {; Parameter_List : [mode] Parameter_Type})] is
    -- local declarations (if any)
begin
    -- Procedure_Name
    -- local statements
    -- may OPTIONALLY include one or more return statements:
    [return;]
    -- to terminate execution of procedure body and return to caller
end Procedure_Name;
```


INSTRUCTOR NOTES

FOR THE 3 MODES

- in CALLER PASSES VALUE TO CALLED
- out CALLED PASSES VALUE TO CALLER
- in out CALLER PASSES VALUE TO AND ACCEPTS VALUE FROM CALLED.

PARAMETERS

PARAMETERS PROVIDE THE MEANS THROUGH WHICH INFORMATION OR DATA IS TRANSFERRED BETWEEN THE CALLED SUBPROGRAM AND ITS CALLER. THIS TRANSFER MAY WORK IN THE FOLLOWING DIRECTIONS:

CALLER -----> CALLED
CALLER <----- CALLED
CALLER <----> CALLED

THE PARAMETER MODE SPECIFIES WHICH DIRECTIONS ARE ALLOWED:

in
out
in out
(no mode specified) -- equivalent to in

INSTRUCTOR NOTES

1. "out and in out parameters must be some object that could appear on the left-hand side of an assignment (i.e., a variable)."

2. A curious fact about out parameters with which to dazzle and confuse the class:

if the called subprogram doesn't update the value of an out parameter, the value is undefined (even if it was previously defined), e.g.:

```
.
.
.
A := 5.2;
Grunt (A);
-- value of A is now
-- undefined (unless
-- Today is Tuesday)
.
.
.
```

```
procedure Grunt (X : out Float) is
    if Today = Tuesday
    then
        X := 2.3;
    end if;
end Grunt;
```

3. out parameters are write only. you cannot read the value of an out parameter inside the procedure.

```
procedure P (X : out Integer) is
begin -- P
    X := 6;
    if X > 3 then -- ILLEGAL
        null;
    end if;
end P;
```

PARAMETER MODES

in (CALLER ----->CALLED)

- ACT AS LOGICAL INPUTS
- ACT AS A CONSTANT AND MAY NOT BE UPDATED
- THIS MODE ASSUMED IF in IS NOT EXPLICITLY STATED

THUS THE CALLED SUBPROGRAM CANNOT RETURN ANY VALUE THROUGH THE in
PARAMETER TO THE CALLER

out (CALLER <----- CALLED)

- ACT AS LOGICAL OUTPUTS

THUS THE CALLED SUBPROGRAM CANNOT RECEIVE ANY INFORMATION OR DATA
FROM THE CALLER THROUGH AN out PARAMETER. THE CALLED SUBPROGRAM CAN,
HOWEVER, PASS DATA BACK TO THE CALLER BY ASSIGNING A VALUE TO THE out
PARAMETER.

in out (CALLER <-----> CALLED)

- ACT AS VARIABLES WHOSE VALUES MAY BE UPDATED DURING
EXECUTION OF THE PROCEDURE

THUS THE CALLED SUBPROGRAM BOTH RECEIVES DATA FROM THE CALLER AND PASSES
DATA BACK TO THE CALLER THROUGH THE in out PARAMETER

INSTRUCTOR NOTE

REMIND STUDENTS THAT IN PARAMETERS ACT AS LOCAL CONSTANTS.

A TYPICAL ERROR

```
with Text_IO; use Text_IO;
procedure This_Open (Name : in File_Type) is
begin -- This_Open
    Create (Name, Out_File, "Data.dat"); -- ** ILLEGAL
end This_Open;
```

- NAME IS A PARAMETER OF MODE "in"
- CREATE EXPECTS PARAMETER OF MODE "in out"
- ERROR MESSAGE GENERATED INDICATES PARAMETER NO. 1 IN CALL IS NOT A VARIABLE

INSTRUCTOR NOTES

"BASICALLY ANYTHING YOU ARE NOT ALLOWED TO ASSIGN TO NORMALLY CANNOT BE PASSED TO AN OUT OR IN OUT FORMAL PARAMETER."

TO EXPLAIN "PARAMETER TO FUNCTIONS"

procedure Example (X : Integer) is

procedure P(I : in out Integer) is

begin -- P

null;

end P;

begin -- Example

P (X); -- ILLEGAL

end Example;

NOTES

SOME OBJECTS CAN ONLY BE PASSED AS in MODE PARAMETERS,
NAMELY THOSE OBJECTS ONE CANNOT ASSIGN TO:

- CONSTANTS
- in PARAMETERS OF THE CALLING SUBPROGRAM
- LOOP PARAMETERS OF A "for" LOOP
- DISCRIMINANTS

INSTRUCTOR NOTES

POINT OUT THAT THE VALUE RETURNED CAN BE A VALUE OF A COMPOSITE TYPE.

FUNCTION

- DEFINES A COMPUTATION WHICH RETURNS A VALUE
 - VALUE MAY BE OF ANY TYPE (E.G., ARRAY OR RECORD)
 - TERMINATES UPON EXECUTION OF A RETURN STATEMENT
- IS INVOKED WITH A FUNCTION CALL WITHIN AN EXPRESSION
- SYNTAX OF A FUNCTION CALL IS
Function_Name [(Parameter_List)]

INSTRUCTOR NOTES

POINT OUT THAT A FUNCTION CALL IS NOT A STATEMENT. THEREFORE A LOCAL VARIABLE IS REQUIRED TO CAPTURE THE VALUE RETURNED BY THE FUNCTION.

FUNCTION CALL EXAMPLE

```
with Text_IO; use Text_IO;
with Vector_Services; use Vector_Services;
procedure Compute_Tracking_Data Is
    Last_Point, Current_Point : Point_Type;
    Distance                  : Float;
    ...
begin -- Compute_Tracking_Data
    ...
    function call → Distance := Distance_Between (Last_Point, Current_Point);
    ...
end Compute_Tracking_Data;
```

INSTRUCTOR NOTES

POINT OUT (HIGHLIGHT) NON-OPTIONAL ITEMS.

POINT OUT THAT THE PARAMETER MODE MUST BE `in`. (SUGGESTED CONVENTION: WRITE `in` EXPLICITLY FOR PROCEDURES, OMIT IT FOR FUNCTIONS.)

EXPLAIN THAT THE RETURN CLAUSE IN THE SPECIFICATION DEFINES THE TYPE OF THE VALUE RETURNED. THE RETURN STATEMENT IN THE BODY RETURNS THE ACTUAL VALUE.

FUNCTIONS

GENERAL FORM OF A FUNCTION BODY

```
function Function_Name [( Parameter_List : [in] Parameter_Type
    ( ; Parameter_List : [in] Parameter_Type ) ) ] return Type_Name is
    -- local declarations (if any)
begin
    -- Function_Name
    -- local statements
    -- MUST include at least one return statement:
    -- the return statement terminates execution of the subprogram
    -- and specifies the value to be returned
    return Value; -- where Value is of type Type_Name
end Function_Name;
```


EXAMPLE

```
separate (Vector Services)
function Next_Point_After (Last_Point, This_Point : in Point_Type;
    Time_Between_Last, Time_Between_Next : Time_Type)
    return Point_Type is

    Next_Point : Point_Type;

begin -- Next_Point_After

    if Time_Between_Last = 0 then
        return This_Point;
    else
        Next_Point(X) := Last_Point(X) + Float(Time_Between_Next/Time_Between_Last)
            * abs (This_Point(X) - Last_Point(X));
        Next_Point(Y) := Last_Point(Y) + Float(Time_Between_Next/Time_Between_Last)
            * abs (This_Point(Y) - Last_Point(Y));
        return Next_Point;
    end if;

end Next_Point_After;
```


INSTRUCTOR NOTES

JUST ANOTHER EXAMPLE. DISCUSS

- STRUCTURE
- TYPE CONVERSION

ANOTHER EXAMPLE

CONTEXT:

```
type Scores_Type is digits 5 range 0.0 .. 100.0;  
type List_Type is array (1 .. 15) of Scores_Type;
```

EXAMPLE:

```
function Mean (List : in List_Type) return Scores_Type is  
    Sum : Scores_Type := 0.0;  
  
begin -- Mean  
    for I in List'Range loop  
        Sum := Sum + List (I);  
    end loop;  
  
    return Sum / Scores_Type (List'Last);  
  
end Mean;
```

INSTRUCTOR NOTES

"WE MAY CONSIDER A SUBPROGRAM TO HAVE ITS OWN SET OF INTERNAL VARIABLES WHICH IT USES IN ITS CALCULATIONS. THESE ARE THE FORMAL PARAMETERS. THE ACTUAL PARAMETERS ARE USED ONLY UPON ENTRY TO THE SUBPROGRAM (WHEN VALUES ARE COPIED FROM ACTUAL TO FORMAL in AND in out PARAMETERS) AND UPON EXIT (WHEN VALUES ARE COPIED FROM FORMAL TO ACTUAL out AND in out PARAMETERS)." ."

POINT OUT THAT FORMAL PARAMETERS ARE SEPARATED BY SEMICOLONS. ACTUAL PARAMETERS ARE SEPARATED BY COMMAS.

FORMAL AND ACTUAL PARAMETERS

- THE PARAMETERS LISTED IN THE SUBPROGRAM SPECIFICATION ARE CALLED THE FORMAL PARAMETERS. THOSE SPECIFIED IN THE SUBPROGRAM CALL ARE KNOWN AS THE ACTUAL PARAMETERS.
- WHEN THE SUBPROGRAM IS CALLED EACH ACTUAL PARAMETER IS ASSOCIATED WITH A FORMAL PARAMETER. THE TYPE OF THE ACTUAL PARAMETER MUST MATCH THE TYPE OF THE FORMAL PARAMETER.

INSTRUCTOR NOTES

POINT OUT THAT THE SPECIFICATION MAY BE TERMINATED BY A ";" IF IT IS A DECLARATION OR IT MAY BE FOLLOWED BY THE WORD "is" TO INDICATE THAT THE BODY FOLLOWS.

POINT OUT THAT ALL THE CONTEXT IS NOT NECESSARY HERE. THE IDEA IS TO DISTINGUISH BETWEEN FORMAL PARAMETERS AND ACTUAL PARAMETERS.

POINT OUT THIS IS A PROCEDURE.

EXAMPLES

SPECIFICATION:

```
procedure Calculate_Velocity (From, To      : in Point_Type;  
                             In_Time       : in Time_Type;  
                             At_Velocity  : out Float)
```

CALL:

```
Calculate_Velocity (Last_Pt, This_Pt, Time, Velocity);
```

FORMALS:	From	To	In_Time	At_Velocity

ACTUALS:	Last_Pt	This_Pt	Time	Velocity

INSTRUCTOR NOTES

THIS IS FOR A FUNCTION.

ANOTHER EXAMPLE

SPECIFICATION:

function Distance_Between (Point_A, Point_B : in Point_Type) return Float;

CALL:

Distance := Distance_Between (Last_Pt, This_Pt);

FORMALS: Point_A Point_B

ACTUALS: Last_Pt This_Pt

INSTRUCTOR NOTES

POINT OUT THAT THESE ARE ALTERNATIVE NOTATIONS.

ACTUAL PARAMETERS

POSITIONAL NOTATION:

PARAMETERS MUST BE LISTED IN THE SAME ORDER AS THE FORMAL PARAMETERS

NAMED NOTATION:

THERE IS AN EXPLICIT ASSOCIATION BETWEEN THE FORMAL AND ACTUAL
PARAMETERS

EXAMPLE

SPECIFICATION:

```
procedure Calculate_Velocity (From, To : in Point_Type;  
    In_Time : in Time_Type;  
    At_Velocity : out Float)
```

CALL:

POSITIONAL

```
Calculate_Velocity (Last_Pt, This_Pt, Time, Velocity);
```

NAMED

```
Calculate_Velocity (From => Last_Pt, To => This_Pt,  
    In_Time => Time, At_Velocity => Velocity);
```


EXAMPLES

SPECIFICATION:

```
function Distance_Between (Point_A, Point_B : in Point_Type) return Float;
```

CALL:

POSITIONAL

```
Distance := Distance_Between (Last_Pt, This_Pt);
```

NAMED

```
Distance := Distance_Between (Point_A => Last_Pt,  
                               Point_B => This_Pt);
```

INSTRUCTOR NOTES

USE THIS FOIL AS A SUMMARY.

NOTES:

- IN NAMED NOTATION, ACTUAL PARAMETERS MAY BE SPECIFIED IN A DIFFERENT ORDER THAN THE FORMAL PARAMETER LIST.

EXAMPLE:

```
Calculate_Velocity (At_Velocity => Velocity,  
                   From      => Last_Pt,  
                   In_Time   => Time,  
                   To        => This_Pt);
```

- ADVANTAGES OF NAMED NOTATION
 - AVOIDS ERRORS IN PARAMETER POSITION (THE CAUSE OF INSIDIOUS PROGRAM BUGS)
 - IDIOMATIC: IF FORMAL PARAMETERS ARE APPROPRIATELY NAMED THEN THE SUBPROGRAM CALL READS LIKE A SENTENCE.

EXAMPLE:

```
Push ( Element => X, On_To => Stack);
```

- DISADVANTAGES OF NAMED NOTATION
 - LENGTH: SOMETIMES DEDUCTS FROM READING THE PROCEDURE CALL AS A SINGLE ENTITY

SOME GUIDELINES

A SUBPROGRAM REQUIRING A SMALL NUMBER (3 OR LESS) OF PARAMETERS SHOULD PROBABLY
USE A POSITIONAL CALL UNLESS THE ORDER, IS VERY CONFUSING (LIKE A ROUTINE
Move(A,B) WHERE THERE IS CONFUSION: DOES A MOVE TO B? OR DOES B MOVE TO A?)

1980 1981 1982 1983 1984 1985 1986 1987 1988 1989 1990 1991 1992 1993 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014 2015 2016 2017 2018 2019 2020

VG 728.2

11-24i

[illegible]

VG 728.2

1980 1981 1982 1983 1984 1985 1986 1987 1988 1989 1990 1991 1992 1993 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014 2015 2016 2017 2018 2019 2020

11-24i

OPTIONAL ARGUMENTS AND DEFAULT VALUES

- OPTIONAL ARGUMENTS OCCUR WHEN, IN THE SUBPROGRAM SPECIFICATION, A DEFAULT INITIAL VALUE IS SPECIFIED FOR ONE OR MORE PARAMETERS. FOR EXAMPLE:

```
procedure Set_Threshold (Lower: Threshold_Type := 60;  
                        Upper: Threshold_Type := 70) is
```

```
...  
begin -- Set_Threshold  
...  
end Set_Threshold;
```

- WHENEVER THE PROCEDURE Set_Threshold IS CALLED, THE VARIABLES Lower AND Upper WILL HAVE VALUES OF 60 AND 70 RESPECTIVELY, UNLESS OTHER VALUES ARE SPECIFIED IN THE PROCEDURE CALL. IN THE FOLLOWING CALL, THE VALUE 50 OVERRIDES THE LOWER INITIAL VALUE OF 60. BECAUSE THE DEFAULT VALUE IS TO BE USED FOR Upper, IT NEED NOT BE GIVEN IN THE PARAMETER LIST:

```
Set_Threshold ( Lower => 50 );
```

- DEFAULT VALUES MAY BE GIVEN FOR IN PARAMETERS ONLY
- A SUBPROGRAM WITH OPTIONAL ARGUMENTS SHOULD USE NAMED NOTATION.

[illegible]

- 11-25i

DO WE NEED BOTH DECLARATION AND BODY ?

- BOTH A DECLARATION AND BODY MAY APPEAR FOR A GIVEN SUBPROGRAM. IN THIS CASE THE SPECIFICATION IN THE BODY MUST MATCH THE SPECIFICATION IN THE DECLARATION
- ALTERNATIVELY THE BODY MAY APPEAR ALONE. IN THIS CASE THE BODY SERVES AS BOTH DECLARATION AND BODY, DEFINING BOTH THE INTERFACE WITH THE CALLER AND THE IMPLEMENTATION OF THE PROCEDURE

INSTRUCTOR NOTES

THE BODY OF Average IS IN THE BODY OF Vector_Package.

EXAMPLE: SUBPROGRAM SPECIFICATION APPEARING IN A DECLARATION.

```
package Vector_Package is
  type Vector_Type is array (Integer range <>) of Integer;
  function Average (V : Vector_Type) return Float;
end Vector_Package;
```

EXAMPLE: SUBPROGRAM SPECIFICATION APPEARING AS PART OF THE BODY.

```
procedure Calc_Average is
  type Vector_Type is array (Integer range <>) of Integer;
  function Average (V : Vector_Type) return Float is
    Sum : Integer := 0;
  begin -- Average
    for I in V'Range
      loop
        Sum := Sum + V (I);
      end loop;
    return Float(Sum)/Float(V'Length);
  end Average;
begin -- Calc_Average
  ...
end Calc_Average;
```


INSTRUCTOR NOTES

POINT OUT THE CASES WHERE YOU WOULD WANT ONLY THE DECLARATIONS. I.E. IN PACKAGE SPECIFICATIONS AND IN SEPARATE SPECIFICATIONS.

INTRODUCE THIS IN THE FRAMEWORK OF PACKAGES. THE STUDENTS BY THE END OF THIS COURSE SHOULD UNDERSTAND WHY AND HOW TO IMPORT PACKAGES. EXPLAIN THAT SUBPROGRAM DECLARATIONS ARE ONE OF THE THINGS THEY WILL SEE THERE. OTHER THINGS ARE TYPE DECLARATIONS (SO THAT'S WHY WE'VE SPENT HOURS ON END ...)

SPECIFICATION AND DECLARATION

- THE PART OF A SUBPROGRAM BODY PRECEDING THE WORD `is` IS CALLED A SUBPROGRAM SPECIFICATION

EXAMPLE:

```
procedure Calculate_Velocity (From, To : in Point_Type;  
    In_Time : in Time_Type  
    At_Velocity : out Float) is  
  
begin -- Calculate_Velocity  
    ...  
end Calculate_Velocity;
```

- A SUBPROGRAM DECLARATION IS A SUBPROGRAM SPECIFICATION FOLLOWED BY A SEMICOLON ("`;`"). IT DEFINES THE INTERFACE OF THE SUBPROGRAM WITH ITS CALLER

EXAMPLE:

```
procedure Calculate_Velocity (From, To : in Point_Type;  
    In_Time : in Time_Type  
    At_Velocity : out Float);
```


PROCEDURES

GENERAL FORM OF A PROCEDURE DECLARATION

```
procedure Procedure_Name [(Parameter_List : [mode] Parameter_Type  
    {; Parameter_List : [mode] Parameter_Type } )];
```

INSTRUCTOR NOTES

STRESS THAT AS A GENERAL RULE THE PARAMETER MODE SHOULD BE SPECIFIED TO DICTATE ACTUAL USE OF THE PARAMETER. IN OTHER WORDS DON'T MAKE ALL PARAMETERS IN OUT JUST BECAUSE IT IS EASIER NOT TO DECIDE WHICH MODE THE PARAMETER SHOULD BE. THIS METHOD OF DESIGN WILL CAUSE PROBLEMS LATER.

EXAMPLES

PROCEDURE DECLARATIONS

```
procedure Find_First_RI (Message: in Message_Type; Pos: out Position;  
                        Status : out RI_Status_Type);
```

```
procedure Make_Line_Available (Physical_Line: in Physical_Line_Type);
```

```
procedure Draw_Line (Point_1, Point_2 : Point_Type);
```

```
procedure Rewind_Tape;
```

```
procedure Calculate_Median (List : in List_Type;  
                           Midpoint : out Scores_Type);
```


FUNCTIONS

GENERAL FORM OF A FUNCTION DECLARATION

```
function Function_Name [(Parameter_List : [in] Parameter_Type  
    {; Parameter_List : [in] Parameter_Type})] return Type_Name;
```


INSTRUCTOR NOTES

POINT OUT THAT EACH DECLARATION DEFINES A BLACK BOX ACTIVITY.

EXAMPLES

FUNCTION DECLARATIONS

```
function Read_Character_Count (Segment: Segment_Pointer_Type) return Natural;  
  
function Valid_Physical_Line (Logical_Line : Logical_Line_Type;  
    Physical_Line : Physical_Line_Type) return Boolean;  
  
function Read_Segment_Number (Segment: Segment_Pointer_Type) return Segment_Number;  
  
function Hardware_Clock_Reading return Hardware_Clock_Time;  
  
function Is_Line_Available (Logical_Line: Logical_Line_Type) return Boolean;  
  
function Is_Line_To_Be_Intercepted (Logical_Line : Logical_Line_Type;  
    This_Precedence : Precedence_Type;  
    This_Medium : Medium_Type) return Boolean;  
  
function Mean (List : in List_Type) return Scores_Type;
```


LOCAL VS. GLOBAL DATA

- VARIABLES DECLARED WITHIN A SUBPROGRAM ARE KNOWN AS LOCAL VARIABLES:

procedure P is

 Local : Integer; -- local variable

begin -- P

 ...

end P;

Local only known here

- VARIABLES KNOWN WITHIN A SUBPROGRAM BUT DEFINED BY THE CALLER (INSTEAD OF LOCALLY) ARE KNOWN AS GLOBAL VARIABLES.

- ASSIGNMENT TO GLOBAL VARIABLES (SIDE EFFECTS) IS ALLOWED WITHIN BOTH FUNCTIONS AND PROCEDURES
- SUCH PRACTICES ARE FROWNED UPON: THEY ARE THE MAINTENANCE PROGRAMMER'S NIGHTMARE.
- VALUES YOU WISH TO CHANGE SHOULD BE PASSED AS in out PARAMETERS

INSTRUCTOR NOTES

RULE OF THUMB FOR NESTING

- 1) IF THE SUBPROGRAM IS USEFUL ONLY TO ITS PARENT, NEST IT (OR PLACE BOTH IT AND ITS PARENT IN A PACKAGE).
- 2) IF IT IS USEFUL IN SEVERAL PLACES, UNNEST IT. (WE WILL DISCUSS UNNESTING SOON).

REMEMBER NESTING ALLOWS, BUT DOES NOT REQUIRE A PROGRAM HIERARCHY.

NESTING

SUBPROGRAM DECLARATIONS MAY BE NESTED INSIDE ANOTHER SUBPROGRAM.
IT IS NOT RECOMMENDED TO WRITE DEEPLY NESTED SUBPROGRAMS.

EXAMPLE:

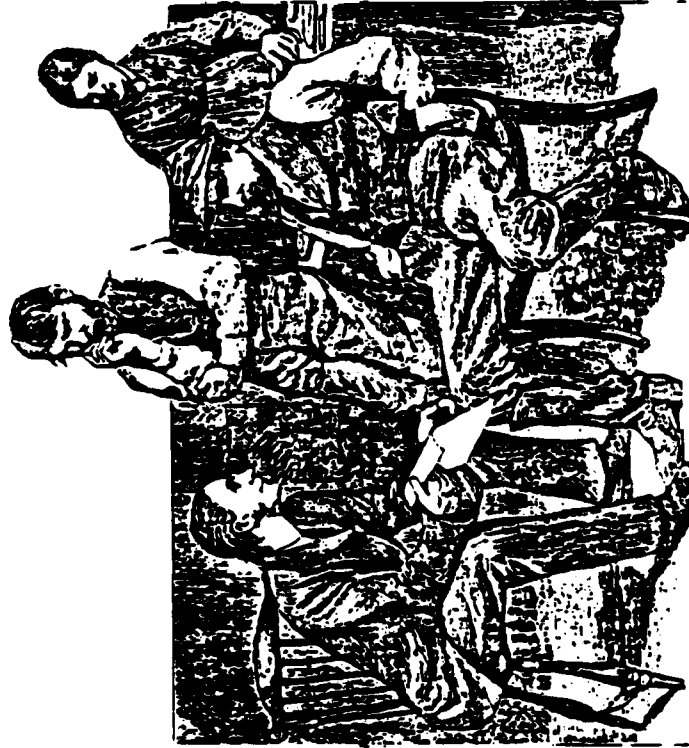
```
procedure Calculate_Median (List : in List_Type;  
                           Midpoint : out Scores_Type) is  
  Temp_List : List_Type := List;  
  ---  
  --- other local declarations  
  ---  
  procedure Sort (Some_List : in out List_Type) is  
    --  
    -- local declarations  
    --  
    begin -- Sort  
      --  
      -- the sort algorithm  
      --  
      end Sort;  
    begin -- Calculate_Median  
      Sort (Temp_List);  
      --  
      -- remaining statements to obtain median  
      --  
      end Calculate_Median;
```

INSTRUCTOR NOTES

"RECALL THAT A PRINCIPAL GOAL OF Ada IS TO SUPPORT 'PROGRAMMING IN THE LARGE'. LARGE PROJECTS ARE UNDERTAKEN BY MANY PERSONS WORKING IN TEAMS. IT IS ESSENTIAL THAT THEY BE ABLE TO WRITE, COMPILE AND TEST MODULES INDEPENDENTLY. AT THE SAME TIME INTERFACES MUST BE WELL-DEFINED TO ENSURE COMPATIBILITY BETWEEN MODULES."

SEPARATE COMPILATION IS A DESIGN ISSUE. POINT OUT DIFFERENCES FROM PASCAL, WHERE ALL SUBPROGRAMS ARE NESTED INSIDE ONE MAIN PROGRAM.

PROGRAMMING IN THE LARGE



TAKING ADVANTAGE OF THE POWER OF
SEPARATE COMPILATION DEPENDS ON
AN UNDERSTANDING OF:

- Ada COMPILATION UNIT
STRUCTURE
- AND
- Ada COMPILATION ORDER

INSTRUCTOR NOTES

CURRENT THOUGHTS ON PROGRAM METHODOLOGY - NEED BOTH BOTTOM-UP AND TOP-DOWN DESIGN.

Ada WAS DESIGNED TO REDUCE SOFTWARE COSTS. NEED REUSABLE SOFTWARE PACKAGES, OFF THE SHELF SOFTWARE COMPONENTS (HELPS WHEN BUILDING SYSTEM BOTTOM-UP).

ADA PROVIDES TWO MECHANISMS FOR SEPARATE COMPILATION

- TOP-DOWN

- LARGE COHERENT PROGRAM BROKEN DOWN INTO SEPARATELY-COMPILED SUBUNITS
- SUBUNITS COMPILED AFTER UNIT ON WHICH THEY DEPEND
- MECHANISM IMPLEMENTED USING "...is separate" AND "separate (...)" NOTATION

- BOTTOM-UP

- TYPICAL APPLICATION IS A "LIBRARY" OF SUBPROGRAMS WRITTEN FOR GENERAL USE (PACKAGES)
- THESE ARE WRITTEN BEFORE UNITS THAT USE THEM
- UNITS THAT DEPEND ON THEM GET ACCESS VIA with AND use CLAUSES

INSTRUCTOR NOTES

LIBRARY UNITS ARE UNITS THAT CAN BE USED IN WITH CLAUSES. SECONDARY UNITS CANNOT APPEAR IN WITH CLAUSES. USING OVERLAYING FOILS SHOW HOW TO UNNEST THE TWO FUNCTIONS ON PAGE 10-33 (ENTITLED "NESTING").

POINT OUT THAT A use CLAUSE WOULD BE ILLEGAL BECAUSE YOU CAN ONLY use PACKAGES.

SEPARATE COMPILATION

- LARGE Ada PROGRAMS MAY BE BROKEN INTO PIECES WHICH ARE COMPILED SEPARATELY
- A COMPILATION CONSISTS OF ONE OR MORE COMPILATION UNITS WHICH ARE SUBMITTED TOGETHER TO THE Ada COMPILER
- COMPILATION UNITS INCLUDE:
 - package declarations
 - subprogram declaration
 - subprogram body
 - package body
 - subunit

INSTRUCTOR NOTES

SUBUNIT? EXCUSE ME,
BUT I DON'T BELIEVE
WE'VE DISCUSSED SUB-
UNITS, HAVE WE?



INSTRUCTOR NOTES

EMPHASIZE THAT STUBBING AND SUBUNITS ARE INDIVISABLE.

POINT OUT THAT IT REPRESENTS A MECHANISM FOR TOP DOWN DEVELOPMENT OF LARGE SYSTEMS USING
TEAMS OF PROGRAMMERS.

SUBUNITS

- IN ADDITION TO BREAKING AN Ada PROGRAM INTO SEPARATELY COMPILED LIBRARY UNITS, ANOTHER SEPARATE COMPILE CAPABILITY IS AVAILABLE BY USING BODY STUBS AND SUBUNITS. THIS IS THE "TOP-DOWN" APPROACH.
- AT THE POINT WHERE A SUBPROGRAM BODY OR PACKAGE BODY WOULD NORMALLY APPEAR IN A COMPILATION, A BODY STUB MAY BE USED INSTEAD

```
procedure Subprogram_Name is separate;
```

THIS IMPLIES THAT THE ACTUAL BODY WILL BE SUPPLIED IN A SEPARATE SUBUNIT.
- THE BODY IS SUPPLIED WITH A PREFIX INDICATING OR NAMING THE COMPILE UNIT WHERE THE CORRESPONDING BODY STUB APPEARED

```
separate (Parent_Unit)      -- note no semicolon
procedure Subprogram_Name is -- body
```
- ALTHOUGH THE SUBUNIT IS SEPARATELY COMPILED THE EFFECT IS EXACTLY AS IF THE ACTUAL BODY WERE GIVEN AT THE POINT OF THE BODY STUB

INSTRUCTOR NOTES

STUBBING IS USEFUL FOR TWO REASONS:

- 1) ALLOWS PROJECTS TO BE SPLIT AMONG SEVERAL PROGRAMMERS, ALLOWING THEM TO
 COMPILE THEIR OWN CODE.
- 2) IT INCREASES READABILITY. NESTED SUBPROGRAMS CAN BE STUBBED OUT. ONLY THE
 SPECIFICATIONS ARE LEFT, WHICH MAKES THE ENCLOSING PROCEDURE MORE READABLE.

STUBBING

NESTED

```
procedure Calculate_Median (...) is
--
-- local declarations
--
procedure Sort (...) is
--
-- local declarations
--
begin -- Sort
...
end Sort;
begin -- Calculate_Median
...
end Calculate_Median;
```

STUB

```
procedure Calculate_Median (...) is
--
-- local declarations
--
procedure Sort (...) is separate;
begin -- Calculate_Median
...
end Calculate_Median;
```

SUBUNIT

```
separate (Calculate_Median)
procedure Sort (...) is
...
begin -- Sort
...
end Sort;
```

INSTRUCTOR NOTES

A PROGRAM LIBRARY IS NOT A LIBRARY OF PROGRAMS, BUT A LIBRARY FOR A PARTICULAR PROGRAM. IT CONTAINS THE DECLARATIVE INFORMATION FOUND IN THE PROGRAM'S LIBRARY UNITS, INFORMATION THAT OTHER UNITS COMPILED LATER MAY NEED ACCESS TO. THIS INCLUDES THE INFORMATION IN PACKAGE DECLARATIONS AND SUBPROGRAM DECLARATIONS. THE COMPILER DETERMINES THE LEGALITY AND MEANING OF A CALL ON A PREVIOUSLY COMPILED SUBPROGRAM, E.G., BY CONSULTING THE PROGRAM LIBRARY.

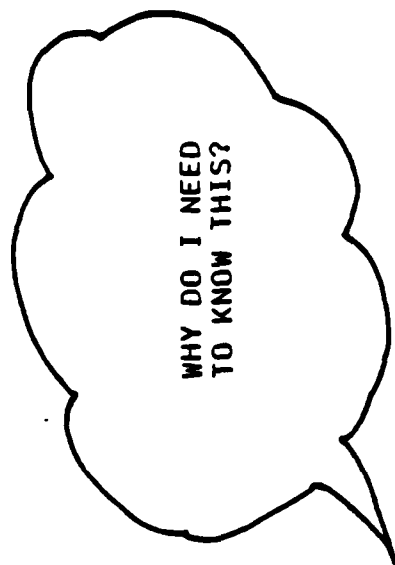
A PROGRAM LIBRARY CONTAINS DECLARATIVE INFORMATION, NOT OBJECT CODE. IT IS LIKE A SYMBOL TABLE FOR THE ENTIRE PROGRAM, BUT ONE WHICH REMAINS INTACT BETWEEN INVOCATIONS OF THE COMPILER.

PROGRAM LIBRARY

- EVERY Ada PROGRAM HAS AN ASSOCIATED PROGRAM LIBRARY
- CERTAIN COMPILATION UNITS (PACKAGE DECLARATIONS, SUBPROGRAM DECLARATIONS, AND SUBPROGRAM BODIES WHEN THERE IS NO CORRESPONDING SUBPROGRAM DECLARATION) ARE LIBRARY UNITS
- IN ESSENCE, LIBRARY UNITS PROVIDE INTERFACE INFORMATION AND OTHER SPECIFICATION DATA NEEDED BY THE COMPILER TO PROCESS OTHER UNITS.
- AS LIBRARY UNITS ARE COMPILED, THEIR DECLARATIONS GO INTO THE PROGRAM LIBRARY
- A PROGRAM LIBRARY CONTAINS DECLARATIONS, NOT OBJECT CODE.
- THE COMPILER USES THE PROGRAM LIBRARY TO PROCESS A REFERENCE TO A PREVIOUSLY COMPILED LIBRARY UNIT.

INSTRUCTOR NOTES

"YOU PROBABLY THINK THIS IS A GOOD QUESTION. IN REALITY YOU NEED TO KNOW THIS IN ORDER TO IMPLEMENT LARGE SYSTEMS. YOU NEED TO KNOW ABOUT COMPILATION ORDER AND INTERFACES IN ORDER TO AVOID WRITING ERRONEOUS PROGRAMS AND DUPLICATING EFFORT."



INSTRUCTOR NOTES

THE use clause could have been added in the last example. Explain its effect.

POINT OUT CONTEXT CLAUSE.

VG 728.2

11-42i

200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769 770 771 772 773 774 775 776 777 778 779 780 781 782 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831 832 833 834 835 836 837 838 839 840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858 859 860 861 862 863 864 865 866 867 868 869 870 871 872 873 874 875 876 877 878 879 880 881 882 883 884 885 886 887 888 889 890 891 892 893 894 895 896 897 898 899 900 901 902 903 904 905 906 907 908 909 910 911 912 913 914 915 916 917 918 919 920 921 922 923 924 925 926 927 928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981 982 983 984 985 986 987 988 989 990 991 992 993 994 995 996 997 998 999 1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1011 1012 1013 1014 1015 1016 1017 1018 1019 1020 1021 1022 1023 1024 1025 1026 1027 1028 1029 1030 1031 1032 1033 1034 1035 1036 1037 1038 1039 1040 1041 1042 1043 1044 1045 1046 1047 1048 1049 1050 1051 1052 1053 1054 1055 1056 1057 1058 1059 1060 1061 1062 1063 1064 1065 1066 1067 1068 1069 1070 1071 1072 1073 1074 1075 1076 1077 1078 1079 1080 1081 1082 1083 1084 1085 1086 1087 1088 1089 1090 1091 1092 1093 1094 1095 1096 1097 1098 1099 1100 1101 1102 1103 1104 1105 1106 1107 1108 1109 1110 1111 1112 1113 1114 1115 1116 1117 1118 1119 1120 1121 1122 1123 1124 1125 1126 1127 1128 1129 1130 1131 1132 1133 1134 1135 1136 1137 1138 1139 1140 1141 1142 1143 1144 1145 1146 1147 1148 1149 1150 1151 1152 1153 1154 1155 1156 1157 1158 1159 1160 1161 1162 1163 1164 1165 1166 1167 1168 1169 1170 1171 1172 1173 1174 1175 1176 1177 1

RESOLUTION

- WHEN A COMPILATION UNIT IS COMPILED INTO A PROGRAM LIBRARY, IT AUTOMATICALLY HAS ACCESS TO THE INFORMATION IN CERTAIN LIBRARY UNITS, E.G.,

- A PACKAGE BODY HAS ACCESS TO ITS SPECIFICATION

- A SUBUNIT HAS ACCESS TO THE UNIT NAMED IN THE separate PREFIX

- IF ACCESS TO OTHER LIBRARY UNITS IS NEEDED, THE COMPILATION UNIT MUST INCLUDE A CONTEXT CLAUSE SPECIFYING THE REQUIRED UNITS:

```
with Text_IO;      -- context clauses
with Vector_Services;
procedure Compute_Tracking_Data is
...
end Compute_Tracking_Data;
```


INSTRUCTOR NOTES

POINT OUT THAT THE ORDER OF COMPILATION IS DEFINED BY THE LRM.

COMPILATION ORDER

- A COMPILATION UNIT MUST BE COMPILED AFTER ALL LIBRARY UNITS NAMED BY ITS CONTEXT
CLAUSE
- A SUBPROGRAM OR PACKAGE BODY MUST BE COMPILED AFTER THE CORRESPONDING SUBPROGRAM
OR PACKAGE SPECIFICATION
- A SUBUNIT MUST BE COMPILED AFTER ITS PARENT COMPILATION UNIT

AS A GENERAL RULE:

A GIVEN UNIT MUST BE COMPILED AFTER ANY UNITS CONTAINING INFORMATION UPON WHICH IT
DEPENDS.

- IF UNIT A MUST BE COMPILED BEFORE UNIT B, THEN IF A IS RECOMPILED B MUST BE
RECOMPILED (UNLESS A SMART COMPILER CAN DETERMINE THAT ANY CHANGES MADE TO A DO
NOT AFFECT B).

INSTRUCTOR NOTES

"NOTE THAT EXECUTABLE STATEMENTS MAY APPEAR INSIDE PACKAGE BODIES. THESE STATEMENTS ARE EXECUTED WHEN THE PACKAGE BODY IS ELABORATED. IN THE CASE OF A LIBRARY PACKAGE, THIS HAPPENS ONCE, BEFORE THE MAIN PROGRAM IS EXECUTED. IN THE CASE OF A PACKAGE DECLARED INSIDE A SUBPROGRAM OR BLOCK STATEMENT, THIS HAPPENS EACH TIME THE SUBPROGRAM OR BLOCK IS ENTERED AND ANOTHER INSTANCE OF THE PACKAGE COMES INTO EXISTENCE. THE INTENDED PURPOSE OF THESE STATEMENTS IS TO INITIALIZE THE VARIABLES IN THE PACKAGE BODY." THE EXECUTION OF THE PACKAGE'S STATEMENTS OCCUR PRIOR TO THE PROGRAM'S STATEMENTS.

SEPARATE COMPILATION EXAMPLE -- LIBRARY UNITS

```
• SINGLE COMPILATION UNIT
  procedure Processor is
    Small : constant := 20;
    Total : Integer := 0;

    package Stock is
      Limit : constant := 1000;
      Table : array (1 .. Limit) of Integer := (1 .. Limit => 0);
      procedure Restart;
    end Stock;

    package body Stock is
      procedure Restart is
        begin -- Restart
          for N in Table'Range
            loop
              Table (N) := 0;
            end loop;
          end Restart;
        end Stock;

      procedure Update (X : Integer) is
        use Stock;
        begin -- Update
          ...
          Table (X) := Table (X) + Small;
          ...
        end Update;
      begin -- Processor
        ...
        Update (10);
        ...
        Stock.Restart; -- reinitializes Table
        ...
      end Processor;
```

INSTRUCTOR NOTES

QUESTION: DO PACKAGE SPECIFICATIONS NEED BODIES?

ANSWER: NO, UNLESS THE PACKAGE SPECIFICATION CONTAINS A DECLARATION (OF A SUBPROGRAM OR PACKAGE) THAT REQUIRES A BODY.

IF TWO PROJECTORS ARE AVAILABLE KEEP 10-44 UP ON DISPLAY.

SEPARATE COMPILATION EXAMPLE

-- LIBRARY UNITS (Continued)

• THREE COMPILATION UNITS

```
package Stock is
  Limit : constant := 1000;
  Table : array (1 .. Limit) of Integer := (1 .. Limit => 0);
  procedure Restart;
end Stock;
```

```
-----
package body Stock is
  procedure Restart is
  begin -- Restart
    for N in Table'Range
    loop
      Table(N) := 0;
    end loop;
    end Restart;
  end Stock;
```

```
-----
with Stock;
procedure Processor is
  Small : constant := 20;
  Total : Integer := 0;
```

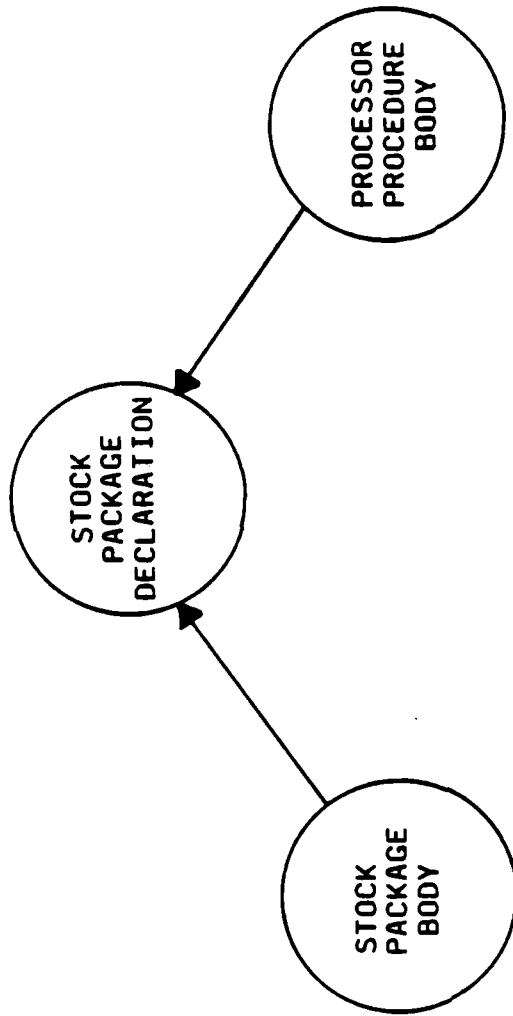
```

  procedure Update (X : Integer) is
  use Stock;
  begin -- Update
    ...
    Table (X) := Table (X) + Small;
    ...
  end Update;
begin -- Processor
  ...
  Update (10);
  ...
  Stock.Restart; -- reinitializes Table
  ...
end Processor;
```

INSTRUCTOR NOTES

IF 2 PROJECTORS ARE AVAILABLE, DISPLAY 10-45 CONCURRENTLY WITH THIS SLIDE.

COMPILATION ORDER DEPENDENCIES



POSSIBLE COMPILATION ORDERS:

1.	STOCK PACKAGE DECLARATION	1.	STOCK PACKAGE DECLARATION
2.	STOCK PACKAGE BODY	2.	PROCESSOR PROCEDURE BODY
3.	PROCESSOR PROCEDURE BODY	3.	STOCK PACKAGE BODY

INSTRUCTOR NOTES

POINT OUT THAT THIS CODE RESIDES IN 1 TEXT FILE.

SEPARATE COMPILATION EXAMPLE --- SUBUNITS

```
• SINGLE COMPILATION UNIT

with Text_IO;
procedure Top is
  type Real is digits 10;
  R, S : Real := 1.0;
  package Facility is
    pi : constant := 3.14159 26536;
    function F (X : Real) return Real;
    procedure G (Y, Z : Real);
  end Facility;
  package body Facility is
    -- some local declarations followed by
    function F (X : Real) return Real is
    begin -- F
      -- sequence of statements of F
      ...
    end F;
    procedure G (Y, Z : Real) is
    -- local procedures using Text_IO
    begin -- G
      -- sequence of statements of G
      ...
    end G;
  end Facility;
  procedure Transform (U : in out Real) is
  use Facility;
  begin -- Transform
    U := F(U);
    ...
  end Transform;
begin -- Top
  Transform (R);
  ...
  Facility.G (R, S);
end Top;
```

INSTRUCTOR NOTES

IF POSSIBLE DISPLAY 10-47 CONCURRENTLY.

POINT OUT USE OF THE SEPARATE CLAUSE.

SEPARATE COMPILATION EXAMPLES

- SUBUNITS (Continued)

- FOUR COMPILATION UNITS

```
procedure Top is
  type Real is digits 10;
  R, S : Real := 1.0;
  package Facility is
    pi : constant := 3.14159 26536;
    function F (X : Real) return Real;
    procedure G (Y, Z : Real);
  end Facility;
  package body Facility is separate;
  procedure Transform (U : in out Real) is separate;

  begin -- Top
    Transform (R);
    ...
    Facility.G (R, S);
  end Top;
```

```
separate (Top)
procedure Transform (U : in out Real) is
  use Facility;
  begin -- Transform
    U := F (U);
    ...
  end Transform;
```

SEPARATE COMPILATION EXAMPLES

FOUR COMPILATION UNITS (Continued)

```
separate (Top)
package body Facility is
    -- some local declarations followed by
    function F (X : Real) return Real is
    begin -- F
        -- sequence of statements of F
        ...
    end F;
    procedure G (Y, Z : Real) is separate;
end Facility;
```

3

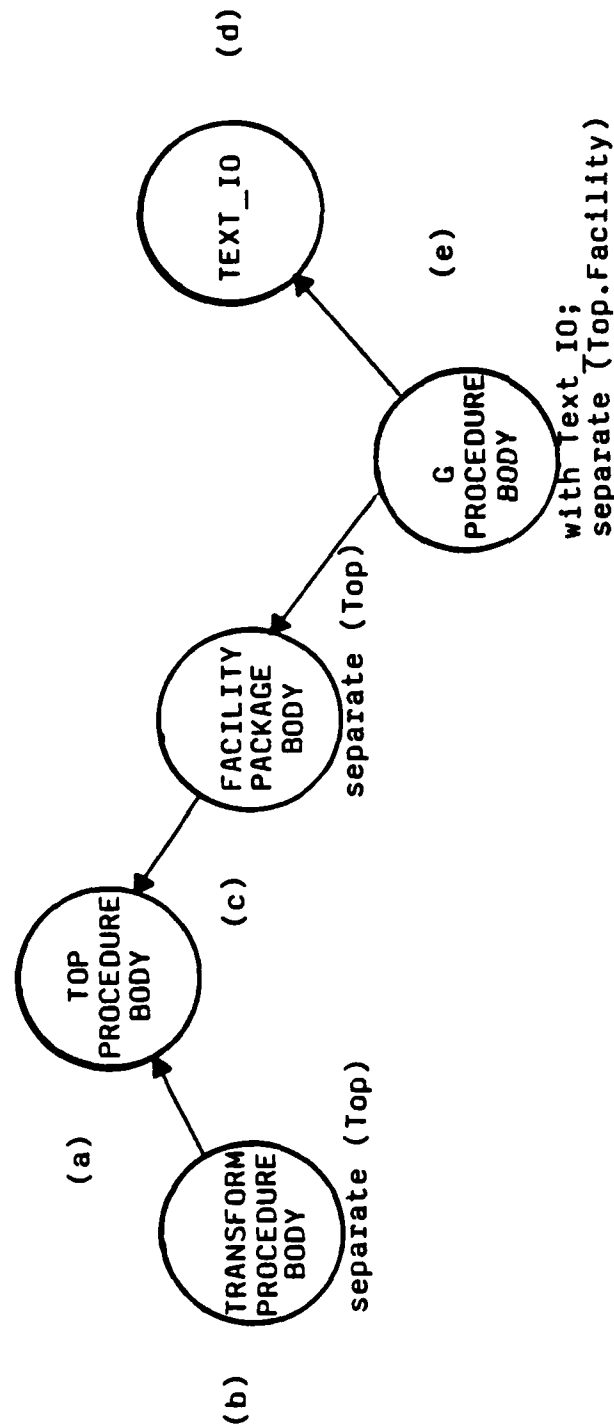
```
-----
with Text_IO;
separate (Top.Facility) -- full name of Facility
procedure G (Y, Z : Real) is
    -- local procedures using Text_IO
    ...
begin -- G
    -- sequence of statements of G
end G;
```

4

INSTRUCTOR NOTES

SPEND SOME TIME DISCUSSING THE VARIOUS POSSIBILITIES AND WHY THESE POSSIBILITIES ARE
LEGAL.

COMPILATION ORDER DEPENDENCIES



POSSIBLE COMPILATION ORDERS:

(A)	(B)	(C)	(D)	(E)	*	(D)	(A)	(B)	(C)	(E)
(A)	(C)	(B)	(D)	(E)		(D)	(A)	(C)	(B)	(E)
(A)	(C)	(C)	(D)	(B)	(E)	(D)	(A)	(C)	(C)	(E)
(A)	(B)	(D)	(D)	(C)	(E)					
(A)	(D)	(B)	(C)	(C)	(E)					
(A)	(D)	(C)	(C)	(B)	(E)					
(A)	(C)	(C)	(D)	(E)	(B)					
(A)	(D)	(C)	(C)	(E)	(B)					

*ACTUALLY, (d) -- Text_IO -- IS "PRE-COMPILED," SO THESE POSSIBILITIES DO NOT ARISE.

INSTRUCTOR NOTES

"AN EXAMPLE OF INFORMATION HIDING IS THE IMPLEMENTATION OF File_Type IN Text_IO. HOW THAT TYPE IS IMPLEMENTED DOES NOT CONCERN (OR INTEREST) THE USERS OF Text_IO. IT IS HIDDEN FROM THEM."

ANOTHER EXAMPLE IS A LIST. OTHER PARTS OF THE PROGRAM MAY NEED TO MANIPULATE LISTS, SO YOU PROVIDE THEM WITH OPERATIONS, BUT NOT WITH THE INTERNAL REPRESENTATION. HELPS MAINTAIN CONSISTENCY.

ASSIGN EXERCISES 25 AND 26 AFTER FINISHING THIS SECTION.

ASSIGN CHAPTER 10 OF THE PRIMER.

SEPARATE COMPILATION PRACTICE

- DECISIONS ON HOW AN Ada PROGRAM SHOULD BE BROKEN INTO SEPARATE COMPILATION UNITS SHOULD BEGIN WITH AN IDENTIFICATION OF LIBRARY PACKAGES.
- THE CRITERIA TO BE USED IN IDENTIFYING LIBRARY PACKAGES INCLUDE:
 - INFORMATION HIDING
 - ACCESS TO COMMON DATA
 - FUNCTIONAL COHESIVENESS
 - SOFTWARE DEVELOPMENT TEAM STRUCTURE

AD-A166 367

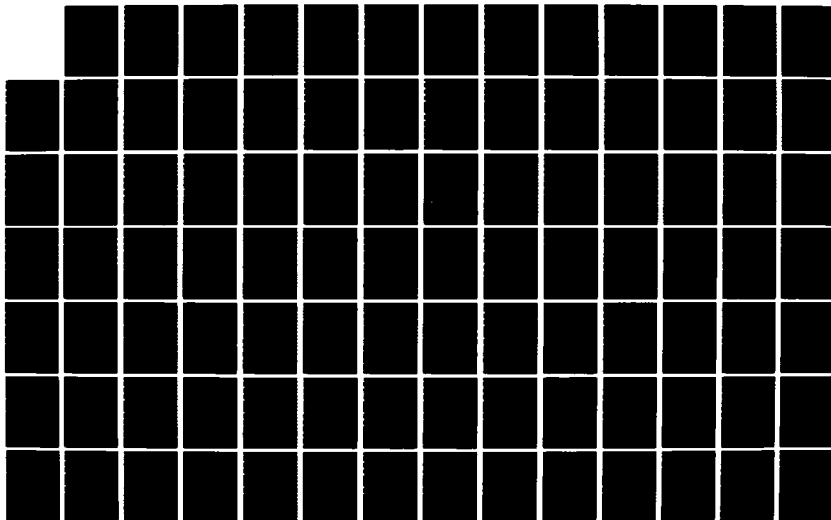
ADA (TRADE NAME) TRAINING CURRICULUM BASIC ADA
PROGRAMMING L202 TEACHER'S GUIDE VOLUME 2(U) SOFTECH
INC WALTHAM MA 1986 DAA007-83-C-K514

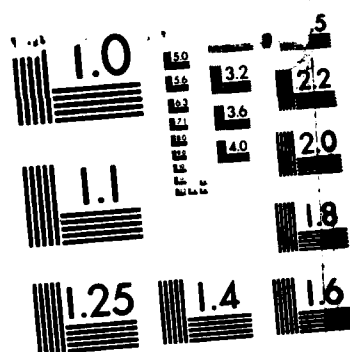
4/8

UNCLASSIFIED

F/G 9/2

NL





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

INSTRUCTOR NOTES

ALLOCATE 3 HOURS OF LECTURE TIME FOR THIS SECTION. AFTER PAGE 28 IS A GOOD BREAKING POINT FOR A 15 MINUTE COFFEE BREAK. ASSIGN EXERCISE 27 OF THE EXERCISE BOOKLET FOR LAB WORK. THE OBJECTIVE OF THIS SECTION IS TO INTRODUCE THE CONCEPT OF SCOPE AND VISIBILITY AND TO INTRODUCE THE STUDENT TO VARIOUS Ada CONSTRUCTS THAT HE/SHE MAY ENCOUNTER IN USING LIBRARY UNITS.

SECTION 12

USING LIBRARY UNITS

INSTRUCTOR NOTES

POINT OUT THAT WITHIN ALL THE PROGRAMS THEY WRITE, THE SCOPE OF ALL OBJECTS, TYPES ETC. THAT ARE DECLARED EXTENDS FROM THAT DECLARATION UNTIL THE END OF THE UNIT IT IS DECLARED IN. THEY ARE ABLE TO MANIPULATE OBJECTS, CALL PROCEDURES, ETC. BECAUSE THOSE IDENTIFIERS ARE IN SCOPE.

SCOPE -- INTUITIVE IDEA

CONTEXT: type Scores_Type is digits 5 range 0.0 .. 100.0;
 type List_Type is array (1 .. 15) of Scores_Type;

EXAMPLE:

function Mean (List : in List_Type) return Scores_Type is
 Sum : Scores := 0.0;

begin -- Mean

 for I in List'Range loop

 Sum := Sum + List (I);

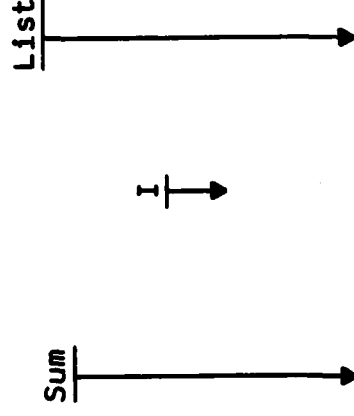
 end loop;

 return Sum / Scores_Type (List'Last);

end Mean;

• SCOPE IS THE AREA IN WHICH AN IDENTIFIER IS KNOWN

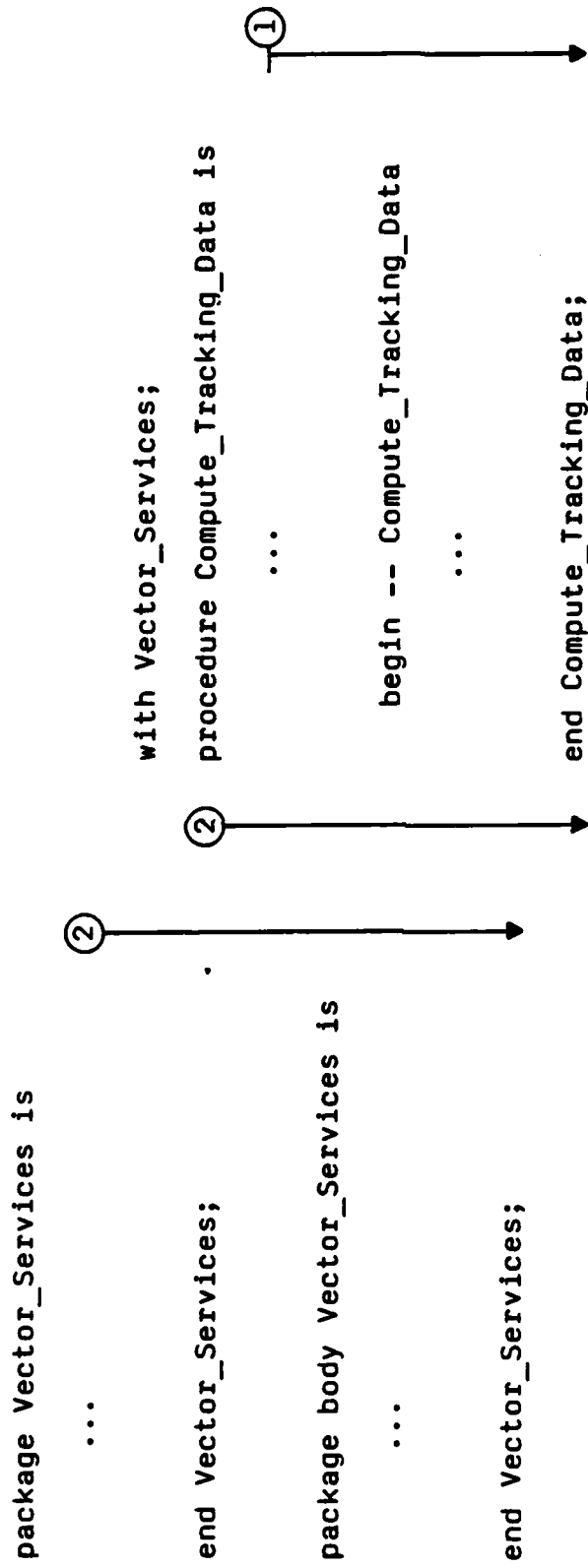
• SCOPE EXTENDS FROM DECLARATION UNTIL END OF UNIT WITHIN WHICH IT IS DECLARED.



INSTRUCTOR NOTES

- ① ANYTHING DECLARED LOCAL TO THE PROCEDURE HAS SCOPE ONLY TO END OF PROCEDURE.
- ② ANYTHING DECLARED IN PACKAGE SPECIFICATION HAS SCOPE WHICH EXTENDS TO THE PACKAGE BODY AND TO ANY UNIT REFERENCING IT.
- ③ ANYTHING DECLARED IN PACKAGE BODY HAS SCOPE WHICH EXTENDS TO PACKAGE BODY ONLY.

SCOPE -- INTUITIVE IDEA

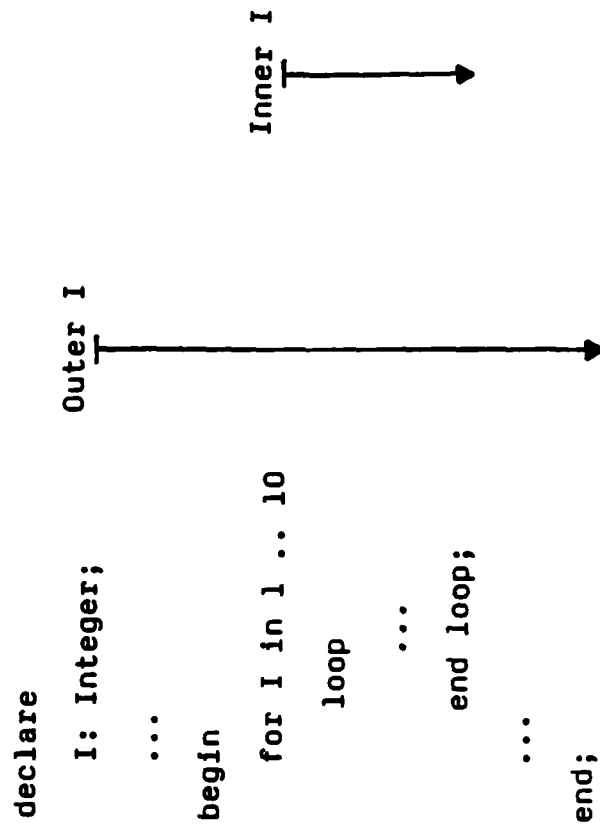


INSTRUCTOR NOTES

POINT OUT THAT THESE TWO I'S ARE DIFFERENT.

SCOPE

- IT IS IMPORTANT TO UNDERSTAND SCOPE



INSTRUCTOR NOTES

POINT OUT THE DISTINCTION BETWEEN WHERE AN ENTITY IS KNOWN AND HOW IT IS REFERENCED.

SCOPE -- DEFINITION

- RECALL THAT A DECLARATION ASSOCIATES AN IDENTIFIER WITH AN ENTITY SUCH AS A VARIABLE, A CONSTANT, A SUBPROGRAM, ETC.
- THE SCOPE OF A DECLARATION IS THE REGION OF PROGRAM TEXT POTENTIALLY INFLUENCED BY THAT DECLARATION.
- WITHIN THE SCOPE OF A DECLARATION AN OCCURRENCE OF THE DECLARED IDENTIFIER MAY BE RECOGNIZED AS A REFERENCE TO THE ENTITY ASSOCIATED BY THE DECLARATION.

INSTRUCTOR NOTES

THESE ARE 2 CONCEPTS RELATED TO SCOPE.

RELATED CONCEPTS

- VISIBILITY

- DETERMINE WHERE WITHIN THE SCOPE THE IDENTIFIER WILL BE RECOGNIZED AS REFERRING TO THE ASSOCIATED ENTITY.

- LIFETIME

- THE DURATION OF PROGRAM EXECUTION DURING WHICH THE ENTITY EXISTS

THESE WILL BE ADDRESSED LATER IN THIS SECTION.

INSTRUCTOR NOTES

THIS IS AN IMPORTANT FOIL. THE SCOPE RULES ARE COMPLEX SO IT IS IMPORTANT FOR THE STUDENT TO UNDERSTAND WHAT BENEFIT THE RULES HAVE.

WHY WORRY ABOUT SCOPE ISSUES

- CONTROL WHAT IS GLOBAL AND WHAT IS NOT
- ALLOW EACH PROGRAMMER TO INVENT HIS OWN NAMES
- ALLOW EACH PROGRAMMER TO INVENT HIS OWN TEMPORARY DATA, BUFFERS, ETC.
- PREVENT A PROGRAMMER'S MODULES FROM INTERFERING WITH ANOTHER PROGRAMMER'S DATA/MODULES NOT INTENDED FOR SHARING

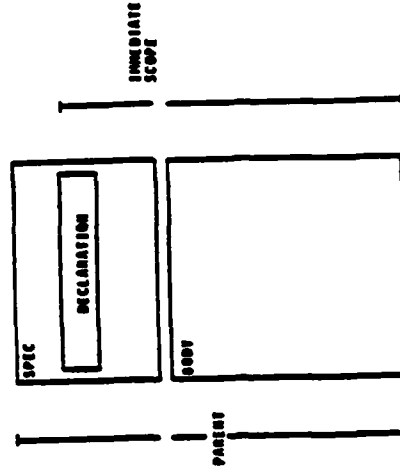
INSTRUCTOR NOTES

DECLARATIVE REGION IS ALMOST SYNONYMOUS WITH IMMEDIATE SCOPE. (IMMEDIATE SCOPE EXCLUDES THE PART OF THE DECLARATIVE REGION UP TO AND INCLUDING THE DECLARATION)

BLOCK STATEMENTS WILL BE DISCUSSED IN THE NEXT SECTION.

IMMEDIATE SCOPE

- THE IMMEDIATE SCOPE OF A DECLARATION EXTENDS FROM THE DECLARATION TO THE END OF THE IMMEDIATELY ENCLOSING DECLARATIVE REGION. THIS IMMEDIATELY ENCLOSING REGION IS REFERRED TO AS THE PARENT OF THE DECLARATION.
- ENCLOSING DECLARATIVE REGIONS INCLUDE:
 - PACKAGE DECLARATIONS
 - SUBPROGRAM DECLARATIONSTOGETHER WITH CORRESPONDING BODIES, IF ANY
- RECORD TYPE DECLARATIONS
- BLOCK STATEMENTS
- LOOP STATEMENTS
- THE IMMEDIATE SCOPE INCLUDES BOTH THE SPECIFICATION AND BODY OF THE PARENT REGION AS IF THEY WERE ONE CONTIGUOUS TEXT REGION.



INSTRUCTOR NOTES

THE PARENT OF Point_Type IS Vector_Services SO THE IMMEDIATE SCOPE OF Point_Type IS THE PACKAGE SPECIFICATION PLUS ITS BODY.

SCOPE - PACKAGE DECLARATIONS

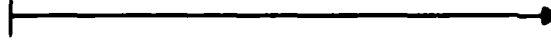
```
package Vector_Services is

    type Coordinate_Type is (X,Y);
    type Point_Type is array (Coordinate_Type) of Float;
    subtype Time_Type is Duration;
    function Distance_Between (Last_Point, This_Point :
                               Point_Type) return Float;

end Vector_Services;

package body Vector_Services is
    ...
    function Distance_Between (Last_Point, This_Point :
                               Point_Type) return Float is
        Dx, Dy : Float;
    begin -- Distance_Between
        Dx := abs (This_Point(X) - Last_Point(X));
        Dy := abs (This_Point(Y) - Last_Point(Y));
        return ( Sqrt (Dx**2 + Dy**2) );
    end Distance_Between;
    ...
end Vector_Services;
```

Immediate Scope
(Point_Type)



INSTRUCTOR NOTES

SCOPE OF D_x & D_y IS THE ENCLOSING DECLARATIVE REGION - THE SUBPROGRAM Distance_Between.

SCOPE - SUBPROGRAM DECLARATION

```

package Vector_Services is
    type Coordinate_Type is (X,Y);
    type Point_Type is array (Coordinate_Type) of Float;
    subtype Time_Type is Duration;
    function Distance_Between (Last_Point, This_Point : Point_Type) return Float;
    ...
end Vector_Services;

package body Vector_Services is
    function Distance_Between (Last_Point, This_Point : Point_Type) return Float is
        Immediate Scope (Dx, Dy)
            Dx, Dy : Float;
        begin -- Distance_Between
            Dx := abs (This_Point(X) - Last_Point(X));
            Dy := abs (This_Point(Y) - Last_Point(Y));
            return ( Sqrt (Dx**2 + Dy**2) );
        end Distance_Between;
    ...
end Vector_Services;

```


INSTRUCTOR NOTES

A LOOP IS AN ENCLOSING DECLARATIVE REGION SO THE SCOPE OF I IS ONLY TO THE END OF THE LOOP.

SCOPE - LOOP STATEMENT

CONTEXT:

```
type Scores is digits 5 range 0.0 .. 100.0;
subtype Scores_Type is Scores range 0.0 .. 1500.0;
type List_Type is array (1 .. 15) of Scores_Type;
```

EXAMPLE:

```
function Mean (List : List_Type) return Scores_Type is
    Sum : Scores := 0.0;
```

```
begin -- Mean
    for I in List'Range loop
        ...
    end loop;
    ...
end Mean;
```

Immediate Scope (I)



INSTRUCTOR NOTES

EXAMPLE - RECORD TYPE DECLARATIONS

```
type Rec is
record
    C1 : Boolean;
    C2 : Character;
    C3 : Integer;
end record;
```

↑ ↓
immediate scope

POINT OUT THAT IF THIS WERE THE ONLY SCOPE, RECORD COMPONENTS COULD NOT BE ACCESSED. BY EXTENDING THE SCOPE TO THE SCOPE OF THE PARENT, ACCESSING RECORD COMPONENTS IS ALLOWED.

STRESS POINT 4. ONLY THESE OBJECTS HAVE EXTENDED SCOPE. ONLY THESE OBJECTS ARE VISIBLE OUTSIDE THE PARENT UNIT.

EXTENDED SCOPE

- HOWEVER SOME DECLARATIONS HAVE AN EXTENDED SCOPE THAT EXTENDS BEYOND THE PARENT.
- THE FULL SCOPE OF A DECLARATION IS THE IMMEDIATE SCOPE PLUS THE EXTENDED SCOPE (IF ANY).
- FOR MOST DECLARATIONS THE FULL SCOPE IS THE SAME AS THE IMMEDIATE SCOPE.
- DECLARATIONS WITH EXTENDED SCOPE INCLUDE:
 - DECLARATIONS IN THE VISIBLE PART OF A PACKAGE
 - SUBPROGRAM PARAMETERS (SO THEY CAN BE USED IN NAMED PARAMETER LISTS ANYWHERE THE SUBPROGRAM CAN BE CALLED)
 - RECORD COMPONENTS (SO THEY CAN BE ACCESSED ANYWHERE THE RECORD CAN BE)
- NOTE THAT DECLARATIONS WITH EXTENDED SCOPE ARE IN A SENSE PROPERTIES OF THE PARENT, SO IT IS REASONABLE THAT THEY HAVE AT LEAST THE SAME SCOPE AS THE PARENT.

INSTRUCTOR NOTES

POINT OUT THAT ITEMS DECLARED IN PACKAGE SPECIFICATIONS HAVE AN EXTENDED SCOPE.

EXTENDED SCOPE - EXAMPLE

- ASSUME NO BODY FOR PACKAGE B.

package A is

...

package B is

Value_1: Integer;

...

end B;

...

end A;

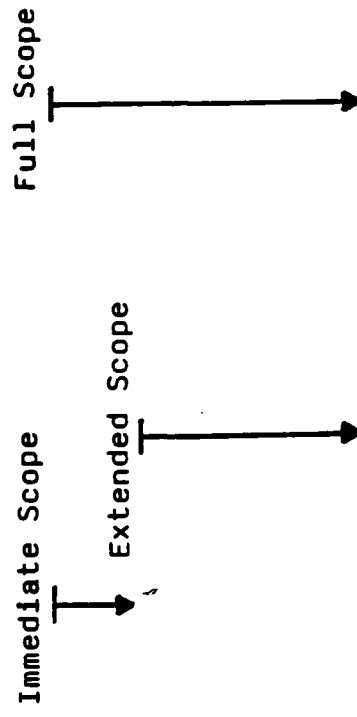
package body A is

...

end A;

- PARENT OF Value_1 IS B

- SCOPE OF Value_1 EXTENDS TO END OF SCOPE OF ENCLOSING DECLARATION (package A)



INSTRUCTOR NOTES

EXTENDED SCOPE - EXAMPLE

- ASSUME B HAS A BODY

package A is

...

package B is

Value_1 : Integer;

end B;

...

end A;

package body A is

package body B is

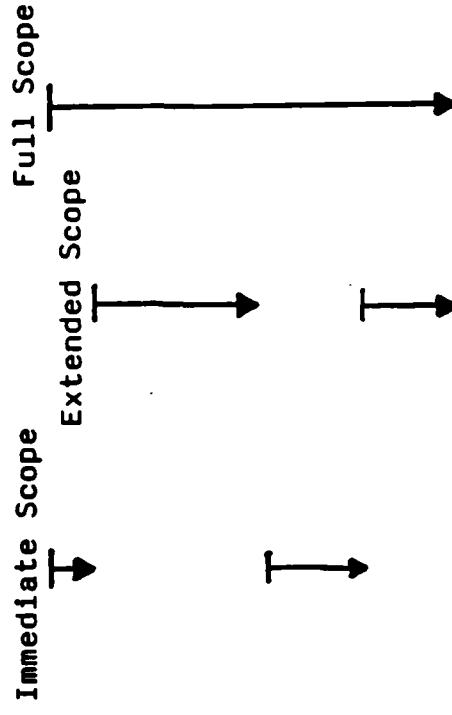
...

end B;

...

end A;

- B IS PARENT OF Value_1



INSTRUCTOR NOTES

POINT OUT THAT IT IS IMPORTANT TO UNDERSTAND SCOPE. IT EXPLAINS WHY THE I'S ARE DIFFERENT IN

declare

I : Integer;

begin

for I in 1 .. 10 loop

end loop;

end;

STRESS THAT ENTITIES IN PACKAGES HAVE EXTENDED SCOPE. ENTITIES IN PROCEDURES DO NOT.

SCOPE -- WRAP UP

- ALTHOUGH THE SCOPE RULES ARE COMPLICATED TO STATE, THEIR IMPLICATIONS ARE STRAIGHTFORWARD:
 - NORMAL DECLARATIONS HAVE A SCOPE THAT INCLUDES ONLY THE CONTAINING REGION (THEY CANNOT BE ACCESSED OUTSIDE OF THAT UNIT)
 - DECLARATIONS VISIBLE OUTSIDE THE PARENT (VISIBLE PACKAGE DECLARATIONS, SUBPROGRAM PARAMETERS, RECORD COMPONENTS) HAVE A LARGER SCOPE

INSTRUCTOR NOTES

"ENTITY EXISTS" - IT IS IN SCOPE

POINT OUT THAT THE ONLY WAY TO ACCESS A VALUE THAT IS NO LONGER IN SCOPE IS TO RE-ENTER ITS SCOPE. (IT WILL PROBABLY HAVE A NEW ADDRESS AND VALUE.)

"ELABORATE" ON THE LIFETIME OF LIBRARY PACKAGES. THEY ARE ELABORATED BEFORE THE MAIN PROGRAM BEGINS, AND REMAIN IN EXISTENCE UNTIL THE MAIN PROGRAM TERMINATES. SINCE THEY ARE, IN ESSENCE, PERMANENT, SO ARE THE ENTITIES DECLARED IN THEM.

THE TERMS USED IN BULLETS 3 AND 4 -- "ENTERING" AND "LEAVING" A SCOPE -- ARE IMPRECISE BUT MAY BE THE BEST WAY TO GET THE INTUITIVE IDEA ACROSS. IF STUDENTS ARE CONFUSED ABOUT "ENTERING" AND "LEAVING" THE SCOPE OF AN ITEM DECLARED IN A PACKAGE, GIVE THE GORY DETAILS:

1. LIBRARY UNITS ARE ELABORATED BEFORE THE MAIN PROGRAM IS EXECUTED. THE DECLARATIONS REMAIN IN EFFECT FOR THE DURATION OF THE MAIN PROGRAM.
2. DECLARATIONS OF BLOCK STATEMENTS OR SUBPROGRAM BODIES ARE ELABORATED BEFORE THE ACCOMPANYING STATEMENTS ARE EXECUTED AND REMAIN IN EFFECT UNTIL THE STATEMENTS HAVE TERMINATED.
3. A DECLARATION IN A PACKAGE SPECIFICATION (BODY) IS ELABORATED WHEN THE PACKAGE SPECIFICATION (BODY) IS ELABORATED. THE DECLARATION REMAINS IN EFFECT AS LONG AS THE DECLARATION OF THE PACKAGE. (IF THE PACKAGE IS A LIBRARY UNIT, USE RULE 1 ABOVE. IF DECLARED IN A SUBPROGRAM OR BLOCK STATEMENT, USE RULE 2 ABOVE. IF NESTED IN AN OUTER PACKAGE, APPLY RULE 3 RECURSIVELY TO THE OUTER PACKAGE.)
4. NEVER MIND TASKS JUST YET!

LIFETIME

- THE LIFETIME OF AN ENTITY IS THE DURATION OF THE TOTAL PROGRAM EXECUTION DURING WHICH THE ENTITY EXISTS (I.E. IT IS IN SCOPE).
- THE LIFETIME OF AN ENTITY IS CONTROLLED BY THE SCOPE OF ITS DECLARATION.
- WHEN THE SCOPE IS ENTERED, THE DECLARATION WILL BE ELABORATED, CAUSING THE DECLARED ENTITY TO BE CREATED AND POSSIBLY INITIALIZED.
- WHEN THE SCOPE IS LEFT, THE ENTITY CEASES TO EXIST (FOR EXAMPLE, ITS STORAGE MAY BE RECLAIMED).
- THE RULES ENSURE THAT WHENEVER CONTROL IS WITHIN THE SCOPE OF AN ENTITY, THAT ENTITY WILL EXIST.
- AN ENTITY DECLARED IN A PACKAGE REMAINS IN EXISTENCE AS LONG AS THE PACKAGE. IN PARTICULAR, A VARIABLE DECLARED IN A PACKAGE BODY RETAINS ITS VALUE BETWEEN CALLS ON THE PACKAGE'S SUBPROGRAMS.
- GLOBAL ENTITIES WHOSE SCOPE IS ESSENTIALLY THE ENTIRE PROGRAM (FOR INSTANCE THOSE DECLARED IN THE MAIN PROGRAMS OR IN THE VISIBLE PART OF A LIBRARY PACKAGE) ARE ESSENTIALLY ALWAYS AROUND, SINCE THEY WILL BE CREATED AT THE BEGINNING OF THE PROGRAM EXECUTION AND WILL LAST UNTIL THE PROGRAM TERMINATES.
- ENTITIES DECLARED IN A SUBPROGRAM EXIST ONLY FOR A SINGLE CALL OF THAT SUBPROGRAM. (NEW ENTITIES WITH THE SAME NAME ARE CREATED THE NEXT TIME THE SUBPROGRAM IS CALLED.)

INSTRUCTOR NOTES

"NOTE THAT EXECUTABLE STATEMENTS MAY APPEAR INSIDE PACKAGE BODIES. THESE STATEMENTS ARE EXECUTED WHEN THE PACKAGE BODY IS ELABORATED." IN THE CASE OF A LIBRARY PACKAGE, THIS HAPPENS ONCE, BEFORE THE MAIN PROGRAM IS EXECUTED. IN THE CASE OF A PACKAGE DECLARED INSIDE A SUBPROGRAM OR BLOCK STATEMENT, THIS HAPPENS EACH TIME THE SUBPROGRAM OR BLOCK IS ENTERED AND ANOTHER INSTANCE OF THE PACKAGE COMES INTO EXISTENCE. THE INTENDED PURPOSE OF THESE STATEMENTS IS TO INITIALIZE THE VARIABLES IN THE PACKAGE BODY. THE EXECUTION OF THE PACKAGE'S STATEMENTS OCCUR PRIOR TO THE PROGRAM'S STATEMENTS.

SEPARATE COMPILATION EXAMPLE -- LIBRARY UNITS

```
• SINGLE COMPILATION UNIT
  procedure Processor is
    Small : constant := 20;
    Total : Integer := 0;

    package Stock is
      Limit : constant := 1000;
      Table : array (1.. Limit) of Integer;
      procedure Restart;
    end Stock;

    package body Stock is
      procedure Restart is
      begin -- Restart
        for N in Table'Range
        loop
          Table (N) := 0;
        end loop;
      end Restart;
    begin -- Stock
      Restart; -- Initialize the package
    end Stock;

    procedure Update (X : Integer) is
    use Stock;
    begin -- Update
      ...
      Table (X) := Table (X) + Small;
      ...
    end Update;

    begin -- Processor
      ...
      Update (10);
      ...
      Stock.Restart; -- reinitializes Table
      ...
    end Processor;
```

INSTRUCTOR NOTES

QUESTION: DO PACKAGE SPECIFICATIONS NEED BODIES?

ANSWER: NO, UNLESS THE PACKAGE SPECIFICATION CONTAINS A DECLARATION (OF A SUBPROGRAM OR PACKAGE) THAT REQUIRES A BODY.

SEPARATE COMPILATION EXAMPLE

-- LIBRARY UNITS (Continued)

• THREE COMPILATION UNITS

```
package Stock is
  Limit : constant := 1000;
  Table : array (1 .. Limit) of Integer;
  procedure Restart;
end Stock;
-----
package body Stock is
  procedure Restart is
  begin -- Restart
    for N in Table'Range
      loop
        Table(N) := 0;
      end loop;
    end Restart;
  begin -- Stock
    Restart; -- Initialize the package
  end Stock;
-----
with Stock;
procedure Processor is
  Small : constant := 20;
  Total : Integer := 0;
  procedure Update (X : Integer) is
    use Stock;
  begin -- Update
    ...
    Table (X) := Table (X) + Small;
    ...
  end Update;
begin -- Processor
  ...
  Update (10);
  ...
  Stock.Restart; -- reinitializes Table
  ...
end Processor;
```


INSTRUCTOR NOTES

"VISIBILITY IS A SUBSET OF SCOPE. IF AN OBJECT IS VISIBLE THEN IT IS IN SCOPE. IF AN OBJECT IS IN SCOPE, IT MAY OR MAY NOT BE VISIBLE."

FOR EXAMPLE: RECORD COMPONENTS ARE IN SCOPE SO LONG AS THE RECORD IS IN SCOPE. RECORD COMPONENTS ARE DIRECTLY VISIBLE ONLY INSIDE THE RECORD TYPE DECLARATION.

VISIBILITY -- DEFINITION

- THE REGION OF VISIBILITY OF A DECLARATION IS NORMALLY THE SAME AS THE SCOPE, THOUGH IN SOME CASES IT MAY BE SMALLER.
- THERE ARE TWO TYPES OF VISIBILITY:
 - DIRECT VISIBILITY IN WHICH THE DECLARED IDENTIFIER MAY BE USED ALONE TO REFER TO THE DECLARED ENTITY.
 - VISIBILITY BY SELECTION IN WHICH THE IDENTIFIER MUST BE PRECEDED BY A QUALIFYING PREFIX OR MUST APPEAR IN A CERTAIN CONTEXT TO BE RECOGNIZED AS AN OCCURRENCE OF THE DECLARED ENTITY.

INSTRUCTOR NOTES

AN EXAMPLE OF POINT 1:

procedure P is

A : Boolean;

A : Integer;

type B is

record

A : Integer;

end record;

function A return Boolean is separate; -- ILLEGAL

function C return Integer is

A : String (1 .. 10); -- LEGAL, NEW DECLARATIVE REGION

begin -- C

return 3;

end C;

begin -- P

null;

end P;

-- ILLEGAL BECAUSE IT IS IN THE SAME DECLARATIVE REGION

-- LEGAL BECAUSE RECORDS START A NEW DECLARATIVE REGION.

DIRECT VISIBILITY

- A DECLARATION IS DIRECTLY VISIBLE ANYWHERE IN ITS IMMEDIATE SCOPE EXCEPT WHERE IT IS HIDDEN BY A DIFFERENT DECLARATION OF THE SAME IDENTIFIER IN AN ENCLOSED DECLARATIVE REGION.
- WHEN A DECLARATION IS DIRECTLY VISIBLE THE IDENTIFIER ALONE MAY BE USED TO REFER TO THE DECLARED ENTITY.
- FOR MOST DECLARATIONS DIRECT VISIBILITY IS THE NORMAL CASE.

```
procedure P is
    A : Boolean := True;
begin -- P
    A := not A; -- A IS DIRECTLY VISIBLE
end P;
```

INSTRUCTOR NOTES

DIRECT VISIBILITY

USE OF A DECLARATION IN AN INNER REGION TO HIDE AN OUTER DECLARATION OF THE SAME IDENTIFIER IS CONFUSING TO THE READER AND SHOULD BE AVOIDED.

Visibility(Outer A) procedure P is

 A : Boolean := True;

 begin -- P

 declare

 A : Boolean := False; -- inner A

 begin

 A := not A;

 -- inner A

 end;

 A := not A;

 -- outer A

 end P;

OUTER A NOT DIRECTLY VISIBLE WITHIN INNER DECLARATIVE REGION.

INSTRUCTOR NOTES

"A NAMED REGION COULD BE A LABELED BLOCK, LABELED LOOP, PACKAGE OR SUBPROGRAM."

EXAMPLE OF NAMED NOTATION:

```
with Text_IO;  
procedure Hello_World is  
begin -- Hello_World  
    Text_IO.Put ("Hello");  
end Hello_World;
```

VISIBILITY BY SELECTION

- A DECLARATION MAY BE MADE VISIBLE BY SELECTION BY PRECEDING THE DECLARED IDENTIFIER WITH A QUALIFYING PREFIX IDENTIFYING THE PARENT REGION CONTAINING THE DECLARATION.

Prefix.Identifier

- THE QUALIFYING PREFIX PROVIDES A SMALL LOCAL WINDOW THROUGH WHICH A DECLARATION IN THE NAMED REGION MAY BE MADE VISIBLE.
- THE MOST USUAL QUALIFYING PREFIX IS JUST THE NAME OF A LIBRARY PACKAGE CONTAINING THE DESIRED DECLARATION.
- IN THE GENERAL CASE THE PREFIX MAY BE A LIST OF NAMES SEPARATED BY DOTS THAT IDENTIFIES A SUCCESSIVELY SMALLER SERIES OF REGIONS.

INSTRUCTOR NOTES

POINT OUT THAT THIS EXAMPLE IS TO ILLUSTRATE VISIBILITY BY SELECTION, IT IS NOT A MODEL
FOR PROGRAMMING PRACTICE.

VISIBILITY BY SELECTION

EXAMPLE:

```
procedure P is
  A : Boolean := True;
begin -- P
  declare
    A : Boolean := False; -- inner A
  begin
    P.A := A;
    end;
    A := not A;
  end P;
```

INSTRUCTOR NOTES

"THE WITH CLAUSE MAKES THE ENTITIES DECLARED IN THE VISIBLE PART OF THE LIBRARY UNIT
VISIBLE BY SELECTION, I.E. YOU MUST QUALIFY THE NAME AS IN

Library_Name.Entity

IF YOU JUST SAID Entity, THE COMPILER WOULD MARK IT AS AN UNDECLARED IDENTIFIER.

with CLAUSES

- TO USE A SEPARATELY COMPILED PROGRAM UNIT IN ANOTHER SEPARATELY COMPILED PROGRAM UNIT, THE SECOND UNIT MUST CONTAIN A with CLAUSE SPECIFYING THE NAME OF THE PROGRAM UNIT BEING USED

```
with Library_unit_name, ..., Library_unit_name;
```
- THE with CLAUSE MAKES THE DECLARATIONS IN THE VISIBLE PART OF THE NAMED PACKAGE VISIBLE BY SELECTION IN THE PROGRAM UNIT.
- THE with CLAUSE MUST APPEAR AT THE BEGINNING OF THE UNIT BEING COMPILED.
- WHY DO YOU NEED THE with CLAUSE? SO THAT READERS KNOW UP FRONT THE CONNECTIONS TO OTHER MODULES AND SO TYPOS CAN BE DETECTED (SIMILAR TO A DECLARATION, IN THIS RESPECT). SO THAT THE COMPILER KNOWS THAT YOU ARE REFERRING TO A PREVIOUSLY COMPILED LIBRARY UNIT, WHOSE DECLARATION WILL BE FOUND IN THE PROGRAM LIBRARY.

```
with Text_IO; -- Get and Put are resources provided by Text_IO
procedure Transfer is
  Line : String (1 .. 80);
begin -- Transfer
  Text_IO.Get (Line);
  Text_IO.Put (Line);
end Transfer;
```

INSTRUCTOR NOTES

SOMEONE MAY ASK "IF A use CLAUSE ALWAYS REQUIRES A with CLAUSE, WHY DOESN'T THE use CLAUSE AUTOMATICALLY INCLUDE THE with?"

ANSWER:

1. THE with CLAUSE MAKES THE NAME OF A LIBRARY UNIT DIRECTLY VISIBLE, AND THEREFORE ITS CONTENTS VISIBLE BY SELECTION.
2. THE use CLAUSE MAKES THE CONTENTS OF A PACKAGE DIRECTLY VISIBLE, PROVIDED THE NAME OF THE PACKAGE IS ALREADY VISIBLE.
3. WHEN PACKAGES ARE LIBRARY UNITS, BOTH CLAUSES ARE REQUIRED TO MAKE CONTENTS OF PACKAGE DIRECTLY VISIBLE.
4. HOWEVER, THE PACKAGE MAY ALREADY BE VISIBLE (FOR EXAMPLE IT MAY HAVE BEEN DECLARED WITHIN THIS PROGRAM UNIT). IN SUCH A CASE, THE use CLAUSE ALONE IS REQUIRED. IN FACT, A with CLAUSE ISN'T EVEN ALLOWED, SINCE IT WOULD MAKE THE COMPILER LOOK FOR A LIBRARY UNIT BY THAT NAME.

POINT OUT THAT SHOULD YOU HAVE A with Some_Package, YOU COULD FOLLOW IT BY SOME DECLARATIONS, THEN APPLY THE use CLAUSE FOR THAT PACKAGE. UNTIL THE use, ANY DECLARATIONS WHICH USE Some_Package MUST USE DOT NOTATION.

REMINDE THE CLASS THAT with CLAUSES ONLY GO AT THE TOP OF A COMPILATION UNIT. A USE CLAUSE MAY FOLLOW THE with CLAUSE OR APPEAR AS A DECLARATION.

use CLAUSES

- A use CLAUSE MAY BE USED IN CONJUNCTION WITH A with CLAUSE TO MAKE DECLARATIONS IN SEPARATELY COMPILED PACKAGES DIRECTLY VISIBLE SO THEY CAN BE USED WITHOUT A QUALIFYING PREFIX

```
use Package_Name, ... , Package_Name;
```
- THE use CLAUSE MAY APPEAR IMMEDIATELY AFTER THE with CLAUSE OR IT MAY APPEAR ANYWHERE AN ORDINARY DECLARATION CAN APPEAR.
- A use CLAUSE AT THE TOP WITH THE with CLAUSE MAKES THE DECLARATIONS IN THE PACKAGE DIRECTLY VISIBLE THROUGH THE COMPILED UNIT.
- A use CLAUSE IN THE PLACE OF A DECLARATION MAKES THE DECLARATIONS DIRECTLY VISIBLE ONLY IN THE DECLARATIVE REGION CONTAINING THE use CLAUSE.
- DECLARATIONS MADE VISIBLE BY A use CLAUSE MAY BE HIDDEN BY A LOWER LEVEL DECLARATION OF THE SAME IDENTIFIER.

INSTRUCTOR NOTES

POINT OUT THAT IN THE BODY OF P, Q.B CANNOT BE ACCESSED. OBJECTS IN Q DON'T HAVE THE
EXTENDED SCOPE NECESSARY TO DO THAT BECAUSE Q IS A PROCEDURE. OBJECTS IN PACKAGE HAVE
EXTENDED SCOPE.

VISIBILITY EXAMPLES

```
with Text_IO; use Text_IO;
procedure Transfer is
  Line : String (1 .. 80);
begin -- Transfer
  Get (Line);
  Put (Line);
end Transfer;
```

```
procedure P is
  A, B : Boolean;

  procedure Q is
    C : Boolean;
    B : Boolean; -- inner B
  begin -- Q
    ...
    B := A;      -- means Q.B := P.A;
    C := P.B;    -- means Q.C := P.B;
  end Q;
begin -- P
  ...
  A := B; -- means P.A := P.B;
end P;
```


NAME CONFLICT

- RATIONALE: IF AN UNWANTED IDENTIFIER IS UNWITTINGLY IMPORTED (BECAUSE IT'S IN THE SAME PACKAGE AS A NEEDED ONE), IT WON'T CAUSE PROBLEMS, BECAUSE A DECLARATION WITHIN THE PROGRAM UNIT HIDES THE IMPORTED ONE.
- IF A PACKAGE DECLARATION P CONTAINS A DECLARATION OF AN ENTITY X, THEN WITHIN THE SCOPE OF ANOTHER ENTITY NAMED X, THE USE CLAUSE
 use P;
USUALLY DOES NOT APPLY TO X. THE X IN P MUST BE REFERRED TO AS P.X DESPITE THE USE CLAUSE.
- IF PACKAGE DECLARATIONS P AND Q BOTH CONTAIN ENTITIES NAMED X, THEN IF THE USE CLAUSES
 use P;
 use Q;
OCCUR IN THE SAME DECLARATIVE REGION, NEITHER APPLIES TO THE CORRESPONDING X. THE ENTITIES MUST BE REFERRED TO AS P.X AND Q.X.
- EXCEPTION: IF THE TWO ENTITIES NAMED X CAN BE INTERPRETED AS OVERLOADED SUBPROGRAMS, THE USE CLAUSE IS APPLIED AND THE SUBPROGRAMS ARE OVERLOADED. (OVERLOADED SUBPROGRAMS ARE TWO SUBPROGRAMS THAT CO-EXIST WITH THE SAME NAME. THEY WILL BE DISCUSSED LATER.)

HIDING EXAMPLE

```

package Numeric_Integration is
    type Integrand is array (0 .. 200) of Float;

    function Simpson (X      : Integrand;
                      Delta_t : Float)
        return Integrand;

    function Runge_Kutta (X      : Integrand;
                          Delta_t : Float)
        return Integrand;

end Numeric_Integration;
-- I want this
-- I don't want this
-- I want this

```

```

with Numeric_Integration; use Numeric_Integration;
procedure Dynamic_System_Simulation is

    Acceleration, Velocity, Displacement : Integrand;
    t, Dt, V0, D0 : Float;

    function Simpson (F : Integrand;
                     Dt : Float)
        return Integrand is
    ...
    end Simpson;
begin -- Dynamic_System_Simulation
    ...
    Velocity := Runge_Kutta (Acceleration, Dt);
    Displacement := Simpson (Velocity, Dt);
    ...
end Dynamic_System_Simulation;
-- My own declaration hides the
-- imported one

```

INSTRUCTOR NOTES

"THE SOFTWARE ENGINEERING GOALS ARE READABILITY, PORTABILITY, AND MAINTAINABILITY."

use CLAUSES ARE APPROPRIATE WHEN THE ENTITIES PROVIDED BY THE PACKAGE ARE:

1. NAMES WITH GENERALLY RECOGNIZABLE MEANINGS (PUT, GET, SIN, COS, TAN)
2. OPERATOR SYMBOLS (BUT TRY NOT TO MENTION THIS SINCE THIS FEATURE HASN'T BEEN COVERED YET)

COMMON PRACTICE

- RECALL THAT ONE OF THE MAIN SOFTWARE ENGINEERING GOALS SUPPORTED BY Ada IS TO PRODUCE READABLE PROGRAMS. IN THIS SPIRIT, WE SHOULD AVOID CONFUSING NESTS OF DECLARATIONS SOME OF WHICH HIDE OTHERS
- EXCEPT FOR DECLARATIONS IN LIBRARY PACKAGES, VIRTUALLY ALL DECLARATIONS WILL BE USED WHERE THEY ARE DIRECTLY VISIBLE
- HIDING SHOULD BE AVOIDED
- THE PRIMARY USE OF VISIBILITY BY SELECTION WILL BE TO ACCESS RECORD COMPONENTS AND TO ACCESS DECLARATIONS IN LIBRARY PACKAGES
- EXCEPT FOR NESTED RECORDS, QUALIFICATIONS WILL SELDOM EXTEND TO MORE THAN ONE PREFIX LEVEL
- EXCESSIVE USE OF use CLAUSES SHOULD BE AVOIDED SINCE THEY RESULT IN LARGE NUMBERS OF NAMES BECOMING DIRECTLY VISIBLE, CONFUSING THE READER OF THE PROGRAM

INSTRUCTOR NOTES

A NEW SUBSECTION STARTS HERE. THE QUESTION DOES NOT REFER BACK TO THE MATERIAL COVERED SO FAR, BUT IS LEAD-IN QUESTION, "WHAT DO SOME OF THE OTHER THINGS WE ENCOUNTER IN PACKAGE SPECIFICATIONS MEAN?"

AS YOU SEE MORE AND MORE PACKAGE SPECIFICATIONS, YOU WILL ENCOUNTER VARIOUS Ada CONSTRUCTS. IN ORDER TO USE THE PACKAGE CORRECTLY, WE MUST LEARN WHAT THEY MEAN.

WHAT DOES THAT MEAN ?

SOME ADDITIONAL CONSTRUCTS THAT YOU MAY ENCOUNTER IN PACKAGE SPECIFICATIONS.

INSTRUCTOR NOTES

"FROM TIME TO TIME YOU WILL ENCOUNTER A PACKAGE CONTAINING A private OR limited private TYPE. THIS IS MAINLY TO PREVENT PROGRAMMERS FROM USING "SHORT CUTS" BASED ON KNOWLEDGE OF THE IMPLEMENTATION (IF THE IMPLEMENTATION IS CHANGED, THE SHORT CUTS WON'T WORK). YOU USE private TYPES AS YOU WOULD ANY OTHER OBJECTS, EXCEPT THAT YOU ARE RESTRICTED AS TO WHICH OPERATIONS ARE ALLOWED." FOR EXAMPLE:

```
package Text_IO is
...
type File_Type is limited private;
procedure Close (File : in File_Type);
...
private
type File_Type is <implementation-defined>;
end Text_IO;
with Text_IO; use Text_IO;
procedure Q is
F : File_Type;
begin -- Q
...
Close (F); -- LEGAL
F := Some_Value; -- ILLEGAL
end Q;
```

(WHEN A PRIVATE TYPE HAS DISCRIMINANTS, EXAMINATION OF THE DISCRIMINANTS IS ALLOWED, INCLUDING USING A MEMBERSHIP TEST TO DETERMINE THE SUBTYPE OF AN OBJECT.)

PRIVATE TYPES: BASIC IDEA

- DECLARE A TYPE BUT CONCEPTUALLY HIDE THE IMPLEMENTATION FROM USER, I.E.
PROGRAMMER CANNOT ACCESS THE IMPLEMENTATION
- DATA ABSTRACTION AND ENCAPSULATION
- CAN BE DECLARED ONLY WITHIN PACKAGE SPECIFICATION
- OPERATIONS ALWAYS AVAILABLE OUTSIDE PACKAGE ARE:
 - ASSIGNMENT
 - EQUALITY/INEQUALITY
 - OPERATIONS SPECIFIED IN VISIBLE PART
- limited private types PERMIT ONLY OPERATIONS SPECIFIED IN VISIBLE PART

INSTRUCTOR NOTES

EXAMPLES OF RENAMING STATEMENTS

```
procedure Write (S: in String) renames Text_IO.Put_Line;  
L : Language_Type renames Native_Language_In;
```

POINT OUT THAT YOU CAN RENAME COMPONENTS OF RECORDS.

ADDITIONAL SCOPE AND VISIBILITY TOPICS

- Ada PROVIDES A FEW OTHER FEATURES THAT WILL NOT BE DISCUSSED HERE IN GREAT DETAIL

- RENAMING DECLARATIONS

SYNTAX:

Newdecl renames Oldname;

Newdecl: Type_Name renames Oldname;

- ALLOWS ASSOCIATING A DIFFERENT IDENTIFIER WITH AN EXISTING ENTITY OVER SOME PART OF THE PROGRAM
- USED PRIMARILY TO AVOID NAME CONFLICTS IN SEPARATELY DEVELOPED PACKAGES OR TO PROVIDE A SHORTHAND NOTATION. A SEQUENCE OF RENAMING DECLARATIONS IS OFTEN BETTER (IN TERMS OF PROGRAM READABILITY) THAN A use CLAUSE.

INSTRUCTOR NOTES

RENAMING A VARIABLE.

RENAMING EXAMPLE

```
procedure P is
  A : Boolean := True;
begin -- P
  declare
    Outer_A : Boolean renames P.A;
    A       : Boolean := False; -- inner A
  begin
    Outer_A := A;
  end;
  A := not A;
end P;
```


OVERLOADING

- IT IS LEGAL FOR MORE THAN ONE DECLARATION OF A SPECIFIC IDENTIFIER TO BE VISIBLE IF REFERENCES TO THE DIFFERENT DECLARATIONS CAN BE DISTINGUISHED BY THE CONTEXT IN WHICH THEY OCCUR
 - THIS IS MOST FREQUENTLY SEEN IN THE CASE OF A FAMILY OF SUBPROGRAM DECLARATIONS WITH THE SAME NAME BUT WITH PARAMETERS OF DIFFERENT TYPES
- EXAMPLE: Get AND Put

INSTRUCTOR NOTES

SUBPROGRAM OVERLOADING

- TWO OR MORE SUBPROGRAMS MAY HAVE THE SAME NAME IF THEY HAVE DIFFERENT NUMBERS OR TYPES OF OPERANDS
- THE SUBPROGRAMS CAN THEN BE DISTINGUISHED BY THE PATTERN OF THEIR OPERANDS
- CONTEXT:
 with Text_IO; use Text_IO;
 -- defines Put and Get for Strings
 package Int_IO is new Integer_IO (Integer); use Int_IO;
 -- defines Put and Get for Integer

EXAMPLES:

```
Put ("NAME:"); -- uses Text_IO.Put  
Put (3); -- uses Int_IO.Put
```

- EXCESSIVE USE OF SUBPROGRAM OVERLOADING IS HAZARDOUS TO THE HEALTH AND PATIENCE OF A PROGRAM READER OR MAINTAINER

INSTRUCTOR NOTES

POINT OUT THAT THIS ALLOWS THE PROGRAMMER TO CHOOSE NAMES WITHOUT FEAR OF CONFLICT.

OVERLOADING - ENUMERATION LITERALS

SAME ENUMERATION LITERAL MAY BE USED IN MORE THAN ONE TYPE DECLARATION

```
type Counting_Format is (Binary, BCD);  
type Tape_Format is (ASCII, EBCDIC, Binary);  
type Interrupt_Status is (Yes, No);  
type Answer is (Yes, No, Maybe);
```

INSTRUCTOR NOTES

POINT OUT THAT WE ARE ALREADY USING OVERLOADED + WHEN WE ADD INTEGERS AND FLOATS SINCE THE LANGUAGE SUPPLIES A DIFFERENT FUNCTION FOR EACH TYPE.

OPERATOR OVERLOADING

IT IS POSSIBLE TO EXTEND OPERATORS TO OTHER DATA TYPES BY USE OF OVERLOADING

EXAMPLE:

OPERATORS (+, -, *, /, **, >, <, >=, <=, =, /=) ARE CONSIDERED TO BE FUNCTIONS IN Ada, FOR THE USER'S NOTATIONAL CONVENIENCE, INFIX NOTATION IS ALLOWED:

a < b RATHER THAN "<(a,b)

SUPPOSE YOU HAVE A PROGRAM THAT HAS A RECORD TYPE Date TO HOLD THE DATE. FOR INSTANCE, AN OBJECT Today OF TYPE Date MIGHT BE ASSIGNED

Today := (March, 4, 1983);

IT WOULD BE NICE, GIVEN TWO Date OBJECTS, TO BE ABLE TO COMPARE THEM:

```
if Due_Date < Today
then
    Put ( " O V E R D U E : " );
end if;
```

INSTRUCTOR NOTES

POINT OUT THAT ONLY PREDEFINED OPERATORS CAN APPEAR INSIDE THE " " FOR OVERLOADING.

PREDEFINED "=" IS NOT AVAILABLE FOR limited TYPES.

IF THE CLASS IS CLAMORING FOR A DEFINITION OF limited TYPES, SAY THEY INCLUDE:

1. TYPES DECLARED AS limited private. (FOR FURTHER DETAILS, TAKE L305)
2. SPECIFIC TYPES KNOWN AS "TASK TYPES." (FOR FURTHER DETAILS, TAKE L401)
3. A RECORD TYPE OR ARRAY TYPE WITH A limited COMPONENT.

OPERATOR OVERLOADING (Continued)

NORMALLY, Ada DOES NOT ALLOW YOU TO COMPARE TWO RECORDS, ONLY THEIR COMPONENTS. HOWEVER, Ada DOES ALLOW YOU TO CREATE A NEW MEANING FOR THE "<" FUNCTION:

```
function "<" (Date_1, Date_2: Date) return Boolean is
begin
    -- "<"
    . . .
end "<";
```

NOW WE CAN WRITE:

```
if Due_Date < Today then ...
```

NOTE: THE ASSIGNMENT OPERATOR (":=") MAY NOT BE OVERLOADED.

THE EQUALITY OPERATOR ("=") MAY ONLY BE OVERLOADED FOR limited TYPES.

THE INEQUALITY OPERATOR ("<") MAY NOT BE EXPLICITLY OVERLOADED, BUT IT IS IMPLICITLY OVERLOADED WHEN YOU OVERLOAD "="

INSTRUCTOR NOTES

POINT OUT THAT THE ' ("TICK") IS NEEDED. THIS IS DIFFERENT FROM TYPE CONVERSION.

OVERLOADING: TYPE AMBIGUITY AND QUALIFICATION

USUALLY THE COMPILER CAN SORT THINGS OUT. WHEN IN TROUBLE, Ada GIVES YOU A WAY OF UNIQUELY IDENTIFYING THE OVERLOADED LITERALS BY MEANS OF TYPE QUALIFICATION

Example:

Counting_Format'(Binary)

Tape_Format'(Binary)

IDENTIFYING NOTATION PRECEDES THE OVERLOADED LITERAL WITH ITS TYPE NAME AND AN APOSTROPHE.

INSTRUCTOR NOTES

THIS IS A MOTIVATIONAL FOIL.

GENERIC PROGRAM UNITS - BASIC IDEA

- PROVIDE EFFECTIVE REUSE OF COMMON CODE
- ARE PARAMETERIZED TEMPLATES OF A PROGRAM UNIT
- WRITE CODE TO BE REUSED
- ENHANCE READABILITY
- BUILD LIBRARIES
- PARAMETERS ARE OBJECTS, TYPES, SUBPROGRAMS

INSTANTIATION PROVIDES THE PHYSICAL CODE FOR MANY SIMILAR BUT DIFFERENT TYPES

INSTRUCTOR NOTES

"INSTANTIATION IS THE PROCESS OF TAILORING THE GENERAL PATTERN TO FIT A SPECIFIC INSTANCE. YOU CAN'T USE A GENERIC PACKAGE LIKE SWAP FOR ANYTHING EXCEPT INSTANTIATION."

"THE INSTANTIATION HAS THE SAME EFFECT AS WRITING A COPY OF Exchange AT THAT POINT, WITH ALL OCCURRENCES OF 'Exchange' REPLACED BY 'Swap' AND ALL OCCURRENCES OF 'Swap_Type' REPLACED BY THE TYPE GIVEN IN THE INSTANTIATION."

"(SO THAT'S WHAT THOSE INCANTATIONS WERE!)"

GENERIC UNITS

EXAMPLE

TEMPLATE:

```
generic -- GENERIC CLAUSE
    type Swap_Type is private;
    procedure Exchange (X, Y: in out Swap_Type);

    procedure Exchange (X, Y: in out Swap_Type) is
        Temp : Swap_Type := X;
    begin -- Exchange
        X := Y;
        Y := Temp;
    end Exchange;
```

INSTANTIATIONS:

```
    procedure Swap is new Exchange (Integer);

    procedure Swap is new Exchange (Character);

    procedure Swap is new Exchange (Day);

    subtype Ten_Char_String is String (1 .. 10);
    procedure Swap is new Exchange (Ten_Char_String);
```


A GENERIC PACKAGE

Enumeration IO

(part of Text IO)

```
generic
    type Enum is (<>);

package Enumeration_IO is
    subtype Field is Integer range 0 .. <implementation-defined>;
    type Type_Set is (Lower_Case, Upper_Case);
    Default_Width : Field := 0;
    Default_Setting : Type_Set := Upper_Case;
    procedure Get (File : in File_Type; Item : out Enum);
    procedure Put (File : in File_Type; Item : in Enum;
        Width : in Field := Default_Width;
        Set : in Type_Set := Default_Setting);
    .
    .
    .
end Enumeration_IO;

package body Enumeration_IO is
    -- implementation-dependent
end Enumeration_IO;
```


INSTRUCTOR NOTES

AN EXAMPLE OF A GENERIC PARAMETER THAT IS A SUBPROGRAM.

```
generic
  type LP is limited private;
  with function Greater_Than (X, Y : LP) return Boolean;
package Pack is
  .
  .
  .
end Pack;
.
.
.
package New_Pack is new Pack (Integer, ">");
```

ASSIGN EXERCISE 27 OF THE EXERCISE BOOKLET.

ASSIGN CHAPTER 11 OF THE PRIMER.

TYPE PARAMETERS FOR GENERICS

type T is private; -- only assignment and equality defined for T; any type
-- actual except limited private

type T is limited private; -- no assignment, equality; any type actual

type T is (<>); -- discrete type

type T is range<>; -- integer type

type T is digits<>; -- floating point

type T is delta<>; -- fixed point

GENERIC PARAMETERS CAN ALSO BE SUBPROGRAMS

INSTRUCTOR NOTES

SECTION 13

PACKAGES

VG 728.2

INSTRUCTOR NOTES

BULLET 1: EMBEDDED SYSTEMS ARE TYPICALLY HUNDREDS OF THOUSANDS OF LINES LONG. SUCH LARGE PROGRAMS CANNOT BE UNDERSTOOD ALL AT ONCE.

BULLET 2: EMBEDDED SYSTEMS ARE TYPICALLY DEVELOPED BY LARGE TEAMS OF TENS OR EVEN A FEW HUNDRED PEOPLE. IF EACH HAD TO BE AWARE OF THE OTHERS' WORK, CHAOS WOULD RESULT.

BULLET 3: CERTAIN ALGORITHMS AND DATA STRUCTURES ARE USED IN MANY DIFFERENT EMBEDDED SYSTEMS. SOFTWARE COSTS CAN BE REDUCED BY PLUGGING "OFF-THE-SHELF COMPONENTS" INTO NEW SYSTEMS INSTEAD OF BUILDING THE COMPONENTS FROM SCRATCH EACH TIME.

IN Ada, THESE CONSIDERATIONS ARE ADDRESSED BY PACKAGES.

PACKAGES

- SOME GOALS OF Ada:
 - BE ABLE TO WRITE HUGE PROGRAMS WHICH CAN BE UNDERSTOOD
PIECE-BY-PIECE.
 - BE ABLE TO DIVIDE A PROGRAMMING PROJECT AMONG TEAMS
WORKING INDEPENDENTLY.
 - BE ABLE TO WRITE GENERAL SOFTWARE COMPONENTS THAT CAN
BE INCORPORATED IN MANY PROGRAMS
- THESE ABILITIES ARE PROVIDED BY PACKAGES

INSTRUCTOR NOTES

UNLIKE A SUBPROGRAM, A PACKAGE DOES NOT EXIST IN ORDER TO BE EXECUTED.

A PACKAGE PROVIDES A COLLECTION OF COMPUTATIONAL RESOURCES THAT MAY BE USED BY ONE OR MORE OTHER PROGRAM COMPONENTS.

THESE RESOURCES INCLUDE VARIABLES, TYPE DECLARATIONS, PROCEDURES, AND FUNCTIONS, AMONG OTHERS.

RESOURCES SHOULD BE PLACED IN THE SAME PACKAGE IF THEY ARE CLOSELY RELATED, AND MEANT TO BE USED IN CONJUNCTION WITH EACH OTHER.



WHAT IS A PACKAGE?

A PACKAGE IS A COLLECTION OF RELATED ENTITIES INCLUDING
(AMONG OTHERS):

- VARIABLES
- TYPE DECLARATIONS
- PROCEDURES
- FUNCTIONS

PACKAGES ARE THE BASIC BUILDING BLOCKS OF LARGE PROGRAMS.

OTHER NAMES FOR PACKAGES:

MODULES, CLUSTERS, PROGRAM COMPONENTS

INSTRUCTOR NOTES

THIS IS A DESCRIPTION OF AN Ada PACKAGE PROVIDING TWO NAMED NUMBERS, THREE SUBTYPES OF TYPE FLOAT, AND 16 FUNCTIONS. THE PACKAGE CONSISTS OF THIS DESCRIPTION PLUS A BODY CONTAINING THE BODIES OF THE 16 FUNCTIONS.

EXAMPLE: A PACKAGE PROVIDING MATHEMATICAL FUNCTIONS AND VALUES

```
package Math_Package is

  Pi : constant := 3.14159265358979;
  e  : constant := 2.71828182845904;

  subtype Non_Negative_Float is Float range 0.0 .. Float'Last;
  subtype Trigonometric_Subtype is Float range -1.0 .. 1.0;
  subtype Principal_Angle_Subtype is Float range -Pi/2 .. Pi/2;

  function Square_Root
    (Square: Non_Negative_Float) return Non_Negative_Float;
  function Sine (Radians: Float) return Trigonometric_Subtype;
  function Cosine (Radians: Float) return Trigonometric_Subtype;
  function Tangent (Radians: Float) return Float;
  function Cotangent (Radians: Float) return Float;
  function Secant (Radians: Float) return Float;
  function Cosecant (Radians: Float) return Float;
  function Arc_Sine
    (Sine: Trigonometric_Subtype) return Principal_Angle_Subtype;
  function Arc_Cosine
    (Cosine: Trigonometric_Subtype) return Principal_Angle_Subtype;
  function Arc_Tangent (Tangent: Float) return Principal_Angle_Subtype;
  function Arc_Cotangent (Cotangent: Float) return Principal_Angle_Subtype;
  function Arc_Secant (Secant: Float) return Principal_Angle_Subtype;
  function Arc_Cosecant (Cosecant: Float) return Principal_Angle_Subtype;
  function Exponential (Exponent: Float) return Non_Negative_Float;
  function Natural_Logarithm (Power: Non_Negative_Float) return Float;
  function Common_Logarithm (Power: Non_Negative_Float) return Float;

end Math_Package;
```

INSTRUCTOR NOTES

IN Ada, A SHARP DISTINCTION IS DRAWN BETWEEN THE INTERFACE OF A PACKAGE AND ITS IMPLEMENTATION.

THE INTERFACE EXPLAINS HOW THE ENTITIES PROVIDED BY THE PACKAGE TO THE OUTSIDE WORLD ARE TO BE USED.

THE IMPLEMENTATION EXPLAINS HOW THESE ENTITIES WORK INTERNALLY.

AN ANALOGY CAN BE DRAWN TO A STEREO RECEIVER: THE FRONT PANEL IS THE INTERFACE AND THE ELECTRONIC COMPONENTS INSIDE THE RECEIVER ARE THE IMPLEMENTATION.

INTERFACE VERSUS IMPLEMENTATION

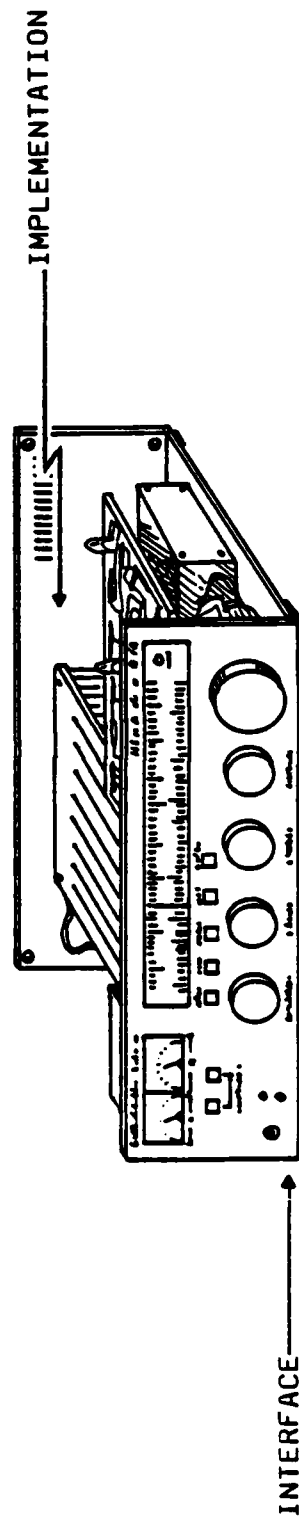
INTERFACE:

EXTERNAL APPEARANCE OF A PACKAGE

IMPLEMENTATION:

INTERNAL WORKINGS OF A PACKAGE

ONLY THE INTERFACE IS RELEVANT TO THE USER OF A PACKAGE



AD-A166 367

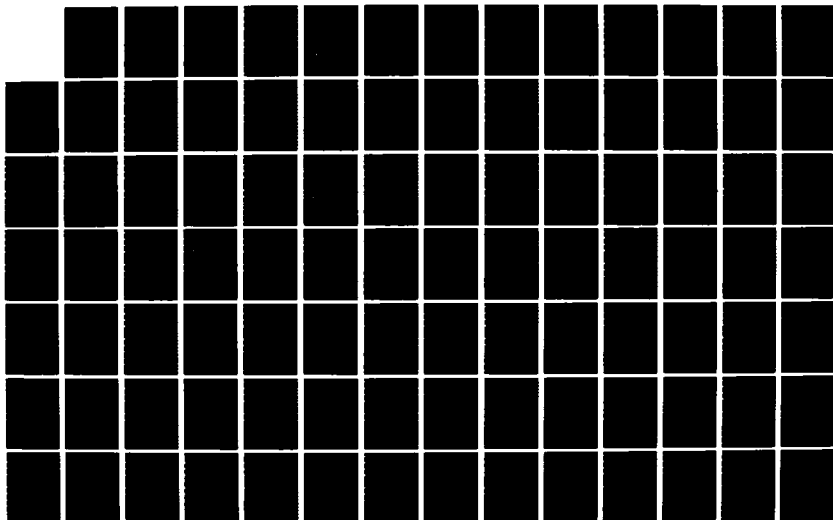
ADA (TRADE NAME) TRAINING CURRICULUM BASIC ADA
PROGRAMMING L202 TEACHER'S GUIDE VOLUME 2(U) SOFTECH
INC WALTHAM MA 1986 DAAB07-83-C-K514

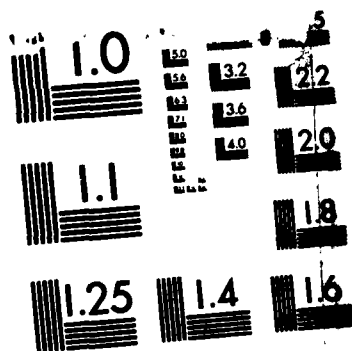
5/8

UNCLASSIFIED

F/G 9/2

NL





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

INSTRUCTOR NOTES

THE INTERFACE OF A PACKAGE INCLUDES SEVERAL KINDS OF INFORMATION DEFINED INSIDE THE PACKAGE FOR USE OUTSIDE THE PACKAGE.

BULLET 1: A PACKAGE MAY PROVIDE VARIABLES THAT CAN BE SET OR EXAMINED BY ALL USERS OF THE PACKAGE.

BULLET 2: A PACKAGE MAY DEFINE NEW TYPES OR SUBTYPES. PROGRAM UNITS USING THE PACKAGE MAY REFER TO THESE TYPES OR SUBTYPES IN VARIABLE AND SUBPROGRAM DECLARATIONS, FOR EXAMPLE.

BULLET 3&4: IF A PACKAGE PROVIDES A SUBPROGRAM TO THE OUTSIDE WORLD, THEN THE PACKAGE INTERFACE MUST INCLUDE INFORMATION ON HOW TO CALL THE SUBPROGRAM. PARAMETER NAMES ARE RELEVANT BECAUSE THEY MAY APPEAR IN NAMED SUBPROGRAM CALLS.

THE PACKAGE INTERFACE MAY INCLUDE INFORMATION BESIDE THAT DESCRIBED HERE.

THE INTERFACE OF A PACKAGE

THE PACKAGE INTERFACE INCLUDES (AMONG OTHER THINGS):

- DECLARATIONS OF VARIABLES AVAILABLE TO ALL USERS OF THE PACKAGE
- TYPES DEFINED IN THE PACKAGE FOR USE OUTSIDE OF THE PACKAGE
- PROCEDURE NAMES, PARAMETER NAMES, AND PARAMETER TYPES FOR PROCEDURES DEFINED IN THE PACKAGE FOR USE OUTSIDE THE PACKAGE
- FUNCTION NAMES, PARAMETER NAMES, PARAMETER TYPES, AND RESULT TYPES FOR FUNCTIONS DEFINED IN THE PACKAGE FOR USE OUTSIDE THE PACKAGE

INSTRUCTOR NOTES

THE IMPLEMENTATION OF A PACKAGE DESCRIBES THE INTERNAL MECHANISMS USED TO PROVIDE THE RESOURCES DESCRIBED IN THE PACKAGE INTERFACE.

THESE MECHANISMS ARE NOT DIRECTLY AVAILABLE OUTSIDE THE PACKAGE.

THE PROGRAMMING PRINCIPLE CALLED SEPARATION OF CONCERNS HOLDS THAT A PROGRAM USING A PACKAGE CAN BE STUDIED INDEPENDENTLY FROM TWO DIFFERENT VIEWPOINTS. ONE CAN STUDY THE RELATIONSHIP BETWEEN THE PACKAGE IMPLEMENTATION AND THE INTERFACE, OR THE RELATIONSHIP BETWEEN THE INTERFACE AND THE OTHER PARTS OF THE PROGRAM THAT USE THE PACKAGE; BUT IT IS NEVER NECESSARY TO CONSIDER THE IMPLEMENTATION AND THE USE OF THE PACKAGE AT THE SAME TIME. THIS MAKES IT POSSIBLE TO UNDERSTAND, DEVELOP, AND MODIFY A HUGE PROGRAM A PIECE AT A TIME.

THE IMPLEMENTATION OF A PACKAGE

THE PACKAGE IMPLEMENTATION INCLUDES (AMONG OTHER THINGS):

- THE ALGORITHMS FOR ALL PROCEDURES AND FUNCTIONS DESCRIBED IN THE INTERFACE
- SUBPROGRAMS USED INTERNALLY BY THOSE PROCEDURES AND FUNCTIONS, BUT NOT CALLED DIRECTLY FROM OUTSIDE THE PACKAGE
- TYPES AND VARIABLES USED INTERNALLY TO IMPLEMENT THE PROCEDURES AND FUNCTIONS DESCRIBED IN THE INTERFACE

INSTRUCTOR NOTES

THIS SLIDE RELATES THE LANGUAGE-INDEPENDENT NOTIONS OF INTERFACE AND IMPLEMENTATION TO Ada SYNTACTIC ENTITIES -- THE PACKAGE SPECIFICATION AND PACKAGE BODY.

A PACKAGE SPECIFICATION IS ESSENTIALLY A LIST OF DECLARATIONS OF ENTITIES TO BE USED OUTSIDE THE PACKAGE.

A PACKAGE BODY HAS THE SAME FORM AS A SUBPROGRAM BODY, EXCEPT FOR THE FIRST LINE. THE DECLARATIONS ARE OF ENTITIES CONSTITUTING THE IMPLEMENTATION OF THE PACKAGE. THESE MAY INCLUDE SUBPROGRAMS. THE INITIALIZATION STATEMENTS ARE EXECUTED ONCE, WHEN THE PACKAGE COMES INTO EXISTENCE.

THE WORD BEGIN AND THE STATEMENTS FOLLOWING IT MAY BE OMITTED (THE COMMENT FOLLOWING BEGIN IS OPTIONAL).

THE TERM PACKAGE GENERALLY REFERS COLLECTIVELY TO THE PACKAGE SPECIFICATION AND PACKAGE BODY.

THE FORM OF A PACKAGE

A PACKAGE HAS TWO PARTS:

- PACKAGE SPECIFICATION -- DESCRIBES THE INTERFACE
package [package name] is
[sequence of declarations]
end [package name];
- PACKAGE BODY -- DESCRIBES THE IMPLEMENTATION
package body [package name] is
[sequence of declarations]
[begin -- [package name]
[sequence of initialization statements]]
end [package name];

INSTRUCTOR NOTES

THIS IS THE SPECIFICATION OF A PACKAGE NAMED Earliest_and_Latest. IT PROVIDES TYPES NAMED Month_Type, Day_Type, and Date_Type, TWELVE ENUMERATION LITERALS FOR Month_Type VALUES, AND THREE SUBPROGRAMS.

THE FUNCTION Earliest_Date_Noted IS TO RETURN THE EARLIEST DATE WHICH HAS BEEN PASSED IN A PREVIOUS CALL TO Note_Date, OR DECEMBER 31 IF Note_Date HAS NOT YET BEEN CALLED. THE FUNCTION Latest_Date_Noted IS TO RETURN THE LATEST DATE WHICH HAS BEEN PASSED IN A PREVIOUS CALL TO Note_Date, OR JANUARY 1 IF Note_Date HAS NOT YET BEEN CALLED.

WITHIN THE PACKAGE SPECIFICATION THE SUBPROGRAMS PROVIDED BY THE PACKAGE ARE DESCRIBED BY SUBPROGRAM SPECIFICATIONS. SUBPROGRAM SPECIFICATIONS ARE IDENTICAL TO THE PART OF A SUBPROGRAM BODY PRECEDING THE WORD is. THEY INDICATE THE KIND OF SUBPROGRAM (PROCEDURE OR FUNCTION), THE NAME OF THE SUBPROGRAM, THE NAMES, MODES, AND SUBTYPES OF PARAMETERS, AND, IN THE CASE OF FUNCTIONS, THE RESULT SUBTYPE.

FOR EACH SUBPROGRAM SPECIFICATION APPEARING IN THE PACKAGE SPECIFICATION, A FULL SUBPROGRAM BODY MUST APPEAR IN THE PACKAGE BODY.

[IF A SECOND PROJECTOR IS AVAILABLE, KEEP THIS SLIDE ON THE SCREEN WHILE SHOWING THE NEXT TWO SLIDES.]

EXAMPLE OF A PACKAGE SPECIFICATION

```
package Earliest_And_Latest is

  type Month_Type is
    (January, February, March, April, May, June, July,
     August, September, October, November, December);

  type Day_Type is range 1 .. 31;

  type Date_Type is
    record
      Month_Part : Month_Type;
      Day_Part   : Day_Type;
    end record;

  procedure Note_Date (Date : in Date_Type);

  function Earliest_Date_Noted return Date_Type;

  function Latest_Date_Noted return Date_Type;

end Earliest_And_Latest;
```

INSTRUCTOR NOTES

Month_Type IS REFERRED TO LATER IN THE SPECIFICATION IN THE DECLARATION OF Date_Type;
Date_Type IS REFERRED TO LATER IN THE SPECIFICATION IN THE DECLARATIONS OF THE
SUBPROGRAMS.

AN ENTITY'S EXPANDED NAME CONSISTS OF THE NAME OF THE PACKAGE AND THE IDENTIFIER FOR THE
ENTITY, SEPARATED BY A PERIOD. Earliest_And_Latest.Month_Type, E.G., MEANS "THE ENTITY
NAMED Month_Type PROVIDED BY THE PACKAGE Earliest_And_Latest."

THE SLIDE LISTS EXPANDED NAMES FOR A TYPE, AN ENUMERATION VALUE, ANOTHER TYPE, AND A
SUBPROGRAM PROVIDED BY PACKAGE Earliest_And_Latest.

AN ENTITY DECLARED IN THE PACKAGE SPECIFICATION CAN BE REFERRED TO

- INSIDE THE PACKAGE: BY ITS IDENTIFIER
 - ANY PLACE IN THE PACKAGE SPECIFICATION FOLLOWING THE ENTITY'S
DECLARATION
 - ANY PLACE IN THE CORRESPONDING PACKAGE BODY

Month_Type

Date_Type

- OUTSIDE THE PACKAGE: BY AN EXPANDED NAME

Earliest_And_Latest.Month_Type

Earliest_And_Latest.January

Earliest_And_Latest.Date_Type

Earliest_And_Latest.Note_Date

INSTRUCTOR NOTES

THIS PACKAGE BODY CONSISTS OF SIX DECLARATIONS, THE VARIABLE DECLARATIONS FOR Earliest So_Far AND Latest So_Far AND THE PROCEDURE BODIES FOR Is_Earlier_Than, Note_Date, Earliest_Date_Noted, AND Latest_Date_Noted.

THE VARIABLES Earliest So_Far AND Latest So_Far COME INTO EXISTENCE WHEN THE PACKAGE COMES INTO EXISTENCE, AND STAY IN EXISTENCE FOR THE LIFETIME OF THE PACKAGE. IF ONE OF THE SUBPROGRAMS IN THE PACKAGE BODY LEAVES SOME VALUE IN ONE OF THOSE VARIABLES, THAT VALUE WILL BE THERE THE NEXT TIME ONE OF THOSE SUBPROGRAMS IS CALLED.

Earliest So_Far AND Latest So_Far ARE MAINTAINED BY Note_Date SO THAT THEY INDICATE THE EARLIEST AND LATEST DATES SEEN SO FAR. THESE VARIABLES CANNOT BE REFERENCED FROM OUTSIDE OF THE PACKAGE BODY, WHICH GUARANTEES THAT THEY WILL KEEP THIS MEANING REGARDLESS OF WHAT HAPPENS OUTSIDE THE PACKAGE.

THE FUNCTION Is_Earlier_Than IS USED BY Note_Date TO DETERMINE WHICH OF TWO DATES OCCURS BEFORE THE OTHER. IT IS PART OF THE PACKAGE'S IMPLEMENTATION RATHER THAN ITS INTERFACE. BECAUSE A SUBPROGRAM SPECIFICATION FOR Is_Earlier_Than DOES NOT OCCUR IN THE Earliest_And_Latest PACKAGE SPECIFICATION, IT MAY NOT BE CALLED FROM OUTSIDE THE PACKAGE BODY.

EXAMPLE OF A PACKAGE BODY

```

package body Earliest_And_Latest is
    Earliest_So_Far: Date_Type := (December, 31);
    Latest_So_Far : Date_Type := (January, 1);
    -- global to package body
    -- global to package body

    function Is_Earlier_Than (Date_1, Date_2 : Date_Type) return Boolean is
    begin -- Is_Earlier_Than
        if Date_1.Month_Part < Date_2.Month_Part then
            return True;
        elsif Date_1.Month_Part = Date_2.Month_Part then
            return Date_1.Day_Part > Date_2.Day_Part;
        else -- Date_1.Month_Part < Date_2.Month_Part
            return False;
        end if;
    end Is_Earlier_Than;

    procedure Note_Date (Date: in Date_Type) is
    begin -- Note_Date
        if Is_Earlier_Than (Date, Earliest_So_Far) then
            Earliest_So_Far := Date;
        end if;
        if Is_Earlier_Than (Latest_So_Far, Date) then
            Latest_So_Far := Date;
        end if;
    end Note_Date;

    function Earliest_Date_Noted return Date_Type is
    begin -- Earliest_Date_Noted
        return Earliest_So_Far;
    end Earliest_Date_Noted;

    function Latest_Date_Noted return Date_Type is
    begin -- Latest_Date_Noted
        return Latest_So_Far;
    end Latest_Date_Noted;

    end Earliest_And_Latest;
-- Maintains Earliest_So_Far
-- and Latest_So_Far

```

INSTRUCTOR NOTES

IF A PACKAGE PROVIDES ONLY TYPE, CONSTANT, OR VARIABLE DECLARATIONS, IT NEED NOT HAVE A PACKAGE BODY.

THE FIRST PACKAGE ON THIS SLIDE PROVIDES ONLY NAMED NUMBERS. A PROGRAM UNIT BEGINNING "WITH Metric_Conversion;" CAN REFER TO Metric_Conversion.Miles_Per_Meter AS IF IT WERE A REAL LITERAL.

THE SECOND PACKAGE DECLARES THE NAMED NUMBER Pi, THE FIXED POINT TYPES Distance AND Angle, AND SEVEN VARIABLES DESCRIBING THE CURRENT ORBIT AND POSITION OF A SATELLITE. ANY PROGRAM UNIT BEGINNING "with Orbit_Status;" CAN EXAMINE OR SET THE VARIABLES Orbit_Status.Major_Semiasis, Orbit_Status.Eccentricity, ETC. THESE VARIABLES ACT AS A COMMON DATABASE. THE PACKAGE Orbit_Status BEHAVES VERY MUCH LIKE A FORTRAN NAMED COMMON BLOCK OR A JOVIAL COMPOOL.

CAVEAT: PACKAGES LIKE Orbit_Status ARE GENERALLY A POOR USE OF PACKAGES. DATA SHOULD BE CLUSTERED WITH THE SUBPROGRAMS THAT USE IT.

PACKAGES WITHOUT BODIES

```
package Metric_Conversion is
  Meters_Per_Mile : constant := 1609.0;
  Kilometers_Per_Mile : constant := Meters_Per_Mile/1000.0;
  Miles_Per_Meter : constant := 1.0/Meters_Per_Mile;
  Liters_Per_Gallon : constant := 4.546;
  Gallons_Per_Liter : constant := 1.0/Liters_Per_Gallon;
...
end Metric_Conversion;

package Orbit_Status is
  Pi : constant := 3.1415926535897932384626433;
  type Distance is delta 0.1 range 0.0 .. 50_000.0;
  type Angle is delta 0.001 range -Pi .. Pi;
  Major_Semixis : Distance;
  Eccentricity : Float;
  Ascending_Node : Angle;
  Argument_Of_Perigee : Angle;
  Inclination : Angle;
  Current_Position : Angle;
end Orbit_Status;
```

INSTRUCTOR NOTES

PACKAGE Factorial Package PROVIDES FACILITIES FOR THE EFFICIENT COMPUTATION OF FACTORIALS OF THE NUMBERS 0 THROUGH 20. THE PACKAGE PROVIDES A NAME FOR THE SUBTYPE OF VALID ARGUMENTS, AN INTEGER TYPE WITH A SUFFICIENTLY WIDE RANGE TO HOLD THE RESULT, AND A FUNCTION Factorial. THE CALL Factorial (X), WHERE $0 < X < 20$, RETURNS THE PRODUCT OF THE FIRST X INTEGERS. (THE PRODUCT OF ZERO INTEGERS IS 1.)

TO AVOID REDUNDANT MULTIPLICATION EACH TIME FACTORIAL IS INVOKED, A TABLE OF FACTORIALS IS SET UP WHEN THE PACKAGE FIRST COMES INTO EXISTENCE, AND THE FACTORIAL FUNCTION MERELY REFERENCES THIS TABLE. THE STATEMENTS BETWEEN "begin" AND "end Factorial Package" ARE EXECUTED ONCE, AT THE BEGINNING OF A PROGRAM USING THE PACKAGE Factorial Package, BEFORE CONTROL IS PASSED TO THE MAIN PROGRAM.

(BECAUSE THE TABLE IS PART OF THE IMPLEMENTATION OF THE PACKAGE, IT IS DECLARED IN THE PACKAGE BODY AND CAN'T BE USED DIRECTLY OUTSIDE THE PACKAGE BODY. THE INITIALIZATION STATEMENTS AND THE Factorial FUNCTION BODY ARE THE ONLY PARTS OF THE PROGRAM WITH ACCESS TO Table.)

(PARENTHETICAL NOTE ON INTEGER TYPES: $20!$ IS APPROXIMATELY EQUAL TO 2.4329×10^{18} . THIS WILL BE GREATER THAN Integer'Last ON MOST IMPLEMENTATIONS, BUT COULD BE LESS THAN Long Integer'Last (THE VALUE FITS IN 64 BITS, BUT NOT IN 32). SINCE IT IS NOT SAFE TO HAVE Factorial RETURN A VALUE OF TYPE Integer, WE DEFINE A NEW INTEGER TYPE, Result_Type. IT IS DECLARED TO BE SLIGHTLY LARGER THAN 2.4329×10^{18} TO MAKE SURE IT IS LARGE ENOUGH TO ACCOMMODATE THE VALUE OF $20!$ AN Ada COMPILER WHICH CANNOT IMPLEMENT THE SPECIFIED RANGE WILL REJECT THE PACKAGE SPECIFICATION. THE ITERATION RULE OF THE FOR LOOP IMPLICITLY DECLARES I TO BE OF TYPE Integer, SO AN EXPLICIT CONVERSION IS REQUIRED TO MULTIPLY I BY Table (I-1), WHICH IS OF TYPE Result_Type.)

PACKAGE INITIALIZATION

```
package Factorial_Package is
  subtype Argument_Subtype is Integer range 0 .. 20;
  type Result_Type is range 1 .. 243E15;
  function Factorial (N : Argument_Subtype) return Result_Type;
end Factorial_Package;

package body Factorial_Package is

  Table: array (Argument_Subtype) of Result_Type;

  function Factorial (N : Argument_Subtype) return Result_Type is
  begin -- Factorial
    return Table (N);
  end Factorial;

  begin -- Factorial_Package

    Table (0) := 1;
    for I in 1 .. Argument_Subtype'Last loop
      Table (I) := Table (I-1) * Result_Type (I);
    end loop;

  end Factorial_Package;
```

INSTRUCTOR NOTES

A SEPARATELY COMPILED PACKAGE CAN BE USED IN ANOTHER COMPILATION UNIT THAT MENTIONS IT IN A with CLAUSE. A with CLAUSE CONSISTS OF THE RESERVED WORD with FOLLOWED BY A LIST OF COMPILATION UNIT NAMES SEPARATED BY COMMAS. THE with CLAUSE ENDS WITH A SEMICOLON. THIS SLIDE SHOWS TWO with CLAUSES, EACH NAMING ONLY ONE COMPILATION UNIT.

AT THE TOP OF THE SLIDE IS A SEPARATELY COMPILED FUNCTION THAT CONVERTS MILES PER GALLON TO KILOMETERS PER LITER. THE with CLAUSE ALLOWS THE FUNCTION TO USE RESOURCES PROVIDED BY THE PACKAGE Metric_Conversion. THESE ARE REFERRED TO BY THEIR EXPANDED NAMES Metric_Conversion.Kilometers_Per_Mile (IN THE ASSIGNMENT STATEMENT) AND Metric_Conversion.Gallons_Per_Liter (IN THE RETURN STATEMENT).

AT THE BOTTOM OF THE SLIDE IS A SEPARATELY COMPILED FUNCTION THAT COMPUTES THE BINOMIAL COEFFICIENT OF n AND k , THE NUMBER OF DISTINCT COMBINATIONS OF k ITEMS THAT CAN BE EXTRACTED FROM A COLLECTION OF n ITEMS. THIS IS EQUAL TO $n!/(k!(n-k)!)$. THE with CLAUSE ALLOWS Binomial_Coefficient TO USE RESOURCES PROVIDED BY Factorial_Package. NOTE THE EXPANDED NAMES Factorial_Package.Argument_Subtype IN THE FUNCTION HEADING AND Factorial_Package.Factorial IN THE RETURN STATEMENT. (THE / AND * OPERATORS TAKE ARGUMENTS IN Factorial_Package.Result_Type AND PRODUCE RESULTS IN THAT TYPE. THE RESULT OF THE DIVISION IS THEN CONVERTED TO TYPE INTEGER. TYPICALLY, THE RIGHT OPERAND OF / IN THIS COMPUTATION IS QUITE LARGE SO THAT THE RESULT IS LESS THAN Integer'Last EVEN WHEN THE OPERANDS OF / ARE NOT.)

A COMPILATION UNIT BEGINNING "with Binomial_Coefficient;" GETS ACCESS TO THIS FUNCTION, BUT IT DOES NOT GET ACCESS TO Factorial_Package. THAT IS, with CLAUSES ARE NOT TRANSITIVE.

USING PACKAGES

```
with Metric_Conversion;

function Kilometers_Per_Liter (Miles_Per_Gallon : Float) return Float is

    Kilometers_Per_Gallon : Float;

begin -- Kilometers_Per_Liter

    Kilometers_Per_Gallon :=
        Miles_Per_Gallon * Metric_Conversion.Kilometers_Per_Mile;
    return Kilometers_Per_Gallon * Metric_Conversion.Gallons_Per_Liter;

end Kilometers_Per_Liter;

-----

with Factorial_Package;

function Binomial_Coefficient (N, K: Factorial_Package.Argument_Subtype) return Integer is

begin -- Binomial_Coefficient
    return
        Integer
            (Factorial_Package.Factorial (N)/
             (Factorial_Package.Factorial (K) * Factorial_Package.Factorial (N-K)));
end Binomial_Coefficient;
```


INSTRUCTOR NOTES

WHEN EXTENSIVE USE IS MADE OF THE RESOURCES PROVIDED BY A PACKAGE, EXPANDED NAMES CAN BECOME CUMBERSOME. A use CLAUSE ALLOWS ENTITIES DECLARED IN A SPECIFIED PACKAGE SPECIFICATION TO BE REFERRED TO BY SIMPLE NAMES (IDENTIFIERS) RATHER THAN EXPANDED NAMES.

THE use CLAUSE "USE Metric Conversion;" ALLOWS Metric Conversion.Kilometers Per Mile AND Metric_Conversion.Gallons Per Liter TO BE REFERRED TO SIMPLY AS Kilometers_Per_Mile AND Gallons_Per_Liter WITHIN THE FUNCTION Kilometers_Per_Liter.

THE use CLAUSE "USE Factorial Package;" ALLOWS Factorial Package.Argument Subtype and Factorial Package.Factorial TO BE REFERRED TO SIMPLY AS Argument_Subtype AND Factorial WITHIN THE FUNCTION Binomial_Coefficients.

A use CLAUSE MAY OCCUR FOLLOWING A with CLAUSE (AS ON THIS SLIDE) OR WITHIN A SEQUENCE OF DECLARATIONS. IN GENERAL, THE RESERVED WORD use MAY BE FOLLOWED BY A LIST OF PACKAGES SEPARATED BY COMMAS.

IF A use CLAUSE WOULD LEAD TO MORE THAN ONE POSSIBLE INTERPRETATION FOR A GIVEN IDENTIFIER (FOR INSTANCE, IF Kilometers Per Liter HAD A LOCAL VARIABLE NAMED Kilometers_Per_Mile), THEN THE use CLAUSE DOES NOT APPLY TO THAT IDENTIFIER.

A use CLAUSE DOES NOT AFFECT WHICH ENTITIES YOU HAVE ACCESS TO, JUST THE WAY YOU NAME THOSE ENTITIES.

USE CLAUSES

```
with Metric_Conversion; use Metric_Conversion;

function Kilometers_Per_Liter (Miles_Per_Gallon : Float) return Float is

    Kilometers_Per_Gallon : Float;

begin -- Kilometers_Per_Liter

    Kilometers_Per_Gallon := Miles_Per_Gallon * Kilometers_Per_Mile;
    return Kilometers_Per_Gallon * Gallons_Per_Liter;

end Kilometers_Per_Liter;
```

```
-----

with Factorial_Package; use Factorial_Package;

function Binomial_Coefficient (N, K : Argument_Subtype) return Integer is

begin -- Binomial_Coefficient

    return Integer (Factorial (N) / (Factorial (K) * Factorial (N-K)));

end Binomial_Coefficient;
```

INSTRUCTOR NOTES

THE NAME `Argument_Subtype` BY ITSELF DOESN'T TELL US MUCH ABOUT THE MEANING OF THE SUBTYPE. THE EXPANDED NAME `Factorial_Package.Argument_Subtype` ANSWERS THE QUESTION

"ARGUMENT OF WHAT?"

IT IS A WITH CLAUSE THAT ALLOWS ENTITIES DECLARED IN SOME OTHER PACKAGE TO BE USED IN A COMPILATION UNIT. IT IS A USE CLAUSE THAT ALLOWS THE ENTITIES TO BE USED WITHOUT REFERRING TO THE PACKAGES THEY WERE DECLARED IN.

PROBLEMS WITH use CLAUSE

- EXPANDED NAMES CAN BE INFORMATIVE

Argument_Subtype VERSUS Factorial_Package.Argument_Subtype

- A use CLAUSE ALLOWS IDENTIFIERS DECLARED OUTSIDE A COMPILATION UNIT TO BE USED INSIDE A COMPILATION UNIT WITHOUT ANY INDICATION OF WHERE AND HOW THEY ARE DECLARED.

- WHEN A COMPILATION UNIT HAS with AND use CLAUSES FOR SEVERAL PACKAGES, IT IS DIFFICULT TO DETERMINE WHICH UNDECLARED IDENTIFIERS IN THE COMPILATION UNIT COME FROM WHICH PACKAGES.

INSTRUCTOR NOTES

FOR A PARTICULAR APPLICATION, THERE MAY BE OTHER PRIMITIVES WHOSE ORIGIN AND MEANING ARE OBVIOUS FROM THEIR NAME.

OPERATOR OVERLOADING IS DISCUSSED IN DETAIL LATER IN THE COURSE. STUDENTS SHOULD BE AT LEAST SUPERFICIALLY FAMILIAR WITH IT FROM PREVIOUS COURSES. AVOID GETTING INTO A DETAILED DISCUSSION NOW.

WHEN TO USE use WITH with

- PACKAGES PROVIDING FAMILIAR NAMES OF BASIC PRIMITIVES

Get, Put

Sine, Cosine, Tangent

- PACKAGES PROVIDING NEW VERSIONS OF OPERATORS

- A PROGRAMMER CAN WRITE A TWO-PARAMETER FUNCTION NAMED "+", FOR EXAMPLE

- IF THIS FUNCTION IS PROVIDED BY A PACKAGE NAMED P, THE use CLAUSE

use P;

ALLOWS THE FUNCTION TO BE CALLED BY THE EXPRESSION

A + B

RATHER THAN

P."+"(A,B)

RENAMING DECLARATIONS

- PROVIDE A NEW NAME FOR SOME ENTITY
- THE NEW NAME MAY BE MORE SUCCINCT OR MORE MEANINGFUL

INSTRUCTOR NOTES

POINT OUT THE RENAMING DECLARATIONS IN THE SECOND VERSION.

IF THE FUNCTION HAD INCLUDED AN ASSIGNMENT TO `Date_1.Month_Part`, IT COULD HAVE BEEN REWRITTEN AS AN ASSIGNMENT TO `Month_1`.

EXAMPLE OF RENAMING DECLARATIONS

- ORIGINAL FUNCTION

```
function Is_Earlier_Than (Date_1, Date_2: Date_Type) return Boolean is
begin -- Is_Earlier_Than
    if Date_1.Month_Part < Date_2.Month_Part then
        return True;
    elsif Date_1.Month_Part = Date_2.Month_Part then
        return Date_1.Day_Part < Date_2.Day_Part;
    else -- Date_1.Month_Part > Date_2.Month_Part
        return False;
    end if;
end Is_Earlier_Than;
```

- REWRITTEN WITH RENAMING DECLARATIONS:

```
function Is_Earlier_Than (Date_1, Date_2: Date_Type) return Boolean is
    Month_1 : Month_Type renames Date_1.Month_Part;
    Month_2 : Month_Type renames Date_2.Month_Part;
begin -- Is_Earlier_Than
    if Month_1 < Month_2 then
        return True;
    elsif Month_1 = Month_2 then
        return Date_1.Day_Part < Date_2.Day_Part;
    else -- Month_1 > Month_2
        return False;
    end if;
end Is_Earlier_Than;
```

INSTRUCTOR NOTES

THE FIRST FORM IS USED FOR RENAMING A VARIABLE OR A CONSTANT.

THE SECOND FORM IS USED FOR RENAMING AN EXCEPTION.

THE THIRD FORM IS USED FOR RENAMING A PACKAGE. IT IS USEFUL FOR PACKAGES NESTED INSIDE OTHER PACKAGES. (ONE OF THE RESOURCES PROVIDED BY A PACKAGE MAY ITSELF BE A PACKAGE.)

THE FOURTH FORM IS USED FOR RENAMING A PROCEDURE OR FUNCTION.

THERE ARE NO RENAMING DECLARATIONS FOR TYPES OR SUBTYPES, BUT A SUBTYPE DECLARATION CAN BE USED TO ACHIEVE THE EFFECT OF RENAMING A TYPE OR SUBTYPE.

THE NEW NAME MAY, BUT NEED NOT, BE IDENTICAL TO PART OF THE OLD NAME.

POSSIBLE FORMS OF RENAMING DECLARATIONS

```
Month_1: Month_Type renames Date_1.Month_Type;  
Singular_Matrix: exception renames Matrix_Package.Singular_Matrix;  
package Keypad_Interface renames Hardware_Interface.Keypad_Interface;  
function Factorial (N: Argument_Subtype) return Result_Type  
renames Factorial_Package.Factorial;
```

```
subtype Factorial_Argument_Subtype is Factorial_Package.Argument_Subtype;
```

INSTRUCTOR NOTES

SOMETIMES IT IS BETTER TO USE SUBTYPE DECLARATIONS AND RENAMING DECLARATIONS INSTEAD OF
use CLAUSES.

THE SUBTYPE DECLARATIONS DECLARE LOCAL SUBTYPES NAMED `Argument_Subtype` AND `Result_Type`
WHICH CONSIST OF THE SAME VALUES AS `Factorial_Package.Argument_Subtype` AND
`Factorial_Package.Result_Type`, RESPECTIVELY.

THE RENAMING DECLARATION FOR `Factorial` ESTABLISHES FACTORIAL AS A LOCAL NAME FOR THE
FUNCTION `Factorial_Package.Factorial`. (THE RENAMING DECLARATION REFERS TO THE SUBTYPES
DECLARED ABOVE IT. THE RETURN STATEMENT REFERS TO THE RENAMED FUNCTION.)

USE OF RENAMING DECLARATIONS INSTEAD OF use CLAUSES

```
with Factorial_Package;

function Binomial_Coefficient (N, K: Factorial_Package.Argument_Subtype) return Integer is

    subtype Argument_Subtype is Factorial_Package.Argument_Subtype;
    subtype Result_Type is Factorial_Package.Result_Type;
    function Factorial (N: Argument_Subtype) return Result_Type
        renames Factorial_Package.Factorial;

begin -- Binomial_Coefficient

    return Integer (Factorial (N) / (Factorial (K) * Factorial (N-K)));

end Binomial_Coefficient;
```

INSTRUCTOR NOTES

RENAMING DECLARATIONS REQUIRE FAR MORE WRITING THAN USE CLAUSES.

HOWEVER, THEY MAKE PROGRAMS FAR EASIER TO UNDERSTAND.

A BASIC PHILOSOPHY UNDERLYING ADA IS THAT EASE OF READING IS MORE IMPORTANT THAN EASE OF WRITING. A PROGRAM IS READ MANY MORE TIMES THAN IT IS WRITTEN.

ADVANTAGES OF RENAMING DECLARATIONS

ORDER use CLAUSES

- THERE IS A LOCAL DECLARATION DESCRIBING THE ENTITY
 - WHETHER IT IS A SUBPROGRAM, AN OBJECT, ETC.
 - THE PARAMETER AND RESULT TYPES OF A SUBPROGRAM, AMONG OTHER THINGS
 - THE TYPE OF AN OBJECT
- THE OLD NAME IN THE RENAMING DECLARATION TELLS WHERE THE ENTITY BEING RENAMED COMES FROM.
- ONE CAN RENAME JUST THE ENTITIES BEING USED RATHER THAN ALL ENTITIES PROVIDED BY SOME OTHER PACKAGE.
- ONE CAN PROVIDE A LOCALLY MORE MEANINGFUL NAME FOR THE ENTITY PROVIDED BY SOME GENERAL PURPOSE PACKAGE.

INSTRUCTOR NOTES

ALLOCATE AT LEAST ONE HOUR AND FIFTEEN MINUTES FOR LECTURE ON THIS SECTION.

ASSIGN EXERCISES 28 THROUGH 31 OF THE EXERCISE BOOKLET FOR LAB ASSIGNMENTS.

THE OBJECTIVE OF THIS SECTION IS TO INTRODUCE EXCEPTIONS, EXCEPTION HANDLING, AND THE BLOCK STATEMENT.

SECTION 14 EXCEPTIONS

VG 728.2

DEFINITIONS

THIS CHAPTER DEFINES THE FACILITIES FOR DEALING WITH ERRORS OR OTHER EXCEPTIONAL SITUATIONS THAT ARISE DURING PROGRAM EXECUTION. SUCH A SITUATION IS CALLED AN EXCEPTION. TO RAISE AN EXCEPTION IS TO ABANDON NORMAL PROGRAM EXECUTION SO AS TO DRAW ATTENTION TO THE FACT THAT THE CORRESPONDING SITUATION HAS ARISEN. EXECUTING SOME ACTIONS, IN RESPONSE TO THE ARISING OF AN EXCEPTION, IS CALLED HANDLING THE EXCEPTION.

Ada LANGUAGE REFERENCE MANUAL

INSTRUCTOR NOTES

POINT OUT THAT Numeric_Error WILL BE RAISED.

EXCEPTION EXAMPLE

- WHEN DIVISOR IS ZERO AN EXCEPTION (CALLED Numeric_Error) WILL OCCUR

```
procedure Example (Dividend, Divisor      : in Float;  
                  Partial_Result, Final_Result : out Float) is  
    . . .  
begin -- Example  
    . . .  
    Partial_Result := Dividend/Divisor;  
    . . .  
    Final_Result := Partial_Result + 1.0;  
end Example;
```

```
with Example;  
procedure Main is  
    A, B, C, D : Float;  
    . . .  
begin -- Main  
    . . .  
    Example (A, B, C, D);  
    . . .  
end Main;
```

INSTRUCTOR NOTES

THESE ARE OPTIONAL WAYS OF HANDLING AN EXCEPTION.

THIS IS A MOTIVATIONAL SLIDE TO SET THE STAGE.

THE CORRECTIVE ACTION THAT ONE MIGHT MAKE IS TO CLOSE UP A FILE OR PRINT A MESSAGE.

IF POSSIBLE DISPLAY 14-2 CONCURRENTLY.

WHAT COULD BE DONE ?

- HANDLE THE EXCEPTIONS LOCALLY IN Example: SET Partial_Result AND Final_Result TO Float'Last AND RETURN TO PROCEDURE Main AS IF NOTHING UNUSUAL HAD HAPPENED
- OR
- TERMINATE EXECUTION OF PROCEDURE Example AND MAKE SITUATION KNOWN TO Main SO THAT IT CAN TAKE CORRECTIVE ACTION
- OR
- PASS THE PROBLEM TO THE OPERATING SYSTEM AND LET IT DO SOMETHING LIKE TERMINATE THE PROGRAM WITH AN ERROR MESSAGE
-
- NONE OF THE ABOVE CHOICES IS RIGHT FOR EVERY APPLICATION
- SO
- Ada ALLOWS THE PROGRAMMER TO CONTROL THE HANDLING OF EXCEPTIONS

INSTRUCTOR NOTES

THESE SLIDES ARE JUST TO SET MOTIVATION FOR THE NEED FOR Ada'S EXCEPTION HANDLING MECHANISM.

ANOTHER EXCEPTION EXAMPLE

- WHEN DATA ENTERED DOES NOT MATCH TYPE OF OBJECT AN EXCEPTION WILL OCCUR

```
with Text_IO; use Text_IO;

procedure Expand_Command is

    type Input_Character_Type is (E, L, H, P, Q);
    Input_Character : Input_Character_Type;

package Input_IO is new Enumeration_IO (Input_Character_Type);

use Input_IO;

...

begin -- Expand_Command
    Get (Input_Character);

    ...

end Expand_Command;
```

INSTRUCTOR NOTES

QUESTION: HOW DO YOU KNOW WHAT STATEMENT RAISED AN EXCEPTION?

ANSWER: DEFINE YOUR OWN AND RAISE IT EXPLICITLY, OR ENCLOSE STATEMENT LIKELY TO RAISE EXCEPTIONS IN BLOCKS WITH HANDLERS. (THIS WILL BE SEEN SOON)

IN GENERAL

- TO CONTROL THE HANDLING OF EXCEPTIONS WE'D LIKE TO

1) NAME EXCEPTIONAL SITUATIONS

2) CALL ATTENTION SOMEWHERE TO THE FACT THAT AN EXCEPTIONAL
SITUATION OCCURRED

3) ATTEMPT TO CORRECT THE SITUATION

- THESE ACTIONS ARE MAPPED TO Ada CONSTRUCTS AS FOLLOWS:

1) EXCEPTION DECLARATION

2) raise STATEMENT

3) EXCEPTION HANDLER

INSTRUCTOR NOTES

CRC_Failure STANDS FOR CYCLE REDUNDANCY CHECK FAILURE.

VG 728.2

14-61

JUN 22 1968 224 527 53A 421 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769 770 771 772 773 774 775 776 777 778 779 780 781 782 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831 832 833 834 835 836 837 838 839 840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858 859 860 861 862 863 864 865 866 867 868 869 870 871 872 873 874 875 876 877 878 879 880 881 882 883 884 885 886 887 888 889 890 891 892 893 894 895 896 897 898 899 900 901 902 903 904 905 906 907 908 909 910 911 912 913 914 915 916 917 918 919 920 921 922 923 924 925 926 927 928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981 982 983 984 985 986 987 988 989 990 991 992 993 994 995 996 997 998 999 1000

USER-DEFINED EXCEPTIONS

EXCEPTION DECLARATION

SYNTAX:

```
Exception_Name | ,Exception_Name | : exception;
```

EXAMPLES:

```
Division_By_Zero : exception;  
CRC_Failure      : exception;  
Sensor_Off       : exception;  
Hell             : exception;
```

INSTRUCTOR NOTES

THESE ARE DECLARED IN PACKAGE STANDARD. THEY ARE RAISED WHEN A RULE OF THE LANGUAGE IS VIOLATED DURING PROGRAM EXECUTION.

JUST READ THE NAMES AND INDICATE THAT EACH ONE IS COVERED ON THE NEXT SLIDES.



PREDEFINED LANGUAGE EXCEPTIONS

THERE EXIST FIVE PREDEFINED LANGUAGE EXCEPTIONS:

- Constraint_Error
- Numeric_Error
- Program_Error
- Storage_Error
- Tasking_Error

INSTRUCTOR NOTES

1. "Tasking_Error PERTAINS TO ACTIVATION OF TASKS, WHICH WILL BE DISCUSSED IN SECTION 14."
2. THESE DEFINITIONS, EXCERPTED FROM Ada LANGUAGE REFERENCE MANUAL, ARE INCOMPLETE, IN ACCORDANCE WITH THE SCOPE OF THIS MODULE.
3. NUMERIC-ERROR NEED NOT BE RAISED FOR TARGET MACHINES WHICH DO NOT ALLOW EASY DETECTION OF OVERFLOW CONDITIONS.
4. DISCRIMINANTS WILL BE DISCUSSED IN SECTION 14.

PREDEFINED LANGUAGE EXCEPTIONS

- Constraint_Error

THIS EXCEPTION IS RAISED IN THE FOLLOWING SITUATIONS (AMONG OTHERS):

- UPON AN ATTEMPT TO VIOLATE A RANGE CONSTRAINT, OR AN INDEX CONSTRAINT
- ASSIGNING 10 TO A VARIABLE DECLARED AS

X : Integer range 0 .. 5;

VIOLATES RANGE CONSTRAINT

- AN OUT OF BOUNDS SUBSCRIPT VIOLATES AN INDEX CONSTRAINT

- Numeric_Error

THIS EXCEPTION IS RAISED BY THE EXECUTION OF A PREDEFINED NUMERIC OPERATION THAT CANNOT DELIVER THE CORRECT MATHEMATICAL RESULT, E.G. OVERFLOW OR ATTEMPTING TO DIVIDE BY ZERO

- AN IMPLEMENTATION IS NOT ALWAYS REQUIRED TO RAISE THIS EXCEPTION

INSTRUCTOR NOTES

POINT OUT THAT THE COMPILER DOES NOT NEED TO CATCH INCORRECT ORDER DEPENDENCIES.

AN EXAMPLE OF THE FIRST POINT:

```
function F (B: Boolean) return Integer;  
A: Integer := F (True);  
function F (B: Boolean) return Integer is  
...  
begin -- F  
...  
end F;
```

AN EXAMPLE OF THE THIRD POINT IN Program_Error IS

```
function F (B: Boolean) return Integer is  
X: Integer := 5;  
begin -- F  
  if B then  
    return X;  
  end if;  
end F;
```

CALLS TO F WITH ACTUAL PARAMETERS WITH VALUE False WILL RAISE Program_Error.

PREDEFINED LANGUAGE EXCEPTIONS (Continued)

- Program_Error

THIS EXCEPTION IS RAISED UPON AN ATTEMPT:

- TO ACTIVATE A UNIT (FOR EXAMPLE TO CALL A SUBPROGRAM) WHOSE BODY HAS NOT YET BEEN ELABORATED
- DEPENDING ON THE IMPLEMENTATION, TO EXECUTE AN ACTION THAT IS ERRONEOUS
- TO EXIT A FUNCTION OTHER THAN BY A RETURN STATEMENT OR ANOTHER EXCEPTION.

- Storage_Error

THIS EXCEPTION IS RAISED WHEN THERE IS INSUFFICIENT STORAGE (MEMORY) TO ALLOCATE NEW OBJECTS OR ACTIVATE NEW PROGRAM UNITS.

- Tasking_Error

INSTRUCTOR NOTES

POINT OUT THAT THESE ARE JUST SOME OF THE SITUATIONS WHICH WILL CAUSE THE EXCEPTION TO OCCUR. FOR A COMPLETE LIST OF SITUATIONS FOR EACH OF THE LISTED EXCEPTIONS, REFER TO THE LRM.

POINT OUT THAT THESE EXCEPTIONS ARE IN THE PACKAGE IO_exceptions.

THE EXCEPTIONS ARE COVERED IN MORE DEPTH IN CHAPTER 13.

TEXT_IO EXCEPTIONS

- Status_Error

- RAISED ON AN ATTEMPT TO OPEN AN ALREADY OPEN FILE

- Mode_Error

- RAISED ON AN ATTEMPT TO PERFORM AN OPERATION NOT APPLICABLE TO FILE
MODE.

- Name_Error

- RAISED ON AN ATTEMPT TO USE A NAME NOT LEGAL FOR AN EXTERNAL FILE

- Use_Error

- RAISED ON AN ATTEMPT TO USE OPERATION NOT APPLICABLE TO EXTERNAL FILE

INSTRUCTOR NOTES

VG 728.2

14-111

TEXT_IO EXCEPTIONS

- Device_Error
 - RAISED BY MALFUNCTION OF HARDWARE
- End_Error
 - RAISED ON AN ATTEMPT TO READ PAST END OF FILE
- Data_Error
 - RAISED WHEN DATA IS NOT OF REQUIRED TYPE
- Layout_Error
 - RAISED ON AN ATTEMPT TO SET COLUMN NUMBER IN EXCESS OF LINE LENGTH

INSTRUCTOR NOTES

POINT OUT THAT:

raise;

IS USED IN EXCEPTION HANDLERS TO PROPAGATE THE EXCEPTION. PROPAGATE MEANS TO PASS THE EXCEPTION UP TO THE NEXT LEVEL FOR HANDLING.

RAISE STATEMENT MAY APPEAR IN THE EXECUTABLE PORTION OF PROCEDURES, FUNCTIONS AND PACKAGE BODIES.

POINT OUT THAT PREDEFINED EXCEPTIONS ARE AUTOMATICALLY RAISED BY THE LANGUAGE.

RAISE STATEMENT

- PREDEFINED EXCEPTIONS AUTOMATICALLY RAISED BY LANGUAGE (OR Text_IO)
- TO RAISE USER DEFINED EXCEPTIONS USE raise STATEMENT

SYNTAX:

```
raise [Exception_Name];
```

EXAMPLES:

```
raise Division_By_Zero;  
raise Hell;  
raise;           -- only in a handler  
raise Sensor_Off;
```

INSTRUCTOR NOTES

POINT OUT THAT THE begin AND end COULD BELONG TO A BLOCK STATEMENT OR TO ANY PROGRAM UNIT. FOR EXAMPLE, THEY COULD BE IN A PROCEDURE, FUNCTION, PACKAGE BODY, TASK, OR GENERIC UNIT.

EXCEPTION HANDLER

AN EXCEPTION HANDLER OCCURS IN A "FRAME"

```

      {
begin
    -- sequence of statements
exception
    -- exception handlers
end;
      }
```

FRAME

INSTRUCTOR NOTES

EXCEPTION HANDLER

A FRAME FOR AN EXCEPTION HANDLER CAN BE ANY OF THE FOLLOWING
LANGUAGE CONSTRUCTS:

- SUBPROGRAM BODY
- BLOCK STATEMENT
- PACKAGE BODY

INSTRUCTOR NOTES

POINT OUT THAT STATEMENTS AFTER Get STATEMENT ARE NOT EXECUTED.

EXCEPTION HANDLING EXAMPLE

```
with Text_IO; use Text_IO;

procedure Expand_Command is

    type Input_Character_Type is (E, L, H, P, Q);
    Input_Character : Input_Character_Type;

    package Input_IO is new Enumeration_IO (Input_Character_Type);
    use Input_IO;

    ...

begin -- Expand_Command
    Get (Input_Character);

    ...
    -- These statements not executed if
    -- Data_Error is raised

exception

    when Data_Error => Put_Line ("Invalid Entry");
                       Put_Line ("Try Again");

end Expand_Command;
```


INSTRUCTOR NOTES

"A BLOCK ALWAYS OCCURS INSIDE THE EXECUTABLE PART OF A PROGRAM UNIT. IT IS AN EXECUTABLE STATEMENT."

BLOCK STATEMENT

- BLOCKS PROVIDE A MECHANISM FOR LOCALIZING EXCEPTION HANDLER(S)
TO A STATEMENT OR SEQUENCE OF STATEMENTS

SYNTAX:

```
[Block_Name:]  
[declare  
    local declarations;]  
begin -- Block_Name  
    sequence_of_statements;  
[exception  
    exception_handler;  
    {exception_handler;}]  
end [Block_Name];
```

- A BLOCK IS A STATEMENT

INSTRUCTOR NOTES

POINT OUT THAT NOW STATEMENTS AFTER Get CAN BE EXECUTED.

POINT OUT THAT THIS DOESN'T ALLOW FOR ANOTHER ATTEMPT BY THE Get STATEMENT. TO ALLOW RE-EXECUTION OF Get, THIS NEEDS TO BE PLACED WITHIN A LOOP.

EXAMPLE WITH BLOCK

```
with Text_IO; use Text_IO;

procedure Better_Expand_Command is
    type Input_Character_Type is (E, L, H, P, Q);
    Input_Character : Input_Character_Type;
    package Input_IO is new Enumeration_IO (Input_Character_Type);
    use Input_IO;

    ...

begin -- Better_Expand_Command
    begin
        Get (Input_Character);
    exception
        when Data_Error => Put_Line ("Invalid Entry");
    end;

    ...
    -- These statements executed whether or not the
    -- exception Data_Error was raised

end Better_Expand_Command;
```

INSTRUCTOR NOTES

POINT OUT THAT NOW MULTIPLE ATTEMPTS ARE SUPPORTED.

EXAMPLE ADDING LOOP WITH EXIT

```
with Text_IO; use Text_IO;
procedure Best_Expand_Command is
    type Input_Character_Type is (E, L, H, P, Q);
    Input_Character : Input_Character_Type;
    package Input_IO is new Enumeration_IO (Input_Character_Type);
    use Input_IO;
    ...
begin -- Best_Expand_Command
    loop
        begin
            Get (Input_Character);
            exit;
        exception
            when Data_Error => Put_Line ("Invalid Entry");
        end;
    end loop;
    ...
end Best_Expand_Command;
```

INSTRUCTOR NOTES

STRESS THAT others MUST APPEAR LAST AND AS THE ONLY CHOICE.

IE.

when Constraint_Error | others =>

IS ILLEGAL. (SEE NEXT SLIDE.)

MULTIPLE EXCEPTION CHOICES IN HANDLER

SYNTAX:

```
when exception_choice ||exception_choice| =>  
    -- sequence of statements
```

EXAMPLE:

```
begin  
    ...  
exception  
    when Division_By_Zero =>  
        -- sequence of statements  
        -- to be executed when Divisor = 0.0  
    when CRC_Failure =>  
        Recover_Message;  
    when others =>  
        -- sequence of statements to be executed when an  
        -- exception other than one of those listed above  
        -- is raised  
end;
```


INSTRUCTOR NOTES

EXCEPTION HANDLER

others MUST BE THE LAST Exception_Choice IN HANDLER AND MUST APPEAR AS ITS ONLY
Exception_Choice

```
-- LEGAL
begin
...
exception
  when Division_By_Zero =>
    ...
  when others =>
    ...
end;

-- **ILLEGAL
begin
...
exception
  when others =>
    ...
  when Division_By_Zero =>
    ...
end;

begin
...
exception
  when Division_By_Zero =>
    ...
  when others | CRC_Failure =>
    ...
end;
```

INSTRUCTOR NOTES

THIS CAN CAUSE PROBLEMS. THE CHOICE IN EXCEPTION HANDLERS NEEDS TO BE MANAGED EFFECTIVELY.

EMPHASIZE FOLLOWING WORDS: "ALL EXCEPTIONS," "INCLUDING," AND "NOT VISIBLE."

OTHERS

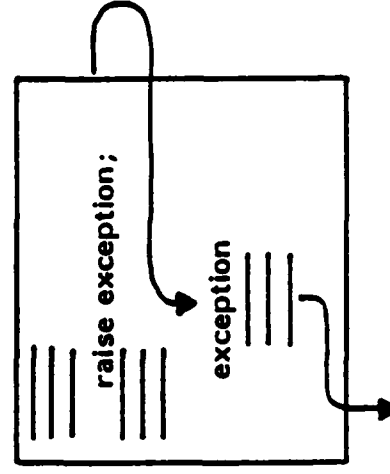
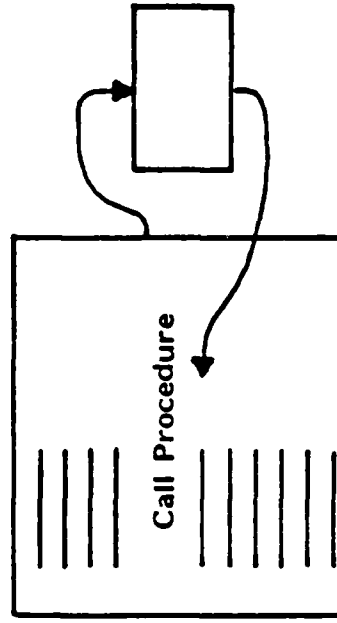
"STANDS FOR ALL EXCEPTIONS NOT LISTED IN PREVIOUS HANDLERS OF
THE FRAME, INCLUDING EXCEPTIONS WHOSE NAMES ARE NOT VISIBLE
AT THE PLACE OF THE EXCEPTION HANDLER."*

*LRM

INSTRUCTOR NOTES

THE GENERAL RULE IS AFTER HANDLING, CONTROL LEAVES THE ENCLOSING FRAME.

EXAMPLE: UNLIKE PROCEDURE CALLS, WHEN AN EXCEPTION IS RAISED EXECUTION IS NOT RESUMED AT THE POINT AT WHICH IT WAS RAISED.



EXCEPTION HANDLING CONTROL FLOW

- WHEN AN EXCEPTION IS RAISED, NORMAL PROGRAM EXECUTION IS SUSPENDED
- IF THERE IS A HANDLER FOR THE EXCEPTION IN THE INNERMOST FRAME, CONTROL IS TRANSFERRED TO IT
- IF THERE IS NO HANDLER, THE EXCEPTION IS PROPAGATED UP TO THE FRAME THAT CAUSED THE FRAME INCURRING THE EXCEPTION TO BE INVOKED -- THE CALLER IN THE CASE OF A SUBPROGRAM OR THE SURROUNDING FRAME IN THE CASE OF A BLOCK
- IF A HANDLER IS FOUND THERE IT IS EXECUTED. IF NOT, THE EXCEPTION IS PROPAGATED UP ONE MORE LEVEL
- IF NO HANDLER IS FOUND IN THE PROGRAM, THE UNDERLYING OPERATING SYSTEM TERMINATES THE PROGRAM (AND CAN TAKE OTHER ACTIONS SUCH AS ISSUING ERROR MESSAGES)
- WHEN A HANDLER IS FOUND, THE EXECUTION OF THE HANDLER REPLACES EXECUTION OF THE FRAME CONTAINING THE HANDLER (AND ANY LOWER LEVEL FRAMES)

INSTRUCTOR NOTES

BE SURE TO POINT OUT WHERE `Division_By_Zero` IS DECLARED.

STRESS THAT IT IS NOT A PREDEFINED EXCEPTION.

THIS EXAMPLE IS CONTRIVED, BUT CONSIDER CHECKING FOR:

- DIVISION BY 0 + 0i IN A COMPLEX NUMBERS PACKAGE
- ATTEMPT TO INVERT A SINGULAR MATRIX

EXAMPLE

EXCEPTION HANDLING LOCALLY

```
procedure Example ( Dividend, Divisor      : in Float;
                    Partial_Result, Final_Result : out Float) is
    Division_By_Zero : exception;      -- exception declaration
begin -- Example
    . . .
    if Divisor = 0.0
    then
        raise Division_By_Zero;      -- raising the exception
    else
        Partial_Result := Dividend/Divisor;
    end if;
    . . .
    Final_Result := Partial_Result + 1.0;
exception
    when Division_By_Zero =>
        Partial_Result := Float'Last;
        Final_Result   := Float'Last;
end Example;
```


INSTRUCTOR NOTES

POINT OUT THAT Main CATCHES ALL EXCEPTIONS. IT DOES NOT KNOW ABOUT Division_By_Zero.
IT WAS DECLARED IN Example.

AN ALTERNATE WAY OF WRITING THIS EXAMPLE IS

```
package Pkg is
    Division_By_Zero: exception;
    procedure Example ...;
end Pkg;
with Pkg; use Pkg;
procedure Main is
    ...
exception
    when Division_By_Zero =>
        ...
end Main;
```

EXAMPLE

EXCEPTION PROPAGATED TO CALLER

```

procedure Example ( Dividend, Divisor      : in Float;
                    Partial_Result, Final_Result : out Float) is
    Division_By_Zero : exception;
begin -- Example
    . . .
    if Divisor = 0.0
    then
        raise Division_By_Zero;
    else
        Partial_Result := Dividend/Divisor;
    end if;
    . . . -- calculations
end Example;

with Example;
procedure Main is
    A, B, C, D : Float;
begin -- Main
    . . .
    Example (A, B, C, D);
    . . .

exception
    when others =>
        ... -- whatever
end Main;

```

-- these will not be executed if
-- Division_By_Zero is raised in Example.
-- Can be corrected with use of block
-- construct as shown earlier.

INSTRUCTOR NOTES

POINT OUT THAT AFTER EXECUTING THE EXCEPTION HANDLER, EXECUTION CONTINUES WITH
"REMAINING CALCULATIONS."

ASSIGN EXERCISES 28 THROUGH 31 OF THE LAB MANUAL.

ASSIGN CHAPTER 12 OF THE PRIMER.

USING BLOCK TO LOCALIZE EXCEPTION HANDLING

```
with Example;
procedure Main is
    A, B, C, D : Float;
    ...
begin -- Main
    ...
    begin -- BLOCK
        Example (A, B, C, D);
    exception
        when others =>
            -- whatever
    end; -- BLOCK
    -- remaining calculations
end Main;
```

INSTRUCTOR NOTES

ALLOCATE ONE AND ONE HALF HOUR OF LECTURE FOR THIS SECTION. ASSIGN EXERCISE 32 OF THE EXERCISE BOOKLET FOR LAB ASSIGNMENT.

THE OBJECTIVE OF THIS SECTION IS TO INTRODUCE THE I/O OPERATIONS AVAILABLE IN Ada.
EMPHASIS IS PLACED ON Text_IO.

SECTION 15

INPUT/OUTPUT

INSTRUCTOR NOTES

- STRESS THAT THE EXTERNAL FILE IS OUTSIDE THE PROGRAM.
- STRESS THAT IT CAN BE ANYTHING THAT CAN ACCEPT OR PRODUCE A CHARACTER

I.E., A CARD READER,
TAPE,
DISK FILE,
I/O PORT

I/O TERMINOLOGY

- EXTERNAL FILE - ANYTHING OUTSIDE THE PROGRAM THAT CAN PRODUCE A VALUE FOR INPUT OR ACCEPT A VALUE FOR OUTPUT.
- INTERNAL FILE - OBJECT WITHIN THE PROGRAM ABLE TO BE ASSOCIATED WITH AN EXTERNAL FILE.
- FILE - REFERS TO THE INTERNAL FILE
- OPEN FILE - INTERNAL FILE ASSOCIATED WITH AN EXTERNAL FILE
- CLOSED FILE - INTERNAL FILE NOT ASSOCIATED WITH AN EXTERNAL FILE

INSTRUCTOR NOTES

- BASIC SCREEN LAID FIRST

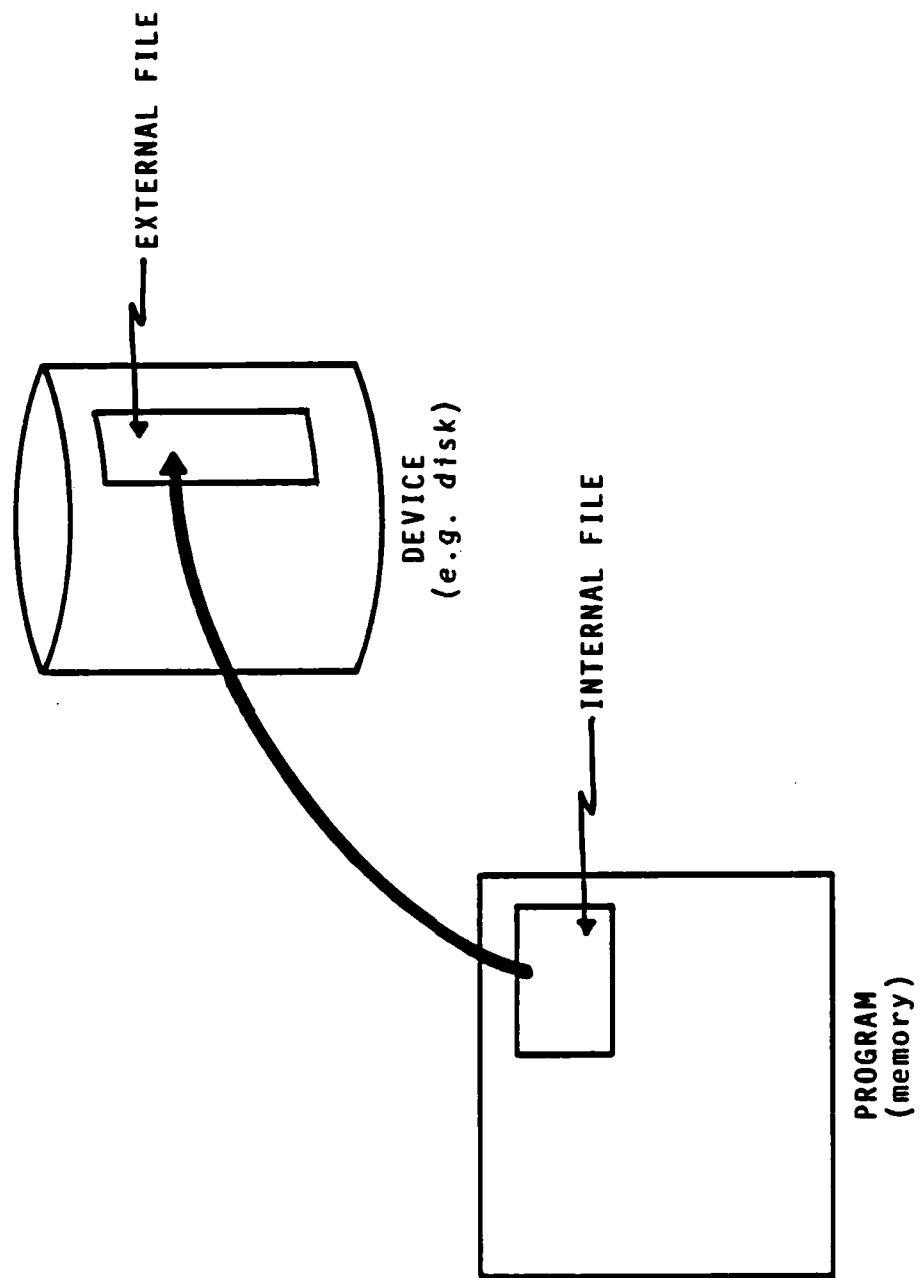
Identify External File

Identify Internal File

- LAY OVERLAY (ARROW) TO EXPLAIN AN OPEN FILE

STRESS THAT THE INTERNAL FILE IS THE ONE CONSIDERED OPEN

- REMOVE OVERLAY TO ILLUSTRATE CLOSING A FILE.



INSTRUCTOR NOTES

- AN ISAM FILE IS A FORM OF RANDOM ACCESS FILE.
- EACH OF THESE IS A PREDEFINED LIBRARY UNIT WHICH MUST BE "WITH"ED TO BE USED.

I/O PACKAGES SUPPLIED BY LANGUAGE

- Text_IO - FOR I/O ON TEXTUAL (I.E. READABLE) FILES
- Sequential_IO - FOR I/O ON SEQUENTIAL ACCESS FILES (EXAMPLE: TAPES)
- Direct_IO - FOR I/O ON DIRECT ACCESS FILES (EXAMPLE: ISAM FILES)

INSTRUCTOR NOTES

- THESE ARE BASIC FILE COMMANDS, COMMON TO EACH OF THESE I/O PACKAGES.
- THESE COMMANDS ARE EXPLAINED IN DETAIL IN A FEW SLIDES.

BASIC FILE COMMANDS

- Create
- Open
- Close
- Delete
- Reset

AD-A166 367

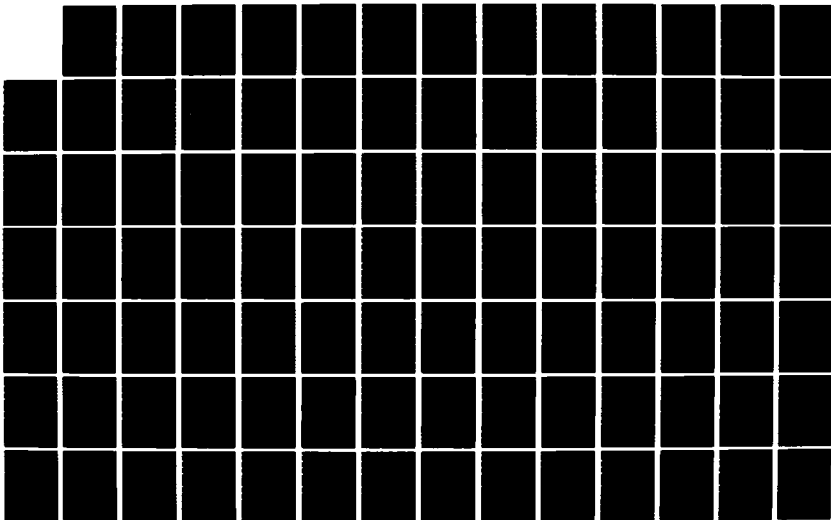
ADA (TRADE NAME) TRAINING CURRICULUM BASIC ADA
PROGRAMMING L202 TEACHER'S GUIDE VOLUME 2(U) SOFTECH
INC WALTHAM MA 1986 DAB07-83-C-K514

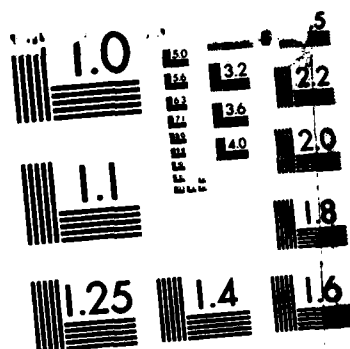
6/1

UNCLASSIFIED

F/G 9/2

NL





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

INSTRUCTOR NOTES

- STRESS THAT FORM WILL VARY - IT IS COMPLETELY IMPLEMENTATION DEPENDENT.
- MENTION THAT FORM IS INTENDED TO BE USED FOR SPECIFYING PERMISSIONS, SIZE LIMITS, ETC.
- File_Mode IS AN ENUMERATION TYPE IN Text_IO. ITS ENUMERATION LITERALS ARE In_File AND Out_File.

THE Create COMMAND

```

procedure Create (File: in out File_Type;
  Mode: in File_Mode := Default_Mode;
  Name: in String := "";
  Form: in String := "M");
• ESTABLISHES A NEW EXTERNAL FILE AND ASSOCIATES IT WITH AN INTERNAL FILE
• PARAMETERS
  • File
    - THE INTERNAL FILE ASSOCIATED WITH THE NEWLY CREATED EXTERNAL
      FILE
  • Mode
    - DEFINES WHETHER THE FILE IS AN INPUT (In_File) OR AN
      OUTPUT (Out_File) FILE
    - THE Default_Mode IS Out_File FOR Text_IO AND Sequential_IO;
      Inout_File FOR Direct_IO
  • Name
    - IDENTIFIES THE EXTERNAL FILE
    - THE DEFAULT IS AN "UNNAMED" TEMPORARY EXTERNAL FILE
    - THIS FILE WILL CEASE TO EXIST WHEN THE PROGRAM TERMINATES
  • Form
    - SPECIFIES THE FILE FORM
    - WILL VARY FROM SYSTEM TO SYSTEM
    - WILL DEFAULT TO THE SYSTEM DEFAULT

```

INSTRUCTOR NOTES

- POINT OUT THAT AS IN ALL PROCEDURE CALLS, THE PARAMETERS ARE SEPARATED BY COMMAS.
- Create creates a new external file, associates it with the internal file, and leaves the internal file open.
- THE EXAMPLE AT BOTTOM OF THE PAGE IS TO ILLUSTRATE WHAT COULD CAUSE Use_Error TO BE RAISED.
- AN EXAMPLE OF RAISING Status_Error IS WHEN YOU TRY TO ASSOCIATE THE SAME INTERNAL FILE WITH MORE THAN ONE EXTERNAL FILE.

THE Create COMMAND (Continued)

EXAMPLE

```
Data_File : File_Type;          -- THE PROCEDURE CREATE WILL GIVE Data_File
                                -- A VALUE
Create (File => Data_File,       -- CREATES A WRITE-ONLY FILE
       Mode => Out_File,        -- Data.Out. Form IS LEFT
       Name => "Data.Out");      -- WITH THE DEFAULT VALUE
```

NOTE: CREATE BOTH CREATES AND OPENS THE EXTERNAL FILE

• EXCEPTIONS

- Status_Error: WHEN THE INTERNAL FILE IS ALREADY OPEN
- Name_Error: WHEN THE NAME STRING DOES NOT PROPERLY NAME A FILE
- Use_Error: WHEN THE COMBINATION OF MODE, NAME, AND FORM DOES NOT ALLOW THE SYSTEM TO CREATE A FILE; FOR EXAMPLE:

CREATING A FILE WITH Mode Out_File AND Form Read_Only_Access

INSTRUCTOR NOTES

THE Open COMMAND

```

procedure Open (File: in out File_Type;
               Mode: in File_Mode;
               Name: in String;
               Form: in String := "");

```

- FUNCTION

OPENS AN INTERNAL FILE BY ASSOCIATING IT WITH AN EXISTING EXTERNAL FILE

- PARAMETERS

- File

- THE INTERNAL FILE TO BE ASSOCIATED WITH THE EXTERNAL FILE

- Mode

- THE MODE OF THE EXTERNAL FILE (In_File, Out_File, OR Inout_File)
 - THERE IS NO DEFAULT

- Name

- IDENTIFIES AN EXTERNAL FILE TO BE OPENED
 - THERE IS NO DEFAULT

- Form

- SPECIFIES THE FILE FORM
 - WILL VARY FROM SYSTEM TO SYSTEM
 - WILL DEFAULT TO THE SYSTEM DEFAULT

INSTRUCTOR NOTES

- STRESS THAT Open IS TO BE USED WHEN AN EXTERNAL FILE ALREADY EXISTS.
- EXPLAIN THAT TEMPORARY FILES CANNOT BE USED BY AN Open COMMAND. POINT OUT THAT THERE IS NO DEFAULT FOR Name. A NULL STRING WOULD RAISE Name_Error.
- STRESS THAT YOU CANNOT OPEN AN INTERNAL FILE THAT IS ALREADY OPEN. I.E., IT IS ILLEGAL TO OPEN A FILE THAT HAS JUST BEEN USED IN A Create STATEMENT.
- THE EXAMPLE AT THE BOTTOM OF THE PAGE ILLUSTRATES HOW Use_Error MIGHT BE RAISED.

THE Open COMMAND (Continued)

EXAMPLE

```
Test_File : File_Type;  
Open (File => Test_File,  
      Mode => In_File,  
      Name => "Test.Dat");
```

NOTE: OPEN DOES NOT FOLLOW THE CREATE COMMAND FOR THE SAME FILE

• EXCEPTIONS

- Status_Error: WHEN THE INTERNAL FILE IS ALREADY OPEN
- Name_Error: WHEN THE NAME STRING DOES NOT IDENTIFY AN EXISTING FILE
- Use_Error: WHEN THE COMBINATION OF Mode, Name, AND Form DOES NOT ALLOW THE SYSTEM TO OPEN THE FILE; FOR EXAMPLE:
ATTEMPTING TO ASSOCIATE THE CARD READER TO A FILE
WITH Mode Out_File

INSTRUCTOR NOTES

"DELETES THE ASSOCIATION BETWEEN THE INTERNAL AND EXTERNAL FILE."

THE Close COMMAND

procedure Close (File: in out File_Type);

- FUNCTION

SEVERES THE ASSOCIATION BETWEEN THE INTERNAL AND EXTERNAL FILE

- PARAMETERS

- File

- INTERNAL FILE OBJECT TO BE CLOSED

EXAMPLE

Close (Test_File);

- EXCEPTIONS

- Status_Error: RAISED WHEN THE INTERNAL FILE IS NOT OPEN

INSTRUCTOR NOTES

POINT OUT THAT FOR Delete TO WORK, THE FILE MUST BE OPEN, I.E. THERE MUST BE A LINK BETWEEN THE INTERNAL FILE AND THE EXTERNAL FILE.

"AFTER EXECUTING THE Delete COMMAND, THE EXTERNAL FILE SHOULD NO LONGER APPEAR IN YOUR DIRECTORY."

QUESTION: DOES Delete DELETE ALL REVISIONS OF THE FILE?

ANSWER: THAT IS IMPLEMENTATION DEPENDENT.

THE Delete COMMAND

procedure Delete (File: in out File_Type);

- FUNCTION

TO DELETE THE EXTERNAL FILE ASSOCIATED WITH THE GIVEN INTERNAL FILE AND
CLOSE THE INTERNAL FILE

- PARAMETERS

- File

- THE INTERNAL FILE ASSOCIATED WITH THE EXTERNAL FILE TO BE
DELETED

EXAMPLE

Open (Junk_File, In_File, "Junk.Dat");
Delete (Junk_File);

- EXCEPTIONS

- Status_Error: WHEN THE INTERNAL FILE IS NOT OPEN
- Use_Error: WHEN THE EXTERNAL FILE CANNOT BE DELETED

INSTRUCTOR NOTES

- STRESS THE NOTE, FILES OF Mode In_File MAY BE RESET TO Mode Out_File AND VICE VERSA.
- POINT OUT THAT THERE ARE TWO FORMS OF RESET. IT IS AN OVERLOADED PROCEDURE NAME.
- "RESET ALLOWS YOU TO OVERWRITE WHAT IS WRITTEN IN A FILE OR READ WHAT WAS JUST WRITTEN."

THE Reset COMMAND

```
procedure Reset (File:  in out File_Type;  
                 Mode:  in File_Mode);
```

```
procedure Reset (File:  in out File_Type);
```

- FUNCTION

ALLOW READING FROM OR WRITING TO A FILE TO BE RESTARTED AT THE BEGINNING
OF THE FILE

- PARAMETERS

- File

- THE INTERNAL FILE TO BE RESET

- Mode (IF SPECIFIED)

- THE Mode THE RESET FILE IS TO HAVE AFTER THE RESET

(NOTE: FILES OF Mode In_File MAY BE RESET TO Mode Out_File
AND VICE VERSA)

EXAMPLE

```
Reset (Data_File, In_File);
```

INSTRUCTOR NOTES

- THE EXAMPLE AT THE BOTTOM OF THE PAGE ILLUSTRATES HOW Use_Error MIGHT BE RAISED.

THE Reset COMMAND (Continued)

- EXCEPTIONS
 - Status_Error: WHEN THE INTERNAL FILE IS NOT OPEN
 - Use_Error: WHEN THE SYSTEM DOES NOT SUPPORT Reset AT ALL, OR WHEN RESETTNG TO THE SPECIFIED MODE FOR A GIVEN EXTERNAL FILE IS NOT SUPPORTED; FOR EXAMPLE:
RESETTNG THE CARD READER TO Mode Out_File

INSTRUCTOR NOTES

THEY ARE COMMON TO THE THREE I/O PACKAGES ALSO. Status_Error IS RAISED IF THE SPECIFIED FILE IS NOT OPEN.

- MODE - RETURNS THE MODE OF THE FILE. THE MODE CAN BE In_File OR Out_File FOR Text_IO AND Sequential_IO FILES. IT CAN BE In_File, Out_File, OR Inout_File FOR Direct_IO FILES.
- NAME - RETURNS THE NAME OF THE EXTERNAL FILE.
- FORM - RETURNS THE FILE'S FORM. THIS IS IMPLEMENTATION DEPENDENT. MAY CONTAIN INFORMATION SUCH AS ACCESS PERMISSIONS.
- Is_Open - RETURNS TRUE IF THE INTERNAL FILE SPECIFIED IS ASSOCIATED WITH AN EXTERNAL FILE; FALSE OTHERWISE.

COMMANDS FOR OBTAINING FILE INFORMATION

- function Mode (File: in File_Type) return File_Mode;
- function Name (File: in File_Type) return String;
- function Form (File: in File_Type) return String;
- function Is_Open (File: in File_Type) return Boolean;

CONTEXT:

Data_File : File_Type;

EXAMPLE:

```
if Is_Open (Data_File)
then
    Put (Data_File, "Hello");
end if;
```

INSTRUCTOR NOTES

- AN EXAMPLE OF "HUMAN READABLE" IS A FILE GENERATED BY A TEXT EDITOR. A "NON-HUMAN READABLE" FILE WOULD BE A TAPE FILE.
- WITHOUT THE WITH CLAUSE, THE I/O SUBPROGRAMS WILL BE UNDEFINED. ALL PROGRAMS WHICH PERFORM I/O ON HUMAN READABLE FILES MUST IMPORT Text_IO.
- THE DEFAULT FILES ARE DISCUSSED NEXT.
- Get AND Put ARE DISCUSSED IN DETAIL LATER.
- WE WILL SEE THE FORMATTING COMMANDS IN A FEW SLIDES.

USING Text_IO

- OPERATES ON HUMAN READABLE FILES
- MUST BE IMPORTED VIA A with CLAUSE
EXAMPLE: with Text_IO;
- HAVE DEFAULT FILES AND FORMATTING COMMANDS
- MAJOR OPERATIONS ARE Get AND Put

INSTRUCTOR NOTES

- EXPLAIN THAT THE DEFAULT FILES ARE ASSOCIATED WITH FILES DEFINED BY THE IMPLEMENTATION. EXACTLY WHAT THEY ARE WILL VARY FROM IMPLEMENTATION TO IMPLEMENTATION.
- STRESS THAT THEY ARE OPEN AND CANNOT BE CLOSED OR RESET BY THE PROGRAMMER.
- EXPLAIN THAT IN FUTURE SLIDES WHEN THE PROCEDURE OR FUNCTION SPECIFICATION APPEARS WITHOUT A FILE PARAMETER, THAT PROCEDURE OR FUNCTION WILL BE OPERATING ON ONE OF THE CURRENT DEFAULT FILES.
- EXPLAIN THE DIFFERENCE BETWEEN CURRENT DEFAULT FILES AND STANDARD DEFAULT FILES.
- THE CURRENT DEFAULT FILES START OUT AS THE STANDARD DEFAULT FILES.
- THE STANDARD DEFAULT FILES CANNOT BE CHANGED
- THE CURRENT DEFAULT FILE CAN BE CHANGED (TO ANY OTHER FILE OF THE SAME MODE OR BACK TO THE STANDARD DEFAULT FILES)

DEFAULT FILES

- CURRENT DEFAULT FILES

- CURRENT INPUT FILE - THE INPUT FILE USED WHEN NO FILE IS SPECIFIED
IN OPERATIONS LIKE Get
- CURRENT OUTPUT FILE - THE OUTPUT FILE USED WHEN NO FILE IS SPECIFIED
IN OPERATIONS LIKE Put

- STANDARD FILES

- STANDARD INPUT FILE - AN INPUT FILE OPEN AT THE BEGINNING OF THE PROGRAM
(TYPICALLY THE TERMINAL KEYBOARD OR DATA CARDS IN A BATCH DECK)
- STANDARD OUTPUT FILE - AN OUTPUT FILE OPEN AT THE BEGINNING OF THE PROGRAM
(TYPICALLY THE TERMINAL DISPLAY OR THE LINE PRINTER)

- INITIALLY, THE STANDARD INPUT AND OUTPUT FILES ARE THE SAME AS THE CURRENT DEFAULT FILES, BUT THE PROGRAM CAN CHANGE THE IDENTITY OF THE DEFAULT FILES.

INSTRUCTOR NOTES

- SETS THE SPECIFIED FILE TO THE DEFAULT INPUT FILE
I.E. MAKES IT THE CURRENT INPUT FILE.
- SETS THE SPECIFIED FILE TO THE DEFAULT OUTPUT FILE
I.E. MAKES IT THE CURRENT OUTPUT FILE.
- RETURNS THE STANDARD INPUT FILE
- RETURNS THE STANDARD OUTPUT FILE
- RETURNS THE CURRENT INPUT FILE
- RETURNS THE CURRENT OUTPUT FILE
- MENTION THAT

My_File := Standard_Input;

IS ILLEGAL. THESE FUNCTIONS ARE INTENDED TO BE USED IN CONJUNCTION WITH OTHER I/O ROUTINES.

- THESE FUNCTIONS CAN BE USED AS PARAMETERS TO OTHER SUBPROGRAMS.
- TO Reset THE DEFAULT INPUT FILE TO Standard_Input SAY Set_Input (Standard_Input);

COMMANDS DEALING WITH DEFAULT FILES

- procedure Set_Input (File: in File_Type);
- procedure Set_Output (File: in File_Type);
- function Standard_Input return File_Type;
- function Standard_Output return File_Type;
- function Current_Input return File_Type;
- function Current_Output return File_Type;

EXAMPLE:

```
Set_Input (Standard_Input);
```


- VG 728.2

15-17i

Text_IO COMMANDS FOR OBTAINING FILE INFORMATION

- function Line_Length (File: in File_Type) return Count;
 *function Line_Length return Count;
- function Page_Length (File: in File_Type) return Count;
 *function Page_Length return Count;
- function End_Of_Line (File: in File_Type) return Boolean;
 **function End_Of_Line return Boolean;
- function End_Of_Page (File: in File_Type) return Boolean;
 **function End_Of_Page return Boolean;
- function End_Of_File (File: in File_Type) return Boolean;
 **function End_Of_File return Boolean;

*OPERATES ON THE CURRENT OUTPUT FILE

**OPERATES ON THE CURRENT INPUT FILE

type Count is range 0 .. <implementation defined>; -- defined in package Text_IO
IT IS IMPORTANT TO REALIZE THAT Count IS DEFINED AS A TYPE.

INSTRUCTOR NOTES

- STRESS THAT THE DEFAULT FILE IS THE CURRENT OUTPUT FILE.
- MENTION THAT THESE COMMANDS MAY BE USED ON FILES OF MODE `In_File` OR `Out_File`.
I.E. `Col (Standard_Input_File)`
IS A LEGAL FUNCTION CALL.
- RESTATE THAT THE FILES MUST BE OPEN.
- POINT OUT THE TYPE Count. OBJECTS THAT ARE ASSIGNED `Page_Length` AND `Line_Length` MUST BE OF TYPE Count.

Text_IO COMMANDS FOR OBTAINING FILE INFORMATION (Continued)

- function Col (File: in File_Type) return Positive_Count;
 *function Col return Positive_Count;
- function Line (File: in File_Type) return Positive_Count;
 *function Line return Positive_Count;
- function Page (File: in File_Type) return Positive_Count;
 *function Page return Positive_Count;

*OPERATES ON THE CURRENT DEFAULT OUTPUT FILE

TYPE Positive_Count IS A POSITIVE INTEGER, A SUBTYPE OF Count

INSTRUCTOR NOTES

- STRESS THAT THESE OPERATE ON THE CURRENT OUTPUT FILE.
- MENTION THAT EACH OF THESE OPERATIONS MUST BE APPLIED TO A FILE OF MODE Out_File.
Mode_Error IS RAISED OTHERWISE.
- RESTATE THAT THE FILE MUST BE OPEN.
- THERE ARE NO PREDEFINED SCREEN MANIPULATION COMMANDS.

Text_IO OPERATIONS ON FILES

- procedure Set_Line_Length (File: in File_Type;
 To: in Count);
*procedure Set_Line_Length (To: in Count);
- procedure Set_Page_Length (File: in File_Type;
 To: in Count);
*procedure Set_Page_Length (To: in Count);
- procedure New_Line (File : in File_Type;
 Spacing: in Positive_Count := 1);
*procedure New_Line (Spacing: in Positive_Count := 1);
- procedure New_Page (File: in File_Type);
*procedure New_Page;

*OPERATES ON THE CURRENT DEFAULT OUTPUT FILE

INSTRUCTOR NOTES

- STRESS WHICH DEFAULT FILE EACH OPERATION APPLIES TO.
- MENTION THAT EACH OPERATION CAN BE APPLIED TO FILES OF BOTH MODES: In_File AND Out_File.
- EXPLAIN Set_Col
FOR INPUT:
READS AND DISCARDS CHARACTERS, LINE TERMINATORS, AND PAGE
TERMINATORS UNTIL THE NEXT CHARACTER TO BE READ HAS A COLUMN NUMBER
EQUAL TO THE SPECIFIED VALUE.
FOR OUTPUT:
OUTPUTS SPACES UNTIL THE CURRENT COLUMN NUMBER EQUALS THE SPECIFIED
VALUE
- MENTION THAT THE OTHER OPERATIONS WORK SIMILARLY
- RESTATE THAT THE FILE MUST BE OPEN

Text_IO OPERATIONS ON FILES (Continued)

```
•   procedure Skip_Line (File:   in File_Type;  
        Spacing: in Positive_Count := 1);  
    **procedure Skip_Line (Spacing: in Positive_Count := 1);  
  
•   procedure Skip_Page (File: in File_Type);  
    **procedure Skip_Page;  
  
•   procedure Set_Col (File: in File_Type;  
        To:   in Positive_Count);  
    *procedure Set_Col (To:   in Positive_Count);  
  
•   procedure Set_Line (File: in File_Type;  
        To:   in Positive_Count);  
    *procedure Set_Line (To:   in Positive_Count);  
  
    *OPERATES ON THE CURRENT DEFAULT OUTPUT FILE  
    **OPERATES ON THE CURRENT DEFAULT INPUT FILE
```


INSTRUCTOR NOTES

- AN EXAMPLE OF Mode_Error BEING RAISED WOULD BE Get FROM A FILE WITH MODE Out_File.

- AN EXAMPLE OF Layout_Error WOULD BE

Put (File1, MONDAY, 4);

WHERE MONDAY IS AN ENUMERATION LITERAL.

- AN EXAMPLE OF Data_Error WOULD BE

type Day is (Monday, Tuesday, Wednesday, Thursday, Friday);

Day_Of_Week: Day;

.

.

.

Get (File1, Day_Of_Week);

WHEN File1 CONTAINS

Saturday

Get AND Put PROCEDURES

- EXIST FOR:

- CHARACTER TYPES

- STRING TYPES

- INTEGER TYPES

- FIXED POINT TYPES

- FLOATING POINT TYPES

- ENUMERATION TYPES

- EXCEPTIONS

- BOTH RAISE Status_Error WHEN THE INTERNAL FILE IS NOT OPEN

- BOTH RAISE Mode_Error WHEN THE MODE IS NOT CORRECT FOR THE OPERATION

- Get RAISES End_Error IF IT TRIES TO READ PAST THE END OF THE FILE

- Put RAISES LAYOUT_ERROR IF IT CANNOT FIT THE CHARACTER

REPRESENTATION OF THE VALUE IN THE SPACE PROVIDED

- GET RAISES Data_Error IF THE VALUE OF THE ITEM IS OF AN INCORRECT

TYPE

INSTRUCTOR NOTES

- POINT OUT THAT Get OPERATES ON THE CURRENT INPUT FILE WHEN NO FILE IS SPECIFIED
- EXPLAIN THE DIFFERENCES BETWEEN Get (WHERE ITEM IS A STRING) AND Get_Line
 - Get (WHEN ITEM IS A STRING), FILLS THE STRING, IGNORING LINE AND PAGE TERMINATORS. IF N IS THE LENGTH OF THE STRING, N CHARACTERS ARE READ.
 - Get_Line READS UNTIL A LINE TERMINATOR IS THE NEXT CHARACTER TO BE READ, OR UNTIL N CHARACTERS ARE READ, WHICHEVER INVOLVES THE LEAST AMOUNT OF GETS.

Character AND String Get

- procedure Get (File: in File_Type;
Item: out Character);
procedure Get (Item: out Character);
- procedure Get (File: in File_Type;
Item: out String);
procedure Get (Item: out String);
- procedure Get_Line (File: in File_Type;
Item: out String;
Last: out Natural);
procedure Get_Line (Item: out String;
Last: out Natural);

Last IS THE INDEX VALUE IN THE STRING OF THE LAST CHARACTER READ.

INSTRUCTOR NOTES

- POINT OUT THAT Put OPERATES ON THE CURRENT OUTPUT FILE WHEN NO FILE IS SPECIFIED
- EXPLAIN THE DIFFERENCE BETWEEN Put WHERE ITEM IS A STRING AND Put_Line
 - Put WHEN ITEM IS A STRING OUTPUTS EACH CHARACTER IN THE STRING. THE STRING CAN SPAN OVER SEVERAL LINES IF THE LINES ARE BOUNDED (I.E. Set_Line_Length HAS BEEN CALLED) AND THE STRING IS LONGER THAN THE LINE LENGTH. NO LINE TERMINATORS ARE AUTOMATICALLY OUTPUT.
 - Put_Line OUTPUTS THE STRING ON A SINGLE LINE, FOLLOWED BY A LINE TERMINATOR. THE STRING MUST FIT ON A SINGLE LINE. Layout_Error IS RAISED OTHERWISE.

Character AND String Put

- procedure Put (File: in File_Type;
 Item: in Character);

 procedure Put (Item: in Character);
- procedure Put (File: in File_Type;
 Item: in String);

 procedure Put (Item: in String);
- procedure Put_Line (File: in File_Type;
 Item: in String);

 procedure Put_Line (Item: in String);

INSTRUCTOR NOTES

- Integer_IO IS A GENERIC PACKAGE CONTAINED IN Text_IO.
- Get AND Put WORK PRIMARILY THE SAME WAY AS FOR Character AND String.
- STRESS THE DIFFERENCES
- POINT OUT THAT Get CAN "READ" A VALUE FROM A STRING. Last WORKS THE SAME WAY.
- WHEN NO FILE IS SPECIFIED THE CURRENT DEFAULT FILE IS USED.
- EXAMPLE OF READING FROM A STRING
 Get ("45 ", item, last);
ITEM WILL EQUAL 45
LAST WILL EQUAL 2
- A WIDTH OF 0 MEANS "READ THE LONGEST SEQUENCE OF CHARACTERS THAT CAN BE INTERPRETED AS A NUMBER." IT IS THE WIDTH USED WHEN NONE IS SPECIFIED IN A CALL.

Get AND Put FOR Integer TYPES

TO ALLOW I/O OF Integer TYPE Int WRITE:

```
package Int_IO is new Integer_IO (Int); use Int_IO;
```

THIS GIVES THE FOLLOWING:

- GET OPERATIONS
 - procedure Get (File: in File_Type;
Item: out Num;
Width: in Field := 0);
 - procedure Get (Item: out Num;
Width: in Field := 0);
- procedure Get (From: in String;
Item: out Num;
Last: out Positive);

TYPE Field IS A NON-NEGATIVE INTEGER SUBTYPE.

INSTRUCTOR NOTES

- POINT OUT THAT PUT CAN "WRITE" A VALUE INTO A STRING
- RESULT IS RIGHT JUSTIFIED
- STRESS THAT Int IS Num
- EXPLAIN UNBOUNDED VS. BOUNDED.
UNBOUNDED - NO SPECIFIED FORMAT
BOUNDED - VALUE MUST BE EXPRESSED IN WIDTH SPACES
- MENTION THAT Default_Base AND Default_Width CAN BE MODIFIED
- EXAMPLE
STR : String (1 .. 5);
...
Put (STR, 31, Base => 10);

STR WILL BE " 31".

Get AND Put FOR Integer TYPES (Continued)

```
• Put OPERATIONS
  • procedure Put (File: in File_Type;
    Item: in Num;
    Width: in Field := Default_Width;
    Base: in Number_Base := Default_Base);

    procedure Put (Item: in Num;
      Width: in Field := Default_Width;
      Base: in Number_Base := Default_Base);

  • procedure Put (To: out String;
    Item: in Num;
    Base: in Number_Base := Default_Base);
```

NOTE: Num IS WHATEVER TYPE APPEARS IN THE DECLARATION; IN THIS CASE Int.
Default_Width IS ZERO, I.E. "UNBOUNDED"
Default_Base IS TEN

NOTE: FOR EACH INTEGER TYPE THE PROGRAMMER DEFINES, HE MUST WRITE
THIS INSTANTIATION IN THE DECLARATIVE PART:
package Package_Name is new Integer_IO (Integer_Type_Name);
use Package_Name;

INSTRUCTOR NOTES

- Fixed_IO AND Float_IO ARE GENERIC PACKAGES CONTAINED IN Text_IO
- RESTATE THAT Put AND Get WORK THE SAME AS FOR Characters AND Strings
- POINT OUT THAT THESE ALSO OPERATE ON Strings AS WELL AS Files.
- POINT OUT THAT NUM IS THE GENERIC TYPE PARAMETER.

Get AND Put FOR Fixed AND Float TYPES

TO ALLOW I/O FOR FLOATING POINT TYPE Flt, WRITE:

```
package Flt_IO is new Float_IO (Flt);  
use Flt_IO;
```

TO ALLOW I/O FOR FIXED POINT TYPE Fix WRITE:

```
package Fix_IO is new Fixed_IO (Fix);  
use Fix_IO;
```

THIS GIVES THE FOLLOWING:

```
• Get OPERATIONS  
  • procedure Get (File: in File_Type;  
    Item: out Num;  
    Width: in Field := 0);  
  procedure Get (Item: out Num;  
    Width: in Field := 0);  
  
  • procedure Get (From: in String;  
    Item: out Num;  
    Last: out Positive);
```

INSTRUCTORS NOTES

- NOTICE THAT YOU CAN SPECIFY THE WIDTH FOR BEFORE THE DECIMAL POINT (Fore), AFTER THE DECIMAL POINT (Aft), AND FOR THE EXPONENT (Exp).

Get AND Put FOR Fixed AND Float TYPES (Continued)

- Put OPERATIONS
 - procedure Put (File: in File_Type;
Item: in Num;
Fore: in Field := Default_Fore;
Aft: in Field := Default_Aft;
Exp: in Field := Default_Exp);

procedure Put (Item: in Num;
Fore: in Field := Default_Fore;
Aft: in Field := Default_Aft;
Exp: in Field := Default_Exp);
 - procedure Put (Io: out String;
Item: in Num;
Aft: in Field := Default_Aft;
Exp: in Field := Default_Exp);

INSTRUCTOR NOTES

Get AND Put FOR Fixed AND Float TYPES (Continued)

NOTE: Num IS THE TYPE USED IN THE I/O DECLARATION, E.G., Flt OR Fix.

FOR Float_IO

Default_Fore IS 2

Default_Aft IS Num'Digits -1

Default_Exp IS 3

FOR Fixed_IO

Default_Fore IS Num'Fore

Default_Aft IS Num'Aft

Default_Exp IS ZERO

INSTRUCTOR NOTES

- Enumeration_IO IS A GENERIC PACKAGE CONTAINED IN Text_IO.
- RESTATE THAT THEY WORK JUST LIKE Get and Put for CHARACTERS.
- POINT OUT THAT THESE ALSO WILL OPERATE ON STRINGS AS WELL AS FILES.

Get AND Put FOR ENUMERATION TYPES

TO ALLOW I/O OF ENUMERATION TYPE Enum, WRITE:

```
package Enum_IO is new Enumeration_IO (Enum); use Enum_IO;
```

THIS GIVES THE FOLLOWING:

- Get OPERATIONS
 - procedure Get (File: in File_Type;
Item: out Enum);
 - procedure Get (Item: out Enum);
- procedure Get (From: in String;
Item: out Enum;
Last: out Positive);

INSTRUCTOR NOTES

- HERE NOTICE THAT YOU CAN SPECIFY THE CASE OF THE OUTPUT.
- "WE HAVE NOW COVERED Text_10."

Get AND Put FOR ENUMERATION TYPES (Continued)

- Put OPERATIONS

- procedure Put (File: in File_Type;

- Item: in Enum;

- Width: in Field := Default_Width;

- Set: in Type_Set := Default_Setting);

- procedure Put (Item: in Enum;

- Width: in Field := Default_Width;

- Set: in Type_Set := Default_Setting);

- procedure Put (To: out String;

- Item: in Enum;

- Set: in Type_Set := Default_Setting);

Default_Width IS ZERO

Default_Setting IS Upper_Case

(OTHER CHOICE IS Lower_Case)

INSTRUCTOR NOTES

- Read AND Write WILL BE EXPLAINED IN DETAIL IN THE NEXT TWO SLIDES.
- POINT OUT THAT Sequential_IO CAN TECHNICALLY BE INSTANTIATED WITH ANY TYPE, BUT EACH IMPLEMENTATION WILL DECIDE WHICH TYPES TO SUPPORT (I.E. UNCONSTRAINED ARRAY TYPES MAY NOT BE SUPPORTED).
- REMIND THEM THAT THE MODES FOR Sequential_IO ARE In_File AND Out_File.
- THE CURRENT INDEX POINTS TO THE NEXT LOCATION THE I/O OPERATION WILL ACT ON.
(I.E. POINT TO THE NEXT ITEM TO BE READ OR TO THE PLACE THE NEXT ITEM WILL BE WRITTEN.)

USING Sequential_IO

- ALLOWS BINARY INPUT/OUTPUT ON SEQUENTIAL ACCESS FILES.
- IS A LIBRARY UNIT AND MUST BE IMPORTED VIA A with CLAUSE
EG: with Sequential_IO;
- IS A GENERIC PACKAGE AND MUST BE INSTANTIATED WITH A type.
type Array_Type is array (1 .. 10) of Boolean;
package Arr_IO is new Sequential_IO (Array_Type);
use Arr_IO;
- HAS A "CURRENT INDEX" WHICH IS SEQUENTIALLY INCREMENTED.
- MAJOR OPERATIONS ARE Read AND Write

INSTRUCTOR NOTES

- Element_Type IS WHATEVER TYPE THE PACKAGE WAS INSTANTIATED WITH.
- THE EXAMPLE DOES NOT SHOW THE Open. THE FILE MUST HAVE BEEN OPENED WITH MODE In_File.

Sequential_IO OPERATIONS

```
procedure Read (File : in File_Type;  
               Item : out Element_Type);
```

- FUNCTION

RETURNS THE NEXT VALUE OF TYPE Element_Type FROM THE SPECIFIED FILE.
OPERATORS OF FILES OF MODE In_File.

- EXCEPTIONS

Status_Error, RAISED WHEN THE FILE IS NOT OPEN.
Mode_Error, RAISED WHEN THE MODE IS NOT In_File.
Data_Error, RAISED WHEN THE VALUE READ IS NOT OF TYPE Element_Type.

- CONTEXT

```
type Array_Type is array (1 .. 10) of Boolean;  
Boolean_Array : Array_Type;  
package Arr_IO is new Sequential_IO (Array_Type);  
use Arr_IO;  
Bool_File : File_Type;
```

- EXAMPLE

```
Read (Bool_File, Boolean_Array);
```


INSTRUCTOR NOTES

- Sequential_IO IS A GENERIC PACKAGE. IT IS INSTANTIATED WITH TYPES TO DO I/O ON OBJECTS OF THOSE TYPES. REMIND THEM THAT IT IS A LIBRARY UNIT WHICH MUST BE "with*ED."
- Sequential_IO IS USED ON SEQUENTIAL ACCESS FILES (TAPE)
- Sequential_IO USES Read AND Write INSTEAD OF Get AND Put. THESE OPERATIONS WILL BE DISCUSSED IN DETAIL ON THE FOLLOWING SLIDES.
- REMIND THEM THAT Create, Open, Close, Delete, Reset, Mode, Name, Form, AND Is_Open ARE DEFINED IN Sequential_IO.
- REMIND THEM THAT In_File AND Out_File ARE THE ALLOWABLE MODES FOR Sequential_IO FILES.

Sequential_IO OPERATIONS

procedure Write (File : in File_Type; Item : in Element_Type);

- FUNCTION

OUTPUTS THE VALUE OF ITEM AT THE END OF THE FILE.
OPERATES ON FILES OF MODE Out_File.

- EXCEPTIONS

Status_Error, RAISED WHEN THE SPECIFIED FILE IS NOT OPEN
Mode_Error, RAISED WHEN THE MODE IS NOT Out_File

- CONTEXT

type Array_Type is array (1 .. 10) of Boolean;
Boolean_Array : Array_Type := (True, True, False, False, True, True, False, True, True, False,
True, False, True);

package Arr_IO is new Sequential_IO (Array_Type);
use Arr_IO;

Bool_File : File_Type;

- Example

Write (Bool_File, Boolean_Array);

INSTRUCTOR NOTES

- THIS COMPLETES THE COMMANDS FOR Sequential_IO.
- IN THE EXAMPLE, File_End WOULD BE A USER DEFINED EXCEPTION.

Sequential_IO OPERATIONS

function End_Of_File (File : File_Type) return Boolean;

- FUNCTION

RETURNS TRUE IF NO MORE ELEMENTS CAN BE READ FROM THE FILE: FALSE OTHERWISE.
OPERATES ON FILES OF MODE In_File.

- EXCEPTIONS

Status_Error, RAISED WHEN THE FILE IS NOT OPEN
Mode_Error, RAISED WHEN THE MODE IS NOT In_File.

- CONTEXT

File_End : exception;

- EXAMPLE

```
if End_Of_File (Bool_File) then
    raise File_End;
else
    Read (Bool_File, Boolean_Array);
    .
    .
    .
end if;
```

INSTRUCTOR NOTES

- Read AND Write ARE EXPLAINED IN DETAIL IN THE FOLLOWING SLIDES.
- THEY ARE SLIGHTLY DIFFERENT THAN Read AND Write FOR Sequential_IO.
- Direct_IO CAN ALSO BE INSTANTIATED WITH ANY TYPE, BUT IMPLEMENTATIONS ARE NOT REQUIRED TO SUPPORT THEM.
- THE MODES FOR Direct_IO ARE In_File, Out_File, AND Inout_File.
- WE WILL LATER SEE THE COMMANDS TO CHANGE THE CURRENT INDEX. REMIND THEM WHAT THE CURRENT INDEX IS.

USING Direct_IO

- ALLOWS BINARY INPUT/OUTPUT ON DIRECT ACCESS FILES
- IS A LIBRARY UNIT WHICH MUST BE IMPORTED VIA A WITH CLAUSE
EXAMPLE: with Direct_IO;
- IS A GENERIC PACKAGE WHICH MUST BE INSTANTIATED WITH A TYPE
EXAMPLE: type Array_Type is array (1 .. 10) of Boolean;
package Arr_IO is new Direct_IO (Array_Type);
use Arr_IO;
- MAJOR OPERATIONS ARE Read AND Write
- HAS A CURRENT INDEX WHICH CAN BE DIRECTLY CHANGED

INSTRUCTOR NOTES

- Element_Type IS WHATEVER TYPE Direct_IO WAS INSTANTIATED WITH
- IN THE EXAMPLE, THE FILE MUST HAVE BEEN OPENED WITH MODE In_File OR Inout_File.
- IN THE EXAMPLE, THE THIRD "RECORD" WOULD BE READ AND ASSIGNED TO RESULT.
- End_Error WOULD BE RAISED IF FROM HAD BEEN GREATER THAN 5.
- THE MISSING CODE FOR THE EXAMPLE IS:
package Bool_IO is new Direct_IO (Boolean);
use Bool_IO;
File : File_Type;
.
.
.
Open (File, In_File, "Data.Fil");
- THEY HAVE SEEN Positive_Count IN Text_IO.

Direct_IO OPERATIONS

```
procedure Read (File : in File_Type;  
               Item : out Element_Type;  
               From : in Positive_Count);
```

```
procedure Read (File : in File_Type;  
               Item : out Element_Type);
```

- FUNCTION

RETURNS THE VALUE LOCATED AT THE POSITION SPECIFIED BY THE Current_Index.
(THE VALUE OF FROM, IF SPECIFIED: THE NEXT VALUE OTHERWISE)

- EXCEPTIONS

Status_Error, RAISED IF THE FILE IS NOT OPEN

Mode_Error, RAISED IF THE MODE IS NOT In_File OR Inout_File.

Data_Error, RAISED IF THE VALUE READ IS NOT OF TYPE Element_Type

End_Error, RAISED WHEN AN ATTEMPT IS MADE TO READ PAST THE END OF THE FILE.

- EXAMPLE

File:

TRUE

FALSE

TRUE ← Result

TRUE

FALSE

Result: Boolean;

.

.

.

Read (File, Result, 3);

INSTRUCTOR NOTES

- AGAIN Element_Type IS THE TYPE THE PACKAGE WAS INSTANTIATED WITH
- THE MODE MUST BE Out_File OR Inout_File.
- MISSING CODE FOR EXAMPLE

```
package Bool_IO is new Direct_IO (Boolean);  
use Bool_IO;  
File: File_Type;  
.  
.  
.  
Open (File, Out_File, "Data.Dat");
```

Direct_IO OPERATIONS

```
procedure Write (File : in File_Type;  
                Item : in Element_Type;  
                To   : in Positive_Count);
```

```
procedure Write (File : in File_Type;  
                Item : in Element_Type);
```

- FUNCTION

OUTPUT THE VALUE TO THE POSITION SPECIFIED BY THE CURRENT INDEX (THE VALUE OF TO IF SPECIFIED, THE NEXT POSITION OTHERWISE)

- EXCEPTIONS

Status_Error, IF THE FILE IS NOT OPEN

Mode_Error, IF THE MODE IS NOT Out_File OR Inout_File

- EXAMPLE

FILE BEFORE	CODE	FILE AFTER
TRUE		TRUE
FALSE		FALSE
FALSE	Write(File, TRUE,3);	TRUE
FALSE		FALSE

INSTRUCTOR NOTES

- Set_Index SETS THE CURRENT INDEX TO THE SPECIFIED VALUE.
- Index RETURNS THE VALUE OF THE CURRENT INDEX.
- Size RETURNS THE SIZE (NUMBER OF RECORDS) IN THE FILE
- End_of_File RETURNS TRUE WHEN THE CURRENT INDEX EXCEEDS THE SIZE OF THE FILE:
FALSE OTHERWISE

Direct_IO OPERATIONS

```
procedure Set_Index (File : in File_Type;  
                    To   : in Positive_Count);  
  
function Index (File : in File_Type)  
    return Positive_Count;  
  
function Size (File : in File_Type)  
    return Count;  
  
function End_Of_File (File : in File_Type)  
    return Boolean;
```

INSTRUCTOR NOTES

- REMIND THEM THAT A FILE IS ANYTHING THAT CAN SEND OR RECEIVE A VALUE. THE INTERNAL FILE OBJECT IS ASSOCIATED WITH THE EXTERNAL PHYSICAL FILE.
- THE COMMON COMMANDS ARE THE COMMANDS WHICH ALL THREE PACKAGES CONTAIN AND THAT WORK THE SAME FOR ALL THREE KINDS OF I/O.
- ASSIGN EXERCISE 32 OF THE EXERCISE BOOKLET.
- ASSIGN CHAPTER 13 OF THE PRIMER.

I/O SUMMARY

- I/O OPERATES ON FILES, INTERNAL AND EXTERNAL
- THREE KINDS
 - Text_IO
 - Sequential_IO
 - Direct_IO
- COMMANDS IN COMMON
 - Open, Close, Create,
 - Delete, Reset, Name,
 - Mode, Form, Is_Open, End_of_File
- Text_IO MAJOR COMMANDS
 - Get AND Put
- Sequential_IO MAJOR COMMANDS
 - Read AND Write
- Direct_IO MAJOR COMMANDS
 - Read AND Write

INSTRUCTOR NOTES

ALLOCATE AT LEAST ONE AND ONE HALF HOURS FOR LECTURE ON THIS SECTION.

THERE ARE NO ASSIGNMENTS AT THE END OF THIS SECTION. HAVE STUDENTS CONTINUE WITH PREVIOUS WORK.

THE PURPOSE OF THIS SECTION IS TO PROVIDE A READING KNOWLEDGE OF SOME OF THE MORE ADVANCED FEATURES OF THE LANGUAGE. IT DOES NOT ATTEMPT TO TEACH THESE CONCEPTS. THEY WILL BE COVERED IN LATER COURSES.

SECTION 16

OVERVIEW OF OTHER LANGUAGE FEATURES

INSTRUCTOR NOTES

- STRESS THAT THE EXISTENCE OF A PRAGMA IS NOT ALLOWED TO CHANGE THE MEANING OF PROGRAM.

A PROGRAM WITH A PRAGMA COMPILED BY ONE IMPLEMENTATION SHOULD "MEAN" THE SAME WHEN COMPILED BY A DIFFERENT IMPLEMENTATION (I.E. ONE THAT DOES NOT RECOGNIZE THE PRAGMA).
- MENTION THAT NOT ALL PRAGMAS ARE REQUIRED TO BE SUPPORTED BY THE COMPILER.
- APPENDIX F SPECIFIES WHICH PRAGMAS ARE SUPPORTED BY THE COMPILER.

PRAGMAS -- BASIC IDEA

- CONVEY INFORMATION TO THE COMPILER
- SOME ARE LANGUAGE-DEFINED
- OTHERS ARE IMPLEMENTATION-DEFINED
- MAY AFFECT PERFORMANCE OF PROGRAM
- DO NOT CHANGE MEANING OF PROGRAM (I.E.,
NO RESULTS PRODUCED)

INSTRUCTOR NOTES

- "PACK ASKS THE COMPILER TO PACK AN OBJECT, E.G. AN ARRAY. In_Line ASKS THE COMPILER TO EMBED THE CODE RATHER THAN HAVE SUBROUTINE CALLS."
- "PRAGMAS MAY APPEAR ANYWHERE IN THE DECLARATIVE PART OF A PROGRAM."
- UNDERSTANDING PRAGMAS IS NOT VITAL TO UNDERSTANDING THE LANGUAGE.
- EXAMPLES OF OPTIMIZE - TIME: GENERATE FAST CODE
SPACE: GENERATE TIGHT CODE FOR LIMITED MEMORY.

PRAGMAS --- EXAMPLES

- `pragma Pack (Type_Name);`
- `pragma Inline (Subprogram_Name);`
- `pragma Optimize (Time);`
- `pragma Optimize (Space);`
- `pragma Page;`

INSTRUCTOR NOTES

RECURSIVE DATA STRUCTURES WERE MENTIONED BRIEFLY IN SECTION 10, ACCESS TYPES, AND ARE COVERED MORE THOROUGHLY IN L305.

"THE INCOMPLETE TYPE SPECIFICATION TELLS THE COMPILER TO ACCEPT THIS TYPE (I.E. ADD IT TO THE SYMBOL TABLE) EVEN THOUGH THE FULL DECLARATION HAS NOT BEEN SEEN."

USE YOUR PEN OR PENCIL TO COVER UP THE INCOMPLETE TYPE SPECIFICATION AND EXPLAIN THE CIRCULARITY PROBLEM. AFTER THE EXPLANATION, REMOVE THE PENCIL AND SHOW HOW THE INCOMPLETE TYPE DECLARATION SOLVES THE PROBLEM.

USING ACCESS TYPES:
LINKED LISTS

CONTEXT:

```

type Vehicle_Type is (Ship, Tank, Plane);
type Vehicle_Record is
  record
    Type_Of_Vehicle   : Vehicle_Type;
    Weight            : Natural;  -- tons
    Date_Of_Acquisition : Date;
  end record;

```

EXAMPLE:

```

type Vehicle_List_Entry; -- INCOMPLETE TYPE SPECIFICATION
type Link is access Vehicle_List_Entry;
type Vehicle_List_Entry is
  record
    V_Rec : Vehicle_Record;
    Next  : Link;
  end record;

```

P,Base : Link := null; -- INITIAL VALUE IS NULL EVEN IF NOT STATED EXPLICITLY

...

```

for Vehicle in Vehicle_Type
loop

```

```

  P := new Vehicle_List_Entry;
  P.V_Rec.Type_Of_Vehicle := Vehicle;  -- set up new entry
  P.V_Rec.Weight := 250;
  P.V_Rec.Date_Of_Acquisition := (3, 8, 1983);
  P.Next := Base;
  Base := P;
end loop;

```

INSTRUCTOR NOTES

IF POSSIBLE, LEAVE THE SLIDE WITH THE CODE UP WHILE THIS IS EXPLAINED.

NOTE THAT THE OBJECTS ARE ADDED TO THE FRONT OF THE LINKED LIST.

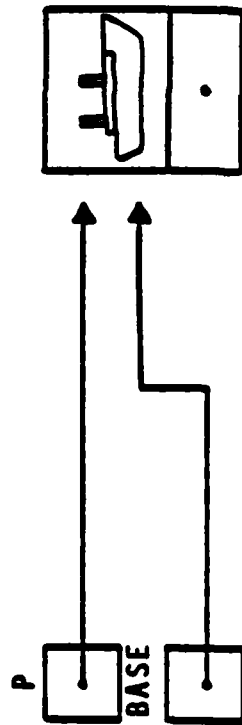
"THE LINKED LIST EXISTS AS LONG AS THE OBJECTS ARE IN SCOPE.

LINKED LIST DIAGRAM

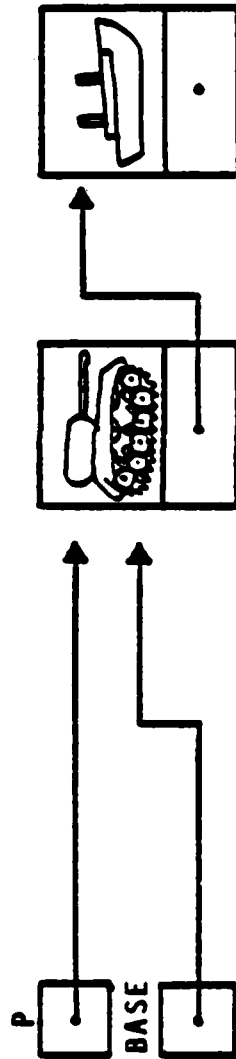
INITIAL



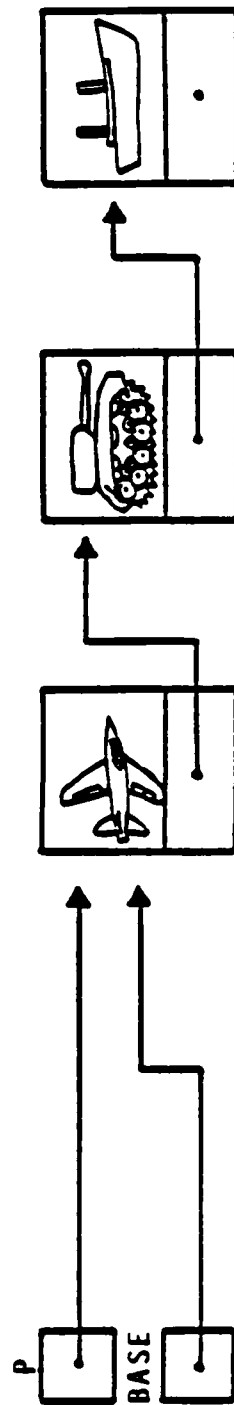
1st ITERATION



2nd ITERATION



3rd ITERATION



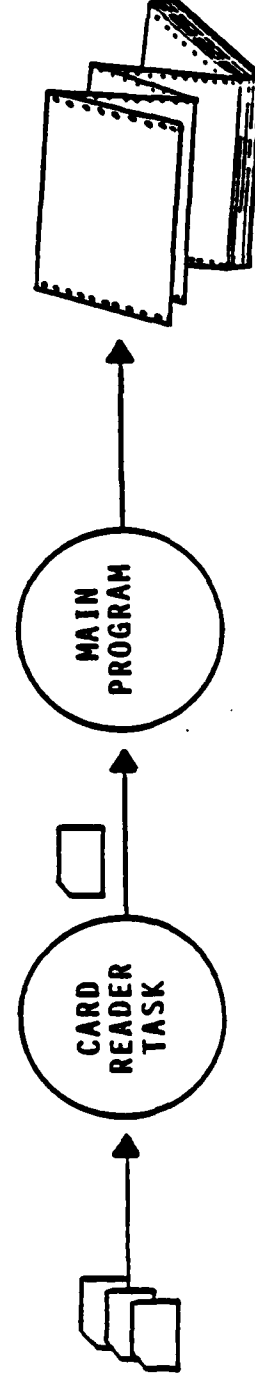
INSTRUCTOR NOTES

POINT OUT THAT THERE IS A PRAGMA CALLED PRIORITY WHICH CAN SET TASK PRIORITIES.
IMPLEMENTATIONS ARE NOT REQUIRED TO SUPPORT THIS.

TASKS -- BASIC IDEA

- TASKS ALLOW MULTIPLE THREADS OF CONTROL
- REAL OR APPARENT CONCURRENCY
- RENDEZVOUS FOR SYNCHRONIZATION AND EXCHANGE OF INFORMATION BETWEEN TASKS
- RUNTIME SYSTEM HANDLES TASK SCHEDULING

e.g COPY CARD IMAGES FROM "STANDARD INPUT" TO "STANDARD OUTPUT" --
READING IS IMPLEMENTED AS TASK EXECUTING CONCURRENTLY WITH MAIN
PROGRAM



INSTRUCTOR NOTES

ENTRIES ARE WHERE RENDEZVOUS CAN OCCUR. STATEMENTS MAY BE EXECUTED DURING RENDEZVOUS.

TASK SYNTAX

```
task Task_Name is
.
.
.
end Task_Name;
```

SPECIFICATION

- Visible Part
- Only Entry Declarations

```
task body Task_Name is
.
.
.
end Task_Name;
```

BODY

- Hidden
- Additional Declarations
- Statements
- Accept Entries

INSTRUCTOR NOTES

- REMIND THE STUDENTS THAT Card_Reader KNOWS ABOUT Card BECAUSE ITS SCOPE IS AS IF IT WERE IN Main BECAUSE OF THF separate.
- "THE RENDEZVOUS COORDINATES THE Get AND Put."
- THE TASK IS ACTIVATED JUST PRIOR TO THE begin IN Main.

MAIN PROGRAM

```

with Text_IO;
procedure Main is
  type Card is String (1 .. 80);
  C : Card;

  task Card Reader is
    entry Get (C: out Card);
  end Card_Reader;

  task body Card_Reader is separate;

begin -- Main
  loop
    Card_Reader.Get (C);
    Text_IO.Put (C);
  end loop;
end Main;

```

SPECIFICATION

BODY

ACCEPT

ENTRY CALL

CARD READER TASK

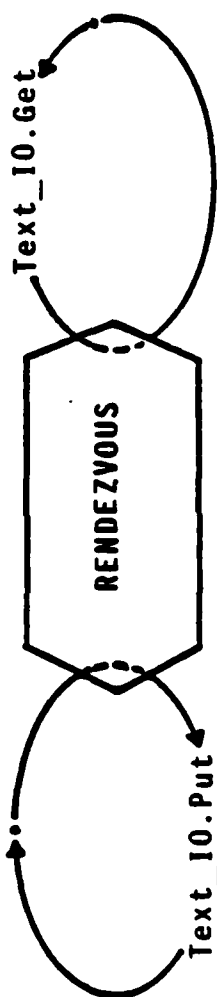
```

separate (Main)

task body Card_Reader is
  Latest_Card : Card;

begin -- Card_Reader
  loop
    Text_IO.Get (Latest_Card);
  accept Get (C : out Card) do
    C := Latest_Card;
  end Get;
  end loop;
end Card_Reader;

```



INSTRUCTOR NOTES

VG 728.2

16-81

TASKS -- RENDEZVOUS

WHEN EITHER A CALLED TASK OR ITS CALLER ARRIVE AT A RENDEZVOUS POINT, THE ONE WHO ARRIVES FIRST WAITS FOR THE OTHER ONE

DURING THE RENDEZVOUS (THE EXECUTION OF THE STATEMENTS IN THE ACCEPT) THE CALLER IS SUSPENDED. THE CALLER RESUMES EXECUTION AFTER THE RENDEZVOUS IS COMPLETED.

INSTRUCTOR NOTES

"SELECT STATEMENTS ALLOW A TASK TO WAIT FOR MORE THAN ONE RENDEZVOUS."

SELECT STATEMENTS

TASK CAN WAIT FOR ONE OF SEVERAL ALTERNATIVES

THREE FORMS

- SELECTIVE WAIT : TASK BEING CALLED
- CONDITIONAL ENTRY : CALLING TASK
- TIMED ENTRY : CALLING TASK

INSTRUCTOR NOTES

11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100

TASK TYPES -- BASIC IDEA

- ALLOWS THE USER TO CREATE A TASK TEMPLATE
- TASK OBJECTS CAN THEN BE CREATED FROM THIS TEMPLATE - OBJECTS ARE LIKE ANY OTHER TASK

INSTRUCTOR NOTES

- POINT OUT THAT OBJECTS OF TASK TYPES ALSO BEGIN EXECUTION WITH THE begin STATEMENT.

TASK TYPES -- BASIC IDEA

EXAMPLE:

```
task type Simple_Channel is
    entry Send (M : in Message);
    entry Receive (M : out Message);
end Simple_Channel;

task body Simple_Channel is
    -- LIKE A TASK DEFINITION
end Simple_Channel;

Channel_1, Channel_2 : Simple_Channel;
.
.
.
Channel_1.Receive (Msg); -- GET A MESSAGE
Channel_2.Send (Msg); -- AND SEND ON OTHER CHANNEL
```

INSTRUCTOR NOTES

POINT OUT THAT THESE ARE LOW LEVEL FEATURES OF THE LANGUAGE.

REPRESENTATION SPECIFICATIONS -- BASIC IDEA

- MAP ABSTRACT DATA TO PHYSICAL HARDWARE
- DECLARED AFTER THE ITEMS THEY REPRESENT
- CAPABILITIES:
 - SPECIFY INTERNAL CODES FOR LITERALS OF AN ENUMERATION TYPE
 - SPECIFY RECORD LAYOUT
 - SPECIFY AMOUNT OF STORAGE ASSOCIATED WITH A TYPE
 - SPECIFY REQUIRED STORAGE ADDRESS FOR AN ENTITY

AD-A166 367

ADA (TRADE NAME) TRAINING CURRICULUM BASIC ADA
PROGRAMMING L202 TEACHER'S GUIDE VOLUME 2(U) SOFTECH
INC WALTHAM MA 1986 DAAB07-83-C-K514

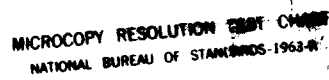
7/8

UNCLASSIFIED

F/G 9/2

NL





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

INSTRUCTOR NOTES

- POINT OUT THAT THE INTEGER CODES SPECIFIED FOR THE TYPE MUST HAVE THE SAME ORDERING AS THE ENUMERATION LITERALS.

I.E.

AN ENUMERATION CLAUSE FOR Boolean CANNOT BE
for Boolean use (False => 1, True => 0);

BECAUSE False MUST HAVE A VALUE LESS THAN True. POINT OUT THAT THE TWO REPRESENTATION CLAUSES ARE JUST TWO WAYS OF STATING THE SAME THING.

ENUMERATION TYPE REPRESENTATION CLAUSES

- MAPPING FROM ELEMENTS OF A TYPE TO SPECIFIC INTERNAL CODES

type Op_Codes is ('00, ST0, ADD, SUB, JNZ);

for Op_Codes use (1, '0, 11, 12, 15);

INSTRUCTOR NOTES

- at CLAUSE IS USED TO SPECIFY AN OBJECT'S PHYSICAL ADDRESS
- ALIGNMENT CLAUSE SPECIFIES STORAGE BOUNDARY FOR THE RECORD

RECORD TYPE REPRESENTATION CLAUSES

- MAPS
 - ORDER OF RECORD COMPONENTS
 - COMPONENT POSITION
 - COMPONENT SIZE
 - GLOBAL ALIGNMENT OF RECORD
- at CLAUSE
 - LOCATES COMPONENT RELATIVE TO START OF RECORD
 - MEASURED IN STORAGE UNITS
- RANGE IN BITS
 - LOCATION AND EXTENT OF A COMPONENT RELATIVE TO A STORAGE_UNIT
- ALIGNMENT CLAUSE
 - STORAGE BOUNDARY FOR BEGINNING OF AN OBJECT

INSTRUCTOR NOTES

LENGTH SPECIFICATION

- USED TO OVERRIDE THE AMOUNT OF STORAGE NORMALLY ASSOCIATED WITH AN ENTITY
- ATTRIBUTES
 - DATA TYPE
 - T'Size : NUMBER OF BITS FOR OBJECTS OF TYPE T
 - ACCESS TYPE
 - T'Storage_Size : NUMBER OF STORAGE UNITS TO BE RESERVED FOR ALLOCATING OBJECTS
 - TASK OR TASK TYPE
 - T'Storage_Size : NUMBER OF STORAGE UNITS TO BE RESERVED FOR EACH ACTIVATION OF THE TASK OBJECT
- for Attribute use Integer_Expression;
for Vehicle_Record'Size use 1000;

INSTRUCTOR NOTES

- "THE REPRESENTATION CLAUSE MUST APPEAR AFTER THE DECLARATION OF THE ENTITY IT REFERS TO. IT MUST BE IN THE SAME DECLARATIVE REGION."

FOR EXAMPLE:

```
declare
type Rec is
record
    Cl : Boolean;
end record;
begin
declare
for Rec use -- **ILLEGAL
record at mod 2
    Cl at 0 range 0 .. 0;
end record;
begin
    null;
end;
null;
end;
```

ADDRESS SPECIFICATION

- SPECIFIES THE LOCATION OF AN OBJECT IN STORAGE
 - SPECIFIES THE STARTING ADDRESS OF A PROGRAM UNIT
 - CAN BE USED TO ASSIGN AN ENTRY TO A HARDWARE INTERRUPT
- FORMAT:

for Name use at Interrupt_Address;

- Name IS

- OBJECT

- SUBPROGRAM, PACKAGE, OR TASK

- ENTRY

e.g.

for Boot_Program use at 256;

task Fire_Control_Interrupt_Handler is
entry Fired;
for Fired use at 176;
end Fire_Control_Interrupt_Handler;

INSTRUCTOR NOTES

THE INTERFACE PRAGMA IS A WAY OF TELLING THE COMPILER THAT A SUBPROGRAM BODY IS WRITTEN
IN SOME OTHER LANGUAGE, E.G. ASSEMBLER OR FORTRAN.

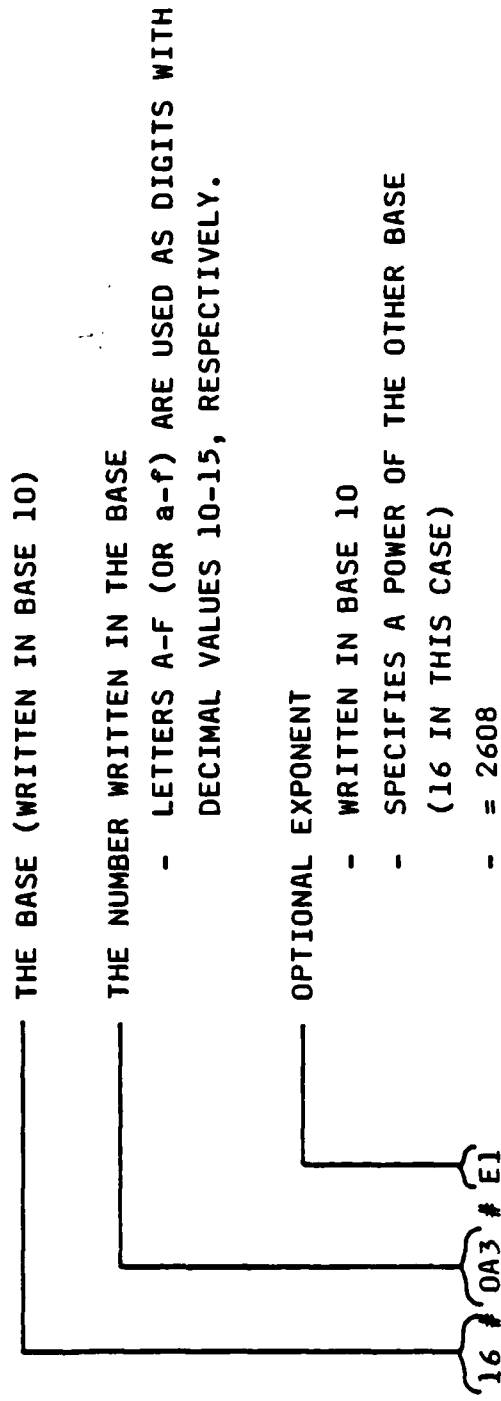
MACHINE CODE INSERTIONS -- BASIC IDEA

- Ada PERMITS INSERTION OF MACHINE CODE INSTRUCTIONS, USING THE CODE STATEMENTS
- CODE STATEMENTS ARE RECORD AGGREGATES WHOSE COMPONENTS CORRESPOND TO FIELDS OF A MACHINE LANGUAGE INSTRUCTION.
- SPECIFICS ARE IMPLEMENTATION-DEPENDENT
- CAN ONLY BE USED IN A PROCEDURE IN WHICH THERE ARE NO DECLARATIONS AND EVERY STATEMENT IS A CODE STATEMENT.
- SHOULD BE USED ONLY FOR VERY SHORT SEQUENCES OF MACHINE LANGUAGE INSTRUCTIONS (1-5 INSTRUCTIONS). FOR LONGER SEQUENCES, WRITE AN ASSEMBLY LANGUAGE MODULE AND USE THE INTERFACE PRAGMA.
- IN GENERAL, A VERY CLUMSY WAY OF WRITING MACHINE INSTRUCTIONS IN Ada.

INSTRUCTOR NOTES

BASED NUMBERS -- BASIC IDEA

- NUMERIC LITERALS THAT ARE WRITTEN IN BINARY, OCTAL, OR HEXADECIMAL (OR ANY OTHER BASE FROM 2 TO 16.)
- MAY BE APPROPRIATE WHEN DEALING DIRECTLY WITH HARDWARE OR WITH ASSEMBLY LANGUAGE SOFTWARE.
- BASED INTEGER LITERAL (NO RADIX POINT, EXPONENT MAY NOT BE NEGATIVE):



- BASED REAL LITERAL (RADIX POINT WITH AT LEAST ONE DIGIT ON EACH SIDE):
2#0.1#E-3 (- (1/2) * 2.0 ** (-3) = 1/16)

INSTRUCTOR NOTES

USE OF BASED NUMBERS

for Boot_Program use at 16#0100#; -- in Address Specification
for Fired use at 16#0A30#; -- associated with Task Entry Specification

END

Dtic

5-86

AD-A166 367

ADA (TRADE NAME) TRAINING CURRICULUM BASIC ADA
PROGRAMMING L202 TEACHER'S GUIDE VOLUME 2(U) SOFTECH
INC WALTHAM MA 1986 DAB807-83-C-K514

8-18

UNCLASSIFIED

F/G 9/2

NL





MICROCOPY

CHART

SUPPLEMENTARY

INFORMATION

AD - A166366
Material: Basic Ada Programming (L202), Volume I ~~AD-166366~~

We would appreciate your comments on this material and would like you to complete this brief questionnaire. The completed questionnaire should be forwarded to the address on the back of this page. Thank you in advance for your time and effort.

1. Your name, company or affiliation, address and phone number.

2. Was the material accurate and technically correct?

Yes ☐

No ☐

Comments:

3. Were there any typographical errors?

Yes ☐

No ☐

If yes, on what pages?

4. Was the material organized and presented appropriately for your applications?

Yes ☐

No ☐

Comments:

5. General Comments:

END

DTic

6-86